В. А. Вишняков Д. Ю. Буланже
О. В. Герман

АППАРАТНО-ПРОГРАММНЫЕ СРЕДСТВА ПРОЦЕССОРОВ ЛОГИЧЕСКОГО ВЫВОДА

В. А. Вишняков Д. Ю. Буланже
О. В. Герман

АППАРАТНО-ПРОГРАММНЫЕ СРЕДСТВА ПРОЦЕССОРОВ ЛОГИЧЕСКОГО ВЫВОДА

Москва
"РАДИО И СВЯЗЬ"
1991

УД[redacted]1.3.06 : 519.681

Вишняков В. А., Буланже Д. Ю., Герман О. В. Аппаратно-программные средства процессоров логического вывода. — М.: Радио и связь, 1991. — 264 с.: ил. — **ISBN 5-256-00606-1.**

Рассмотрены теоретические аспекты построения аппаратно-программных средств процессоров, поддерживающих механизмы логического вывода в версиях языка Пролог. Описаны архитектура и системы команд процессоров логического вывода (ПЛВ), механизмы компиляции Пролог-программ и организации скомпилированных программ. Указаны подходы к моделированию систем логического вывода. Даны решения по выполнению специализированных компонентов ПЛВ в виде СБИС. Подробно рассмотрены вопросы построения программно-аппаратных эмуляторов Пролог-машины на основе существующих персональных ЭВМ, технологии микропрограммирования ПЛВ, а также программно-аппаратные средства отладки Пролог-процессоров.

Приведены модели и методы логического вывода, а также схемы параллелизма в логических программах и подходы к эффективному представлению системных структур данных в Прологе.

Для научных работников; может быть рекомендована инженерам.

Табл. 21. Ил. 82. Библ. 53 назв.

Рецензент: канд. техн. наук В. Н. Захаров

Редакция литературы по информатике и вычислительной технике

В $\frac{2404010000\text{-}130}{046(01)\text{-}91}$ **126-91**

ISBN 5-256-00606-1

ПРЕДИСЛОВИЕ

Широкий спектр исследований, проводимых с начала 80-х годов в области искусственного интеллекта, программа создания ЭВМ пятого поколения, провозглашенная Японией и поддержанная ведущими индустриальными странами мира, инициировали поток разработок и публикаций в этой области. Одним из направлений этих исследований является логическое программирование как технология и язык программирования логического вывода.

Практическая реализация логического вывода ориентирована на решение широкого диапазона задач — от чисто научных до производственных. Для практической реализации логического вывода необходимо решить три основные концептуальные проблемы.

Первая проблема связана с разработкой методов логического вывода. В этом смысле, вероятно, отправным методом следует считать разработанный Дж. Робинсоном (1965 г.) метод резолюций. Существуют и другие не менее сильные методы логического вывода: обратный дедуктивный метод Маслова, вывод на основе интерпретаций Бета, сравнений по образцу (pattern matching), семантических сетей и др. Тем не менее именно для метода резолюций была разработана Р. Ковальским идея процедурной интерпретации логического вывода, которая дала начало собственно логическому программированию как языку, технологии и парадигме. Поэтому методы эффективной программной реализации логического вывода образуют вторую проблему. Несомненно, что среди языков логического программирования ведущее место в этом отношении занимает язык Пролог и его многочисленные версии.

Наконец, третья глобальная проблема — это аппаратная поддержка логического вывода, необходимость которой обусловлена следующими причинами:

переборным характером процедуры вывода;

использованием нетрадиционных сложных структур данных (списков, функторов, альтернативных типов);

наличием специфических операций, не имеющих аналогов в других языках (унификация, возврат, отсечение, двунаправленная передача параметров из вызывающей и вызываемой процедур).

При таком рассмотрении исходная Пролог-программа должна быть скомпилирована в программу специализированного процессора и обработана.

Отечественной литературы по аппаратному и программному обеспечению логического вывода явно недостаточно. В предлагаемой книге рассматриваются вопросы компиляции Пролог-программ на уровне абстрактной машины логического вывода, определяются основные понятия и принципы организации абстрактной машины (на примере машины Уоррена). Описаны аппаратные реализации процессоров логического вывода, в частности эмулятор процессора логического вывода на базе персональной ЭВМ, а также перспективные направления логического вывода и соответствующие математические модели.

Главы 1, 2 написаны Д. Ю. Буланже, введение, заключение, § 6.1, 6.2, 9.1, 9.4—9.6 — О. В. Германом, § 3.1, 4.1, 4.3, 4.4, 5.2, 5.6, 6.5, 7.5, 8.3, 9.3 — совместно В. А. Вишняковым и О. В. Германом, остальной материал — В. А. Вишняковым.

ВВЕДЕНИЕ

Язык Пролог, с одной стороны, выступает как декларативный язык со своей семантикой и синтаксисом, а с другой — как система логического вывода, основанная на формальном исчислении Хорна. В своей первой ипостаси Пролог ориентирован на широкий круг пользователей-программистов, а во второй — представляет интерес для специалистов в области формальных систем и математической логики. Рассмотрение этих сторон логического программирования необходимо для понимания компиляции Пролог-программ и аппаратной реализации его основных механизмов.

В.1. СИНТАКСИС И СЕМАНТИКА ПРОЛОГ-ПРОГРАММ

Пролог-программа записывается как множество правил со структурой «если — то». Нотация

a :— b, c, d, ... , e

означает «если b&c&d& ... &e, то верно a» (& — логическая операция И), где a, b, c, ... , e — имена процедур. Единственным логическим действием процедуры является присваивание значений ее параметрам. Цель программы — найти такие значения для переменных, которые удовлетворяют данным правилам. Рассмотрим следующие правила:

(1) площадь (X, Y) : — фигура (X, Y), write ("X=", X, "Y=", Y).
(2) фигура (квадрат, "сторона*сторона").
(3) фигура (круг, "π/2*радиус*раднус").
(4) фигура (треугольник, "высота*основание*1/2").
(5) фигура (_, "неизвестно").

Пусть целью будет площадь (круг, Z). Это значит, нужно найти значение Z, соответствующее площади круга. Пролог пытается сначала найти правило с именем «площадь». Отождествим имя правила с именем процедуры, которое в конструкции "если (...), то (...)" указано в скобках после "то (...)". В данном случае правило 1 имеет вид

площадь (X, Y) : — фигура (X, Y).

Параметры обеих процедур здесь совпадают и являются переменными (которые принято записывать прописными буквами).

Первый шаг, выполняемый Пролог, — это передача значений X и Y из цели: X=круг, Y=Z. В нашем случае X получает конкретное значение "круг", запись Y=Z просто устанавливает факт равенства двух переменных. Как только переменная Y получает значение, Z присваивается то же значение. Теперь правило 1 трансформируется в правило следующего вида:

площадь (круг, Y) : — фигура (круг, Y).

Пролог обязан запомнить вид этого правила, что равносильно следующей записи:

если (фигура (круг, Y)), то (площадь (круг, Y)).

Теперь Пролог формирует новую цель (фигура (круг, Y)). Для этой новой цели он ищет подходящее правило. В нашей программе таких правил четыре (2—5). Эти правила имеют усеченный вид: в структуре "если (...), то (...)" отсутствует часть "если (...)". С точки зрения логики это означает, что независимо от исходных условий, записываемых в части "если (...)", эти правила утверждают истинность частей "то (...)". Следовательно, доказывать такие правила не надо.

Итак, для цели "фигура (круг, Y)" выбирается правило 2. Как и на первом шаге, выполняется попытка передачи значений переменным. Иначе говоря, получается следующее:

круг=квадрат; Y="сторона*сторона".

Учитывая, что "круг" и "квадрат" не переменные (первый символ "к" строчной), Пролог вместо присваивания сравнивает их. Результат сравнения — "ложь". Следовательно, правило 2 использовать нельзя. Однако есть еще подходящее правило с именем "фигура". Поэтому Пролог выбирает следующее по порядку — правило 3. Текущая цель — фигура (круг, Y). Результат присваивания значений:

круг=круг; Y="π/2*радиус*радиус".

Цель доказана, поскольку доказывать правило 3 не надо (о чем было сказано выше).

Поскольку выше было принято, что Z=Y, то Z автоматически получает значение Z="π/2*радиус*радиус".

Стандартная процедура write выведет найденные значения X и Y, напечатав:

X=треугольник, Y="π/2*радиус*радиус".

Из данного примера видим, что выбор подходящего правила для доказательства цели не всегда возможен из-за несоответствия значений параметров. Поведение Пролога полностью соответствует стратегии поиска выхода из лабиринта.

Пометим стрелкой (на стене лабиринта) направление нашего движения. Предположим, мы попали в место развилки. Если все пути в развилке помечены стрелкой, то возвращаемся по пути в направлении против стрелки. Если таких путей несколько, то возвращаемся по тому, по которому шли в последний раз (для этого следует присвоить каждому пути номер, такой, чтобы последний новый путь получал номер старше всех номеров уже пройденных путей). Если на развилке есть непройденный путь, то помечаем его стрелкой и

следуем по этому пути. Если при движении возник тупик, то возвращаемся по этому пути обратно до предыдущей развилки. Эта стратегия полная. Иначе говоря, она выведет нас из лабиринта, если, и только если есть хотя бы один маршрут, выводящий из лабиринта.

Механизм вывода в Прологе аналогичен описанному. Выбор нового пути в лабиринте соответствует выбору нового правила для доказательства текущей цели. Возврат против стрелки означает неудачу при попытке доказать текущую цель и соответствует переходу к доказательству предыдущей цели (заходу в предыдущую развилку). Этот процесс называется возвратом (backtracking). При этом восстанавливаются все переменные, которым были присвоены значения «ниже» этой позиции в программе.

Рассмотренный выше пример позволяет выделить следующие шаги выполнения программы:

1. Выполнение подходящего правила для текущей цели.
2. Передача и согласование параметров цели и правила.
3. В зависимости от исхода шага 2 определение текущей (новой) цели или возврат к предыдущей цели с отменой значений для переменных, полученных в результате выполнения шага 2 для последней цели.
4. Создание точек ветвления в программе (по аналогии с движением в лабиринте это соответствует заходу в развилку).
5. Инициализация переменных процедуры и восстановление их значений при возврате (движение против стрелки в лабиринте).

Чтобы приведенный выше программный фрагмент представлял законченный модуль на языке Turbo Prolog 2.0, необходимо описать структуру программы, включающую разделы данных, объявления процедур, цели и правил. Законченный вариант имеет следующий вид:

```
domains                              /*раздел данных*/
    тип_фигуры=symbol
    выражение_для_площади=string
predicates                           /*раздел процедур*/
    площадь (тип_фигуры, выражение_для_площади)
    фигура (тип_фигуры, выражение_для_площади)
goal
    площадь (круг, Z).
clauses
    площадь (X, Y) :— фигура (X, Y), write ("X=", X, "Y=", Y).
    фигура (квадрат, "сторона*сторона").
    фигура (круг, "π/2*радиус").
    фигура (треугольник, "высота*основание*1/2").
    фигура (_, "неизвестно").
```

В разделе данных domains объявлены типы данных пользователя "тип_фигуры" и "выражение_для_площади", указано, что "тип_фигуры" эквивалентен стандартному символьному типу (symbol), а тип "выражение_для_площади" — стандартному строковому типу (string).

Другие стандартные типы данных на языке Turbo Prolog 2.0 — это целый (integer), вещественный (real), знаковый (char), файловый ((file). Кроме того, в языке можно объявлять списки, смешанные и составные типы.

Список представляет совокупность элементов указанного типа. Например, для объявления списка имен вводим

```
domains
list_of_names=symbol*
```

указателем списочного типа является *. Здесь объявлен списочный тип list_of_names, представляющий собой список элементов символьного типа, например следующий: [яблоко, апельсин, слива, дыня].

Пролог предоставляет возможность выделить первый элемент списка, отделив его от остальных. Это выполняется с помощью следующей формы: [X|Rest_of_List], в которой переменная X получает значение первого элемента некоторого списка, а переменная Rest_of_List связывается с его оставшейся частью. Рассмотрим пример программы подсчета суммы элементов целочисленного списка:

```
domains
        list_of_integer=integer*
predicates
        sum(list_of_integer, integer, integer)
goal
        sum([1, 2, 3, 4, 5], Ø, Summa),
        write("Сумма элементов=", Summa).
clauses
(1)     sum([ ], R, R).
(2)     sum([X|Rest], Sumcurrent, R) :-
        SumNew=Sumcurrent+X,
        sum(Rest, SumNew, R).
```

В этом примере ставится цель — найти сумму элементов списка [1, 2, 3, 4, 5], а полученное значение суммы присвоить переменной Summa и вывести его на печать. В разделе правил всего два правила относятся к процедуре sum. Правило 1 соответствует случаю, когда список пуст ([ ]). Второй параметр в sum, определяющий накопленную сумму, присваивается третьему параметру, в котором фиксируется результат. Заметим, что в исходной цели второй параметр равен 0, определяя начальную накопленную сумму, а третий параметр не определен, поскольку он должен быть вычислен. Правило 2 рекурсивно, поскольку ссылается на процедуру с тем же именем sum. Посмотрим, как выполняется эта программа.

Сначала в качестве доказываемой цели берется sum([1, 2, 3, 4, 5,], 0, Summa). Правило 1 для доказательства этой цели не подходит, поскольку сравнение первых аргументов невозможно: [1, 2, 3, 4, 5]≠[ ]. Пороролог пытается использовать правило 2. В результате передачи значений параметров получается

$$X=1,\ Rest=[2, 3, 4, 5],\ SumCurrent=0,\ R=Summa.$$

Далее вычисляется значение SumNew: SumNew=0+1=1. Отметим, что арифметическое выражение для вычисления SumNew не является логической «конструкцией» языка Пролог. «Внешние» процедуры в языке Пролог (арифметика, ввода-вывод, работа со строками и файлами и др.) реализованы в виде стандартных библиотечных модулей. Эти нелогические «конструкции» обрабатываются как обычные операторы в процедурных языках.

После вычисления SumNew Пролог создает новую текущую цель: sum([2, 3, 4, 5], 1, R), после чего опять ищет правило для этой цели. Правило 1 по-прежнему не годится, поэтому снова выбирается правило 2. Происходит установка значений параметров:

$$X=2,\ Rest=[3, 4, 5],\ SumCurrent=1,\ R1=R2.$$

Необходимо четко различать, что фактически речь идет не об отмене старых значений переменных X, Rest, SumCurrent и R, а о создании новых копий этих переменных в памяти ЭВМ с новыми значениями. Этот момент принципиальный. Не продолжая далее «раскрутку» программы, приведем формируемые значения для переменных в виде следующей таблицы:

Номер шага	Номер правила	Устанавливаемые значения переменных процедур
1	2	X=1, Rest=[2, 3, 4, 5], SumCurrent=0, R1=Summa
2	2	X=2, Rest=[3, 4, 5], SumCurrent=1, R2=R1
3	2	X=3, Rest=[4, 5], SumCurrent=3, R3=R2
4	2	X=4, Rest=[5], SumCurrent=6, R4=R3
5	2	X=5, Rest=[ ], SumCurrent=10, R5=R4
6	1	R6=SumNew=15. Значение SumNew определяется в вызове sum([ ]), 15, R)

Чтобы не было путаницы в присваивании R=R, мы использовали индекс, который определяет номер копии данной переменной. Из таблицы нетрудно видеть следующую цепочку равенств:

$$Summa=R1=R2=R3=R4=R5=R6=SumNew=15,$$

устанавливающую результирующее значение для искомой переменной Summa, которое будет напечатано командой write.

На примере данной программы мы познакомились с рекурсией — одним из основных механизмов Пролога. Как видно из представленной выше таблицы, на каждом шаге Пролог определяет, какое правило использовать, и создает для него область, в которую помещает устанавливаемые значения параметров процедур. Присваивание значений выполняется в следующей форме:

"переменная"="константа"

или

"переменная"="переменная".

Поскольку рекурсия в Прологе один из важнейших механизмов, используемых, в частности, для построения циклов, то ее реализация в значительной степени определяет эффективность самих Пролог-программ. Проблема реализации рекурсии в Прологе сводится к проблеме памяти. Когда одна процедура вызывает другую, то состояние вызывающей процедуры в момент вызова должно быть запомнено, чтобы впоследствии можно было возобновить вызывающую процедуру с этого состояния. Это значит, что если процедура рекурсивно вызы-

вает себя k раз, то необходимо k раз сохранять ее состояния. Это может привести к быстрому исчерпыванию памяти и невозможности продолжать выполнение программы.

Эффективным способом решения указанной проблемы в Прологе является использование хвостовой рекурсии.

Обратимся к примеру подсчета суммы ряда чисел от 1 до K:

```
goal
    write("Введите K").          /* запрос числа K */
    readint(K),                  /* чтение числа K с клавиатуры */
    count(K, 0, Result),         /* вызов правила для подсчета */
    write("ymma=", Result).      /* суммы 1+2+ ... +K */
clauses
(1) count (X, CurrentSum, Summa) :-
        X<>0,
        XNew=X-1,
        SummaNew=CurrentSum+X,
        count(XNew, SummaNew, Summa).
(2) count( 0 , Result, Result).
```

В этом примере два правила для count(...). Сначала используется правило для накапливания промежуточной суммы путем выполнения оператора

SummaNew=CurrentSum+X.

После этого оператора следует рекурсивный вызов правила count(...) с новыми параметрами. В момент вызова Пролог сохраняет состояние логического вывода, соответствующее текущему выполняемому шагу программы. Причем единственное, что «заставляет» его это сделать, — наличие неиспользованной альтернативы (правила 2), поскольку не ясно, успешно ли закончится правило 1. При выполнении этой программы Пролог создает ровно K копий состояний логического вывода только потому, что альтернативное правило к моменту рекурсивного вызова остается неопробованным. В этом случае принцип хвостовой рекурсии потребует поменять местами правила 1 и 2. Такой простой шаг обеспечит то, что рекурсивный вызов будет последним и к моменту его выполнения не окажется неиспользованной альтернативы (поскольку теперь правило 1 станет последним непомеченным путем в развилке лабиринта с именем count). Поэтому нет необходимости запоминать состояние логического вывода в этот момент.

Другой способ обеспечить хвостовую оптимизацию — использовать оператор отсечения «!». При этом структуру программы менять не придется, за исключением того, что раздел процедур примет следующий вид:

```
clauses
(1)     count (X, CurrentSum, Summa) :-
        X<>0,
        XNew=X-1,
        SummaNew=CurrentSum+X,
        !,
(*)     count(XNew, SummaNew, Sum).
(2)     count( 0, Result, Result).
```


Оператор отсечения «!» запрещает использовать другие альтернативные правила с тем же именем (т. е. помечает оставшиеся пути в развилке как пройденные, хотя фактически это не так), в данном случае заставляет игнорировать правило 2 как несуществующее.

Специфическим типом данных, обрабатываемых в Прологе, является *составной тип данных*, или *функторы*. Пусть мы хотим создать структуру данных, описывающую название и автора книги. В этом случае приемлемой могла бы быть структура следующего вида:

книга (название ("Три мушкетера"), автор ("А. Дюма")).

Эта сложная запись представляет объект данных с именем «книга», состоящий из двух объектов «название» и «автор». Примером объекта с именем «название» является строка "Три мушкетера", а объекта с именем «автор» — "А. Дюма". Объявление функтора «книга» имеет следующий вид:

```
domains
    first_obj=название (string)
    second_obj=автор (string)
    third_obj=книга (first_obj, second_obj)
```

В этом объявлении составной тип данных third_obj имеет структуру функтора «книга».

Использование функторов демонстрирует программа, которая определяет, какие из перечисленных книг написал В. Гюго:

```
domains
    first_obj=название (string)
    second_obj=автор (string)
    third_obj=книга (first_obj, second_obj)
predicates
    что_написал (string)
    библиотека (third_obj)
goal
    что_написал(«В. Гюго»).
clauses
(1)   что_написал (X) :-
(*)   библиотека (книга (название(A), автор (X))),
      write (A),
      nl
      fail
(2)   что_написал(_).
(3)   библиотека (книга (название ("Труженики моря"), автор ("В. Гюго"))).
(4)   библиотека (книга (название ("Граф Монте-Кристо"), автор ("А. Дюма"))).
(5)   библиотека (книга (название ("Утраченные иллюзии"), автор ("О. Бальзак"))).
(6)   библиотека (книга (название ("Отверженные"), автор ("В. Гюго"))).
```

Эта программа также полезна, поскольку в ней явно использован оператор неудачи fail, отличающий Пролог от других языков. Этот оператор инициирует безусловный возврат в программе (т. е. имитирует ситуацию тупика в лабиринте).

Программа работает следующим образом. Для цели «что_написал ("В. Гюго")» выбирается одноименное правило 1. Переменной X присваивается значение X="В. Гюго". Далее создается новая текущая цель:

библиотека (книга (название (A), автор ("В. Гюго"))).

Для этой новой цели могут быть использованы правила 3—6. По порядку выбирается сначала правило 3. В результате переменная A получает значение A=="Труженики моря". Таким образом, текущая цель доказана. Выполняется переход к новой цели: write (A), причем A уже имеет значение. Команда write печатает название книги, затем выполняет перевод курсора на новую строку по команде nl и доходит до оператора fail. Этот оператор не имеет операндов и инициирует возврат в предыдущую точку ветвления в программе. Одна из задач компилятора — создавать точки ветвления. Каждая точка ветвления соответствует случаю, когда при доказательстве текущей цели имеется несколько одноименных процедур. В нашем примере две точки ветвления: первая соответствует процедуре с именем «что_написал», а вторая — процедурам с именем «библиотека» (3)—(6). Возврат произойдет в точку ветвления с условным именем «библиотека» (помечена как (*) в тексте программы). В месте программы, отмеченном (*), переменная A была инициализирована, т. е. ей было присвоено значение. Поскольку теперь будет испробовано новое правило с именем «библиотека» (которому соответствует еще не пройденный путь в лабиринте), то отменяется значение для A и оно становится неопределенным. Общее правило таково: должны быть отменены значения всех переменных, которые получили их ниже последней точки ветвления. Компилятор обязан поддерживать информацию о всех таких переменных и назначениях.

Далее используется правило 4. Оно оказывается несовместимым с целью из-за несоответствия фамилий авторов. Аналогичная неудачная попытка осуществляется и в отношении правила 5, после чего выполняется последнее правило 6 и печатается новая строка "Отверженные". После выполнения правила 6 точка ветвления с условным именем «библиотека» помечается как закрытая. Это означает, что не осталось никаких других альтернатив, которые еще не пройдены из данного места программы (все пути у развилки помечены). Теперь, когда оператор fail снова инициирует возврат, он уже распространится поверх точки (*) и дойдет до точки ветвления с именем «что_написал» (правило 1). Единственная альтернатива — правило 2 — выполняется безусловно при любых аргументах. Символ (_) вместо имени аргумента означает, что аргумент может быть любым.

Как видим из примеров, механизм повторения может быть реализован с помощью рекурсии и оператора fail. Последний пример также связан с проблемой представления в памяти составленных структур и списков.

Наконец, еще одним специфическим механизмом Пролога является оператор отсечения (cut), обозначаемый «!». Его можно представить как заграждение или барьер, устанавливаемый в пути лабиринта. Как только такой барьер установлен, он блокирует возможность отступления. Программа не в состоянии при возврате перейти через барьер. Это значит, что все переменные, которые получают значения «до барьера», сохраняют их до завершения доказательства правила. Это свойство очень важно: позволяет экономить память, поскольку отпадает необходимость копировать значения переменных, так как они в данном

правиле уже не могут быть отменены. Кроме того, «!» часто используется для управления логикой программ. Рассмотрим пример такого использования:

```
predicates
    rd
    count
goal
    rd.
clauses
(1) rd :— count.
(2) rd.
(3) count :—
        readln(X),        /* считывается строка с экрана и */
        X="*",            /* присваивается переменной X */
        !,
        fail.
(4) count :— count.
```

Эта программа осуществляет чтение строк с клавиатуры до тех пор, пока не будет введена строка «*». Вводимая строка сравнивается с «*». При совпадении Пролог «обходит» «!» в правиле 3, а затем начинает возврат, инициируемый оператором fail. Поскольку оператор «!» блокирует возврат, то нельзя вернуться в развилку с именем count, а это значит, что правило завершается неудачей, хотя для него еще осталась неиспытанной альтернатива 4. В свою очередь, неудачей завершается правило 1 (rd :— count). В связи с этим для цели rd выбирается правило 2, которое заканчивается успешно.

Мы дали краткое введение в синтаксис и семантику Пролога, достаточную для понимания сути его основных механизмов. Подробные сведения по языку можно получить из [15].

В.2. ОСНОВЫ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Рассмотрим основные понятия формализованной теории и логического программирования как одной из них. В качестве формализованной теории рассмотрим исчисление предикатов первого порядка (или просто исчисление предикатов). Под *предикатом* понимается переменное высказывание об объектах некоторой предметной области. Это высказывание может иметь два значения «истина» и «ложь». С формальной точки зрения природа участвующих в высказывании объектов значения не имеет. Примерами переменных высказываний являются следующие:

"если x — вещественное число, то $(x-2)(x-5)=x^2-7x+10$"

или

"каждый студент сдает экзамены".

Любое рассуждение в математических доказательствах состоит из простых шагов, каждый из которых заключается в порождении предложений (формул), являющихся логическими следствиями других. Эти шаги могут рассматриваться как элементарные логические рассуждения, называемые правилами вывода.

Под *правилом вывода* понимается операция, которая конечной последова-

тельности формул $a_1, a_2, \ldots, a_n$ языка ставит некоторую формулу b этого языка, являющуюся логическим следствием формул $a_1, \ldots, a_n$:

$$a_1, \ldots, a_n \vdash b.$$

Формулы $a_1, \ldots, a_n$ называются *посылками*, а b — *заключением* правила вывода. Выводимость формулы b из формул $a_1, \ldots, a_n$ понимается в смысле тождественной истинности формулы следующего вида:

$$(a_1 \& a_1 \& \ldots \& a_n) \to b,$$

где $\to$ — знак импликации.

Для записи формул используются знаки и правила, позволяющие строить правильные формулы этого языка. Знаками являются:

1, 2, 3, ... — для постоянных объектов.

+, —, ... — для постоянных функций,

t (истина) и f (ложь) — для постоянных высказываний,

=, <, > — для постоянных признаков,

&, $\vee$, $\to$, , $\neg$ — для логических переменных операторов,

(,) — вспомогательные,

a, b, ..., m и n, ..., x, y, z — для свободных и связанных переменных,

A, B, C, ... — для переменных высказываний.

Предметные переменные называются аргументами переменного высказывания.

Понятие формулы задается индуктивно:

1) знаки t и f есть формулы;

2) переменная высказывания с некоторым числом (возможно, равным нулю) следующих за ней свободных предметных переменных есть формула;

3) если U и V — формулы, то таковыми являются и $\overline{U}$, $\overline{V}$, $U\&V$, $U \vee V$, $U \to V$, $U \sim V$;

4) из формулы, в которую не входит связанная предметная переменная x, получается новая формула с использованием кванторов, т. е. если x — переменная, а $U(x)$ — формула, то $\forall x(U(x))$ и $\exists x(U(x))$ также формулы, других формул в языке нет.

Логический оператор $\forall x$ называется *квантором всеобщности*, а логический оператор $\exists x$ — *квантором существования*. Формула $\forall x(U(x))$ интерпретируется как *утверждение*: $U(x)$ истинно для любых значений x, т. е. не зависит от содержания (смысла) $U(\cdot)$ и x, а истинно только в силу формы выражения для U. Областью действия оператора $\forall x$ в формуле $\forall x(U(x))$ является само выражение $U(x)$. Формула $\exists x(U(x))$ эквивалентна формуле $\overline{\forall x}(\overline{U}(x))$. В логике предикатов переменные формулы, связанные кванторами $\forall$ и $\exists$ и находящиеся в области их действия, называются *связанными*, несвязанные переменные называются *свободными* или *пустыми*.

Формулы вида 1, 2, называются элементарными. Примеры более сложных формул:

$$\forall x(\forall y(P(a, x, y) \vee Q(b, x, y))),$$

$$R(a, x, w) \sim \overline{T}(a, x, w).$$

В дальнейшем, чтобы не перегружать формулы скобками, будем их опускать,

где это возможно, с учетом старшинства логических операторов, устанавливаемого следующей схемой:

уровень 0: $\exists$, $\forall$, $\neg$

уровень 1: &

уровень 2: $\vee$

уровень 3: $\to$, $\sim$

Операторы, расположенные на уровне с меньшим номером, являются старшими по отношению к операторам, расположенным на уровне с бо́льшим номером. В силу сказанного, например, формула $\forall x P(x) \& Q(x)$ должна пониматься как $(\forall x(P(x))) \& Q(x)$.

Формула, все переменные которой связаны, называется *высказыванием* или *замкнутой формулой*. Очевидно, что формула без переменных — это частный вид высказывания.

Выводимость формулы F из G обозначим $G \vdash F$. Будем говорить, что формула A выводима из A_0 с помощью правил вывода $R_1, \ldots, R_r$, если существует такая конечная последовательность формул

$$a_1, a_2, \ldots, a_n,$$

что A совпадает с a_n, для каждого $i \leqslant n$ либо $a_i \in A_0$, либо a_i является непосредственным следствием некоторых из формул $a_1, \ldots, a_{i-1}$ в силу одного из правил вывода $R_1, \ldots, R_r$.

Приведем два основных правила вывода [20, 21], достаточных для получения всех следствий некоторой исходной системы формул логики первого порядка: правило отделения (modus ponens) и обобщения (generalization).

Правило отделения $\dfrac{A,\ A \to C}{C}$ утверждает, что формула C выводима из формул A и $A \to C$.

Правило обобщения устанавливается следующей теоремой [27]. Пусть H — множество правильно построенных формул, U — формула и x — предметная переменная, не входящая свободно ни в одну из формул H. Тогда из выводимости $H \vdash U$ следует выводимость $H \vdash \forall x(U)$. В частности, если H пусто, имеет место $U \to \forall x(U)$ при условии, что U — тождественно истинная формула.

Обратимся к рассмотрению метода резолюций.

Рассмотрим один из основных механизмов логического вывода — метод резолюций Робинсона.

Определим дизъюнкт как дизъюнкцию переменных высказывания (литер). Будем запись дизъюнкта $RVPV\ \overline{Q}$ представлять в виде $\{R, P, \overline{Q}\}$, а пустой дизъюнкт обозначим $\blacksquare$.

Под стандартной формой представления формулы F будем понимать представление F с помощью множества дизъюнктов с приведением любой формулы F к стандартной форме. (Правила преобразования правильно построенных формул логики предикатов к стандартному виду можно найти, например, в [28].)

Рассмотрим следующие дизъюнкты: $C_1 : \overline{P}$, $C_2 : P \vee R$; тогда нетрудно доказать, что $C_1\ C_2 \vdash R$ [26, 27]. В этом случае R называется *резольвентой* дизъюнктов C_1 и C_2.

Пусть S множество дизъюнктов. Под множеством дизъюнктов $S_1, S_2, \ldots, S_n$ понимаем их логическое произведение $S_1 \& S_2 \& \ldots \& S_n$. Резолюционный вывод C из S есть такая конечная последовательность $C_1, \ldots, C_k$ дизъюнктов, что каж-

дый C_i либо принадлежит S, либо является резольвентой дизъюнктов, предшествующих C_i, и $C_k = C$.

В системах, построенных по принципу резолюции, доказательство выводимости F из P реализуется через доказательство формулы $\overline{F}\&P = \blacksquare$.

Главными механизмами метода резолюций являются: унификация, возврат, стратегия порождения резольвент.

Унификация необходима для порождения резольвент дизъюнктами, содержащими переменные. Чтобы применить резолюцию к предложениям, содержащим переменные, необходимо иметь возможность найти такую подстановку, которая, будучи примененной к родительским предложениям, обеспечит одинаковые символы для переменных, являющихся аргументами предложений. Например, для дизъюнктов $C_1 : G(y) \vee Q(y, f(x))$, $C_2 : \overline{Q}(a, w)$. Полагая для y подстановку a, а для w подстановку $f(x)$ в C_1, получим следующие дизъюнкты, резольвентой которых являются $G(a)$:

$$G(a) \vee Q(a, f(x)) \text{ и } \overline{Q}(a, f(x)).$$

Подстановкой Θ будем называть конечное множество вида

$$\{E_1/t_1, E_2/t_2, \ldots, E_k/t_k\},$$

где E_i — унифицируемые аргументы, а t_i — подставляемые вместо них значения. Результат применения подстановки Θ к E обозначим ΘE. Подстановка называется *унификатором* для множества аргументов $\{E_1, E_2, \ldots, E_k\}$, если имеет место равенство $\Theta E_1 = \Theta E_2 = \ldots = \Theta E_k$.

Пусть L — множество одноименных литералов (с отрицаниями или без), аргументы которых нужно унифицировать, например:

$$L = \{P(a, f(Y), f(X)), \bar{P}(X, Y, Z), P(Y, V, f(a))\}.$$

Алгоритм унификации

1. Для каждой позиции i ($i = 1, 2, \ldots$) аргумента в литерале определим множества унифицируемых аргументов M_i. В нашем примере это

$$M_1 = \{a, X, Y\},$$
$$M_2 = \{f(Y), Y, V\},$$
$$M_3 = \{f(X), Z, f(a)\}.$$

Полученное исходное семейство множеств обозначим μ.

2. Выполним проверку совместимости аргументов:

если в некоторое множество M_i входит две разные константы, то оно несовместимо;

если в некоторое множество M_i входят функторы (не являющиеся аргументами других функторов) с различными функциональными символами, то оно несовместимо;

если в некоторое множество M_i входит функтор и простая константа (т. е. константа, не имеющая структуру функтора), то это множество несовместимо.

При наличии несовместимого множества M_i алгоритм завершается неудачей, иначе переход к п. 3.

3. Из множества M_i, содержащего функторы, сформируем подмножества $M^j{}_{i\varphi}$, где φ — имя (индекс) функтора; j — номер позиции аргумента функтора, содержащие аргументы, стоящие в функторе φ на позиции j. П.3 применяется

только к таким множествам, рекурсивно повторяется для вновь сформированных множеств $M^j{}_{i\varphi}$, если они вновь содержат функторы (так как функторы могут быть аргументами других функторов). В нашем примере получаем расширенное семейство μ':

$$M_1 = \{a, X, Y\},$$
$$M_2\{Y, V\},\ M^1{}_{2f} = \{Y\},$$
$$M_3 = \{Z\},\ M^1{}_{3f} = \{X, a\}.$$

Поскольку для μ' не надо выполнять п. 3 алгоритма, переходим к п. 4.

4. Условия унификации каждого множества из μ':

все константы в каждом множестве должны совпадать;

переменные получают значение константы, если есть константа, или приравниваются друг другу.

Например, проиндексировав переменные, присваиваем переменным со старшим индексом значения переменной с наименьшим индексом. В нашем примере получим

$$X = Y = a \quad /{*} \text{ из } M_1 {*}/,$$
$$V = Y \quad /{*} \text{ из } M_2 {*}/,$$
$$X = a \quad /{*} \text{ из } M^1{}_{3f} {*}/.$$

5. Подставим найденные значения в исходное семейство μ:

$$M_1 = \{a, a, a\},$$
$$M_2 = \{f(a), a, a\},$$
$$M_3 = \{f(a), Z, f(a)\}$$

и выполним для него проверку согласно п. 2. Если проверка завершается успешно, то переходим к п. 6.

6. Найдем множество M_i, содержащее константы и переменные (вариант А) или переменные и функторы, не являющиеся константами (вариант В). Если такого множества нет, то конец с признаком «удача»: полученные значения переменных унифицируют μ. В противном случае все переменные этого множества получают значение константы этого множества (в варианте А) или функтора этого множества (в варианте В). С этими новыми значениями переход к п. 5.

В нашем примере в M_2 есть две разные константы: a и $f(a)$, а это означает, что M_2 несовместимо в силу п. 2 и весь алгоритм завершается неудачей. Если бы исходные литералы имели следующий вид:

$$L = \{P(a, a, f(X)), \bar{P}(X, Y, Z), P(Y, V, f(a))\},$$

то было бы получено следующее решение:

$$L = \{P(a, a, f(a)), \bar{P}(a, a, f(a)), P(a, a, f(a))\}.$$

Приведем пример без сопроводительных пояснений, записывая получаемые решения в соответствии с шагами представленного алгоритма

$$L = \{P(X, a, f(X, a), Y), P(Y, Z, f(Y, Z), T)\}.$$

1. $M_1 = \{X, Y\}$,
   $M_2 = \{a, Z\}$,
   $M_3 = \{f(X, a), f(Y, Z)\}$,
   $M_4 = \{Y, T\}$.

2. $M_1=\{X, Y\}$,
$M_2=\{a, Z\}$,
$M^1{}_{3f}=\{X, Y\}$,
$M^2{}_{3f}=\{Z, a\}$,
$M_4=\{Y, T\}$.

3. Замена $X_{(1)}=Y_{(2)}$, $a=Z_{(3)}$, $Y_{(2)}=T_{(4)}$ дает

$$L=\{P(X, a, f(X, a), X), P(X, a, f(X, a), X)\}.$$

Механизм возврата соответствует возврату процесса в ранее пройденное состояние для выбора новой альтернативы развития. Такой возврат необходим, если в текущем состоянии нельзя применить правила вывода или нет возможности получить с их помощью новые следствия.

Рассмотрим некоторые известные варианты метода резолюций.

Линейная резолюция независимо предложена Лавлендом и Лаккемом [26]. Для множества S дизъюнктов и дизъюнкта C_0 из S линейный вывод дизъюнкта C_n с верхним дизъюнктом C_0 — это вывод, представленный в виде дерева

$$\begin{array}{l} C_0 \\ \vdots \\ C_1\langle -B_0 \\ \vdots \\ \cdots \\ \vdots \\ C_{n-1}\langle -B_{n-2} \\ \vdots \\ C_n\langle -B_{n-1}, \end{array}$$

где для $i=0, 1, \ldots, n-1$ дизъюнкт C_{i+1} есть резольвента дизъюнктов C_i и B_i; каждый B_i либо принадлежит S, либо есть C_j для некоторого $j<i$.

Семантическая резолюция, предложенная Слэйглом, использует понятие интерпретации. Интерпретация языка L логики первого порядка характеризуется [38]:

непустым множеством D, называемым областью интерпретации;

соответствием между константами языка L и элементами множества D;

соответствием между каждой n-местной функциональной зависимостью в языке L и отображением $D^n\to D$;

соответствием между каждым n-местным предикатом в языке L и отображением $D^n\to\{f, t\}$, где f и t — константные символы, обозначающие «ложь» и «истина». Это определение устанавливает понятие модели языка L.

В [28] интерпретация трактуется более узко: если $S_0, S_1, S_2, \ldots, S_k$ — формулы (предикатные) языка L и D — область интерпретации, то под интерпретацией понимается множество значений, принимаемых формулами S_i в $\{f, t\}$ для любых допустимых наборов аргументов в D.

Любая интерпретация i разбивает множество дизъюнктов на два непересекающихся подмножества дизъюнктов T_i и F_i, истинных и ложных в интерпретации i. Ясно, что резольвента любых двух дизъюнктов из одного и того же подмножества (T_i или F_i) принадлежит тому же подмножеству. Отсюда следует, что, находя резольвенты для дизъюнктов из одного и того же подмножества (не важно какого), мы никогда не получим пустого дизъюнкта, поскольку пара контрарных (взятых с отрицанием) дизъюнктов не может принадлежать только T_i либо только F_i. Это позволяет уменьшить число рассматриваемых

сочетаний дизъюнктов при формировании резольвент. Для этого в резолюции участвуют по одному представителю из T_i и F_i, а резольвента затем помещается в то подмножество, которое определяет для нее интерпретация i.

На сокращение числа лишних резольвент ориентировано также произвольное упорядочение предикатных символов в дизъюнктах. Правило получения резольвент в семантической резолюции требует, чтобы удаляемый предикатный символ (литера) был старшим в T_i или F_i. Нетрудно убедиться, что это ограничение не нарушает условия полноты резолюции (т. е. условия, означающего, что если система противоречива, то пустой дизъюнкт будет получен обязательно). В самом деле, допустим, что на некотором этапе вывода нельзя удалить старшую литеру ни в одном из дизъюнктов. Даже если порождение новых резольвент возможно за счет удаления других литер, получить резольвенту, содержащую контрарную литеру для данной старшей литеры, не удается, а значит, никогда не удастся получить пустой дизъюнкт как резольвенту контрарных однолитерных дизъюнктов. Настанет момент, когда дальнейшее порождение резольвент окажется невозможным и ни один дизъюнкт нельзя будет сократить. В этом случае каждый дизъюнкт можно представить в следующем виде:

$$\begin{array}{l} R_1 \vee \_ \\ R_2 \vee \_ \\ \cdots \\ R_k \vee \_ \end{array}$$

где _ соответствует остальным литерам дизъюнкта. По предположению, в записи дизъюнктов не найдется контрарной литеры ни для одного R_i ($i=1, \ldots, k$), т. е. ни один дизъюнкт не сократим до пустого дизъюнкта ■. Формализация этого метода описана в [28].

Рассмотрим особенности логического вывода, описываемого на языке Пролог. Для терминологической ясности (поскольку терминология логического программирования и языка Пролог не во всем совпадает) предварительно введем необходимые определения.

Элементарная формула языка, взятая с отрицанием или без него, называется *литералом*. Из данного ранее определения следует, что литерал соответствует переменной высказывания с приданными ей аргументами.

Клозом (дизъюнктом) называется формула

$$\forall x_1 \ldots \forall x_n (L_1 \vee L_2 \ldots \vee L_m),$$

где L_i — литералы ($i=1, \ldots, m$).

Сгруппируем отдельно литералы с отрицанием и без него:

$$\forall x_1 \ldots \forall x_n (M_1 \vee M_2 \ldots M_i \vee \overrightarrow{M}_{i+1} \vee \ldots \vee \overrightarrow{M}_m),$$

где каждое $M_j(\overrightarrow{M}_j)$ соответствует L_j в предыдущей записи. Полученная форма называется *префиксной* (предваренной) *стандартной формой*. (Процедура перевода любой правильно построенной формулы языка логики предикатов в префиксную стандартную форму изложена в [28].)

Предполагая в дальнейшем наличие кванторов общности перед формулой и опуская их для простоты, последнее выражение, используя эквивалентные преобразования

$$A\to B=\bar{A}\vee B,\ \overline{A\&B}=\bar{A}\vee\bar{B},$$

может переписать:

$$M_{i+1}\&M_{i+2}\& \ldots \&M_m \rightarrow M_1 \vee M_2 \vee \ldots \vee M_i.$$

Наконец последнюю формулу заменим множеством формул вида

$$\begin{gathered} M_{i+1}\&M_{i+2}\& \ldots M_m \rightarrow M_1, \\ \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \\ M_{i+1}\&M_{i+2}\& \ldots M_m \rightarrow M_i. \end{gathered} \qquad \text{(B.1)}$$

Каждая из формул является так называемым *хорновским клозом*, а исчисление на основе хорновских клозов — хорновским исчислением. Если антецедент хорновского клоза (левая часть до стрелки) отсутствует, то формула получает следующий частный вид:

$$\rightarrow M_s. \qquad \text{(B.2)}$$

Если отсутствует сукцедент (правая часть после стрелки), то формула получает вид

$$M_{i0}\&M_{i1}\& \ldots \&M_{iz} \rightarrow . \qquad \text{(B.3)}$$

Каждая из формул типа (B.1) называется также *правилом* (редукцией, утверждением), а каждая частная формула типа (B.2) — *фактом*. Антецедент правила называется его *телом*, а сукцедент — *заголовком*. Как видим, факт — это правило без тела. При процедурной интерпретации программ предикаты тела называются *процедурными вызовами* или *операторами*.

Формула (B.3) называется целью (т. е. определяет формулы, которые нужно доказать) и также является частным примером хорновского клоза.

Пролог использует хорновское исчисление, в котором программа представляется с помощью ряда правил и фактов. Это множество правил и фактов называется *базой данных* [14—16].

В Прологе вывод состоит в доказательстве непротиворечивости цели исходной базе данных. *Факты* — простые утверждения, описывающие объекты и отношения между ними. Редукция осуществляет разбиение задачи на подзадачи, причем задача считается решенной, если решены все составляющие ее подзадачи. Аксиомы, т. е. факты и правила, а также теорема (формулировка задачи) представляются в виде дизъюнктов. Переменные, входящие в дизъюнкты, связываются кванторами всеобщности; кванторы существования вообще не используются. Пролог базируется на линейной входной резолюции. При использовании этой стратегии выбор дизъюнкта начинается с применения правила резолюций к целевому дизъюнкту и к одной из гипотез, что дает новый дизъюнкт и т. д. (см. схему линейной резолюции выше на стр. 18).

Другие механизмы логического вывода связаны с использованием моделей, допускающих представление на основе структурированных и вычислительных моделей. Речь идет, в частности, о семантических и вычислительных сетях, фреймах [20, 21].

ГЛАВА 1.

АБСТРАКТНАЯ ПРОЛОГ-МАШИНА

В отличие от традиционных языков программирования, таких как Фортран, Паскаль или Си, схема реализации интерпретатора для Пролога достаточно сложна. Это объясняется тем, что если семантика традиционных языков во многом является отражением принципов функционирования современных ЭВМ, то семантика Пролога никоим образом не связана с архитектурой современных ЭВМ. Поэтому практически невозможно адекватно описать процесс выполнения Пролог-программы в терминах традиционных ЭВМ. Чтобы преодолеть эту трудность, вводится понятие абстрактной машины, в терминах которой уже относительно просто объясняется процесс интерпретации логической программы.

Абстрактная машина служит одной из основных спецификаций компилятора в следующем смысле. Компилятор для языка Пролог должен представлять исходный текст программы в виде последовательности команд абстрактной машины и совокупности необходимых данных. Таким образом, абстрактная машина включает следующие основные компоненты: структуры данных, систему команд — и позволяет рассматривать задачу реализации Пролога как совокупность двух более простых задач:

разработка компилятора языка Пролог, выполняющего преобразование исходной программы в последовательность команд абстрактной машины;

разработка интерпретатора для системы команд абстрактной машины, обеспечивающего выполнение кода абстрактной машины с помощью универсальной ЭВМ, или специального процессора, предназначенного для эффективного выполнения команд абстрактной машины.

Далее подробно описаны основные элементы абстрактной машины для языка Пролог на основе процедурной семантики логических программ [50, 51].

Программа на языке Пролог с точки зрения процедурной семантики представляет собой неупорядоченное множество предикатов

$$\mathbf{P}=\{P1, P2, P3, \ldots, Ps\},$$

интерпретируемое как множество процедур, вызовы которых реализуются при выполнении программы. Каждый предикат Pi программы **P** однозначно идентифицируется своим именем и числом аргументов (арностью). Это означает, что если предикаты Pi и Pj входят в состав программы **P** и $i \neq j$, то предикаты Pi и Pj различаются по крайней мере именами либо числом аргументов. Каждый предикат Pi программы **P** является упорядоченным множеством утверждений

$$Pi=\langle S1, S2, \ldots, Sr\rangle,$$

каждое из которых имеет следующий вид:

$$H \leftarrow G1, G2, \ldots, Gq,$$

где H — головной литерал (атом) утверждения; G1, G2, ..., Gq — литералы (атомы) тела утверждения. Утверждение называется *правилом*, если $q>0$, и *фактом*, если $q=0$.

Литерал (или атом) определяется функциональным символом f/n (n — арность символа), а также значениями аргументов A1, A2, ..., An:

$$L=f(A1, A2, \ldots, An),\ n \geqslant 0.$$

Для всех утверждений S предиката P головной литерал имеет одно и то же имя и число аргументов, определяющих предикат P.

Логическая программа **P** замкнута, если для каждого ее предиката P тело любого утверждения содержит только такие литералы, что для каждого из них существует предикат программы **P** с тем же именем и числом аргументов. Для программ реальных систем программирования на Прологе условие замкнутости, как правило, не выполняется из-за необходимости использования встроенных предикатов для выполнения операций ввода-вывода, арифметических и операций над самой программой (метаопераций). С точки зрения реализации механизмов логического вывода встроенные предикаты особого интереса не представляют и могут быть опущены при анализе схемы выполнения программы.

Аргументы литералов *L*, входящих в состав утверждений программы, могут быть только *термами*. Для эффективной реализации механизма логического вывода (особенно его части,

обеспечивающей унификацию) термы классифицируются следующим образом.

Терм, являющийся аргументом литерала, называется *термом нулевого уровня*. Подтермы, являющиеся аргументами некоторого терма нулевого уровня, называются *вложенными термами*. Простым термом является константа (ее функциональный символ f/0). Более сложным случаем терма нулевого уровня является первое или непервое (последующее) вхождение логической переменной в утверждение; определяется при просмотре его слева направо. Утверждение и только оно является областью определения логической переменной. Если терм нулевого уровня является первым вхождением логической переменной, то он называется *свободной переменной*, а если повторным вхождением логической переменной, то *связанной*. Последним и наиболее сложным случаем являются термы нулевого уровня, представляющие собой структуры вида

$$f(T1, T2, \ldots, Tk),$$

где T1, T2, ..., Tk — вложенные (ненулевого уровня) термы; f/k — главный функтор терма. В свою очередь, вложенные термы могут быть переменными, константами или содержать другие вложенные термы.

Основная операция логического вывода — *унификация термов*. При унификации термов вычисляются значения логических переменных, обеспечивающие символьное равенство термов при подстановке этих значений вместо соответствующих логических переменных. Если набор значений переменных, удовлетворяющих этим условиям, существует, то при унификации они вычисляются в наиболее общем виде. В противном случае унификация завершается неуспешно.

При процедурной интерпретации программы механизм унификации используется как средство передачи параметров вызываемой процедуре, но фактически данные могут передаваться в обоих направлениях, что является одним из наиболее существенных отличий реализации языка Пролог от других процедурных языков программирования. Процедурная интерпретация программы состоит в следующем.

Процесс логического вывода над программой **P** выполняется как последовательность вложенных вызовов процедур, каждой из которых соответствует предикат P логической программы **P**. Литералы тела утверждения интерпретируются как операторы вызова соответствующих им процедур, т. е. предикатов программы **P**, имеющих имя и число аргументов, совпадающих с именем и числом аргументов литерала. Литералы тела процедуры интерпретируются как операторы вызова процедуры, а их параметры — как фактические параметры вызова. Головные литералы утверждений рассматриваются как точки входа в процедуру, а их параметры — как формальные параметры процедуры, обрабатываемые в процессе унификации.

Процедура, тело которой определяется предикатом P программы **P**, может иметь несколько точек входа, каждой из которых соответствует утверждение данного предиката. Причем в момент вызова процедуры точка входа в общем случае не определена и возможен повторный вход в уже ранее вызванную процедуру. Это отличает способ реализации языка Пролог от других языков программирования. Именно поэтому схема выполнения программы получила название недетерминированных вычислений. Реально недетерминизм используется, если вызов определенной процедуры предиката завершается неуспешно. Такая ситуация возникает, если невозможно унифицировать формальные и фактические параметры. В этом случае автоматически выполняется повторный вход в одну из вызванных ранее процедур, что и является недетерминированным входом в процедуру. Точка повторного входа определяется стратегией механизма логического вывода, используемого при данной реализации. Но обычно при реализации Пролога используется стратегия логического вывода, обеспечивающая регулярный обход текущего И-ИЛИ-дерева логического вывода программы. Вершины этого дерева соответствуют средам процедур. Каждая вершина снабжена указателями на следующую вершину, на альтернативную вершину и на вершину, которая ей предшествует в графе, — И-вершину. Если при вызове вершины соответствующая ей процедура закончилась неудачей, то управление передается на альтернативную вершину, соответствующую ближайшей точке ветвления, — ИЛИ-вершину. Если ИЛИ-вершина отсутствует, то управление возвращается в предшествующую точку ветвления в программе. Таким образом, на абстрактном уровне структура вызовов процедур должна воспроизводить дерево И-ИЛИ, соответствующее исходной программе.

1.2. ПРИНЦИПЫ ФУНКЦИОНИРОВАНИЯ АБСТРАКТНОЙ МАШИНЫ ЛОГИЧЕСКОГО ВЫВОДА

Процедурная интерпретация программ позволяет выполнить компиляцию программы в набор команд абстрактной машины логического вывода. Эти команды могут быть использованы для реализации либо специального процессора логического вывода, непосредственно реализующего данную систему команд высокого уровня, либо интерпретатора соответствующего байт-кода или скомпилированы в код ЭВМ.

Схема компиляции программы в систему абстрактных команд основана на замкнутости этой программы и полной независимости предикатов программы друг от друга. Скомпилированный код программы состоит из кодов каждого предиката программы. Причем код каждого предиката строится независимо и содержит только имена других вызываемых предикатов программы, т. е. никакая информация о внутренней структуре других предикатов программы не используется. В силу этого оказывается относительно несложно добавлять новые предикаты к программе или модифицировать уже имеющиеся предикаты во время выполнения программы.

Код для предиката P программы **P** строится следующим образом. Абстрактная машина логического вывода имеет специальный набор регистров — регистров аргументов вызова процедуры A1, A2, ..., Am, ... Непосредственно до начала выполнения первой команды кода программы первые m регистров должны быть загружены аргументами литерала тела некоторого утверждения, где m — число аргументов вызываемой процедуры. Этот литерал определяет вызываемую процедуру и значения, которые должны быть загружены в регистры аргументов. Обычно для запуска Пролог-машины используется специальное утверждение, не имеющее головного литерала:

$$\leftarrow G1, G2, \ldots, Gn.$$

Это утверждение называется целью программы. Структура кода вызова, т. е. последовательности команд абстрактной машины логического вывода, выполняющих загрузку регистров аргументов и передачу управления первой команде кода вызываемой процедуры предиката, во многом определяется конкретным вариантом модели логического вывода, используемой для реализации языка Пролог.

Память абстрактной машины логического вывода состоит из двух областей:

области кода программы **P**, являющейся квазистатической областью, так как изменения в этой области могут происходить только если сама программа модифицируется во время выполнения;

динамической области, где размещается трасса логического вывода и некоторая вспомогательная информация, необходимая для выполнения повторных входов в уже вызванные ранее процедуры.

Трасса логического вывода — это последовательность элементов ENV (environment — среда), каждый из которых был создан при первом входе в вызванную процедуру предиката. Текущее состояние трассы логического вывода T является упорядоченным множеством

$$T = \langle ENV1, ENV2, \ldots, ENVi, \ldots, ENVj, \ldots \rangle.$$

Элемент ENV трассы логического вывода T называется *средой процедуры*. Если элементы ENVi и ENVj принадлежат T и $i<j$, то среда процедуры ENVj младше среды процедуры ENVi. Последнее означает, что элемент ENVi был создан раньше среды ENVj. Наиболее младшая среда ENV называется текущей и образуется в момент выполнения первых команд кода вызываемой процедуры, т. е. после завершения операций вызова. Повторный вход

в процедуру может быть выполнен только для процедуры, среда которой ENV уже содержится в трассе вывода T. В этом случае она модифицируется.

В общем случае среда процедуры содержит ADM — управляющую информацию, ARGS — копии содержимого регистров аргументов в момент вызова данной процедуры, VARS — вектор переменных утверждения, использованного для входа в данную процедуру, и STRS — структуры данных, используемых для представления термов:

$$ENV = \{ADM, ARGS, VARS, STRS\}.$$

Управляющая информации ADM среды процедуры ENV в трассе логического вывода T содержит указатели на ближайшие вершины текущего дерева логического вывода, которыми являются элементы ENV трассы T. Информация в ADM всегда определяет ближайшую И-вершину и может содержать указатель ближайшей ИЛИ-вершины. Если ADM содержит указатель ИЛИ-вершины, то в ADM должен быть и указатель точки повторного входа процедуры, соответствующей этой ближайшей ИЛИ-вершине. Управляющая информация ADM всегда содержит указатель точки входа, которая будет использована для последующего вызова процедуры после завершения данной. Эта информация соответствует понятию точки возврата, используемой при вызове процедур в традиционных процедурных языках.

Для ускорения доступа к элементам текущей трассы абстрактная машина имеет специальный набор регистров E, C и B. Регистр E содержит указатель текущей среды процедуры, которую далее будем обозначать ENV(E), регистр C — указатель вершины дерева вывода, связанный отношением И с текущей. Эта вершина ENV(C) является ближайшей И-вершиной к текущей. Вершина дерева логического вывода ENV(B), указатель которой хранится в регистре B, называется ближайшей точкой выбора (ветвления) и является ближайшей ИЛИ-вершиной к текущей. Вершина ENV(B) связана отношением ИЛИ с текущей вершиной ENV(E).

Состояние дерева логического вывода программы изображено на рис. 1.1 в момент первого входа в процедуру предиката b. При этом предполагается, что все предшествующие вызовы процедур предикатов a1 и a2 завершились успешно. Возврат после завершения выполнения процедуры предиката a показан на рис. 1.1 штриховой стрелкой, цифры соответствуют последовательности создания элементов трассы логического вывода.

В момент входа в процедуру предиката b выполняются соотношения

$$ENV(E) = \langle \text{среда процедуры предиката b} \rangle,$$

$$ENV(C) = \langle \text{среда процедуры предиката p} \rangle.$$

Если при входе в процедуру предиката a используется первое утверждение соответствующего предиката, то

$$ENV(B) = \langle \text{среда процедуры предиката a} \rangle.$$

Для доступа в область кода абстрактная машина имеет регистры P, CP и BP. Регистр P всегда содержит указатель выполняемой команды, регистр CP — указатель начала последовательности команд вызова процедуры, которая должна быть вызвана после завершения текущей процедуры. Для выполнения текущей процедуры всегда имеется среда ENV(E). Регистр BP содержит указатель точки для повторного входа в процедуру, среда которой ENV(B) уже создана. В момент входа в процедуру предиката b (рис. 1.1) регистр CP содержит указатель на последовательность команд вызова предиката c, а регистр BP — указатель точки повторного входа в процедуру предиката, соответствующую второму утверждению предиката a.

Управляющая информация ADM содержит копии содержимого регистров C и CP в момент входа в процедуру. Сохранение содержимого регистров B и BP необходимо, только если используемая точка входа не последняя в процедуре предиката, т. е. если используемое в качестве точки входа утверждение не последнее для данного предиката. При этом условии в текущей среде процедуры сохраняется содержимое регистра B, а в среде ENV(B) — регистра BP. Затем регистры B и BP получают новые значения, так как текущая среда становится ближайшей точкой выбора.

Таким образом, управляющая информация ADM в каждом элементе трассы логического вывода T содержит все отношения между вершинами дерева логического вывода. Вершинами этого дерева являются только элементы трассы T. Цепочки И-вершин и ИЛИ-вершин образуют копии содержимого регистров C и B, хранящиеся в разделах ADM соответственно. Головные элементы этих цепочек определяются текущим содержимым регистров C и B.

В момент создания среды ENV(E) каждая ячейка вектора переменных VARS этой среды содержит специальное пустое значение, соответствующее свободной переменной. В общем случае ячейки вектора переменных VARS наряду с пустым значением могут

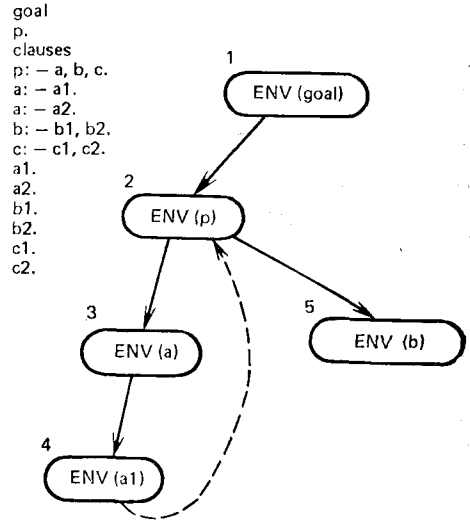


Рис. 1.1. Логическая программа и цель (*а*) и дерево логического вывода в момент вызова процедуры предиката b (*б*)

иметь еще значения константы, структуры и ссылки на другую переменную, принадлежащую любому вектору переменных текущей трассы T. Последнее означает, что ссылка может определять адрес ячейки переменной не только в данном элементе трассы логического вывода. Эти значения образуются в результате унификации содержимого регистров A1, A2, ..., Am с аргументами головного литерала утверждения вызванной процедуры. Поэтому унификация может рассматриваться как способ передачи параметров при вызове процедуры. Если ячейка вектора VARS имеет значение структуры, то это означает, что она содержит ссылку на структуру, расположенную в разделе STRS одного из элементов текущей трассы логического вывода.

Таким образом, регистры E, C, B, P, CP, BP и трасса логического вывода T полностью определяют текущее состояние логического вывода над программой **P**. Но для возврата в ближайшую точку выбора ENV(B) и повторного входа в соответствующую процедуру этой информации недостаточно.

При возврате в ближайшую точку выбора необходимо вернуть абстрактную машину в то состояние, которое она имела в момент последней модификации среды ENV(B). Это преобразование выполняется специальной командой абстрактной машины fail. Для выполнения команды fail в регистр E записывается указатель на среду ENV(B), т. е. E: =B. В результате ближайшая точка выбора становится текущей средой трассы логического вывода. Содержимое регистров C, B и CP восстанавливается из раздела ADM среды ENV(B), а регистр P получает значение BP, т. е. P: =BP. Содержимое ARGS среды ENV(E) (новой текущей среды) не изменяется, а разделы VARS и STRS уничтожаются. Во время повторного выполнения процедуры эти разделы создаются заново. В разделе ADM среды ENV(E) сохраняется только информация, связанная с последней точкой выбора.

После всех этих преобразований уничтожаются все элементы трассы логического вывода, которые оказались младше среды ENV(E), а содержимое регистров аргументов A1, A2, ..., Am восстанавливается из раздела ARGS среды ENV(E).

Для полного приведения абстрактной машины в требуемое состояние этих операций недостаточно. Дело в том, что при унификации содержимого регистров A1, A2, ..., Am с формальными параметрами головного литерала соответствующего утверждения предиката может происходить двунаправленная передача значений переменных. В результате некоторые ячейки векторов VARS, имевшие до этого момента пустые значения, могут получить другие значения, причем эти ячейки могут принадлежать элементам трассы логического вывода, которые значительно старше текущей среды ENV(E).

Для корректного завершения команды fail необходима специальная информация. Эта информация сохраняется в динамической области, называемой следом логического вывода (trail). *След логического вывода* представляет собой стек, элементами которого являются указатели (адреса) ячеек векторов переменных VARS. Для фиксации вершины этого стека имеется специальный регистр TR абстрактной машины, который всегда содержит указатель первой свободной ячейки стека следа логического вывода. Используется также регистр BTR, который хранит содержимое регистра TR в момент создания точки выбора ENV(B).

Модификация стека следа логического вывода происходит следующим образом. Всякий раз, когда при унификации содержимого регистров аргументов и формальных параметров некоторая пустая ячейка v переменной получает некоторое значение, проверяется, что среда этой переменной старше среды последней точки выбора ENV(B). И если это так, то адрес переменной v записывается в стек следа логического вывода, а содержимое регистра TR получает приращение на единицу.

Кроме того, всякий раз, когда содержимое регистров B и BP записывается в соответствующий раздел ADM трассы логического вывода (по алгоритму, описанному выше), в этот же раздел ADM текущей среды ENV(E) записывается и содержимое регистра TR. В дополнение к этому выполняется операция инициализации регистра BTR (BTR: =TR).

При выполнении серии повторных входов в одну и ту же процедуру наступает момент, когда используется последнее утверждение соответствующего предиката, т. е. точка выбора оказывается исчерпанной. В этом случае регистры B, BP и BTR должны получить новые значения. В результате предыдущая точка выбора, параметры которой определены в уже исчерпанной точке выбора в разделе ADM соответствующего элемента трассы логического вывода, становится ближайшей точкой выбора. Для этого содержимое регистра B восстанавливается из среды ENV(B), а регистров B и BP — из среды ENV(B^), где B^ — только что восстановленное содержимое регистра B. Эта операция называется восстановлением предыдущей точки выбора.

Этот алгоритм сохранения и восстановления содержимого регистров, используемых при выполнении возврата, позволяет завершить алгоритм выполнения команды fail следующим образом.

Из стека следа логического вывода удаляются все элементы в диапазоне [BTR, TR], и каждый из них используется для отмены значения переменной, адрес которой определяет данный элемент. В результате все ячейки переменных, адреса которых были удалены из стека следа логического вывода, получают пустые значения (undef). Затем выполняется операция инициализации регистра вершины стека следа логического вывода TR: =BTR. Этот алгоритм, завершающий выполнение команды fail абстрактной машины логического вывода, называется *распаковкой следа при возврате* (undo_trail).

Описанный выше алгоритм выполнения команды fail обеспечивает полное восстановление состояния абстрактной машины логического вывода в момент последней модификации ближайшей

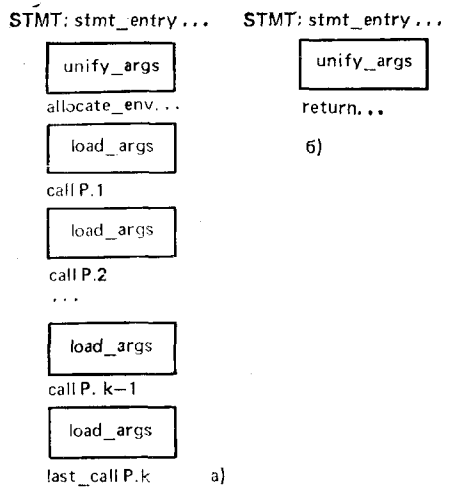


Рис. 1.2. Структура кода для правила (*а*) и факта (*б*)

точки выбора. Однако необходима проверка, что среда ENV(B) остается точкой выбора. Если это не так, то должна быть восстановлена предыдущая точка выбора.

Все остальные команды абстрактной машины логического вывода могут быть распределены по двум группам: команды управления логическим выводом; команды выполнения унификации.

В терминах команд абстрактной машины логического вывода код, создаваемый компилятором для каждого правила, имеет вид, показанный на рис. 1.2,*а*, а код, создаваемый для факта, — на рис. 1.2,*б*. Далее в тексте термины «правило» и «факт» заменяем на термин «утверждение».

Код утверждения всегда начинается с команды stmt_entry. Эта команда имеет несколько операндов, которые содержат размеры разделов VARS и STRS, используемые для приведения в исходное состояние среды ENV(E), что необходимо для последующей унификации содержимого регистров аргументов с формальными параметрами вызванной процедуры. Приведение в исходное состояние вектора переменных VARS заключается в том, что все его переменные получают пустые значения.

После выполнения команды stmt_entry выполняется следующий за ней блок команд унификации unify_args, которые выполняют унификацию содержимого регистров аргументов A1, A2, ..., Am с формальными параметрами процедуры, т. е. со значениями аргументов головного литерала данного утверждения. Для каждого из m аргументов имеется не менее одной команды, выполняющей унификацию содержимого соответствующего регистра и формального параметра. Для формальных аргументов, являющихся структурами, генерируется несколько команд унификации. Команды унификации имеют два операнда. Первый — номер соответствующего регистра, а второй — либо аргумент литерала, либо ссылка в раздел DATA, используемый для представления скелетонов структурированных термов предиката. Если формальный аргумент — переменная, то второй операнд — ссылка на ячейку этой переменной, расположенную в разделе VARS среды ENV(E). В результате выполнения блока команд унификации unify_args ячейки векторов VARS трассы логического вывода (причем они необязательно должны принадлежать текущей среде ENV(E)) получат некоторые значения.

Однако при выполнении команд унификации может возникнуть ситуация, когда некоторая пара термов окажется неунифицируемой. В этом случае считается, что вектор переменных VARS не определен, автоматически генерируется и выполняется команда fail. Если унификация прошла успешно, то команда fail не генерируется и выполняется команда, следующая за блоком команд унификации.

Если утверждение является правилом (рис. 1.2,*а*), то после успешной унификации содержимого регистров аргументов выполняется команда allocate_env, позволяющая вычислить размеры области динамической памяти, занимаемой текущей средой ENV(E). Это необходимо для размещения в трассе логического вывода нового элемента. Команда allocate_env выполняет сохранение содержимого регистров C и CP в разделе ADM текущей среды ENV(E). После чего эта текущая среда фиксируется как ближайшая И-вершина, т. е. C : =E. Затем в трассу логического вывода помещается новый (пустой) элемент, а его адрес записывается в регистр E (для этой последней операции необходимы параметры команды, вычисляемые компилятором). Команда allocate_env завершает выполнение тела процедуры и подготавливает вложенные вызовы других процедур. После этой команды всегда выполняется следующая команда.

Этими командами могут быть команды блока загрузки аргументов load_args либо одна из команд call или last_call. Команда load_args не используется лишь тогда, когда соответствующий литерал тела (первый в данном случае) не имеет аргументов. Команда last_call может использоваться только для последнего литерала тела утверждения.

Команды блока загрузки аргументов load_args выполняют загрузку регистров аргументов соответствующими значениями фактических параметров вызываемого предиката, т. е. аргументами литерала. В регистры A1, A2, ..., Am загружаются значения фактических параметров, которые содержатся в операндах команд блока load_args (фактический аргумент есть константа), либо ссылки на переменную, расположенную в среде ENV(C) (если фактический аргумент переменная), либо ссылки на структуру, расположенную в разделе STRS текущей среды ENV(E). Если структура содержит переменные, то эти переменные располагаются в среде ENV(C).

Если утверждение есть факт, то после успешного окончания команд унификации выполняется команда return (рис. 1.2,*б*), действие которой эквивалентно выполнению команды allocate_env с последующей оптимизацией, выполняемой командой last_call, и загрузкой регистра P без предварительной модификации регистра CP, т. е. (P : =CP).

После завершения команд блока загрузки аргументов всегда выполняется команда либо call, либо last_call. Команда call загружает в регистр CP адрес команды, которая должна быть выполнена после успешного завершения вызова данной процедуры,

а в регистр P — адрес основной точки входа процедуры предиката. Тем самым управление передается вызываемой процедуре.

Команда last_call более сложная, выполняет оптимизированный хвостовой вызов процедуры, определяемой последним литералом тела утверждения. Основное назначение этой команды — освободить динамическую область памяти, которая занята ставшими уже ненужными элементами трассы логического вывода T. Реально команда last_call может освободить память, занимаемую только разделами ADM, ARGS и только частично разделом VARS некоторых элементов трассы логического вывода T. Если в момент выполнения команды last_call среда ENV(C) младше среды ENV(B), то память, используемая разделами ADM, ARGS и частью раздела VARS всех элементов ENV трассы логического вывода не старше элемента ENV(C), может быть освобождена. Не могут быть освобождены только части разделов VARS и разделы STRS, так как на структуры, созданные в процессе логического вывода, могут быть ссылки из элементов трассы логического вывода, которые еще необходимо сохранить. По этой же причине сохраняется содержимое некоторых ячеек переменных. Если среда ENV(C) не младше среды ENV(B), то эта оптимизация занимаемой памяти не может быть выполнена, так как вся информация необходима для выполнения повторного входа в процедуру, которой соответствует среда ENV(B). Выполнение команды last_call может приводить к частичному уничтожению среды вызывающей процедуры ENV(C).

Оптимизация памяти, выполняемая командами last_call и return, чрезвычайно важна. Дело в том, что рекурсивные правила — единственное средство в языке Пролог, позволяющее выполнять циклическую обработку. Если при выполнении цикла не удается использовать оптимизацию, то в трассе логического вывода T появляется столько новых элементов, содержащих полностью все разделы, сколько итераций необходимо для выполнения этого цикла. Если оптимизация используется (для этого необходимо, чтобы рекурсивный вызов был хвостовым и выполнялись указанные выше условия), то на каждой итерации в трассу логического вывода T фактически добавляются только разделы GLOB_VARS (см. ниже) и STRS, а если рекурсивное утверждение не содержит структур, то и эти разделы будут пустыми. В последнем случае при выполнении цикла используется фиксированный участок динамической области, размер которого не зависит от числа итераций цикла.

Для выполнения команд last_call и return раздел VARS организуется в виде вектора локальных переменных LOC_VARS и вектора глобальных переменных GLOB_VARS. При выполнении указанных выше условий векторы LOC_VARS освобождаются. Векторы GLOB_VARS и раздел STRS освобождаются только вместе с соответствующей средой при выполнении команды fail. (Необходимость введения двух типов ячеек для вектора VARS подробно рассматривается в следующем разделе.)

Кроме указанной выше оптимизации используемой части динамической области памяти, команда last_call восстанавливает содержимое регистров C и CP из среды ENV(C) и загружает в регистр P значение своего единственного операнда, которое определяет точку входа в вызываемую процедуру предиката, т. е. передает управление этой процедуре.

Код для каждого предиката (рис. 1.3) строится независимо от других предикатов этой программы с точностью до значений операндов команд call и last_call. Операнды этих команд определяют только адреса основных точек входа в вызываемые процедуры предикатов (такая точка на рис. 1.3 отмечена меткой PRED). Основная точка входа PRED: может быть использована только для первого входа в процедуру. Это означает, что непосредственно перед передачей управления на эту точку входа необходимо выполнить загрузку регистров аргументов по командам, обозначенным на рис. 1.2,*а* блоком команд load_args, и только после этого по командам call или last_call управление передается на эту точку входа. Эти последовательности команд, реализующие первый вход в процедуру предиката (рис. 1.2,*а*), образуют тело каждого утверждения предиката.

Таким образом, в момент начала выполнения первой команды кода предиката proc_entry регистр P содержит адрес этой команды (т. е. основной точки входа), а регистры аргументов A1, A2, ..., Am — фактические параметры вызова.

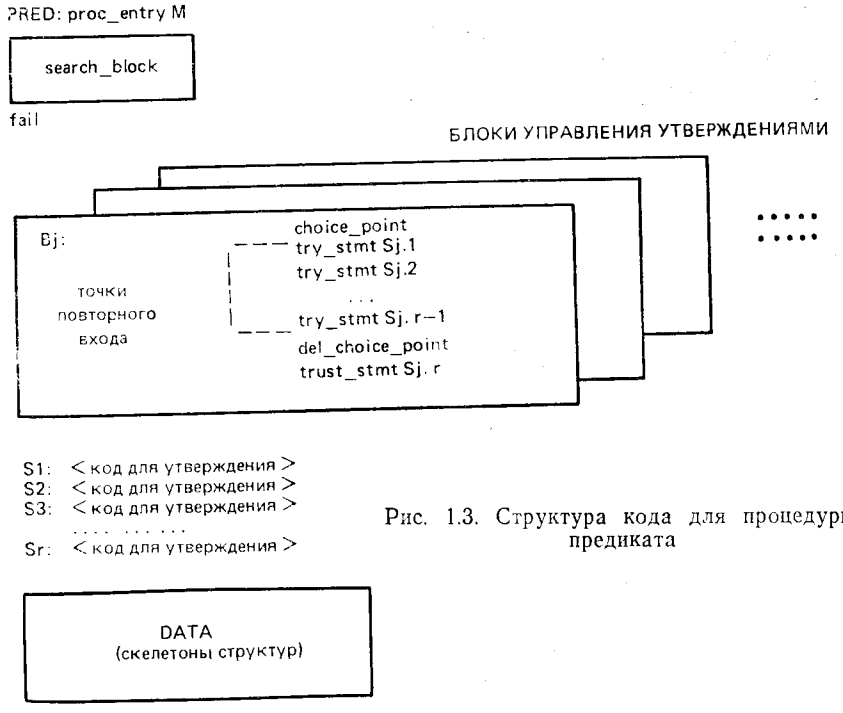


Рис. 1.3. Структура кода для процедуры предиката

До начала выполнения команды proc_entry среда ENV(E) пустая, а среды ENV(B) и ENB(C) полностью определены. Регистр CP содержит указатель блока команд load_args, которые должны быть выполнены после успешного завершения данной процедуры, в которую выполняется вход. Если блок команд вызова данной процедуры принадлежит цели и это последний литерал цели, то регистр CP должен содержать адрес подпрограммы успешного окончания доказательства цели. Эта подпрограмма заканчивает функционирование машины логического вывода. Код утверждения цели имеет такую же структуру, как и код для правила (рис. 1.2,*а*), но только без команд, предшествующих команде allocate_env. Регистр BP содержит указатель повторной точки входа в процедуру, среда которой есть ENV(B) (точки повторного входа изображены на рис. 1.3). Команда proc_entry сохраняет значения регистров A1, A2, ..., Am в текущей среде ENV(E), имеет только один операнд, равный числу аргументов m данного предиката.

Далее выполняется блок команд search_block, вычисляющих адрес одной из точек входа Bj. Каждую точку входа Bj определяет блок команд управления утверждениями. Операнды некоторых команд этого блока содержат адреса точек входа в утверждения предиката Sj.1, Sj.2, ... , Sj.r. Эти утверждения и составляют блок управления утверждениями. Блоки управления утверждениями строятся компилятором и не имеют аналогов в исходном тексте программы на Прологе. Каждый такой блок определяет упорядоченное множество тех, и только тех утверждений предиката, для которых формальные аргументы головных литералов могут унифицироваться с содержимым регистров аргументов.

Условия, которые определяют блок управления утверждениями и используются для анализа содержимого регистров аргументов, могут быть следующие: регистр Ai имеет значения определенной структуры, константы или пустой переменной. Могут использоваться условия, включающие проверку содержимого нескольких регистров аргументов в определенной последовательности. Этот порядок проверки содержимого регистров аргументов и их число могут указываться в исходном тексте определения предиката на Прологе в форме специального описателя схемы индексации утверждений. Если описатель не задан, то по умолчанию используется только регистр A1, т. е. значения только первых формальных аргументов головных литералов утверждений предиката.

Константы и структуры, используемые для генерации команд проверки условий search_block, выбираются компилятором из исходного кода предиката. Например, если первый аргумент головного литерала каждого утверждения предиката есть константа и все эти константы различны, то компилятор построит r+1 блок управления утверждениями (r — число утверждений в предикате) при условии, что дополнительной индексации для данного предиката не задано. Условия, удовлетворение которым содержимого регистра A1 проверяется командами search_block, следующие.

1. Если содержимое регистра A1 есть константа, равная первому аргументу головного литерала утверждения Si предиката P, то необходимо использовать блок Bi, содержащий точку входа в это утверждение.

2. Если значение регистра A1 есть константа или структура, но ни одно из вышеперечисленных условий не выполнено, то необходимо выполнить команду fail абстрактной машины логического вывода.

3. Если значение регистра A1 есть пустая переменная, то необходимо использовать блок B.r+1, содержащий все утверждения предиката, так как в этом случае схема индексации не может выделить никакие подгруппы утверждений.

Если при этих условиях некоторые головные литералы утверждений предиката могут содержать одинаковые константы, то число блоков будет меньше и некоторые из них будут содержать несколько утверждений. В один и тот же блок попадут утверждения, имеющие одинаковые константы в качестве первого формального параметра. Объединение всех блоков всегда задает все множество утверждений предиката. В этом последнем случае целесообразно использовать индексацию еще по какому-либо дополнительному аргументу, чтобы «расщепить» блоки управления утверждениями, содержащие более одного утверждения. Для этого необходимо задать описатель индексации. Тогда компилятор может построить условия так, что все блоки опять будут содержать по одному утверждению. Но предикат может иметь такую структуру, что такой результат индексации в принципе невозможен.

Схема индексации, реализуемая командами search_block, может быть весьма сложной, но в любом случае она гарантирует только то, что утверждения предиката, не включенные в блок управления утверждениями, не могут быть использованы для выполнения вызова процедуры, если условия данного блока выполнены для содержимого регистров аргументов. Поэтому если содержимое регистров аргументов не удовлетворяет ни одному из условий, определяющих блоки управления утверждениями, то команда fail выполняет возврат машины логического вывода в ближайшую точку выбора. Схема индексации позволяет ускорить доступ к утверждениям предиката. Но главное ее достоинство состоит в следующем. В большинстве случаев компилятор может установить, что при определенных условиях вызов процедуры детерминированный (т. е. блок управления утверждениями, на который передается управление, содержит всего одно утверждение). А это, в свою очередь, приводит к существенному сокращению используемой памяти для хранения трассы логического вывода (этот вопрос далее обсуждается подробнее).

Блок управления утверждениями для общего случая (когда определяет более одного утверждения) обязательно начинается командой choice_point (рис. 1.3), предназначенной для создания новой ближайшей точки выбора — текущей среды ENV(E). Это необходимо, так как имеются еще другие утверждения в блоке.

Команда choice_point сохраняет содержимое регистров B и TP в разделе ADM текущей среды ENV(E), регистра BP в разделе ADM среды ENV(B), т. е. в среде текущей ближайшей точки выбора, которая перестает быть таковой при выполнении команды choice_point. После чего выполняется последовательность регистровых передач

$$B := E,$$

$$BTR := TR,$$

в результате текущая среда становится ближайшей точкой выбора. После выполнения команды choice_point всегда выполняется следующая за ней команда.

Предпоследняя команда блока управления утверждениями — команда del_choice_point. Эта команда, всегда являющаяся парной к команде choice_point, выполняет операции по уничтожению ближайшей точки выбора, созданной командой choice_point данного блока. Команда del_choice_point расположена в блоке управления утверждениями так, что она выполняется при повторном входе в процедуру через последнюю точку входа, т. е. в тот момент, когда точка выбора после выполнения последнего утверждения блока будет исчерпана. Возникновение ситуации, при которой ближайшая точка выбора оказалась исчерпанной, не проверяется — момент уничтожения такой точки выбора обозначен этой командой явно. Алгоритм этой команды следующий. Содержимое регистра B восстанавливается из раздела ADM текущей среды ENV(E), так как всегда в момент выполнения команды выполнено условие B=E. Если обозначить через B^ восстановленное содержимое регистра B, то содержимое регистров BP и BTR восстанавливается из раздела ADM среды ENV(B^).

После выполнения команды del_choice_point выполняется команда try_stmt, которая имеет один операнд, содержащий указатель точки входа Si одного из утверждений предиката. Алгоритм выполнения этой команды следующий. В регистр BP загружается адрес следующей команды — либо try_stmt, либо del_choice_point, после чего управление передается по адресу, содержащемуся в операнде команды. Эти команды и являются точками повторного входа в процедуру (рис. 1.3). Никаких других повторных точек входа нет: в данном блоке управления утверждением находятся все возможные варианты повторного входа (это обеспечивается схемой индексации). Так как команда fail всегда передает управление по адресу, содержащемуся в регистре BP, и повторный вход в процедуру происходит только по этой команде, то данная схема организации блока управления утверждениями обеспечивает требуемое поведение машины логического вывода при возвратах.

Группа команд, начинающаяся командой choice_point и завершающаяся командой del_choice_point, имеется только в блоках управления утверждениями, содержащих более одного утверждения. Если блок управления утверждением содержит только одно

утверждение, то он состоит только из одной команды trust_stmt, имеющей, так же как и команда try_stmt, только один операнд, содержащий адрес соответствующего утверждения Si. Команда trust_stmt выполняет загрузку в регистр P адреса, содержащегося в ее операнде, т. е. передает управление на код утверждения, используется также для последнего утверждения блока управления утверждениями (рис. 1.3) и всегда завершает блок управления утверждениями. Различие между командами try_stmt и trust_stmt состоит в том, что последняя не выполняет модификации адреса следующей точки повторного входа в процедуру.

1.3. КОМАНДЫ УНИФИКАЦИИ

Конкретная реализация алгоритмов выполнения команд load_args и unify_args определяется способом представления термов. Для представления термов широко используются методы разделения и копирования структур. Метод копирования структур (Structure Coping — SC), используемый наиболее широко, впервые был применен в абстрактной машине Уоррена, метод разделения структур (Structure Sharing — SS) более сложен и впервые был применен при реализации Пролога на ЭВМ DEC-10 [50].

Команды блока загрузки аргументов load_args могут быть четырех типов в соответствии с четырьмя основными типами термов нулевого уровня (см. § 1.2). Для каждого фактического параметра (аргумента литерала тела утверждения) кроме структур используется одна команда.

1. Если аргумент — константа, то команда

load_cons Ai C

загружает значение второго операнда C (внутреннее представление константы) в регистр A.

2. Если аргумент — переменная, имеющая первое вхождение в тело утверждения, то команда

load_var Ai V

загружает в регистр Ai указатель (адрес) ячейки вектора VARS, находящегося в среде ENV (C), V — смещение этой ячейки относительно начала вектора и используется для вычисления указателя ячейки как

$$addr(C, V) = \langle \text{значение регистра C} \rangle + V.$$

Кроме того, при выполнении этой команды в ячейку с указанным адресом записывается пустое значение, так как эта ячейка осталась неопределенной после выполнения унификации (ее первое вхождение в теле утверждения, и, следовательно, эта переменная не участвовала в процессе унификации).

3. Если аргумент — связанная переменная (т. е. переменная, имеющая не первое вхождение в утверждении), то команда

load_ref Ai V

загружает в регистр Ai указатель ячейки переменной V, вычисляемый так же, как для предыдущей команды. Но запись пустого значения в ячейку переменной не выполняется, так как эта переменная уже имеет какое-то значение, полученное либо при унификации, либо при выполнении какой-либо команды загрузки аргумента.

Алгоритмы выполнения всех этих команд практически не зависят от способа представления термов.

Если аргумент является структурой, то используется команда

load_str Ai S,

которая определяет загружаемую структуру S в регистр Ai. Алгоритм выполнения этой команды и семантика второго аргумента полностью определяются способом представления термов.

Если используется SC-метод представления термов, то для аргумента литерала, являющегося структурой, генерируется несколько команд, создающих специальные структуры в разделе STRS среды ENV(E), для каждого терма ненулевого уровня — своя команда. Например, для терма t=f1(X, Y, f2(X, f3(a, X), f4(b,

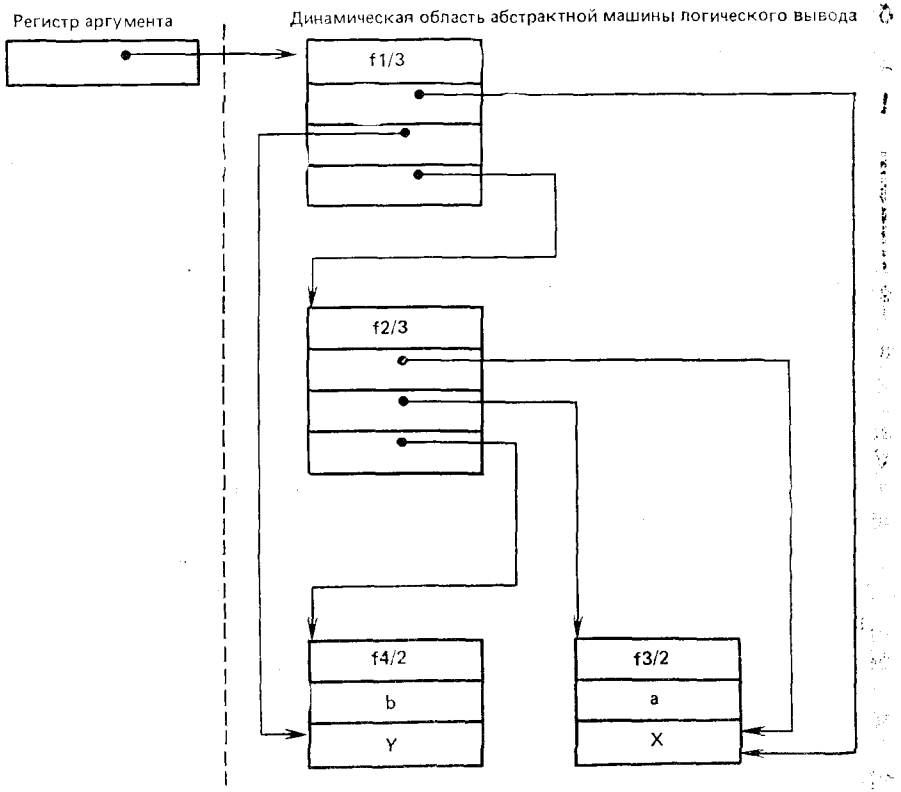


Рис. 1.4 Размещение структуры терма в динамической области памяти и загрузка ее в регистр при копировании структур

Y))) создается структура, изображенная на рис. 1.4. Сначала размещаются подтермы f3 и f4, затем, используя их указатели, подтерм f2, и только после этого размещается сам терм

t=f1 (X, Y) ⟨ссылка на уже размещенный терм⟩.

При этом физически имеется только одна ячейка для переменной, а остальные ее вхождения представляются ссылками на эту ячейку. Ячейки, представляющие переменные X и Y, могут иметь произвольные значения, определяемые их другими вхождениями в утверждение (в том числе и ссылки на другие переменные).

Таким образом, при использовании SC-метода все переменные утверждения, имеющие вхождение в структурах, размещаются в разделе STRS среды ENV(E). Вектор GLOB_VARS среды ENV(E) не используется, можно считать, что он размещается в разделе STRS. Остальные логические переменные утверждения размещаются в векторе LOC_VARS среды ENV(C).

Если используется SS-метод, то загрузка структуры в регистр выполняется только одной командой

load_str Ai S,

где операнд S — указатель скелетона, загружаемый в регистр структуры. Скелетон терма расположен в статической области, кода абстрактной машины логического вывода в разделе DATA (рис. 1.5), структура скелетона аналогична той, которая динамически создавалась при использовании SC-метода. Отличие состоит в том, что скелетон терма строится компилятором и никогда не копируется в динамическую область. Кроме того, ячейки скелетона (по одной на каждый аргумент функтора), соответствующие переменным, содержат не значения этих переменных, а смещения ячеек переменных относительно начала вектора GLOB_VARS, где расположены сами значения переменных. Эти ячейки называются шаблонами переменных. Кроме шаблонов переменных аргументами скелетона может быть константа и указатель подструктуры (другого скелетона). Последний случай имеет место, если аргумент структуры также является структурой.

Все переменные, входящие в утверждение и имеющие хотя бы одно вхождение в структуру, размещаются в векторе GLOB_VARS. Такие переменные называются глобальными. Каждая такая переменная при компиляции получает соответствующее смещение, которое используется при построении скелетона структуры. Остальные переменные, размещаемые в векторе LOC_VARS, называются локальными.

При использовании метода разделения структур команда

load_str Ai S

выполняется следующим образом. В разделе STRS среды ENV(E) создается структура, называемая молекулой. Молекула

mol(PTR1, PTR2)

представляет собой пару указателей: PTR1 — адрес начала век-

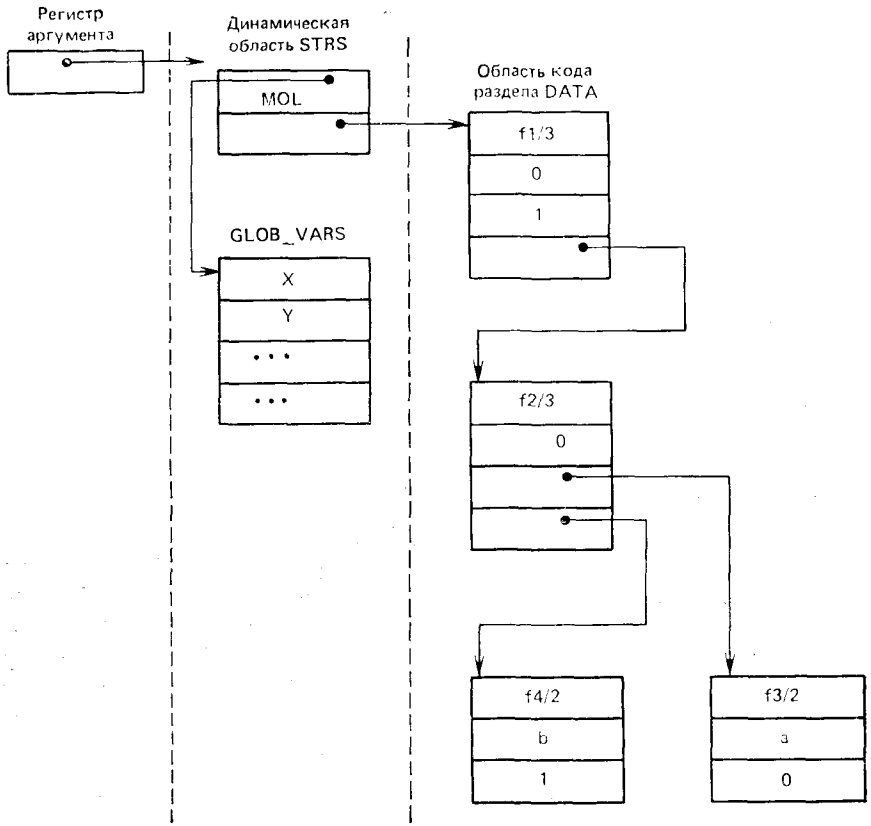


Рис. 1.5. Загрузка структуры терма в регистр при разделении структур

тора GLOB_VARS среды ENV(E), а PTR2 — адрес скелетона S (значение второго операнда команды). Указатель созданной молекулы помещается в регистр Ai. Молекула связывает в единый объект вектор GLOB_VARS и скелетон терма S. В векторе GLOB_VARS находятся значения переменных, которые с помощью скелетона S упорядочены в виде структуры. Молекула полностью определяет терм.

Если обозначить

$$LOC\_ENV = ADM \cup ARGS \cup LOC\_VARS$$

$$GLOB\_ENV = GLOB\_VARS \cup STRS,$$

то при использовании как SC-метода, так и SS-метода GLOB_ENV содержит информацию, связанную с созданием структур. Так как ссылки на структуры, расположенные в GLOB_ENV, загружаются в регистры аргументов и при унификации могут передаваться в качестве значений в вызываемую процедуру или в обратном направлении, то ссылка на некоторую структуру может

оказаться в любой ячейке переменной текущей трассы логического вывода. Поэтому память, занимаемая элементами разделов GLOB_ENV, может быть освобождена только при выполнении команды fail. Чтобы освободить эту память при выполнении команд last_call и return, необходимо определить, действительно ли используются ячейки GLOB_VARS или нет. Для этого необходимо просмотреть значения переменных практически всей трассы логического вывода. Очевидно, что это слишком трудоемкая операция, чтобы ее выполнять в рамках команд return и last_call.

Из этой сравнительной схемы видно, что загрузка аргументов при использовании SS-метода намного эффективнее как по быстродействию, так и по использованию динамической памяти абстрактной машины логического вывода.

При использовании и SS-метода и SC-метода для эффективной реализации доступа и манипулирования элементами трассы логического вывода T достаточно иметь всего два стека — один для хранения LOC_ENV (локальный), а второй стек для хранения GLOB_ENV (глобальный). Первый стек более динамичен — он может освобождаться при выполнении команд last_call и return, второй только при выполнении команды fail.

Команды блока унификации unify_args во многом аналогичны командам загрузки аргументов с точки зрения использования разделов STRS и VARS элементов трассы логического вывода, могут быть четырех типов в соответствии с типами термов нулевого уровня.

Для каждого аргумента головного литерала, не являющегося термом, используется только одна команда, для структур — нескольких команд.

Формат всех команд унификации следующий:

unify_ ... Ai T,

где Ai — регистр аргумента, содержащий фактический параметр, который должен быть унифицирован с термом T — вторым операндом команды (формальный параметр вызываемой процедуры).

Любая команда унификации в качестве самой первой операции выполняет дереференсирование содержимого регистра Ai.

При дереференсировании проверяется, что содержит регистр A — ссылку на пустую переменную, структуру (молекулу) или константу. Если это условие выполнено, то состояние регистра A не изменяется. В противном случае (т. е. если регистр содержит ссылку на переменную, которая сама содержит ссылку или является константой) в регистр A загружается значение, которое находится в ячейке переменной, адрес которой до этого момента находился в регистре A, т. е. выполняется операция

$$A := (A),$$

где (A) — содержимое ячейки с адресом, находящимся в регистре A. После чего выполняется первый шаг алгоритма дереференсирования, описанный выше.

В результате дереференсирования в регистре будет ссылка на пустую пе-

ременную или структуру, либо константа, либо адрес любой переменной трассы логического вывода Т. Операция дереференсирования необходима, чтобы при наличии нескольких связанных ячеек переменных определить корень этой связки, т. е. ячейку, на которую ссылаются все остальные переменные. При унификации запись значения возможна только по этому адресу. Именно из-за необходимости выполнять дереференсирование результаты последующей унификации могут быть записаны в произвольный элемент трассы логического вывода.

1. Если соответствующий аргумент головного литерала — константа, то используется команда

unify_const Ai C,

где второй операнд С является внутренним представлением константы. Эта команда выполняется следующим образом. Сначала выполняется дереференсирование содержимого регистра Ai. Далее проверяется, является ли оно ссылкой на пустую переменную. Если это так, то, используя содержимое регистра Ai как адрес, по этому адресу записывается значение второго операнда команды. После этого выполняется операция фиксации адреса, куда была записана константа, в стеке следа логического вывода opt_trail Ai. Эта операция необходима для правильного восстановления состояния машины логического вывода при выполнении команды fail, выполняется следующим образом.

Если в регистре Ai адрес логической переменной, расположенной в среде, которая старше среды ENV(B), то содержимое регистра Ai записывается в стек следа (содержимое регистра TR получает соответствующее приращение). В противном случае никаких действий операция opt_trail не выполняет.

Если содержимое регистра Ai не является ссылкой на пустую переменную и, кроме того, не есть С (т. е. не совпадает со вторым операндом команды), то автоматически выполняется команда fail. Это означает, что фактические и формальные параметры не унифицируемы. Если содержимое регистра Ai совпадает со вторым операндом команды, то команда успешно завершается и никакие пересылки не выполняются.

2. Если формальный аргумент i — свободная переменная (первое вхождение переменной), то используется команда

unify_var Ai V,

где V — относительный адрес переменной (см. описание команд загрузки аргументов). Выполнение этой команды состоит в следующем. После дереференсирования содержимого регистра Ai, т. е.

Ai := dereference (Ai),

вычисляется адрес ячейки переменной addr(E, V), определяющий ячейку, находящуюся в текущей среде. По этому адресу записывается содержимое регистра Ai.

3. Для формального аргумента i, являющегося повторным вхождением переменной, используется команда

unify_ref Ai V,

которая после дереференсирования регистра Ai и вычисления адреса переменной addr(E, V) запускает алгоритм унификации произвольных термов

general_unify Ai T,

где терм Т определяется адресом addr(V, E) [38]. Эта команда используется в тех случаях, когда на этапе компиляции невозможно определить, как будет происходить процесс унификации. В этом случае нет информации о термах, которые будут унифицироваться, поскольку переменная к моменту унификации (вхождение которой не первое) может иметь произвольное значение.

4. Для формального аргумента i, являющегося структурой, генерируется последовательность команд, начинающихся с команды

unify_str Ai S,

где S — формальный аргумент, являющийся термом. В этом случае сначала выполняется дереференсирование регистра, после чего анализируется полученное содержимое регистра.

Если содержимое регистра пустое, то при использовании SS-метода операнд S интерпретируется как указатель скелетона терма в области кода логической программы. В результате в разделе STRS среды ENV(E) создается молекула

mol(PTR, S),

где PTR — указатель начала вектора глобальных переменных среды ENV(E). Затем ссылка на созданную молекулу записывается по адресу пустой переменной (содержимое регистра Ai) и выполняется операция opt_trail Ai. Оставшиеся команды унификации для данного аргумента не выполняются.

Если содержимое регистра Ai пусто, то при использовании SC-метода в разделе STRS среды ENV(E) создается структура с помощью всей последовательности команд, которые следуют за данной. Эти команды имеют тот же смысл, что и соответствующие команды загрузки аргумента, только загрузка происходит в ячейку, определяемую регистром Ai. Операция opt_trail Ai также должна быть выполнена.

Наиболее сложный случай, когда содержимое регистра Ai не пусто. При использовании SS-метода, если содержимое регистра не является ссылкой на молекулу, то автоматически выполняется команда fail; в противном случае молекула распаковывается. Операция распаковки молекулы

⟨V, SKEL⟩ = decomp(MOL),

(где V, SKEL — специальные унификационные регистры абстрактной машины; MOL — адрес молекулы) загружает в регистр V

адрес начала вектора глобальных переменных, а в регистр SKEL — адрес скелетона структуры. После чего содержимое регистра SKEL сравнивается с функтором, определяемым операндом S. Если функторы неидентичны, то автоматически генерируется команда fail. Если же функторы идентичны, то выполняются последующие команды унификации аргументов структур, т. е. выполняется унификация вложенных термов ненулевого уровня.

При использовании SC-метода эта часть алгоритма несколько проще. Функторы унифицируемых структур доступны непосредственно — указатель первого функтора есть содержимое регистра Ai, а второй функтор содержится в операнде S. Если они совпадают, то выполняются команды глубокой унификации. В противном случае автоматически выполняется команда fail.

Для выполнения глубокой унификации необходим унификационный стек, если в структуре имеются подструктуры. Для каждого аргумента структуры (терма ненулевого уровня) генерируется своя последовательность команд глубокой унификации. Если аргумент структуры не является, в свою очередь, структурой, то для него генерируется всего одна команда. Для аргументов структуры, не являющихся структурами, генерируются команды

unify_cons_deep C,

unify_var_deep V,

unify_ref_deep V.

Операнды этих команд имеют точно такой же смысл, как и команды унификации нулевого уровня (см. выше).

При использовании SS-метода каждая из этих команд в качестве самой первой операции выполняет вычисление адреса аргумента скелетона терма в области кода программы. Для этого может использоваться специальный регистр N, который устанавливается в нулевое состояние командой unify_str. Поэтому каждая команда глубокой унификации выполняет операцию

N := N+1,

после чего вычисляется адрес аргумента скелетона структуры

(SS, N) = SS+N.

Далее вычисляется значение самого терма, соответствующего аргументу скелетона. Если аргумент скелетона есть константа, то это и есть требуемый терм. Если аргумент скелетона есть указатель подтерма S, то в разделе STRS среды ENV(E) создается молекула

mol(PTR, S),

где PTR — указатель начала вектора VARS среды ENV(E). Адрес этой молекулы и есть требуемый терм. Если аргумент скелетона есть шаблон переменной, то вычисляется ее адрес как

offs + ⟨адрес начала вектора VARS среды ENV(E)⟩.

После чего выполняются соответствующие алгоритмы команд

unify_cons A C,

unify_var A V,

unify_ref A V,

где операнд A был вычислен на основе аргумента скелетона.

При SC-методе вместо регистра используется его аналог — регистр H. Первоначальное содержимое регистра H устанавливается командой unify_str равным адресу функтора структуры фактического аргумента, которая находится в динамической области в разделе STRS. Каждая команда в качестве первой операции вычисляет адрес соответствующего аргумента структуры H := H+1. Далее, используя этот адрес и значение операнда команды глубокой унификации, выполняется операция элементарной унификации по соответствующему алгоритму команды унификации нулевого уровня. Если очередной аргумент структуры нулевого уровня есть структура, то происходит переход на следующий более глубокий уровень унификации. Эта операция выполняется следующим образом.

В специальный унификационный стек помещается вся необходимая информация о текущем уровне унификации — содержимое регистров V, SKEL и N для SS-метода и регистра H для SC-метода. Эти операции выполняются специальной командой

new_deep_level.

Для этой команды имеется парная команда

end_of_deep_level,

которая восстанавливает из унификационного стекла содержимое указанных регистров и генерируется после команд и глубокой унификации последнего аргумента вложенной структуры. После команды new_deep_level используются команды глубокой унификации по тем же правилам, что и выше.

Алгоритм операции

general_unify

тот же, что и для команд унификации нулевого уровня. Но только для каждого элемента пары унифицируемых термов требуются все регистры V1, SKEL1, V2, SKEL2, N, загружаемые при распаковке молекул первого и второго аргументов, для SS-метода и H1, H2 для SC-метода.

1.4. ПРИМЕРЫ КОДОВ АБСТРАКТНОЙ МАШИНЫ ЛОГИЧЕСКОГО ВЫВОДА

В программе, инвертирующей списки,

Goal:

S0: ← naive_reverse(c(1, c(2, c(3, c(4, nil)))), X).

```
Pred1:
   S1:  naive_reverse(c(L, X), Z)←naive_reverse(X, Y),
                                  append(Y, c(L, nil), Z).
   S2:  naive_reverse(nil, nil).
Pred2:
   S3:  append(c(L, X), Y, c(L, Z))←append(X, Y, Z).
   S4:  append(nil, Y, Y).
```

```
        ПРОЦЕДУРА GOAL/0
PRED0:  proc_entry          0
   S0:  stmt_entry          (...)
        allocate_env        (...)
        load_skel           A1        SKELETON1
        load_var            A2        X
        last_call           PRED1
```

а)

```
        ПРОЦЕДУРА naive_reverse/2
PRED1:  proc_entry          2
        serch_block:        if A1=variable goto B1
                            if A1="nil"    goto B2
                            in A1="c/2"    goto B3
   B1:  choice_point
        try_stmt            S1
        del_choice_point
        trust_stmt          S2
   B2:  trust_stmt          S2
   B3:  trust_stmt          S1
   S1:  stmt_entry          (...)
        unify_str           A1        SKELETON5
        unify_var_deep      L
        unify_var_deep      X
        unify_var           A2        Z
        allocate_env        (...)
        load_var            A1        X
        load_var            A2        Y
        call                PRED1
        load_var            A1        Y
        load_skel           A2        SKELETON6
        load_var            A3        Z
        last_call           PRED2
   S2:  stmt_entry          (...)
        unify_const         A1        "nil"
        unify_const         A2        "nil"
        return              (...)
```

б)


```
        РАЗДЕЛ ДАННЫХ DATA
SKELETON1:  functor     "c/2"
            const       1
            pointer     SKELETON2
SKELETON2:  functor     "c/2"
            const       2
            pointer     SKELETON3
SKELETON3:  functor     "c/2"
            const       3
            pointer     SKELETON4
SKELETON4:  functor     "c/2"
            const       4
            const       "nil"
            const       "c/2"
SKELETON5:  functor     0
            var_offset  1
            var_offset  "c/2"
SKELETON6:  functor     0
            var_offset  "nil"
            const       "c/2"
SKELETON7:  functor     0
            var_offset  1
            var_offset  "c/2"
SKELETON8:  functor     0
            var_offset  3
            var_offset  "c/2"
```

```
    ПРОЦЕДУРА append/3
PRED2:  proc_entry      3
        search_block:   if A1=variable goto B4
                        if A1="nil"    goto B5
                        if A1="c/2"    goto B6
   B4:  choice_point
        try_stmt        S3
        del_choice_point
        trust_stmt      S4
   B5:  trust_stmt      S4
   B6:  trust_stmt      S3
   S3:  stmt_entry      (...)
        unify_str       A1        SKELETON7
        unify_var_deep  L
        unify_var_deep  X
        unify_var       A2        Y
        unify_str       A3        SKELETON8
        unify_ref_deep  L
        unify_var_deep  Z
        allocate_env    (...)
        load_var        A1
        load_var        A2
        load_var        A3
        last_call       PRED2
   S4:  stmt_entry      (...)
        unify_const     A1        "nil"
        unify_var       A2        Y
        unify_ref       A3        Y
        return          (...)
```

в)

Рис. 1.6. Структура кодов абстрактной машины

списки представляются в виде структуры, пустой список обозначен константой nil. С помощью этих обозначений список [1, 2, 3] (стандартное обозначение, используемое в языке Пролог) будет иметь вид

c(1, c(2, c(3, nil))),

а конструкция языка Пролог [X|Y] имеет вид c(X, Y). Список, который должен быть реверсирован этой программой, состоит из четырех целых чисел и задан в виде первого аргумента цели (утверждение S0). После выполнения этой программы логическая переменная X должна получить значение списка, в котором заданные числа расположены в обратном порядке. Эта программа часто используется в качестве теста, позволяющего определить производительность процессоров логического вывода.

Структуры кодов абстрактной машины для этой программы приведены на рис. 1.6. Этот код построен в предположении, что при представлении термов используется метод разделения структур. Статическая область кода для этой программы содержит четыре фрагмента. Первые три (рис. 1.6,*а—в*) представляют собой код для цели и двух предикатов программы, а четвертый фрагмент (рис. 1.6,*г*) содержит данные, построенные компилятором.

Первый фрагмент (рис. 1.6,*а*) содержит команды, которые фактически инициализируют машину логического вывода. В результате выполнения этих команд в регистр A1 загружается адрес молекулы, построенной на основе скелетона структуры, представляющей заданный список из четырех целых чисел, — SKELETON1. Этот скелетон расположен компилятором в разделе DATA (рис. 1.6,*г*) и фактически состоит из четырех элементарных скелетонов с функтором c/2.

SKELETON1—4, расположенные во фрагменте данных, представляют исходный список, заданный в утверждении S0 исходной программы. Второй аргумент каждого из этих скелетонов содержит ссылку на следующий скелетон. Последний скелетон SKELETON4 в качестве второго аргумента имеет константу nil. В результате выполнения первого фрагмента программы в регистр A2 загружается адрес переменной, в которую будет помещена ссылка на реверсированный список, строящийся в процессе выполнения программы. Эта переменная размещена и инициализирована пустым значением при выполнении команд stmt_entry и allocate_env, которые предшествовали операциям загрузки регистров аргументов. Выполнение первого фрагмента завершается передачей управления на основную точку входа PRED1, после чего начинается выполнение собственно программы.

Структура кода программы такова, что управление больше никогда не будет передано в первый фрагмент. Это вытекает из семантики цели, назначение которой в данном случае — только инициализация регистров абстрактной машины и размещение переменной для окончательного результата.

Выполнение второго фрагмента (рис. 1.6,*б*) начинается с сохранения содержимого регистров аргументов A1 и A2 в текущей среде трассы логического вывода. В данный момент в регистре C адрес среды, созданной при выполнении первого фрагмента программы, в регистре CP — адрес команды успешного завершения процесса логического вывода, в регистре BP — адрес команды неуспешного завершения процесса логического вывода. Содержимое регистра B несущественно, но должно определять фиктивный элемент трассы логического вывода, который старше начального (среды, созданной при инициализации машины логического вывода). Регистры TR и BTR находятся в исходном состоянии и содержат адрес начала стека следа логического вывода.

Процедура реверсирования списков (рис. 1.6,*б*) использует SKELETON5 и SKELETON6 в области данных (рис. 1.6,*г*). Ячейки, соответствующие переменным в этих скелетонах, имеют смещения переменных в векторе VARS: переменная, которая имеет самое левое вхождение в утверждение, 0, следующая переменная — 1 и т. д. (т. е. L имеет смещение 0, X — смещение 1).

После сохранения содержимого регистров аргументов выполняется анализ содержимого первого регистра. Так как этот регистр содержит указатель структуры с функтором c/2, то управление передается в блок управления утверждениями с меткой B3, а затем на точку входа утверждения S1. Выполнение утверждения S1 начинается с инициализации вектора переменных VARS текущей среды трассы логического вывода.

Затем выполняется унификация содержимого регистра A1 со структурой, определяемой SKELETON5. Так как содержимое регистра A1 непусто, то выполняются все три команды унификации первого аргумента с содержимым регистра A1. В результате переменная L получает значение первого элемента исходного списка, а переменная X — ссылки на молекулу, которая связывает SKELTON2 и некоторый вектор глобальных переменных (в данном случае он не имеет значения, так как исходный список не содержит переменных). Эта молекула определяет исходный список без первого элемента. В результате унификации содержимого регистра A2 с переменной Z последняя получает значение ссылки на пустую переменную X, расположенную в среде ENV(C). Это и есть адрес, где должен быть размещен окончательный результат.

После завершения унификации создается новый текущий элемент трассы логического вывода, а в регистр C загружается указатель на среду, в которой расположены только что вычисленные значения переменных L, X и Z. Затем выполняется загрузка регистров A1 и A2 и вызов опять же процедуры реверсирования списков. При этом в регистр CP загружается адрес команды, с которой начинается последовательность команд вызова процедуры склейки списков. В данном случае это склейка исходного реверсированного списка без первого элемента и списка, состоящего только из первого элемента. Этот повторный вызов процедуры реверсирования списков отличается от первого только тем, что в регистре A1 находится ссылка на молекулу, определяющая исходный список без первого элемента. Так возникает цикл, который выполняется по точно такой же схеме еще три раза. В результате будут созданы еще четыре копии переменной L, содержащие четыре элемента исходного списка. Однако при четвертом вызове процедуры склейки списков четвертая копия переменной X получит значение nil. Поэтому пятый вызов процедуры склейки списков будет выполнен уже по другой схеме.

Так как при пятом вызове процедуры реверсирования списков в регистре A1 будет значение константы nil, то при выполнении команд search_block управление будет передано на блок управления утверждениями B2, т. е. на утверждение S2. Так как утверждение S2 есть тривиальный факт, то будет выполнен возврат управления командой return по адресу, содержащемуся в регистре CP. При этом четвертая копия переменной Y получит значение nil.

В результате выполнения пяти вызовов процедуры реверсирования списков создано четыре копии для каждой переменной L, X, Y и Z. Эти копии имеют следующие значения:

```
1 : L1=1   Z1→X0   Y1=empty
2 : L2=2   Z2→Y1   Y2=empty
3 : L3=3   Z3→Y2   Y3=empty
4 : L4=4   Z4→Y3   Y4=empty
5 :                Y4=nil,
```

где стрелками показаны значения ссылок на ячейки переменных, а X0 обозначает ячейку, созданную при выполнении первого фрагмента. В X0 должен быть помещен окончательный результат. Значения копий переменной X не показаны, так как для дальнейшего они несущественны.

После выполнения этих пяти вызовов процедуры реверсирования списков в регистре CP будет адрес последовательности команд вызова процедуры склейки списков. При этом в регистр A1 будет загружена константа nil, а в регистр A2 — ссылка на молекулу, определяющую список из одного четвертого элемента исходного списка (четвертая копия переменной L). При этом в регистр A3 будет загружен адрес ячейки четвертой копии переменной Z.

Далее произойдет первый вызов процедуры склейки списков (рис. 1.6,*в*). Процедура склейки списков использует SKELETON7 и SKELETON8 в области данных (рис. 1.6,*г*). Переменная утверждения S4 имеет следующие смещения: L—0, X—1, Y—2, Z—3.

Так как в регистре A1 константа nil, то управление будет передано на утверждение S4, которое является фактом. После унификации, в результате которой Z4 получит значение c(4, nil) (такое же значение будет иметь и Y3), управление будет передано по адресу в регистре CP (предварительно восстановленному из трассы логического вывода), указывающему на команду вызова процедуры склейки списков по адресу среды, созданной при третьем вызове процедуры реверсирования списков, в регистре C. Поэтому в регистр A1 будет загружена ссылка на c(4, nil), а в регистр A2 — ссылка на c(3, nil), построенное на основе третьей копии переменной L. В регистр A3 будет загружена ссылка на Z3.

Таким образом, при втором вызове процедуры склейки списков будет использовано утверждение S3, в результате происходит склейка списков, заданных регистрами A1 и A2 в момент этого второго вызова. Склейка этих списков будет завершена и выполнена с помощью еще одного «внутреннего» вызова процедуры склейки списков с использованием утверждения S4. В результате переменные Z3 и Y2 получат значение c(4, c(3, nil)). Эта последовательность вызовов процедуры склейки списков повторится еще 2 раза, в результате копии переменной Z получат следующие значения:

$$Z2=Y1=c(4,\ c(3,\ c(2,\ nil))),$$
$$Z1=X0=c(4,\ c(3,\ c(2,\ c(1,\ nil)))),$$

а регистр CP — адрес команды успешного завершения процесса логического вывода. После чего машина логического вывода остановится.

ГЛАВА 2.

СТРУКТУРЫ ДАННЫХ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

Данная глава логически заканчивает спецификацию абстрактной машины логического вывода — посвящена представлению данных и предикатов в памяти абстрактной машины. Простые (неструктурированные) данные относятся к стандартному типу. Структурированные данные создаются и поддерживаются на основе механизма разделения структур. Каждый структурированный объект (молекула) состоит из двух слов: первое определяет адрес переменных, участвующих в записи молекулы, а второе — адрес фрейма структуры (скелетона). Достоинство метода разделения структур заключается в экономии памяти абстрактной машины, поскольку унифицируемые структурированные термы ссылаются на один и тот же фрейм (скелетон).

В главе рассмотрены представления в памяти предикатов; при этом различаются встроенные предикаты и предикаты, определенные пользователем. Приведено описание команд, используемых для работы с данными и предикатами.

2.1. ЭЛЕМЕНТАРНЫЕ СТРУКТУРЫ ДАННЫХ

Эффективность процессора логического вывода во многом определяется тем, насколько удачно выбраны структуры данных для трассы логического вывода, представления термов и т. д. Для представления всех необходимых объектов используются стандартные элементарные структуры данных. Структуры данных, рассмотренные ниже, выбраны с учетом наиболее эффективной реализации процессора логического вывода, использующего метод разделения структур для представления термов.

Элементарная структура данных представляет собой 32-битовое машинное слово, первый (старший) байт которого всегда интерпретируется как флаг (рис. 2.1). Однобайтовый флаг (тег) элемента всегда определяет семантику остальных байтов.

Элементы могут быть двух типов: стандартный и функтор. Стандартные элементы используются для представления всех объектов, кроме функторов. Содержимое стандартного элемента час-

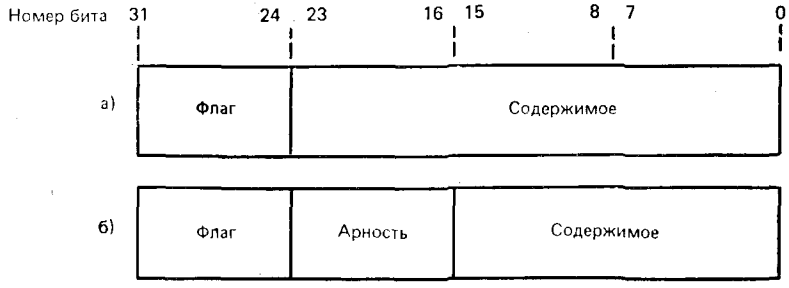


Рис. 2.1. Элементарные структуры данных

то используется как адрес. Этот адрес может быть 24-битовым.

Использование только 24-битового адреса несколько ограничивает максимально допустимый объем программы. Но чтобы использовать полное 32-битовое слово в качестве адреса, необходимо вынести тег за пределы слова. А это, в свою очередь, потребует нестандартной организации памяти, и, следовательно, процессор логического вывода нельзя будет использовать в качестве сопроцессора к ЭВМ с традиционной архитектурой. Поскольку процессор логического вывода целесообразно проектировать только для выполнения операций логического вывода и не дублировать функции традиционных процессоров, то вариант с расширенным словом памяти представляется менее приемлемым.

Для организации области кода программы используются специальные бинарные таблицы символов и предикатов.

Таблица символов содержит все символы, встречающиеся в программе. Каждый элемент таблицы символов представляет собой последовательность байтов, кодирующих символы строки исходного текста программы. Таблица символа представляет собой вектор 32-битовых слов. Каждый символ исходного текста программы обязательно кодируется одним или несколькими 32-битовыми словами таблицы символов.

Каждому символу исходного текста программы приписывается индекс (номер) элемента таблицы символов, с которого начинается строка этого символа в таблице символов. Все идентичные символы исходного текста программы получают один и тот же индекс. Если в строке число символов не кратно четырем, то она дополняется справа пробелами.

Все операции с термами, в которых участвуют символы исходной программы, выполняются процессором логического вывода путем манипулирования индексами этих символов. Реальные значения, хранящиеся в таблице символов, необходимы только при выполнении операций ввода-вывода и добавлении новых термов в программу. При добавлении новых термов (например, нового утверждения или целого предиката) эти термы поступают в исходном виде. Все новые символы должны быть приведены в соответствие с уже имеющимися в таблице, т. е. символы новых термов, которые уже имеются в таблице, не должны получить нового индекса. Чтобы проверить наличие или отсутствие в ней данного символа, необходимо сканирование таблицы символов. Для больших программ это достаточно трудоемкая операция, требующая специальной организации поиска символа в таблице.

Таблица предикатов также представляет собой вектор 32-битовых слов, содержит список всех предикатов программы. Для каждого предиката программы в таблице имеется запись, состоящая из двух 32-битовых слов. Первое слово — функтор, кодирующий имя и арность предиката, второе — стандартный элемент, флаг которого определяет тип предиката, а содержимое — адрес основной точки входа в процедуру предиката.

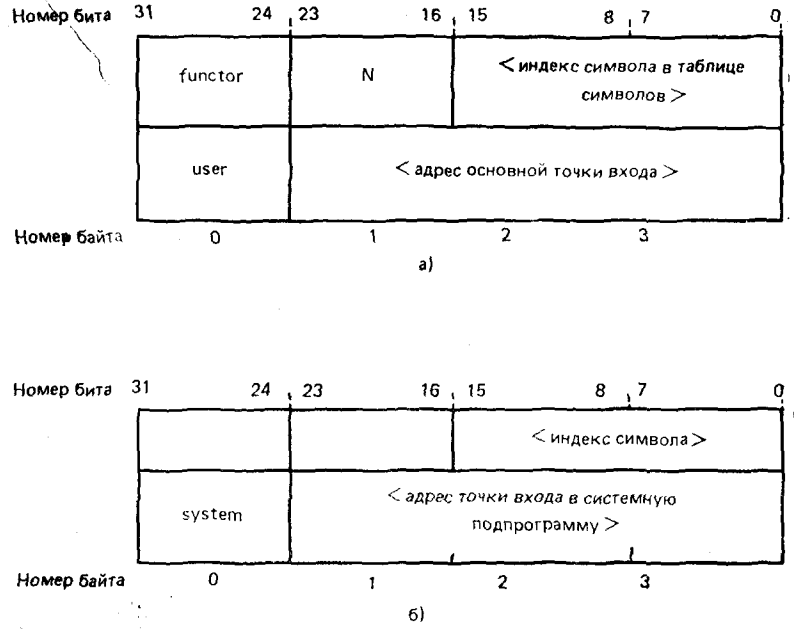


Рис. 2.2. Структура предикатов

Имеется два типа предикатов: предикаты пользователя и системные (встроенные) предикаты. Предикат первого типа (рис. 2.2,*а*) — это предикат, процедура которого имелась в исходном тексте Пролог-программы и была скомпилирована в команды процессора логического вывода. В этом случае второе слово содержит адрес основной точки входа в процедуру предиката в области кода программы. Предикат второго типа (рис. 2.2,*б*) — это системный (встроенный) предикат. В исходном тексте Пролог-программы для него не имеется кода. Обычно многие часто используемые операции, такие как арифметические, операции над строками символов, ввод-вывод, выполняются на процессоре традиционной архитектуры и обрабатываются как внешние подпрограммы. Поэтому второе слово описателя предиката содержит точку входа в подпрограмму. Выполнение предиката второго типа происходит вне процессора логического вывода.

Таблица предикатов, так же как и таблица символов, необходима при добавлении новых предикатов или утверждений в процессе выполнения программы. В этом случае она используется для поиска имеющихся предикатов или утверждений. Но существует еще одна специальная операция, часто используемая в Пролог-программах, для которой наличие таблиц символов и предикатов обязательно. В Пролог-программах можно выполнять метавызов предиката. Это означает, что на этапе компиляции имя вызываемого предиката задано переменной, т. е. не определено. В процес-

се выполнения программы переменная может получить значение некоторого терма, главный функтор которого должен быть интерпретирован как имя вызываемого предиката, а его аргументы — как фактические параметры вызова этого предиката. В этом случае с помощью таблицы символов находится символьное представление функтора, а затем выполняется поиск этого функтора в таблице предикатов, что позволяет вычислить точку входа в процедуру этого предиката. После чего выполняется сам вызов предиката.

В области кода программы кроме бинарных таблиц и кода процессора логического вывода располагаются данные, являющиеся скелетонами структур. Кроме того, в операндах команд загрузки аргументов и унификации могут находиться константы и указатели скелетонов структур. Поэтому для представления программы необходимы следующие элементарные объекты данных: константа; целое число; указатель скелетона структуры; функтор скелетона структуры; шаблон логической переменной. Форматы этих типов данных изображены на рис. 2.3. Все они создаются компилятором на основе предварительно построенной таблицы символов.

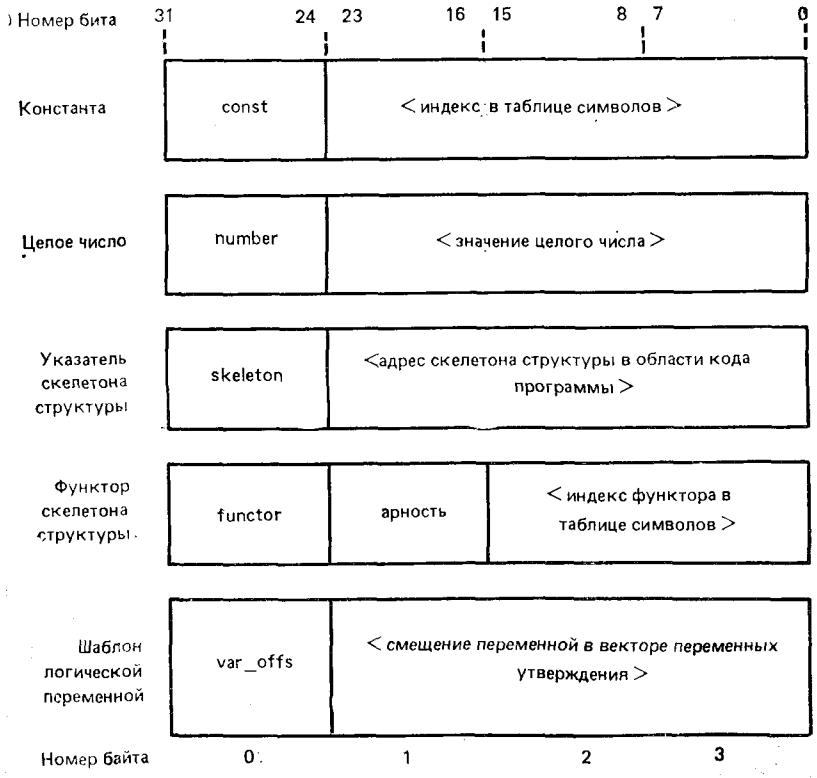


Рис. 2.3. Форматы типов данных

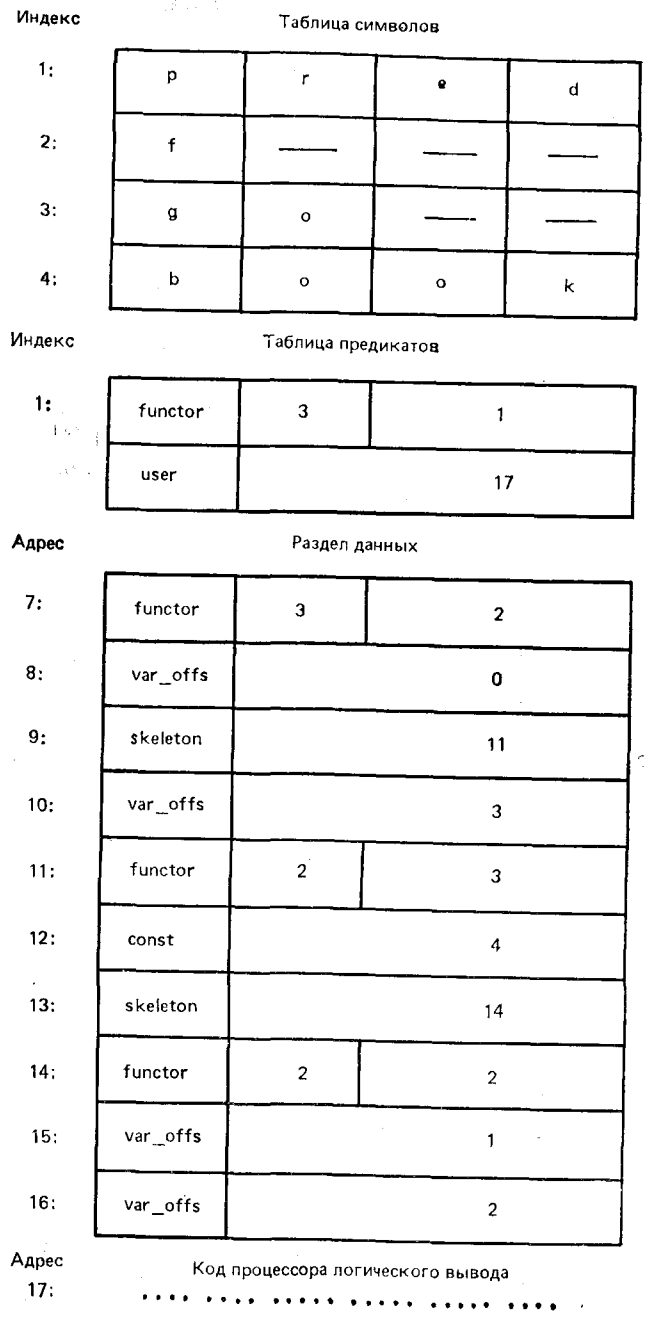


Рис. 2.4. Структура области кода

Для каждого утверждения компилятор планирует два вектора переменных. Первый — это вектор локальных переменных, т. е. переменных, не имеющих вхождений в структурах. Глобальные переменные — это переменные, имеющие хотя бы одно вхождение в структуру. Поэтому значение шаблона переменной, используемое для представления переменной в скелетоне структуры, — это смещение переменной в векторе глобальных переменных утверждения. Например, для логической программы

$$\text{pred (X, f (X, go(book, f(Y, Z))), W)}$$

таблица символов, предикатов и раздел данных будут иметь структуру, изображенную на рис. 2.4.

2.2. СТРУКТУРА ДИНАМИЧЕСКОЙ ОБЛАСТИ И РЕГИСТРЫ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

Динамическая область должна содержать следующие объекты: трассу логического вывода, след логического вывода, унификационный стек (см. гл. 1). Отдельные части элементов трассы логического вывода имеют разное время жизни. Глобальные переменные и структуры, создаваемые в процессе логического вывода, сохраняются вплоть до выполнения команды fail, локальные переменные и административные разделы трассы логического вывода освобождаются значительно чаще. Поэтому целесообразно трассу логического вывода реализовывать в виде двух независимых стеков: локального (основного) и глобального.

В основном стеке располагаются локальные переменные среды процедуры и вектор административного раздела, в глобальном стеке — векторы глобальных переменных и молекулы, создаваемые в процессе логического вывода.

Локальный и глобальный стеки организуются как векторы 32-битовых слов. Стеки должны быть расположены в памяти так, чтобы увеличение размера стеков происходило в направлении больших адресов, локальный стек находился в области старших адресов, а глобальный — в области младших адресов (рис. 2.5). Такое расположение стеков позволяет эффективно организовать целый ряд операций. Например, можно добиться такой структуры ссылок, чтобы все ссылки, имеющиеся в локальном стеке, были направлены в область меньших адресов, и в силу этого упрощается освобождение локального стека — не надо проверять наличие ссылок из глобального стека на освобождаемый участок локального.

Резервные зоны имеют важное значение для организации обработки исключительного состояния, которое может возникнуть при переполнении стеков. В этом случае все данные продолжают записываться в резервную зону, но следующий вызов процедуры не выполняется, и управление передается обработчику прерываний. Резервная зона позволяет сохранить все необходимые данные, и если имеется свободная дополнительная память, то после

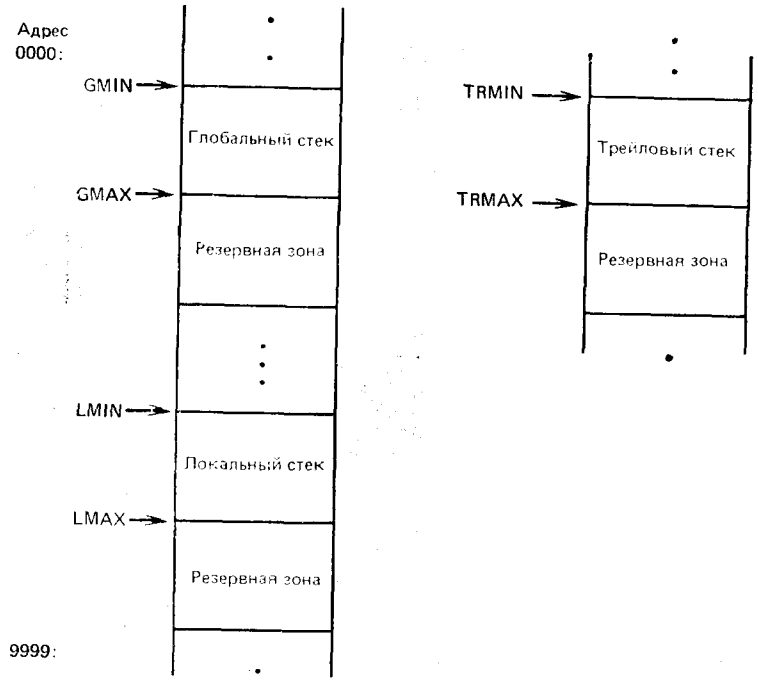


Рис. 2.5. Размещение стеков в памяти

ее выделения процесс логического вывода может быть продолжен. Минимально необходимый размер резервной зоны может быть вычислен компилятором.

Для эффективного доступа процессора логического вывода к этим стекам содержимое регистров абстрактной машины логического вывода должно быть несколько изменено. Вместо регистра E необходимо использовать два регистра L и G. Регистр G содержит указатель начала вектора локальной части элемента трассы логического вывода, а регистр G — адрес начала глобальной части этого же элемента. Все алгоритмы команд, рассмотренные в гл. 1, которые манипулируют регистром E, легко модифицируются — они должны выполнять те же действия, но уже над парой регистров. Точно так же вместо регистра C абстрактной машины необходимо использовать регистры CL и CG, а вместо регистра B — регистры BL и BG. Соответствующим образом должны быть изменены и алгоритмы команд. Кроме указанных изменений, раздел ADM должен содержать вместо одного регистра абстрактной машины пару соответствующих регистров, полученных расщеплением абстрактного регистра. Регистры следа логического вывода TR и BTR изменений не требуют.

Вектор раздела ADM локальной среды можно организовать следующим образом. В этом векторе достаточно иметь всего шесть

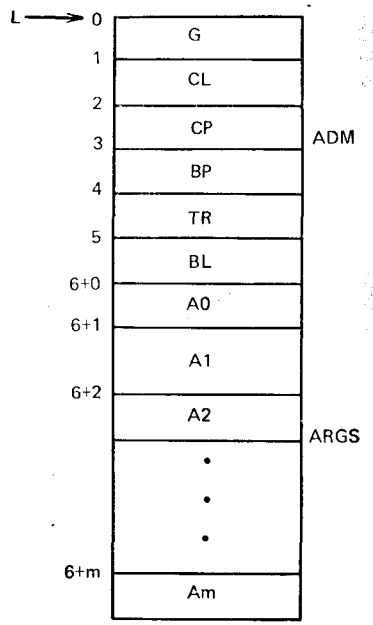


Рис. 2.6. Структура раздела ADM локальной среды

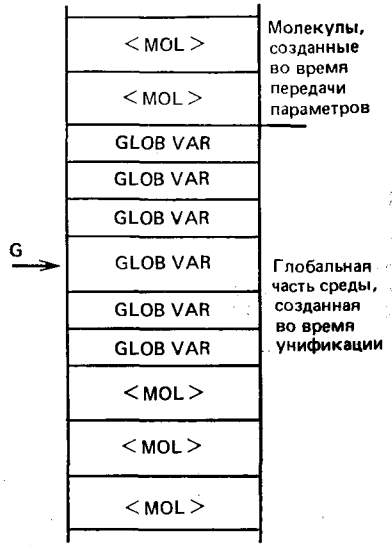


Рис 2.7. Состояние глобального стека после унификации

регистров, как это изображено на рис. 2.6. Если каждая локальная среда процедуры содержит копии содержимого регистров, указанных на рис. 2.6, то содержимое недостающих регистров CG и BG легко может быть восстановлено из соответствующих локальных сред. Такой способ организации административной части позволяет сэкономить место в локальном стеке.

После административной части локальной среды следует вектор регистров аргументов (рис. 2.6). Вслед за ним располагаются ячейки локальных переменных.

Эта структура локальной среды учитывается компилятором при назначении смещений локальных переменных утверждения. Так, если число аргументов предиката равно 3, то переменная, первое вхождение которой в утверждение этого предиката является самым левым, получит смещение $9=6+3$.

Глобальный стек содержит векторы глобальных переменных и молекулы, созданные при загрузке регистров аргументов и унификации. Нет необходимости располагать молекулы непосредственно до или после соответствующего вектора глобальных переменных. Молекула всегда располагается в первых двух свободных ячейках глобального стека, вектор глобальных переменных — сразу же после входа в утверждение. Типичное состояние глобального стека непосредственно после завершения унификации изображено на рис. 2.7.

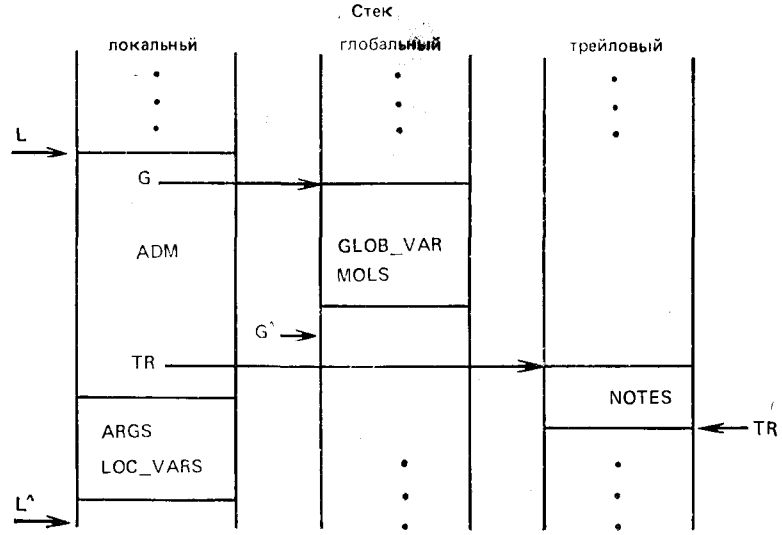


Рис. 2.8. Структура текущей среды для команды allocate_env

Таким образом, элементы ENV трассы логического вывода и дополнительная информация следа логического вывода NOTES, создаваемые в процессе выполнения вызова процедуры, располагаются в трех стеках. Типичная структура текущей среды в момент выполнения команды allocate_env изображена на рис. 2.8. Регистр L указывает на начало расположения локальной части среды, а регистр G содержит адрес начала глобальной части среды. Регистр TR содержит адрес первой свободной ячейки стека следа, а соответствующая ему ячейка в разделе ADM — адрес первой отметки, возникшей при унификации.

После выполнения команды allocate_env регистры L и G будут соответственно иметь значения L^ и G^ (рис. 2.8). Операндами этой команды являются размеры локальной и глобальной частей среды, вычисляемые компилятором.

Стек следа (унификационный) имеет достаточно короткое время жизни — он необходим только во время выполнения одной команды унификации, поэтому может располагаться в свободном пространстве локального стека. Так как время доступа к нему является критическим параметром при выполнении глубокой унификации, для большей эффективности унификации целесообразно для этого стека иметь специальную быстродействующую память.

2.3. ЛОГИЧЕСКИЕ ПЕРЕМЕННЫЕ И РЕГИСТРЫ АРГУМЕНТОВ

При выполнении программы процессор логического вывода должен создавать новые структуры данных, которые могут быть значениями переменных и регистров аргументов следующих типов:

константа, целое число, ссылка, пустое значение, ссылка на молекулу. Кроме того, в глобальном стеке могут создаваться молекулы, а в стеке следа логического вывода — отметки адресов переменных.

Константа и целое число имеют точно такой же формат, как и аналогичные объекты данных, используемые для представления программы в статической области (см. рис. 2.3). Форматы остальных значений переменных и отметок адресов переменных представлены на рис. 2.9. Ссылка и пустое значение переменной имеют один и тот же флаг 0, значение которого жестко фиксировано. При этом предполагается, что все остальные флаги не равны нулю. Это сделано для того, чтобы можно было организовать дереференсирование переменной как можно эффективнее.

Дереференсирование — одна из наиболее часто используемых операций, так как не только каждая команда унификации начинается с этой операции, но и при выполнении алгоритма унификации эта операция также используется очень часто. Алгоритм дереференсирования переменной основан на следующем правиле.

При унификации двух пустых переменных в ячейках одной из них записывается адрес другой (см. гл. 1). Однако для эффективной организации дереференсирования необходимо всегда записывать меньший адрес в ячейку с большим адресом. В результате ссылки всегда будут направлены от больших адресов к меньшим. Это правило построено с учетом относительного расположения локального и глобального стеков в памяти. Унификация пустых переменных в соответствии с этим правилом гарантирует, что ссылок из глобального стека в локальный возникнуть не может. Тогда алгоритм дереференсирования может быть организован следующим образом.

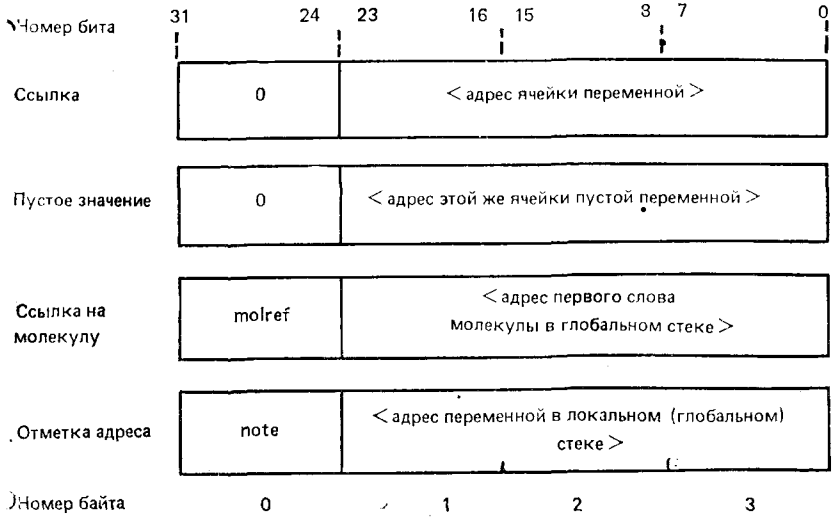


Рис. 2.9. Форматы значений логических переменных

Содержимое ячейки переменной сравнивается с ее адресом, который уже находится в регистре адреса процессора. Если эти значения совпадают, то ячейка имеет пустое значение. Если адрес ячейки меньше, чем ее содержимое, то ячейка имеет непустое значение, так как ссылка не может быть направлена в сторону большего адреса. Это объясняется тем, что все непустые значения переменной имеют отличный от нуля флаг и полное 32-битовое слово, интерпретируемое как адрес, всегда даст адрес, больший, чем адрес любой переменной.

Как в первом, так и во втором случае дереференсирование может быть закончено. Если же ни первое, ни второе условия не выполнены, то в регистр адреса процессора пересылается содержимое ячейки и очередной шаг дереференсирования повторяется. В этом случае содержимое ячейки было ссылкой на другую ячейку логической переменной.

Для использования этого алгоритма необходимо уточнение алгоритмов команд абстрактной машины логического вывода.

Молекула также является структурой, создаваемой процессором логического вывода (рис. 2.10). Первое слово молекулы определяет адрес начала вектора глобальных переменных, среди которых находятся переменные, входящие в терм, описываемый молекулой. Второе слово молекулы — это адрес скелетона структуры терма, используемый для представления данных программы в статической области машины логического вывода.

Ссылка на молекулу создается либо при формировании содержимого регистра аргумента, либо при унификации. Вообще говоря, этот объект данных не является необходимым. Можно записывать саму молекулу в регистр аргумента или ячейку переменной. Поскольку молекула состоит из двух 32-битовых слов (рис. 2.10), то ячейки переменных и регистры аргументов должны быть 64-разрядными, что нецелесообразно.

Отметки адреса могут находиться только в стеке следа логического вывода. Поэтому наличие специального флага, вообще говоря, не обязательно. В ряде реализаций Пролога стек следа логического вывода используется и для других целей (для реализации встроенных предикатов некоторых типов). В этом случае в

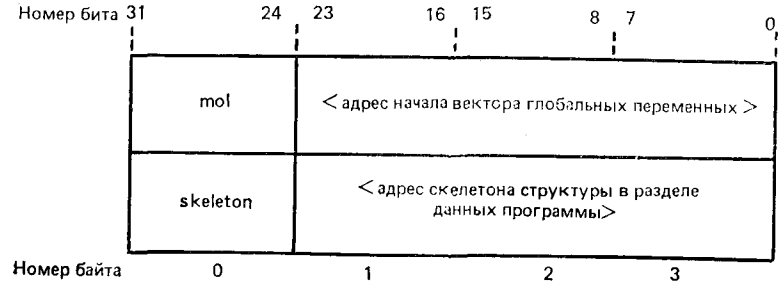


Рис. 2.10. Формат молекул

Таблица 2.1

Флаг	Объект данных	Назначение
const	Константа	Область кода, локальный и глобальный стеки
number	Целое число	То же
skeleton	Указатель скелетона структуры	Область кода в составе молекулы в глобальном стеке
var_offs	Шаблон логической переменной	Область кода в составе скелетона структуры
user	Предикат, определенный пользователем	Таблица предикатов
system	Системный (встроенный) предикат	То же
0	Ссылка или пустая переменная	Локальный и глобальный стеки
molref	Ссылка на молекулу	То же
mol	Молекула (первое слово)	Глобальный стек
note	Отметка адреса переменной	Стек следа логического вывода

стеке следа могут быть не только отметки адресов переменных, но и необходим специальный флаг (рис. 2.9).

Все флаги и назначение объектов данных, которые могут интерпретироваться процессором логического вывода, приведены в табл. 2.1.

2.4. КОМАНДЫ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

В этом параграфе подробно рассматриваются алгоритмы команд процессора логического вывода, ориентированные на структуры данных области кода и динамической области.

Команды процессора логического вывода (табл. 2.2) построены на основе абстрактной машины логического вывода (см. гл. 1).

Система команд процессора логического вывода представляет собой систему команд высокого уровня. Это означает, что эти команды выполняют достаточно сложные алгоритмы, требующие большого количества регистровых передач. Эта система команд отличается от системы команд абстрактной машины следующим.

Во-первых, добавлены команды индексации, которые на уровне абстрактной машины не рассматривались (было указано только их назначение и использование при помощи блока команд search_block).

Во-вторых, команды глубокой унификации исключены из системы команд процессора логического вывода — при аппаратной

Таблица 2.2

Команды ПЛВ		Операнд 1	Операнд 2	Операнд 3
Команды управления предикатом и утверждением	proc_entry	N		
	choice_point			
	del_choice_point			
	try_stmt	ENTRY		
	trust_stmt	ENTRY		
	stmt_entry	GLOBALS		
	allocate_env	LOCALS		
	return			
Команда возврата	fail			
Команды вызова предикатов	call	ENTRY		
	last_call	ENTRY		
Команды индексации	if_const	Ai	CONST	ENTRY
	if_str	Ai	FUNCTOR	ENTRY
	if_empty	Ai	ENTRY	
Команды загрузки аргументов	load_cons	Ai	CONST	
	load_var	Ai	VAR	
	load_ref	Ai	VAR	
	load_str	Ai	SKEL	
Команды унификации	unify_const	Ai	CONST	
	unify_var	Ai	VAR	
	unify_ref	Ai	VAR	
	unify_str	Ai	SKEL	

реализации алгоритма унификации термов нецелесообразно выполнять глубокую компиляцию структур. В этом случае целесообразно все операции глубокой унификации инициировать и выполнять в рамках одной команды unify_str, используя алгоритм общей унификации в рамках выполнения этой команды, т. е. в режиме аппаратной интерпретации. Режим глубокой унификации обычно используется при программной реализации абстрактной машины логического вывода.

Форматы команд процессора логического вывода построены следующим образом. Код операции занимает один байт, а следующие за ним байты отводятся под операнды. Команды имеют не более трех операндов, некоторые не имеют операндов. Все возможные типы операндов команд процессора логического вывода приведены в табл. 2.3.

При описании алгоритмов команд используется специальный псевдоязык. Этот язык содержит следующие основные конструкции:

L, G, G1, CL, CG, BL, BG, TR, BTR, P, CP, BP, Ai — регистры процессора логического вывода, обозначаемые прописными латинскими буквами;

ADDR, SS, SG — дополнительные рабочие регистры;

0, 1, 2, ... — целочисленные константы;

оператор цикла for i: =1 until N;

Таблица 2.3

Тип операнда	Размер, байт	Назначение
N	1	Малое целое число
Ai	2	Код регистра аргумента номер 1
ENTRY	3	Относительный адрес точки входа в процедуру, утверждение или блок управления утверждениями
CONST	4	Константа или целое число (см. гл. 3)
SKEL	4	Указатель скелетона (см. гл. 3)
FUNCTOR	4	Функтор (см. гл. 3)
LOCALS	2	Размер локальной среды для данного утверждения
GLOBALS	2	Размер глобальной среды для данного утверждения
VAR	3	Адрес переменной, представленный как ⟨R, OFFS⟩, где R — код регистра (1 байт), а OFFS — смещение (2 байта)

тело оператора, обозначаемое операторными скобками do и od;

условный оператор if ... then ... else ... fi;

+, —, =, =/=, ⟨, ⟩ — стандартные арифметические операции;

регистровая передача ⟨регистр⟩←⟨регистр⟩+⟨ ... ⟩ или ⟨регистр⟩←⟨ ... ⟩ (пересылка содержимого одного регистра в другой с указанным преобразованием);

cell(ADDR) — содержимое 32-разрядной ячейки с адресом ADDR;

goto ADDR — оператор безусловного перехода по адресу ADDR;

length (⟨команда процессора⟩) — процедура-функция, вычисляющая длину указанной команды процессора.

В момент выполнения команды операнды находятся в рабочих регистрах процессора, обозначаются так же, как регистры.

Команды управления предикатом и утверждением

В алгоритмах команд этой группы

```
proc_entry          N
for i:=1 until N do store Ai in cell((L+(5+i))
                    od

stmt_entry          GLOBALS
G1<—G+GLOBALS

try_stmt            ENTRY
BP<—P+length_of(try_stmt)
goto P+ENTRY

trust_stmt          ENTRY
goto P+ENTRY
choice_point
store TR in cell (L+4)
store BL in cell (L+5)
store BP in cell (BL+3)
BL<—L
BG<—G
BTR<—TR
del_choice_point
restore BL from cell (L+5)
restore BP from cell (BL+3)
restore BG from cell (BL+0)
restore BTR from cell (BL+4)
allocate_env        LOCALS
store G  in cell (L+0)
store CL in cell (L+1)
store CP in cell (L+2)
CL<—L
CG<—G
L<—L+LOCALS
G<—G1
return
if BL=L do store G  in cell (L+0)
           store CL in cell (L+1)
           store CP in cell (L+2)
           CL<—L
           CG<—G
           L<—L+LOCALS
        od
fi
G<—G1
goto CP
```

используются следующие дополнительные процедуры:

store ⟨регистр⟩ in cell (⟨регистр⟩+⟨смещение⟩) — запись содержимого регистра в ячейку памяти с адресом, указанным как содержимое регистра плюс заданное смещение;

restore ⟨регистр⟩ from cell (⟨регистр ⟩+⟨смещение⟩) — запись в регистр содержимого ячейки памяти.

Алгоритм команды proc_entry в точности соответствует алгоритму этой команды на уровне абстрактной машины логического вывода (см. гл. 1).

Алгоритмы команд choice_point и del_choice_point модифицированы по сравнению с алгоритмом их выполнения на уровне абстрактной машины, учитывают расщепление среды процедуры на локальную и глобальную, а также структуру административной части и локальной среды.

Алгоритм команды try_stmt и trust_stmt в точности соответствует алгоритму на уровне абстрактной машины. Однако при их

выполнении регистр текущей команды процессора P содержит адрес первого байта выполняемой команды. По завершении команд регистр P содержит адрес следующей команды, которая должна быть выполнена после завершенной. После команд, не изменяющих содержимого регистра P, выполняется всегда следующая команда, т. е. регистр P автоматически получает приращение на длину выполненной команды.

Команда stmt_entry использует дополнительный регистр процессора логического вывода G1. Этот регистр инициализируется данной командой и после ее завершения содержит адрес в глобальном стеке, начиная с которого можно размещать молекулы. При этом предполагается, что при каждом размещении молекулы в глобальном стеке содержимое этого регистра увеличивается на 2 (размер молекулы).

Алгоритм команды allocate_env на уровне абстрактной машины не рассматривался, так как он полностью определяется структурами данных, используемыми для представления трассы логического вывода. Эта команда формирует административную часть локальной среды процедуры и фиксирует текущую среду в регистрах ПЛВ как ближайшую И-вершину дерева логического вывода. После чего в регистры L и G загружают адреса размещения в локальном и глобальном стеках новой текущей среды процедуры. При этом используется регистр G1, обеспечивающий размещение новой глобальной части среды после всех молекул, созданных при унификации.

Алгоритм команды return оптимизирован с учетом структуры стеков. Если текущая среда является точкой выбора (L=BL), выполняется формирование административной части локальной среды и фиксация текущей среды как ближайшей И-вершины. Если это условие не выполнено, то текущая локальная среда уничтожается, но соответствующая ей глобальная среда сохраняется. После чего управление передается по адресу в регистре CP, т. е. выполняется успешный возврат из процедуры.

Алгоритм команды fail, модифицированный с учетом размещения среды процедуры в двух стеках:

```
fail
if BL<L then do L<—BL
                G<—BG
                restore CP from cell(L+2)
                restore CL from cell(L+1)
                restore CG from cell(CL+0)
                restore BTR from cell(L+4)
              od
fi
undo_trail
TR<—BTR
for i: = 1 until Nmax do restore Ai from cell ((L+(5+i))
                      od
goto BP
```


Если выполнено условие L>BL, то это означает, что ближайшая точка возврата старше текущей среды. Следовательно, необходимо восстановить содержимое регистров из среды точки возврата. В противном случае в этом нет необходимости.

В алгоритме команды fail используется процедура undo_trail — обработка отметок адресов переменных, расположенных в стеке следа логического вывода, в диапазоне адресов этого стека [BTR, TR] по алгоритму, описанному для абстрактной машины (см. гл. 1). При этом выполняется соглашение о специальном способе представления пустой переменной (содержимое ячейки пустой переменной представляет свой собственный адрес). Предпоследняя операция, выполняемая командой fail, состоит в восстановлении содержимого регистров аргументов. В алгоритме указано восстановление всех существующих регистров аргументов (Nmax — максимально возможное число аргументов предиката и соответственно регистров аргументов процессора). Это не самый эффективный способ реализации этой операции — необходимо восстанавливать реальное число регистров аргументов. Но для этого необходимо иметь информацию о числе аргументов процедуры, вызванной в точке выбора, что требует несколько иной организации области кода программы, более сложной, чем было рассмотрено ранее. Детали структуры области кода, позволяющей реализовать команду fail более эффективно, рассмотрены в [50].

Команды вызова предиката

Так же как и в абстрактной машине логического вывода, в ПЛВ достаточно использовать только две команды вызова процедуры предикатов:

```
call            ENTRY
CP<—P+length(call)
goto P+ENTRY

last_call       ENTRY
if BL<CL then do L<—CL
                od
fi
restore CP from cell(CL+2)
restore CL from cell(CL+1)
restore CG from cell(CL+0)
goto P+ENTRY
```

В алгоритме используется тот факт, что вслед за командой call начинается последовательность команд вызова для следующего литерала тела утверждения, и адрес первой из этих команд можно легко вычислить. Этот адрес является точкой возврата после успешного завершения вызываемой процедуры.

Команда last_call освобождает часть локального стека (L<—CL), если ближайшая точка выбора старше ближайшей И-вер-

шины (BL<CL). Кроме того, непосредственно перед передачей управления на основную точку входа вызываемого предиката восстанавливаются все параметры ближайшей И-вершины, которая будет необходима после возврата из вызываемой процедуры.

Команды загрузки аргументов

Для процессора логического вывода можно использовать только четыре команды загрузки регистров аргументов, как и для абстрактной машины:

```
load_cons        Ai    CONST
Ai<—CONST

load_var         Ai    VAR
store Ai in cell(Ai)
Ai<—var_addr(VAR)

load_ref         Ai    VAR
Ai<—var_addr(VAR)

load_str         Ai    SKEL
Ai<— molecule(CG, SKEL)
```

Алгоритмы команд загрузки регистров аргументов соответствуют их алгоритмам команд абстрактной машины логического вывода. В алгоритмах этих команд используются две специальные процедуры-функции:

var_addr(VAR) — вычисление адреса ячейки переменной на основе параметра VAR, который может быть либо ⟨CL, OFFS⟩, либо ⟨CG, OFFS⟩;

molecule — создание молекулы с указанными параметрами и размещение ее в глобальном стеке по адресу в регистре G1; после размещения молекулы содержимое регистра G1 увеличивается на 2 (размер молекулы); затем создается ссылка на только что размещенную молекулу, ссылка и есть результат работы процедуры-функции molecule.

Команды индексации

Последовательность операций, выполняемая командами search_block абстрактной машины логического вывода, может быть реализована командами индексации:

```
if_const          Ai  CONST      ENTRY
 dereference(Ai)
if Ai=CONST then do goto P+ENTRY
                 od
fi_str            Ai  FUNCTOR    ENTRY
 dereference(Ai)
if flag (Ai) = Imolref then do decomp_molref (Ai, SS, SG)
                        if cell(SS) =FUNCTOR then do goto P+ENTRY
                                                 od
fi
if_empty          Ai             ENTRY
dereference(Ai)
if flag(Ai) =0 then do goto P+ENTRY
                    od
fi
goto                             ENTRY
goto P+ENTRY
```

Операнды ENTRY всех команд индексации содержат относительные адреса точек входа в блоки управления утверждениями. Последовательность команд search_block всегда начинается с команды if_empty и завершается командой fail или goto. Между ними может находиться произвольное число команд if_const или if_skel. Это означает, что если содержимое регистра Ai непусто и ни одна из команд if_const или if_skel не передала управление по адресу, определяемому третьим аргументом, то будет выполнена команда fail или goto.

Команда fail используется, если все утверждения предиката в позиции i не содержат переменной на нулевом уровне. Поэтому если содержимое регистра Ai непусто и ни одна из команд if_const или if_skel не обнаружила совпадения содержимого этого регистра со своим вторым операндом, то ни одно утверждение не может быть использовано для данного вызова. Но если есть утверждения с переменными в позиции i, то они могут быть использованы для входа в процедуру с помощью соответствующего блока управления утверждениями, адрес которого содержит команда goto. Команды if_const и if_skel должны охватывать весь диапазон констант и функторов в аргументной позиции i данного предиката.

В алгоритмах команд индексации используются три специальные процедуры:

flag(Ai) вычисляет флаг содержимого регистра Ai;

dereference(Ai) выполняет дереференсирование регистра Ai в соответствии с алгоритмом, подробно рассмотренным ранее;

decomp_molref выполняет декомпозицию ссылки на молекулу, вычисляя адрес скелетона молекулы SS и адрес соответствующего вектора глобальных переменных SG; адрес скелетона SS есть адрес функтора структуры (см. гл. 1), что и используется в алгоритме команды if_skel.

Команды унификации

Алгоритмы команд, унификации процессора логического вывода представляют собой конкретизацию алгоритмов команд абстрактной машины:

```
unify_const       Ai        CONST

dereference(Ai)
```

```
if flag(Ai)=0 then do store CONST in cell(Ai)
                      opt_trail(Ai)
                   od
fi
if flag(Ai)≠0 then do if Ai≠CONST do execute(fail)
                      fi od
                   od
fi
unify_var Ai VAR
dereference(Ai)
if flag(Ai)=0 then do store var addr(VAR) in cell(var_addr(VAR))
                      link_vars(Ai, var_addr(VAR), ADDR)
                      opt_trail(ADDR)
                   od
fi
store Ai in cell(var_addr(VAR))
unify_ref Ai VAR
ADDR<—var_addr(VAR)
general_unify(Ai, ADDR)
unify_str Ai SKEL
dereference(Ai)
if flag(Ai)=0 then do store molecule(G, SKEL) in cell(Ai)
                      opt_trail(Ai)
                   od
fi
if flag(Ai)=molref then do general_unify(Ai, molecule(G, SKEL))
                   od
fi
if flag(Ai) ≠ 0
& flag(Ai) ≠ molref then do execute(fail)
                   od
fi
```

Для выполнения унификации процессором логического вывода используются специальные процедуры:

opt_trail — алгоритм такой же, как и соответствующий алгоритм абстрактной машины логического вывода;

execute — автоматическое выполнение команды процессора логического вывода, не имеющей операндов;

link_vars используется при унификации двух пустых переменных, адреса которых заданы в первом и втором аргументах этой процедуры;

link_vars вычисляет больший из этих двух адресов и записывает в ячейку с этим адресом меньший адрес; больший адрес дублируется в третьем операнде для последующего помещения в стек следа, если это будет необходимо.

Алгоритм последней процедуры реализует правило ссылок, рассмотренное выше.

Основная работа по унификации сложных глубоко вложенных термов выполняется алгоритмом general_unify (TERM1, TERM2), операндами которого могут быть любые допустимые значения логических переменных. Алгоритм унификации является одной из основных операций, выполняемых процессором логического вывода.

ГЛАВА 3.
АППАРАТНО-ПРОГРАММНЫЕ СРЕДСТВА ЗАРУБЕЖНЫХ СПЕЦИАЛИЗИРОВАННЫХ ПРОЦЕССОРОВ

Особенностью аппаратной поддержки логического вывода является прямая реализация модифицированной системы команд Уоррена, выбор которой скорее априорный, чем формализованный. В рамках этого направления разрабатываются как специализированные системы — так называемые Пролог-машины, так и сопроцессоры для включения в состав ЭВМ общего назначения [17, 37, 39—42]. Причем аппаратно поддерживаются программные системы, в которых компилятор выполняет перевод исходной Пролог-программы в систему команд машины Уоррена. Последняя поддерживается либо микропрограммно, либо аппаратно.

В данной главе анализируются основные архитектуры распространенных Пролог-машин. Описываются аппаратные средства поддержки трассы логического вывода и основных механизмов Пролога: унификации, возврата, обработки структурированных данных и др.

3.1. ПЕРЕХОД ОТ АБСТРАКТНОЙ МАШИНЫ К КОНКРЕТНЫМ РЕАЛИЗАЦИЯМ

Существует не один вариант абстрактной машины логического вывода (см. § 1.1). В книге представлен вариант на основе машины Уоррена. Переход к тому или иному техническому проекту базируется на концепции абстрактной машины, отражая общую идеологию реализации логического вывода в современных прикладных системах (рис. 3.1). Прикладные задачи C_1, ..., C_n реализуются на языке Пролог, а затем компилируются в систему команд абстрактной машины логического вывода. Оттранслированный код на языке абстрактной машины поступает в специализированный процессор по тракту обработки вида

$C_i \rightarrow$ Абстрактная машина $\rightarrow$ Специализированный процессор

либо

$C_i \rightarrow$ Абстрактная машина $\rightarrow T_j \rightarrow \Pi_j$,

где T_j — транслятор с языка Пролог в язык машины j; Π_j — построенный исполняемый код для машины j. Последний тракт машинно-зависимый, что является существенным недостатком. Оттранслированный код может содержать интерпретируемую часть,

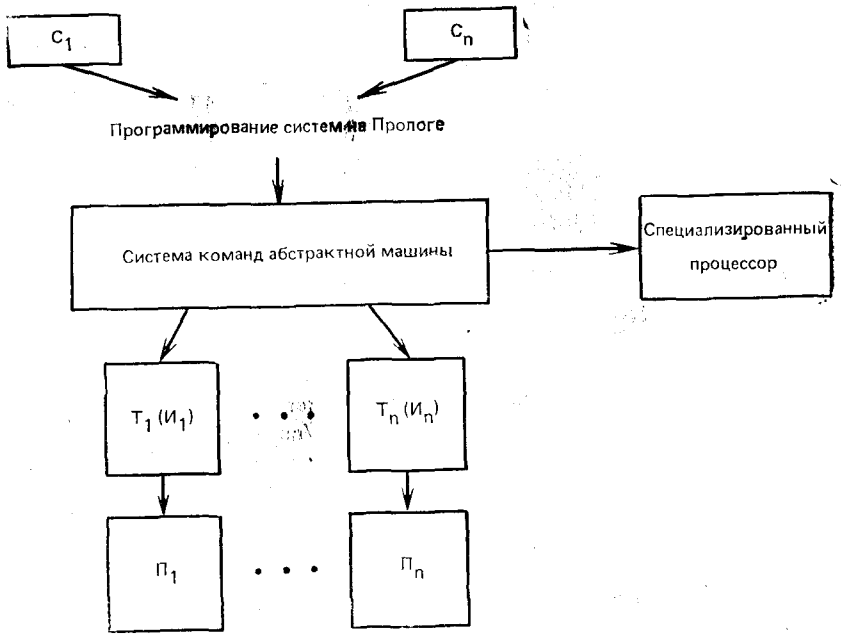


Рис 3.1. Концептуальная схема использования абстрактной машины

которая обрабатывается интерпретатором $И_i$ машины i с учетом ее архитектурных особенностей. При использовании специализированного процессора логического вывода (ПЛВ) в качестве сопроцессора основной ЭВМ также не устраняется архитектурная зависимость его от основной ЭВМ (например, форматы данных, системный интерфейс, обработка прерываний). Однако в целом архитектура специализированного процессора выполняется независимо от основной машины. Кроме того, программа на языке абстрактной машины обладает мобильностью, т. е. переносимостью на разные типы ЭВМ.

Таким образом, при проектировании ПЛВ необходимо решить следующие задачи:

1) реализовать устройства абстрактной машины на физическом уровне;

2) реализовать систему команд абстрактной машины на конкретной архитектуре;

3) обеспечить средства взаимодействия с основной ЭВМ при инициализации, прерываниях и завершении работы специализированного процессора;

4) разработать управление специализированным процессором для реализации им логического вывода.

Обратимся к рассмотрению зарубежного опыта решения этих и других задач, связанных с аппаратной реализацией машин логического вывода. Первые работы в данном направлении были выполнены в институте перспективных исследований (ICOT) по программе разработки ЭВМ пятого поколения [52, 53].

3.2. МАШИНА PSI

Персональная последовательная машина логического вывода PSI (Personal Sequential Inference Machine) [41, 44, 45, 52] представляет собой рабочую станцию (рис. 3.2), разработанную в рамках проекта ЭВМ пятого поколения в качестве средства создания программного обеспечения. Это первая реальная машина, ориентированная на язык Пролог.

В машине PSI использована теговая архитектура. Слово памяти 40-битовое — 8-битовая теговая часть и 32-битовая часть данных. В теговой части 2 бита отведены для сборки мусора, а 6 — для идентификации типов данных. Для выполнения программы используется четыре стека: локальный, глобальный, управляющий и трейловый. В локальном стеке хранятся локальные переменные, в глобальном — составные термы. Трейловый стек содержит адреса границ для возврата. Управляющая информация хранится в регистровом файле и управляющем стеке. В управляющем стеке хранятся 10-словные фреймы для выполнения программы или фиксации точек возврата. Четыре стека расположены в независимом адресном пространстве — в так называемой области кучи. Здесь размещаются перезаписываемые данные — команды и векторы.

В языке высокого уровня KLO, используемом в PSI-машине определены два типа переменных: локальные и глобальные. Гло-

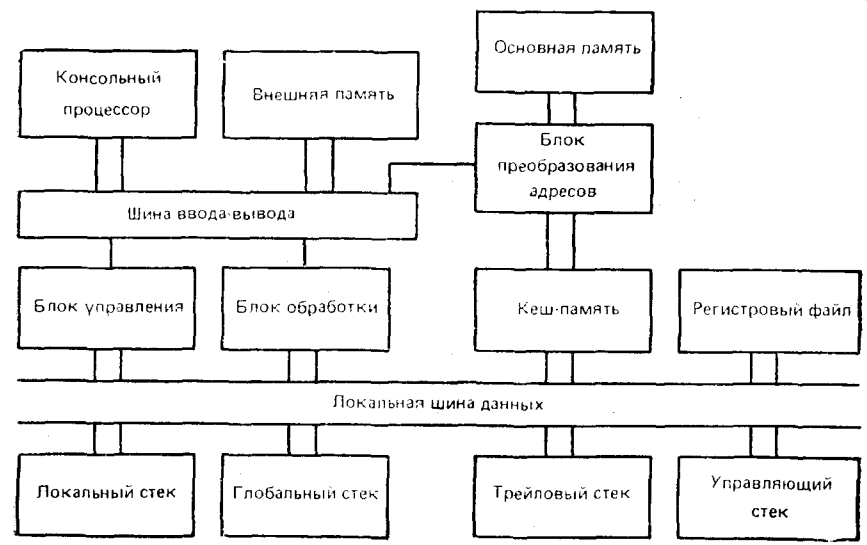


Рис. 3.2. Структурная схема машины PSI

бальные переменные являются элементами некоторого структурированного данного. Поскольку структурированные данные разделяются операторами, то переменные этого типа не могут удаляться до тех пор, пока не выполнен возврат при неудаче или сборке мусора. Это и стало причиной разделения локального и глобального стеков. Локальными переменными называются простые переменные.

При интерпретации KLO используются четыре независимых стека: управляющий, локальный, глобальный и трейловый. Управляющий стек сохраняет различную информацию для управления последовательностью выполнения операторов и восстановления среды при возврате. Локальный или глобальный стек используется только для динамического размещения переменных. Размер каждой переменной хранится в операторе, и выделение пространства для ячейки переменной в локальном или глобальном стеке осуществляется только при вызове этого оператора, который происходит из некоторого другого оператора. Трейловый стек используется для хранения тех адресов ячеек, значения в которых связываются с каким-то объектом. При возврате значения переменных переводятся в несвязанное состояние с использованием этих адресов.

Объектное представление Пролог-программ в машине имеет табличную форму, обеспечивающую удобство чтения и минимальное число обращений к памяти.

Операторы языка KLO транслируются во внутреннюю объектную форму (рис. 3.3) компилятором, написанным на том же языке KLO. Непосредственная интерпретация машинного языка высокого уровня микропрограммным интерпретатором (микроинтерпретатором) позволяет обеспечить высокую скорость выполнения программ.

Объектный код машины PSI состоит из заголовка процедуры и некоторого количества блоков операторов. В заголовке процедуры указывается количество ее аргументов к_арг и размер объектного кода. Каждый оператор представляется в виде заголовка оператора, *аргументов заголовка и совокупности целей тела. В заголовке* оператора указывается его тип и количество переменных в операторе: к_арг — число аргументов оператора, к_лок — число локальных переменных, к_глоб — число глобальных переменных. Каждый аргумент в заголовке оператора компилируется во внутреннее представление, размещаемое в одном слове. Каждой цели тела утверждения соответствует вызов предиката со своим набором аргументов и управляющим разделом ADM, хранящим в регистрах арг1, арг2, арг3 адреса точек входа соответственно в локальную, глобальную и трейловую среду, соответствующую текущему элементу трассы логического вывода. Если аргумент предиката константа (например, атом или целое), то он представляется непосредственным значением. Таким образом, каждый оператор KLO непосредственно транслируется в последовательность слов, которая включает всю информацию об этом операторе.

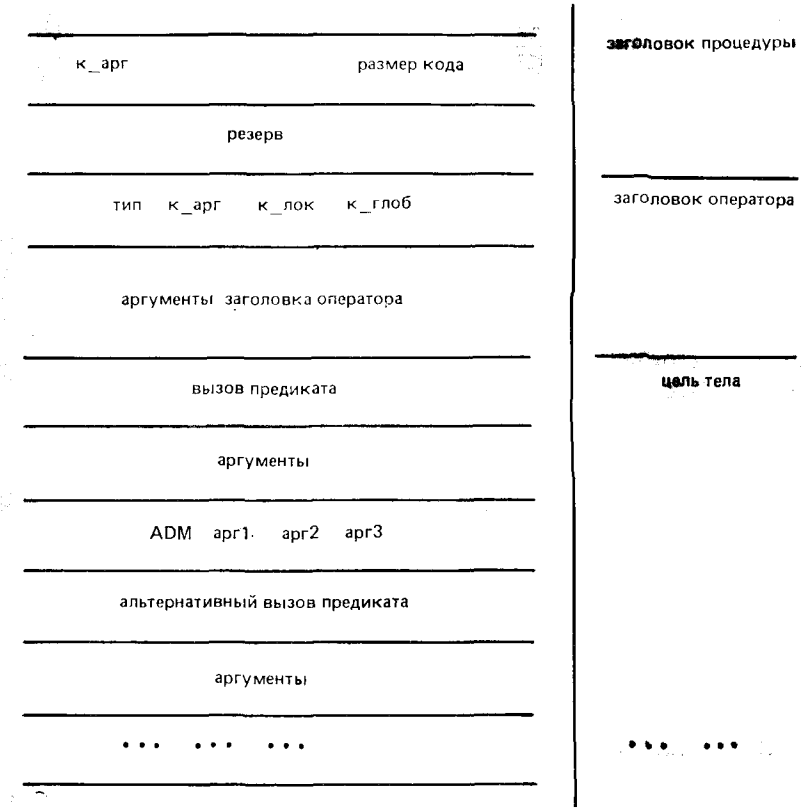


Рис. 3.3. Внутреннее представление процедуры в машине PSI

Объектный код заносится в кучу. Его выполнение осуществляется с помощью интерпретатора путем сравнения аргументов запроса с соответствующими аргументами оператора процедуры. При интерпретации базовые адреса фреймов стека должны храниться в рабочих регистрах. Это прежде всего база фрейма локального стека, база фрейма глобального стека, адрес объектного кода и т. д.

На рис. 3.4 показано аппаратное окружение АЛУ обрабатывающего блока PSI. В состав регистрового файла входят регистры общего назначения и аппаратный буфер для буферизации одного фрейма вершины стека. Помимо этого имеются схема проверки тегов и регистры адреса (AR) и данных (DR) для доступа к средам вызываемого и вызывающего предикатов. Схема проверки тега выделяет поля тегов унифицированных аргументов и вырабатывает признаки, проверяемые в блоке управления PSI. АЛУ осуществляет сравнение аргументов и вычисление адресов переменных. Схема расширения используется для подстановки требуемых байтов в слово данных, сдвинутое на сдвиговом регистре.

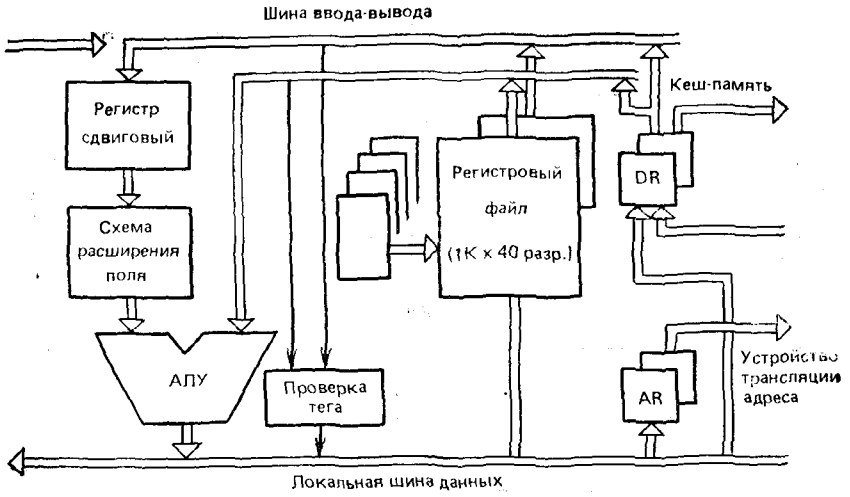


Рис. 3.4. Структурная схема блока обработки машины PSI

Микроинтерпретатор, в состав которого входит и сборщик мусора, содержит порядка 15 000 микрокоманд. Помимо ядра, предназначенного для интерпретации языка программирования, он включает микропрограммы около 160 встроенных предикатов и микропрограммы интерфейса с операционной системой, которые предназначены, например, для обработки прерываний, переключения процессов, вычисления виртуальных адресов.

В машине PSI обеспечивается поддержка до 63 параллельных процессов. Для этого 32-битовое пространство памяти делится на 256 независимых сегментов, каждый из которых используется в качестве области стека. Благодаря этому обеспечивается независимость каждого стека, а логически они могут расширяться до 16 Мслов.

PSI-машина поддерживает высокоскоростную обработку, используя кеш-память (8 Мслов) для быстрого доступа к стекам и области кучи, регистровый файл большой емкости для оптимизации интерпретации, теги для разветвления вычислительного процесса, а также мультипрограммное параллельное управление на базе 64-разрядной микрокоманды с горизонтальной организацией. Такт микрокоманд 200 нс. Скорость выполнения около 30 КЛисп.

Данная архитектура была исследована и сравнена с компилятором. Компилятор быстрее, если программа лучше компилируется, а интерпретатор лучше, если прикладная программа требует больше процессорной обработки. Коэффициент использования кеш-памяти 96%. Соотношение операций чтения-записи для оперативной памяти 3:1. Каждая пятая микрокоманда требует доступа к памяти, причем к области кучи 30—50% от числа всех обращений памяти. К регистровому файлу обращалось 90% всех

команд, но его объем 1 Кслов может быть сокращен. Эффективно показало себя тегирование данных для организации ветвлений в программе.

3.3. МАШИНА PEK

Машина PEK (Prolog Engine of Kobe Univ) [32], разработанная в университете г. Кобе (Япония), — машина с горизонтальным микропрограммированием, предназначенная для высокоскоростного выполнения Пролог-программ (рис. 3.5).

В качестве HOST-процессора используется микропроцессор MC68000, работающий под управлением операционной системы CP/M-68K. На HOST-процессор возлагается поддержка операционной системы CP/M-68K, организация диалога с пользователем, ввод-вывод, компилирование программ, загрузка микропрограмм в память. PEK-процессор является управляемым сопроцессором микропроцессора MC68000. Оба процессора взаимодействуют через общие три регистра и общую память. Пролог-интерпретатор, работающий на PEK-процессоре, обеспечивает быстродействие 60—70 КЛипс при выполнении Пролог-программы.

PEK-процессор собран на пяти печатных платах (300 корпусов 450×250). В PEK-процессоре (рис. 3.6) секвенсор, построенный на

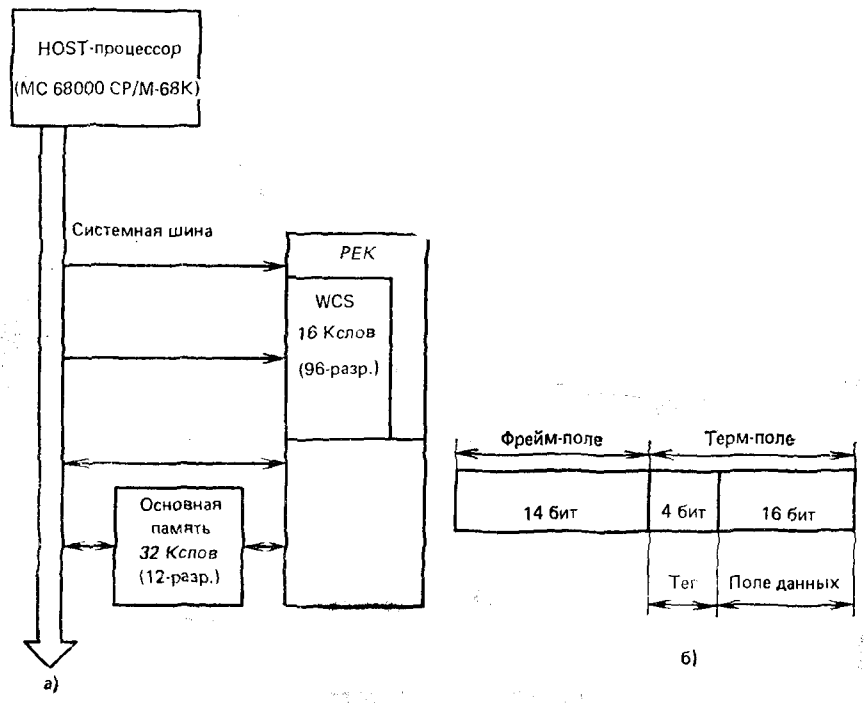


Рис. 3.5. Структурная схема (*а*) и формат слова (*б*) PEK-процессора

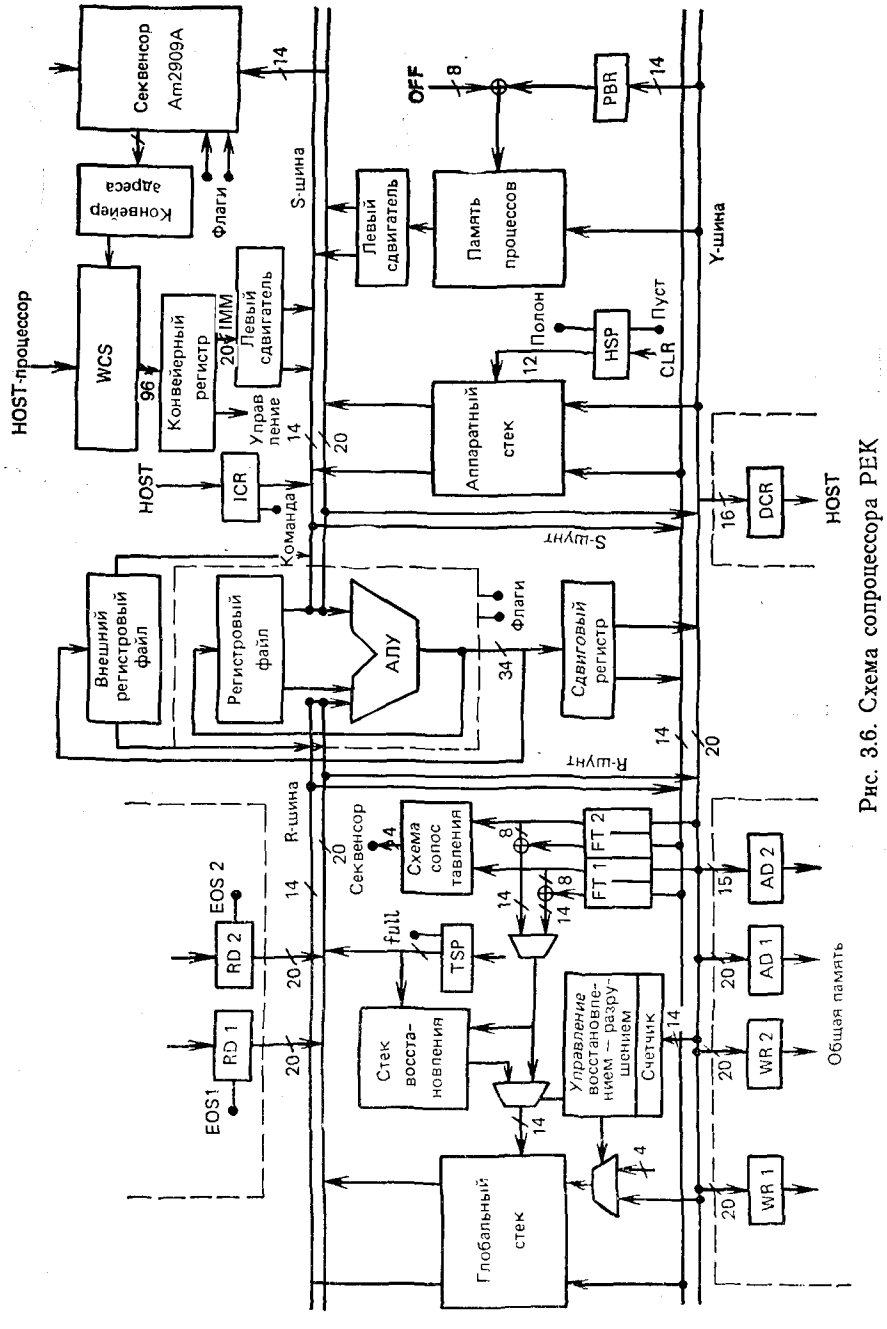


Рис. 3.6. Схема сопроцессора РЕК

процессорных секциях Am2909A, формирует 14-разрядный адрес микропрограммы. Синхрогенератор Am2925 (не показан на рисунке) генерирует четырехфазные импульсы. Время цикла $120+40i$ ($i=0, 1, ..., 7$) зависит от конкретной микрокоманды. Выполнение текущей микрокоманды и выборки следующей осуществляется с перекрытием по времени (одноуровневый конвейер).

В РЕК-процессоре применяется горизонтальное микропрограммирование. Длина микрокоманды составляет 96 бит и содержит 24 поля. Формат микрокоманды приведен на рис. 3.7, а назначение полей в табл. 3.1.

Для организации переходов в микропрограмме используются содержимое микростека секвенсора, шина данных S (содержимое регистров АЛУ, памяти процессоров или аппаратного стека); непосредственные данные поля микрокоманды. Всего в микрокоманде допускается до 36 условий перехода.

Данные в РЕК-процессоре представляются 34-битовыми словами (рис. 3.5,*б*) — 14-битового фрейм-поля и 20-битового терм-поля. В свою очередь, терм-поле содержит 4-битовую теговую часть и 16-битовое поле данных. Тег определяет тип данных, поле данных интерпретируется в зависимости от тега.

Стеки и составные термы запоминаются в последовательных ячейках памяти. Конец списка или составного терма идентифицируется словом, тег которого равен EOS (End Of Structure). Если поле тега соответствует переменной или структуре с переменными, то поле фрейма содержит адрес фрейма переменных. Фрейм переменных запоминается в области стека глобальных переменных и содержит значения переменных. Например, терму f(a, b) соответствует скелетон f(x, y) и фрейм переменных x=a, y=b (рис. 3.8).

Шины данных в РЕК-процессоре также 34-разрядные. Шины R и S — шины источника, а Y — шина назначения. Данные, передаваемые по шинам R и S, обычно обрабатываются в АЛУ, а ре-

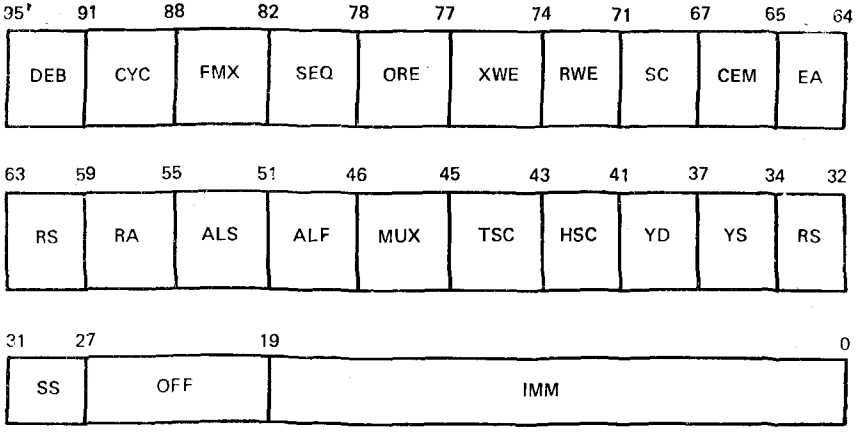


Рис. 3.7. Формат микрокоманды процессора РЕК

Таблица 3.1

Имя поля	Назначение
DEB	Отладка
CYC	Управление циклом
FMX	Управление мультиплексором флагов
SEQ	Управление секвенсором
ORE	Наличие альтернативы
XWE	Наличие внешнего регистра записи
RWE	Наличие регистра записи
SC(CC)	Управление сдвигом (переносом)
CEM	Наличие условия
EA	Наличие операнда, прочитанного из памяти
RA, RB	Регистры A, B
ALS	Операция сдвига в АЛУ
ALF	Функция АЛУ
MUX	Переключение источника адреса
TSC	Управление стеком восстановления
HSC	Управление аппаратным стеком
YD	Приемник С шины Y
YS	Источник для шины Y
RS	Источник для шины R
SS	Источник для шины S
OFF	Смещение относительно содержимого регистра базы процесса
IMM	Данные

зультат записывается в память или регистр через шину Y. Для обеспечения высокоскоростной передачи данных между шинами используются спецнальные шунтирующие цепи.

Память РЕК-процессора состоит из совокупности модулей, каждый из которых имеет собственное функциональное назначение:

WCS (перезаписываемая память микропрограммы) — для хранения микропрограмм; чтение из памяти и запись в память осуществляются HOST-процессором;

общая память (32K×20 разр.) — для хранения Пролог-программы, головы атомов и структуры; используется как HOST-процессором, так и Пролог-процессором;

память процессора (16K×20 разр.) — для хранения управляющей информации, полученной при выполнении Пролог-программы; для каждого вызова предиката создается фрейм управления;

стек глобальных переменных (16K×34 разр.) — для хранения глобальных или локальных переменных;

стек восстановления (4K×34 разр.) — для запоминания адресов переменных, которые должны стать свободными при возврате;

аппаратный стек (4K×34 разр.) — для унификации термов;

регистровый файл (32K×34 разр.).

Механизм выполнения Пролог-программы сложнее, чем при традиционных языках программирования: при выполнении Про-

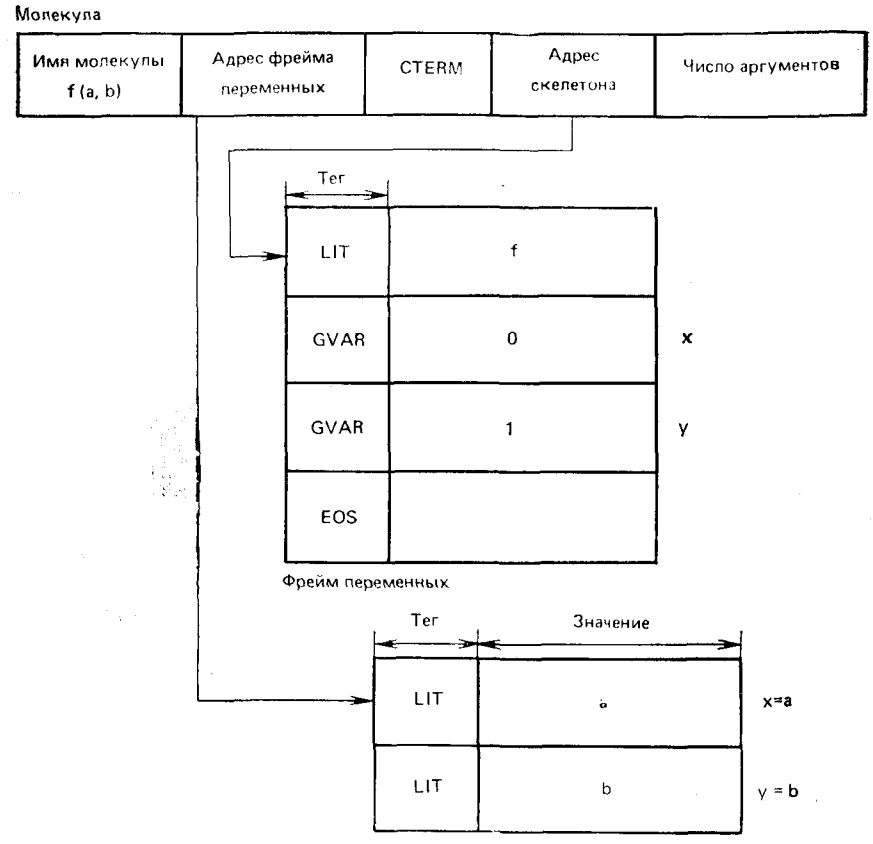


Рис. 3.8. Формат молекулы процессора РЕК

лог-программ управляющая информация сохраняется, так как это необходимо для перезапуска программ при возврате.

Управляющая информация в виде фрейма хранится в памяти процессоров. Каждый управляющий фрейм содержит следующую информацию: указатели на выполняемый и альтернативный операторы, адреса глобального и локального фреймов, указатель в трейловом стеке на начало выполняемой последовательности, количество глобальных и локальных переменных в операторе. Адрес внутри фрейма определяется суммой содержимого базового регистра PBR и поля смещения из микрокоманды. Фрейм переключается изменением содержимого регистра PBR. Если адрес управляющего фрейма загружен в PBR, то доступ к нему осуществляется за одну микрокоманду. Успешный или неуспешный возврат из выполняемой последовательности, включающий перезагрузку регистра PBR, может быть выполнен только за две микрокоманды.

Для унификации двух термов в РЕК-процессоре предусмотрены два множества регистров AD1, RD1, WR1 и AD2, RD2, WR2 (адреса, чтения и записи). Для обеспечения конвейерного чтения структурированные термы запоминаются в последовательных ячейках памяти. Одновременно с процессом чтения данных из регистра чтения RD автоматически выполняется приращение содержимого регистра адреса AD и начинается операция чтения для следующего адреса данных. Период появления данных в результате чтения 250 нс. При чтении в регистр данных адрес глобальной переменной формируется автоматически сложением поля адреса фрейма и смещения переменной относительно начала фрейма. Быстрое выполнение алгоритма унификации обеспечивается эффективными средствами сравнения двух слов. В зависимости от результата сравнения на схеме сопоставления осуществляется переход по одному из 16 возможных адресов. Выбор адреса выполняется в соответствии с полями тегов регистров FT1 и FT2 и однобитовым полем, представляющим результат сравнения полей данных унифицированных термов. Адрес переменной вычисляется автоматически по содержимому регистров FT1 или FT2. Адрес в стеке глобальных переменных определяется как сумма поля фрейма FT1 (или FT2) и индекса переменной. Выбор FT1 или FT2 осуществляется по содержимому поля MUX соответствующей микрокоманды. По флагам UNDEF1 или UNDEF2 можно проверить, связана переменная или нет, без проверки ее значения в АЛУ.

Аппаратная поддержка возврата в РЕК-процессоре обеспечивается цепью автоматического восстановления и цепью автоматического разрушения. Операция разрушения образует значения переменных «не определено». Для ее выполнения адреса связанных переменных должны быть размещены в стеке восстановления. Это делается автоматически, т. е. операция связывания глобальной переменной сопровождается запоминанием ее адреса в стеке восстановления. Операция разрушения выполняется дополнительным секвенсором, работающим параллельно с основным. Дополнительный секвенсор запускается операцией записи количества операций разрушения в счетчик разрушений. Конец выполнения операции разрушения может быть установлен с помощью специального флага.

3.4. МАШИНА PLM

Разработанная в Калифорнийском университете для системы Aquarius PLM (Programmed Logic Machine) [35, 43] предназначена для высококачественного исполнения программ, написанных на языке Пролог, для работы с множеством потоков данных и команд. Язык управления PLM является промежуточной формой языка Пролог и компилируется перед исполнением с этого или другого языка программирования.

В системе Aquarius (рис. 3.9) ЭВМ NCR/32, являющаяся базовой (HOST-машиной), обеспечивает работу с памятью, ввод-вывод и организацию вычислений.

Блок PMI (PLM Memory Interface Unit) выполняет операции обмена с Пролог-процессором PLM, организует буферизацию ввода-вывода.

Блок обмена PMI организует интерфейс между системной шиной ЭВМ NCR/32 и PLM, поддерживает логику протокола шины и несколько модулей, обеспечивающих доступ к памяти и буферизацию из области данных и области программ, а также логику приоритета доступа. Необходимость буферизации доступа к области данных обусловлена разницей циклов PLM (100 нс) и NCR (150 нс).

Первичным буфером является буфер записи. Все операции записи, инициализируемые PLM, ставятся в очередь в буфере записи в ожидании разрешения доступа к памяти. Это позволяет процессору продолжить исполнение, так как эффективное время записи равно одному циклу. Запрос считывания из памяти от PLM также буферизируется в ожидании доступа к PM-шине. В этом случае PLM должна ждать окончания обслуживания текущего запроса. Обслуживание запроса на считывание откладывается до завершения обработки запросов на запись. Реализована простая схема сохранения последовательности данных, хранящихся в памяти.

Кроме буферов считывания и записи PMI содержит специальную кеш-память точки выбора (CP). В кеш-память записывается только текущая точка выбора. Управление кеш-памятью производится с помощью парадигмы сквозной записи и становится недействительным каждый раз при уничтожении точки выбора. Восстановление точки выбора имеет место только как побочный эффект последующего доступа к ней из памяти во время следующей операции fail. Это экономит время на восстановление кеш-памяти. Время доступа процессора к кеш-памяти составляет один цикл.

Кроме буферизации области данных, PMI обеспечивает буферизацию области программ и частичную дешифровку команд с помощью блока предварительной выборки команд, содержащего небольшой буфер команд (до четырех команд). Поскольку длина команд PLM переменная, блок предварительной выборки распознает все изменения направления управляющего потока по мере их выборки. Благодаря простоте этих команд блок предварительной выборки может продолжать выборку через ветви управляющего потока. Только в специальных случаях блок предварительной выборки продолжает выборку, пока блок обработки не возьмет на себя управление потоком. Еще одним источником изменения направления управляющего потока является операция fail. Она заставляет блок предварительной выборки заполнить буфер команд и начать выборку с точки возврата к предыдущему состоянию.

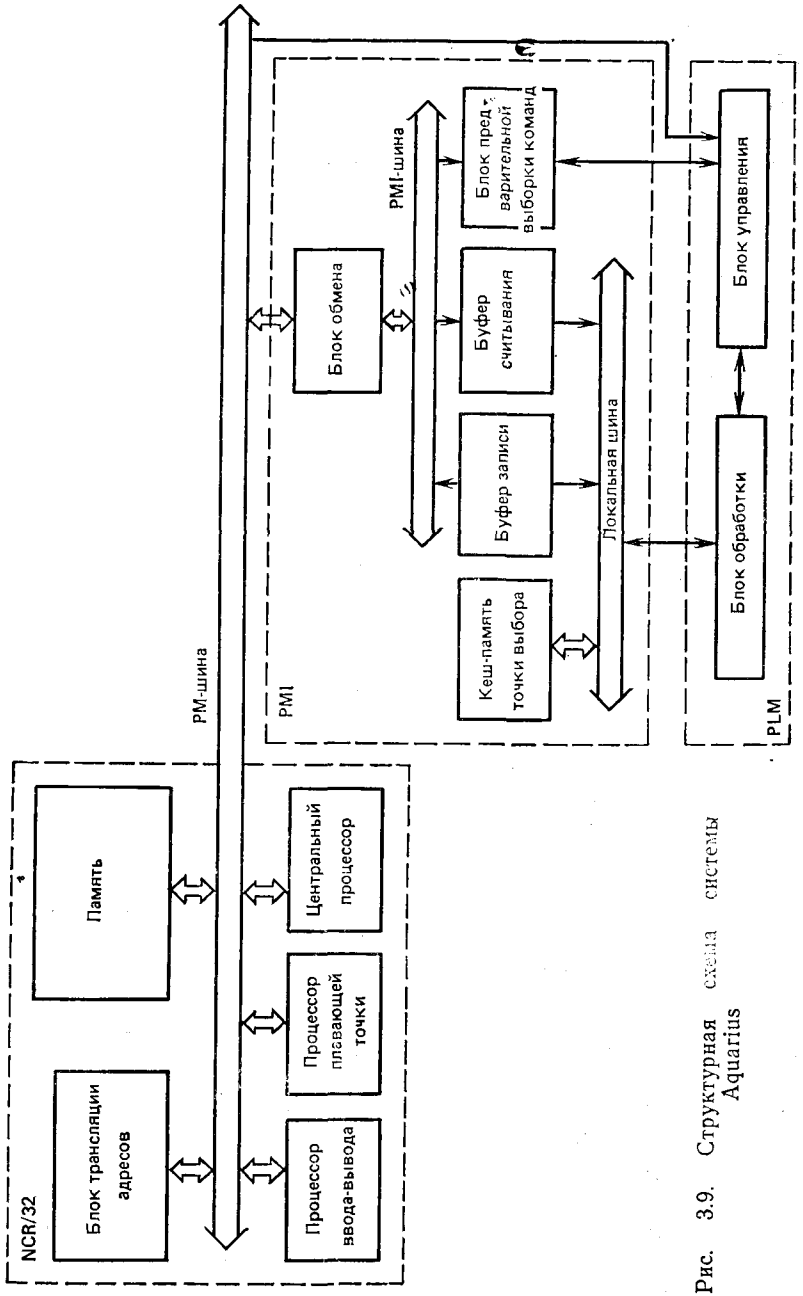


Рис. 3.9. Структурная схема системы Aquarius

Работу различных буферов и блоков PMI координирует блок управления с помощью схемы приоритета, управляющей доступом к PMI. Выборка команд имеет самый низший приоритет. Если буфер команд очищается, то приоритет операции выборки будет наивысшим. Таким образом, предварительная выборка осуществляется, когда для исполнения имеется хотя бы одна команда в буфере команд. Запросы на считывание из памяти имеют низкий приоритет, запросы на запись в память — более высокий приоритет. Это гарантирует задержку запросов на считывание до выполнения записи в память.

Архитектура машины PLM основана на абстрактной модели для исполнения программ на языке Пролог и набора команд (ISA), описанных Уорреном [49] и дополненных Добри, Пэтом и Деспейном [35]. Согласно этой модели адресное пространство PLM делится на область программы и область данных. Область программы представляет собой псевдостатическую область (псевдостатическая из-за утверждения и отмены предикатов Пролога), содержит программу на языке Пролог и другую информацию, используемую для описания атомов (подобно списку свойств на языке Лисп). Элементы в области программы имеют переменную длину, а адресация производится на байтовом уровне. К области программы имеют доступ блок предварительной выборки команд и некоторые встроенные функции.

В области данных содержатся 32-разрядные объекты данных и информация о состоянии программы на языке Пролог. Адресация производится как к 32-разрядным словам. Под теги отводится от 4 до 6 разрядов в зависимости от объекта данных. Процессор с помощью первых двух разрядов тегов может распознавать до четырех типов данных: ссылки, представляющей переменные языка Пролог, как ограниченной, так и неограниченной; константу, представляющую различные типы констант; две формы составных данных — список и структуру (рис. 3.10). Кроме того, в поле тега имеется разряд cdr для поддержки представления составных данных и разряд для сборки мусора c. Встроенные разряды тега служат для дальнейшего уточнения типов данных. Однако обычно они не распознаются набором функций и используются только встроенными функциями.

Область данных делится на три основные области: неупорядоченный массив (куча), стек и трейловую. В неупорядоченном массиве содержатся все элементы составных данных, сгенерированные программой, а также, возможно, и глобальные переменные (так же как и область для «побочных» переменных, используемых некоторыми встроенными функциями). Он организуется по принципу LIFO (последним пришел — первым обслужен) и освобождается только при возврате к предыдущему состоянию. Эта область участвует в сборке мусора.

Стек содержит фрейм среды и фрейм точки выбора. Среда для конкретного правила содержит информацию о состоянии выполнения правила и все переменные (не хранящиеся резидентно

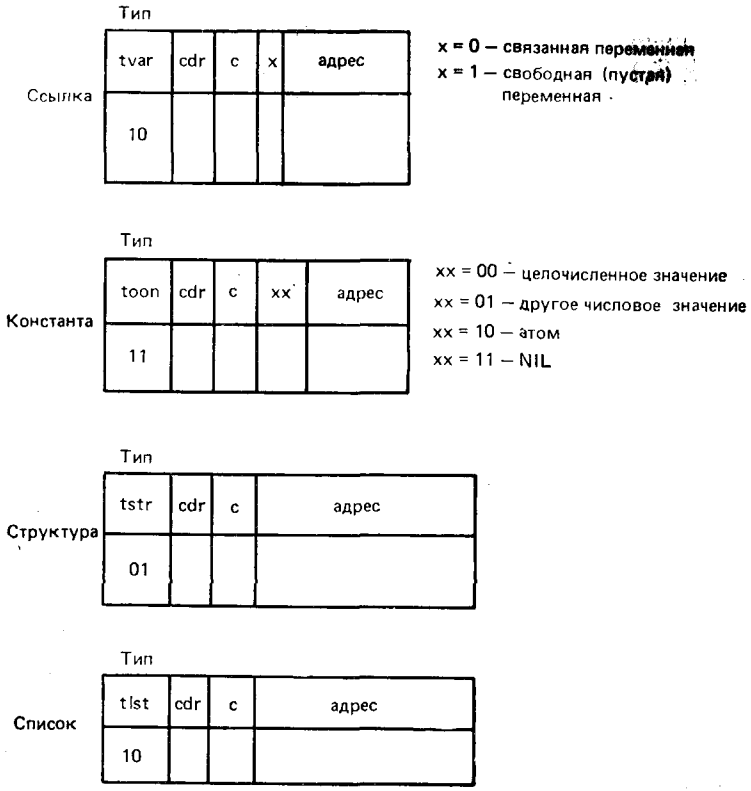


Рис. 3.10. Структура данных процессора PLM

в регистрах), используемые в правиле. В точке выбора содержится информация, используемая для восстановления состояния ПЛВ в случае возврата к предыдущему состоянию. Среды и точки выбора создаются и удаляются соответствующими командами. Стек используется по принципу LIFO и не участвует в сборе ненужных данных. Трейловая область содержит указатели переменных в стеке и неупорядоченном массиве, значения которых должны быть отменены во время возврата к предыдущему состоянию. Эта область работает также по принципу LIFO. Кроме указанных выше областей имеется небольшая область PDL, которая используется во время унификации вложенных списков и структур.

Блок управления состоит из запоминающего устройства, в котором содержится микрокод, и схемы управления последовательностью выполнения микрокоманд, предназначен прежде всего для управления блоком обработки PMI. Имеется также возможность ввода управляющих команд из NCR/32, чтобы центральный процессор мог работать в качестве «передней панели» относительно PLM при отладке и тестировании.

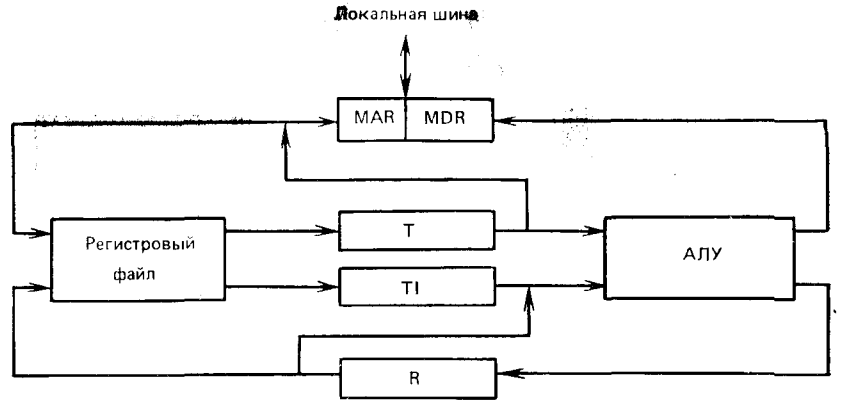


Рис. 3.11. Структурная схема блока обработки процессора PLM

Блок обработки состоит из двух основных блоков (рис. 3.11): регистрового файла и арифметическо-логического устройства (АЛУ).

Регистровый файл содержит следующие регистры:

P — указатель программы, содержит адрес следующей исполняемой команды;

CP — указатель продолжения, содержит адрес команды, которая должна быть выполнена после успешного завершения текущего правила;

E — регистр среды, содержит адрес области данных, образующих текущий фрейм среды в стеке;

N — размер среды — размер последнего фрейма среды в стеке;

B — регистр возврата к предыдущему состоянию, содержит адрес области данных — активного фрейма точки выбора в стеке; используется для возврата к предыдущему состоянию;

H — указатель неупорядоченного массива, содержит адрес области данных, указывающий на текущую верхушку неупорядоченного массива;

HB — указатель возврата к предыдущему состоянию неупорядоченного массива, содержит адрес области данных, указывающий на верхушку неупорядоченного массива в последней точке возврата к предыдущему состоянию; используется для восстановления неупорядоченного массива после возврата к предыдущему состоянию;

S — указатель структуры, представляет адрес в памяти для размещения следующего элемента списка или структуры;

TR — указатель трейловой области, содержит адрес, указывающий на текущую верхушку трейловой области.

Команды PLM выполняются по трехступенчатой схеме:

С-стадия (доступ к регистрам) — выборка данных из регистрового файла и запись их в регистры T и TI;

E-стадия (операция АЛУ) — содержимое регистров T и TI используется в качестве данных для операций в АЛУ; результаты заносятся в регистры R и MDR, могут также производиться операции в памяти;

P-стадия (занесение результатов в регистр) — запись содержимого регистров R и MDR в регистровый файл.

Максимальный параллелизм реализации стадий внутри каждой команды обеспечивает микрокоманда, управляющая блоком обработки. Типичная микропрограмма может вызвать одну операцию С-стадии, несколько операций E-стадии и затем операцию P-стадии.

Доступ к памяти через регистры MDR (регистр данных памяти) и MAR (регистр адреса памяти) осуществляется параллельно.

3.5. МАШИНА CHI

Машина CHI (Cooperative High-speed Inference) имеет шесть областей данных для хранения информации при выполнении Пролог-программ: системную, область кодов, кучу, глобальный, локальный и трейловый стеки. *Системная область* содержит информацию, необходимую для управления памятью и процессами, используется также в качестве рабочей для микропрограмм. *Область кодов* содержит скомпилированные коды Пролог-программ. В *куче* хранятся данные, созданные при выполнении Пролог-программы: факты базы данных, буферы файлов, графические драйверы и др. *Глобальный стек* содержит структуры и списки, созданные в процессе унификации прямым копированием, *локальный стек* — фреймы переменных и управляющие фреймы, а *трейловый* — ссылки на переменные, которые подлежат «развязыванию» (unbinding) при выполнении возврата.

Чтобы разделить вызов процедуры от управления возвратом, локальный стек содержит три типа фреймов: среды, указателя альтернативы и ловушки. В *фрейме среды* хранятся значения переменных и информация, необходимая для продолжения выполнения программы, в *фреймах указателя альтернативы* и *ловушки* — адреса альтернативных операторов и информация о состоянии вычислений, необходимая для возврата, но последний используется для обработки исключительных ситуаций.

Для передачи аргументов используются регистры, что обеспечивает повышение эффективности в случае выполнения рекурсии. В момент вызова оператора в регистры загружаются аргументы вызывающего предиката. В эти же регистры между унификациями загружаются также результаты каждой последующей унификации. Поэтому для рекурсивного вызова достаточно выполнения команды перехода. Следовательно, отпадает необходимость в стековом пространстве: операции выполняются в регистрах аргументов.

Машина CHI является машиной языка высокого уровня для одного пользователя. Она содержит CHI-процессор, HOST-процессор PSI и систему памяти емкостью 64 Мслова. Процессор PSI управляет устройствами ввода-вывода, интерфейсом с CHI-процессором и другими внешними системными интерфейсами, в том числе межмашинным интерфейсом и локальным сетевым интерфейсом. В этой конфигурации большая Пролог-программа загружается от процессора PSI и выполняется в CHI. Процессор CHI, состоящий из Пролог-процессора, кеш-памяти и интерфейса с HOST-процессором, непосредственно интерпретирует внутренние объектные данные и коды, используя свои микропрограммы и аппаратуру. Внутренний объектный код и данные хранятся в основной памяти (36-разрядной). Распараллеливание выполнения операций в процессоре машины CHI (рис. 3.12) реализовано путем разделения работы с базовыми регистрами для управления стеками и работы с данными для унификации по отдельным блокам. Для доступа к аргументам вызывающего и вызываемого предикатов предусмотрены два регистра адреса и данных, а для высокоскоростной коммутации управляющей информации о выполнении программ — два набора управляющих базовых регистров. Чтобы обеспечить высокую производительность машины, в процессоре CHI могут параллельно работать несколько функциональных модулей, осуществляющих выборку, декодирование команды, согласование аргументов между целью и заголовком оператора Пролога, манипуляции стековым указателем, арифметические операции с плавающей и фиксированной точкой. Эти функциональ-

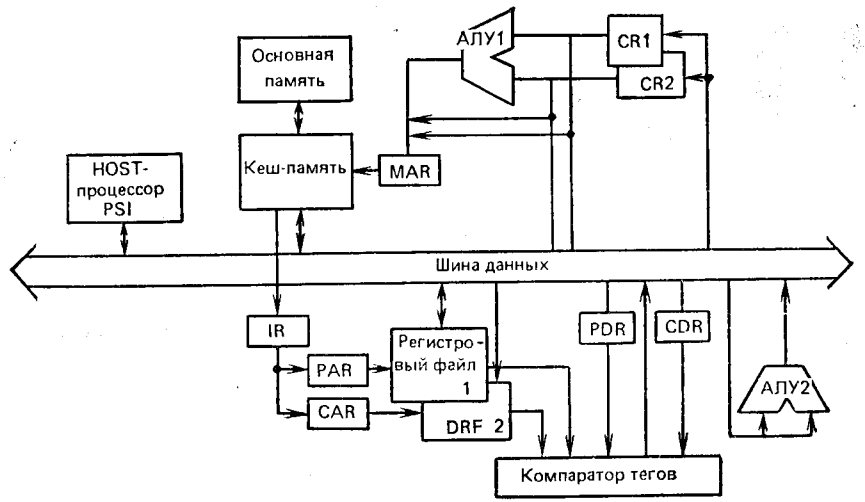


Рис. 3.12. Структурная схема процессора CHI

ные модули подключаются к 36-разрядной шине данных, так же как кеш-память и интерфейс HOST-процессора.

Пролог-процессор выполняет команды последовательно (в режиме конвейера) в три стадии: выборка команды, декодирование и выполнение. На стадии выборки команды регистр команд IR принимает однословную команду из кеш-памяти. На стадии декодирования команда дешифрируется и результаты фиксируются в регистрах PAR и CAR. Эти регистры указывают на адреса соответствующих регистров в двухпортовых 32-разрядных регистровых файлах DRF. Каждый регистровый файл содержит регистры аргументов и рабочие регистры, необходимые для выполнения команд. На стадии выполнения модуль согласования аргументов и модуль управления стековым указателем, в основном, используются параллельно. Сравнение данных осуществляется компаратором. Двойной доступ по чтению-записи к управляющему файлу возможен благодаря делению на два файла. Это деление позволяет сравнивать и модифицировать различные указатели в управляющих регистрах одновременно.

Архитектура CHI требует большого числа управляющих регистров. Из них 48 регистров назначаются стековыми указателями, а другие регистры используются в качестве рабочих.

Для адресации кеш-памяти используется адресный регистр MAR, содержимое которого либо формируется из указателей в регистровом файле, либо вычисляется как некоторое преобразование аргументов в регистрах CR1 и CR2 на АЛУ1. Регистры PDR и CDR используются для хранения унифицированных аргументов предикатов, выбираемых из кеш-памяти. При этом в регистр CDR заносится аргумент вызывающей, а в регистр PDR — аргумент вызываемой процедуры.

Быстродействие машины CHI оценивается в 280 КЛипс при выполнении детерминированной Пролог-программы.

3.6. КОНВЕЙЕРНЫЙ ПРОЛОГ-ПРОЦЕССОР

Архитектура конвейерного Пролог-процессора машины TICK-WARREN [33] очень близка к архитектуре абстрактной машины Уоррена, положенной в основу Пролог-системы ЭВМ DEC-10. Однако при построении термов сложной структуры вместо разделения структур используется их копирование. Разделение структур применяется только для представления целей-резольвент; точки ветвления отделяются от среды унификации (фреймов стека локальных переменных) и становятся необязательными; среда унификации модифицируется в процессе выполнения клоза за счет удаления термов, не нужных для последующего вычисления переменных.

В конвейерном Пролог-процессоре каждый терм представляется словом, 32 бита которого отведено по значению и 8 бит — под тег. Значение в большинстве случаев представляет собой адрес, обеспечивающий доступ к большому адресному пространству. Тег определяет тип терма: ссылки (соответствующие связанным и несвязанным переменным), структуры, списки, константы (включая атомы и целые). Структуры и списки создаются точным копированием функтора и аргументов в последовательные слова памяти.

Основными областями данных являются область кодов, содержащая команды и другие данные, представляющие собственно программу, и три стека: локальный, глобальный (куча) и трейловый. Существует также небольшой обратный магазинный список PDL, используемый при унификации. Стеки заполняются при каждом процедурном вызове и освобождаются при возвратах. Куча содержит все структуры и списки, созданные унификацией и процедурными вызовами. Трейловый стек содержит ссылки на переменные, которые были связаны при унификации и должны быть развязаны при возврате. Локальный стек содержит два типа объектов: среды и точки выбора. Среда состоит из вектора значений переменных тела некоторого оператора и указателя на тело другого оператора с соответствующей ему средой. Точка выбора содержит всю информацию, необходимую для восстановления более раннего состояния вычислений при возврате; создается тогда, и только тогда, когда вызываемая процедура имеет более одного оператора, потенциально согласуемого с вызовом. Сохраняемая информация является указателем на альтернативные операторы и содержимое регистров, загружаемых в момент вызова процедуры: A, P, E, H, TR, B, CR, S, A1 ... AM (где M — количество аргументов в процедуре).

Текущее состояние логического вывода определяется содержимым определенных регистров — указателями в основные области данных:

P — программный указатель в область кода,
CP — текущий программный указатель в область кода,
E — последняя среда в локальном стеке,
B — последняя точка выбора (точка возврата) в локальном стеке,
A — вершина стека,
TR — указатель верхушки трейлового стека,
H — вершина кучи,
HB — точка возврата в куче,
S — указатель структуры в куче,
A1, A2, ... — регистры аргументов.

Описываемая архитектура представляет собой однопользовательский одноконвейерный Пролог-процессор, который состоит из блока команд и блока обработки. Организация памяти характеризуется расслоением с четырьмя циклами доступа. Поэтому обращения к ней могут осуществляться параллельно, что делает время доступа к памяти некритичным. Команда на машинном языке, выбранная блоком команд, преобразуется в последовательность микрокоманд, которые выполняются блоком обработки.

Блок обработки (рис. 3.13) конвейерного процессора представляет собой трехступенчатый конвейер. На стадии C происходит

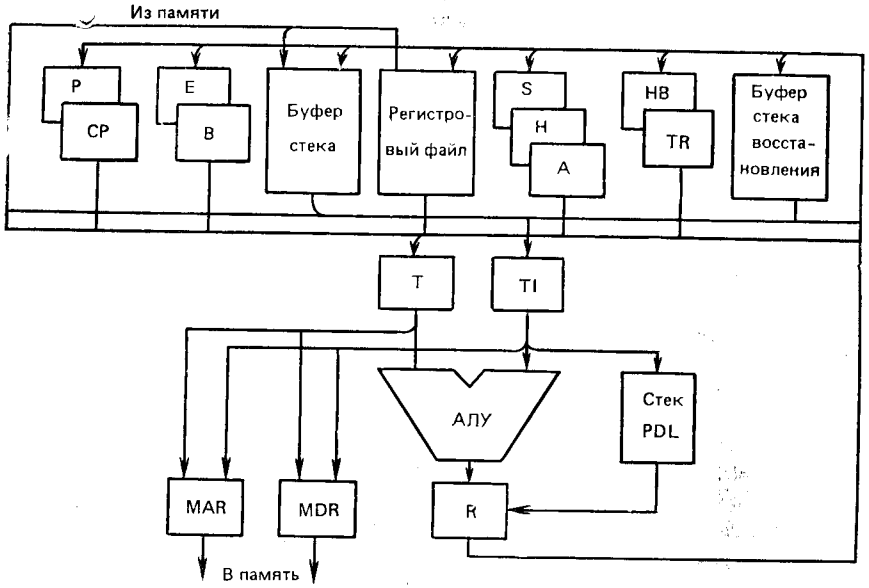


Рис. 3.13. Структурная схема блока обработки машины TICK-WARREN

обращение к буферу стека, регистровому файлу, регистрам и буферу стека восстановления. Полученные данные фиксируются во временных регистрах T и TI. На стадии E выполняются вычисления в АЛУ, а полученные результаты заносятся в регистр R, стек PDL (обратный магазинный список), адресный регистр MAR и регистр данных MDR. На стадии P осуществляется запись в регистры, стеки, а также обращение к основной памяти.

Указатели B, E и A (вершины стека) необходимы для управления стеком. Указатели H и S размещаются в счетчиках, обеспечивая сокращение количества блокировок.

Буфер стека представляет собой массив с быстрым доступом, в котором размещается верхняя часть стека. Стек содержит объекты двух типов: среду и точки возврата. Среда представляет собой множество постоянных переменных, на которые существуют непосредственные ссылки. Каждый из этих объектов может иметь произвольную длину.

В процессе компиляции Пролог-программе ставится в соответствие некоторая последовательность специальных команд. Команда состоит из кода операции и нескольких операндов (чаще всего одного). Код операции определяется обычно типом соответствующего символа Пролог-программы и контекстом его использования. Систему команд конвейерного Пролог-процессора составляют пять групп команд:

чтения — используются для чтения аргументов заголовка клоза и сопоставления с ними аргументов процедуры, заданных в регистрах A;

записи — осуществляют загрузку аргументов в регистры A;

унификации — соответствуют аргументам структуры (или списка) и выполняют унификацию термов и построение новых структур;

работы с процедурами — ставятся в соответствие с клозами Пролога, управляют выполнением Пролог-программы и выделяют память под среду унификации вызываемой процедуры;

индексирования — связывают в цепочки различные литералы, потенциально сопоставимые с заданным вызовом процедуры.

Команды унификации могут работать в режиме чтения или записи. В режиме чтения выполняется унификация со следующими друг за другом аргументами соответствующей структуры, адресуемой через регистр S. В режиме записи формируется последовательность аргументов новой структуры, адресуемой через регистр H.

Кроме рассмотренных архитектур интерес представляют RISC-машина [47], NEC-процессор [22] и другие технические решения, рассмотренные в [11, 12].

3.7. ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ ПРОЛОГ-МАШИН

При компиляции операторы процедур Пролога преобразуются в последовательности команд. Заголовок каждого оператора в этом случае представляет собой последовательность команд, выполняющих проверку возможности унификации аргументов запроса с аргументами самого оператора процедуры.

В машинах PSI и PEK реализована интерпретация Пролога, а в машинах TICK-WARREN, CHI и PLM — его компиляция. Применение компиляции позволяет добиться значительного повышения быстродействия — в 10 раз большее, чем при интерпретации. Однако в специализированных машинах PSI и PEK интерпретатор представлен в виде микропрограмм и широко применяются аппаратные средства, способствующие повышению быстродействия интерпретатора. Поэтому скорость интерпретации повышается настолько, что введение компилятора повышает ее всего лишь в 2—3 раза.

Стандартным средством для определения производительности Пролог-систем принята тестовая программа детерминированной конкатенации списков APPEND. Производительность (быстродействие) некоторых известных систем, полученная на этой программе, приведена в табл. 3.2.

Для оценки производительности Пролог-системы рекомендуется использовать два набора тестовых программ. Первый набор, предложенный Д. Уорреном [49], включает:

программу обращения списка из 30 элементов (NREV);

программу быстрой сортировки списка из 50 элементов (QS4);

программу перечисления списка из 25 элементов (PALIN25);

четыре программы символьного дифференцирования (TIME S10, DIV10, LOG10, OPS8);

Таблица 3.2

Система	Быстродействие, Липс *
BERKELEY PLM (COMPILED)	425000
TICK-WARREN (VLSI)	415000
AQUARIUS I (COMPILED)	305000
CHI	280000
SYMBOLIC 3600 (MICROCODED)	110000
PEK	60000
NCR/32 (PLM)	53000
DEC 2060 (COMPILED)	43000
PSI (MICROCODED)	30000
IBM 3030 (WATERLOO)	27000
VAX-780 (MACROCODED)	15000
SUN-2	14000
ALPHA	12000
LMI/LAMBDA	11000
VAX-780 (POPLOG)	8000
SYMBOLIC 3600 (INTERPRETER)	1500
Z-80 (MICRO PROLOG)	120
APPLE-II (INTERPRETER)	8

* Липс — LIPS (Logical Interences Per Second) — число логических выводов в секунду.

программу извлечения информации из базы данных (QUERY).

Второй набор, предложенный Беркли [35], предназначен для более глубокой проверки реализации системы команд Уоррена, включает:

две программы конкатенации, формирующие список из пяти элементов, причем первая программа осуществляет детерминированную конкатенацию (CON1), а вторая — недетерминированную (CON6);

программу решения задачи ханойской башни с 8 переставляемыми дисками (HANO1);

программу решения задачи о двух ферзях на шахматной доске размером 4×4 (QUEENS);

программу доказательства теоремы (MUTEST);

Таблица 3.3

Имя тестовой программы	VAX 8600	NCR/32	PLM	DEC 2060
NREV	116000	25000	115000	9000
QS4	98000	35000	174000	11200
PALIN25	67000	21000	134000	10500
TIMES10	48000	13000	63000	7700
DIV10	42000	11000	55000	7800
LOG10	56000	15000	79000	7800
OPS8	650000	21000	106000	11200
QUERY	20000	89000	367000	31900

Таблица 3.4

Имя тестовой программы	VAX 8600	NCR/32	PLM	DEC 2060
CON1	95000	53000	305000	43000
CON6	38000	110000	465000	—
HANOI	106000	59000	310000	—
MUTEST	73000	17000	89000	—
PR12	28000	7000	191000	—
QUEENS	77000	50000	148000	—
CKT2	—	17000	—	—

программу, реализующую алгоритм Решета для поиска всех простых чисел меньше 100 (PR12);

программу проектирования на базе элементов И—НЕ мультиплексора «4→1» (CKT2).

Результаты измерений на этих двух наборах быстродействия (Липс) четырех систем приведены в табл. 3.3 и 3.4.

ГЛАВА 4.
РАЗРАБОТКА АРХИТЕКТУРЫ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

В известных процессорах реализована система команд, близкая к командам абстрактной машины Уоррена, в виде микропрограмм. Для дальнейшей аппаратной реализации возможны два пути. Первый связан с реализацией микропрограмм на секционированных микропроцессорных комплектах, а второй — с реализацией процессора логического вывода на основе RISC-технологии. Реализовать первый вариант процессора проще, однако в типовых БИС не учитывается специфика алгоритмов обработки Пролог-программ. Второй вариант устраняет это последнее ограничение. Вместе с тем высокое быстродействие существующих БИС сохраняет «конкурентную» основу первого варианта.

4.1. ЗАДАЧИ И ОСОБЕННОСТИ РЕАЛИЗАЦИИ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

Для реализации своего прямого назначения ПЛВ должен выполнять следующие функции:

обработку команд объектной Пролог-программы;

установку содержимого регистров абстрактной машины, поддерживаемых аппаратно;

эффективное вычисление адресов операндов и работу с памятью;

унификацию операндов;

управление и поддержку локального и глобального стеков;

взаимодействие с HOST-машиной;

синхронизацию работы блоков и элементов.

Для обеспечения этих функций необходимо решить следующие задачи при проектировании:

определить набор функциональных блоков и алгоритмов их работы;

определить систему команд и разработать микропрограммы;

разработать интерфейс между функциональными блоками и систему синхронизации их работы;

провести анализ эффективности ПЛВ путем моделирования.

При проектировании следует принять во внимание:

использование в качестве промежуточного звена между стеками и регистрами абстрактной машины быстрой буферной памяти;

предварительную выборку данных и команд и их обработку;

аппаратную реализацию ряда команд и их обработку;

оптимизацию работы с основной памятью при доступе к ней через шину;

использование локального параллелизма;

сокращение длины трактов передачи информации.

В микропрограммном ПЛВ стеки данных и команд, а также тракт обработки команд и данных разъединены. Наличие стека команд позволяет реализовать кольцевой буфер для выборки команд. Стек данных (локальный и глобальный) выполняется в конструктиве с двумя буферными кеш-памятями, которые поочередно работают на запись (когда один буфер работает в режиме записи информации, второй либо не используется, либо его содержимое записывается в основную память). Микропрограммный ПЛВ имеет четкую функциональную структуру и специализацию блоков.

В комбинационном RISC-процессоре обеспечивается работа напрямую (не через шину) с основной памятью ПЭВМ, локальный параллелизм, используется регистровый файл, допускающий и чтение, и запись, универсализм функциональной структуры — отсутствие четкой функциональной специализации блоков.

Для реализации микропрограммного ПЛВ подходит отечественный микропроцессорный комплект К1804, а из зарубежных — Am29300. Выбор последнего удобен с точки зрения структуры обрабатываемых данных (32-разрядные операнды) и наличия в нем регистрового файла для представления регистров абстрактной машины.

4.2. СИСТЕМА КОМАНД ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

Все команды ПЛВ могут быть разделены на следующие группы.

Команды установки режима адресации — предназначены для установки видов адресации при вычислении значений параметров в Пролог-командах, имеющих аргументы типа PAR, VAR. В качестве аргумента PAR может выступать содержимое одного из

регистров аргументов или локальная переменная вызывающей или вызываемой процедуры. Идентификатор VAR является ссылкой на переменную, которая может быть локальной или глобальной и принадлежать либо вызываемой, либо вызывающей процедуре. Адрес PAR или VAR вычисляется как

(R)+смещение,

где (R) — содержимое регистров L, G, CL или CG (напомним, что в регистре L (G) — адрес первой ячейки текущей среды в L (G)-стеке, а в регистре CL (CG) — соответствующий адрес фрейма процедуры в локальном (глобальном) стеке.

Выполнение команд установки режима адресации приводит к подготовке в соответствующем регистре (PA для PAR, VA для VAR) адреса, который складывается со смещением в поле операнда команды. Их формат одно- и двухбайтовый для регистрового типа адресации (рис. 4.1,*а*).

К *командам передачи аргументов* относятся команды загрузки константы, переменной, структуры, а также последовательности команд загрузки (рис. 4.1,*б*). Основное назначение команд этой группы — установка содержимого ячеек памяти по указанным адресам.

Команды управления предикатом осуществляют разгрузку регистров аргументов, управление точкой выбора и передачу управления утверждению (рис. 4.1,*в*).

Команды управления утверждением подготавливают унификацию, осуществляют вход в тело утверждения и выход из него (рис. 4.1,*г*).

Команды вызова предиката осуществляют непосредственно вызов предиката, а также подготовку регистров и стеков для выполнения вызова (рис. 4.1,*д*).

Команды возврата и отсечений выполняют возврат процессора в последнюю точку выбора и уничтожают все точки выбора, созданные с момента вызова текущей процедуры (рис. 4.1,*е*).

Комбинированные команды генерируются для оптимизации выполнения частного случая структуры утверждения и имеют однобайтовый формат.

Команды унификации выполняют унификацию константы, скелетона, переменной с фактическим параметром PAR (рис. 4.1,*ж*).

Команды индексации осуществляют ветвления по типу аргумента PAR, определение утверждения и выполнение вспомогательных операций (рис. 4.1,*з*).

Команды передачи и загрузки аргументов

Аргументы (константы, переменные, адресные ссылки и структурированные термы) могут быть переданы либо через регистры, либо через область памяти (стек). Описание команд загрузки параметров:

для константы

load_cons(PAR, CONST),

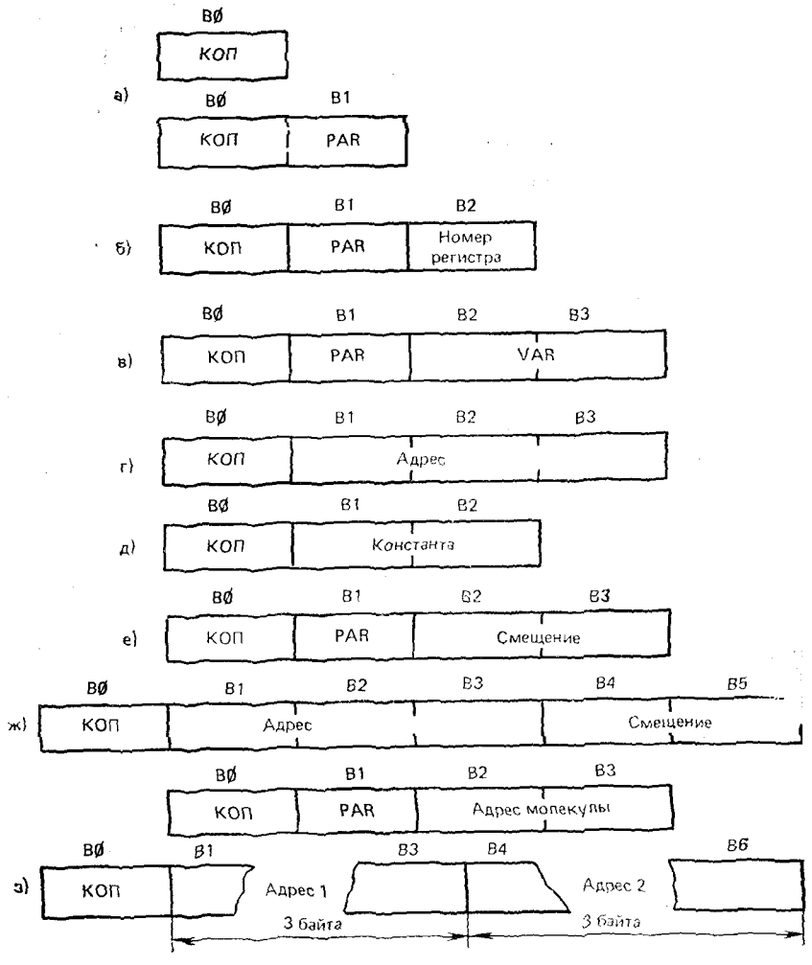


Рис. 4.1. Форматы основных команд ПЛВ

где CONST — константа; PAR — параметр, которому присваивается значение константы;

для переменной

load_var(PAR, VAR, Number),

где Number — число последовательно расположенных переменных, пересылаемых с ячейки VAR в область стека, адресуемую аргументом PAR;

для адресной константы

load_ref(PAR, Address),

где Address — адресная ссылка, присваиваемая параметру PAR;

для структурированного терма

load_str(PAR, str_addr),

где str_addr — адрес молекулы, копируемой в область памяти, адресуемую PAR;

null_var (PAR, VAR)

передает в PAR значение пустой переменной (такая переменная ссылается на саму себя, т. е. ее содержимым является адрес ее самой).

Вызов предиката и передача аргументов выполняются по схеме

call ⟨predicate_name⟩

{ ...
  load_args
  ... }

где фигурная скобка ограничивает совокупность команд загрузки аргументов, передаваемых данному предикату. Идентификатор load_args соответствует любой из перечисленных выше команд загрузки аргументов.

Если передача аргументов выполняется через области памяти, то необходимы команда загрузки специального регистра базы, следующая вслед за командой вызова предиката start_arg, и команда восстановления состояния этого регистра, завершающая последовательность команд загрузки аргументов end_arg.

При загрузке переменных возможны следующие дополнительные ситуации:

1) переменная имеет пустое значение, и это значение не будет изменено;

2) значение переменной в момент передачи аргументов не определено, и ей надо присвоить пустое значение;

3) значение переменной в момент передачи аргументов не определено, но можно отложить ее инициализацию до момента входа в вызываемую процедуру, впоследствии она будет инициализирована;

4) локальная переменная входит в последний литерал тела.

Все вхождения локальной переменной должны быть в теле утверждения, но по крайней мере одно ее вхождение не должно принадлежать данному последнему литералу тела. Соответствующая команда загрузки генерируется только при этих условиях. Если они выполнены для локальной переменной, то она может быть уничтожена при выполнении хвостового вызова. При этом ссылка на эту переменную станет «висячей», т. е. в ней не может быть необходимая для дальнейшего информация.

Команды управления предикатом

Для каждого предиката генерируется следующая цепочка команд логической машины:

⟨сохранить_состояние_логической_машины_в_момент_вызова⟩
⟨создать_точку_возврата⟩
⟨выполнить_альтернативный_предикат_1⟩
. . .
⟨выполнить_альтернативный_предикат_n⟩
⟨уничтожить_точку_возврата⟩

proc_entry (M) — команда сохранения состояния логической машины, запоминает содержимое M регистров аргументов вызываемой процедуры, используется только в разделе основной точки входа.

choice_point и del_choice_point — команды создания и удаления точки возврата. Команда choice_point генерируется как первая команда предиката, имеющего более одного правила, точка возврата создается в текущей среде локального стека. Алгоритм команды:

сохранить BR, TR, BL в L-стеке;
BL := L;
BG := G;

del_choice_point — команда восстановления содержимого регистров:

восстановить BP, BL из L-стека;
BG := G (BL)

exec (statement) — команда выполнения предиката, соответствует передаче управления:

BP := ⟨адрес_следующей_команды⟩
goto (метка)

где метка идентифицирует вызываемый предикат, передает управление на начало раздела Пролог-кода, описывающего утверждение, сначала загрузив в регистр BP адрес следующей Пролог-команды; генерируется для предикатов, содержащих более одного утверждения.

trust_stmt — команда выбора детерминированного (неальтернативного) предиката.

fail — восстановление содержимого памяти, соответствующего последней точке возврата.

cut_def — команда удаления точек возврата, созданных в текущей процедуре, соответствующей области файлового стека и всех локальных фреймов, сохранившихся после текущего локального фрейма.

cut_fail — команда удаления всех точек возврата, созданных в текущей процедуре, восстанавливает содержимое памяти, соответствующее последней точке возврата.

Команды управления утверждением

Утверждение (правило) может иметь или не иметь тело. Если правило не имеет тела, то генерируется следующая цепочка команд:

⟨установка_адресного_указателя_для_молекулы⟩
(если есть аргумент_молекула)

⟨команды_унификации⟩
⟨вызов_следующей_инструкции⟩

Если утверждение имеет тело, то генерируется следующая цепочка команд:

⟨установка_адресного_указателя_для_молекулы⟩
(если есть аргумент_молекула)
⟨команды_унификации⟩
⟨создание_локального_и_глобального_фреймов_для_правила⟩
⟨вызов_предиката_1⟩/*предикаты, входящие в правило*/
. . .
⟨вызов_предиката_n⟩
⟨возврат⟩

Команды унификации:

константы и терма

unify_const (A, T),

где A — адрес константы, T — терм;

переменной и терма

unify_var (A, V),

где V — переменная, A — адрес терма;

структурированных термов

unify_str (Ai, S),

где S — структурированный терм.

Этим трем случаям соответствуют следующие обобщенные алгоритмы унификации, записанные в нотации языка Пролог:

```
unify_const(S, T):—
        atomic(S),
        (var(T), T:= S; S== =T).
unify_var(S, T) :— var(S),
                   S:=T.
unify_str(S, T) :—
                   struct(S),
                   functor(S, F, N),
                   (var(T), functor(T, F, N);
                   struct(T), functor(T, G, M),
                   F== =G, N== =M),
            T= ..[_ | TL], S= ..[_ | TL],
      unify_arg(SL, TL).
      unify_arg([ ], [ ]).
            unify_arg([S | SL], [T | TL]) :—
      unify(S, T), unify_arg(SL, TL).
      unify(S, T) :—
            unify_const(S, T);
            unify_var(S, T);
            unify_str(S, T).
```

Здесь предикаты atomic (S), var (S), struct (S) проверяют, явля-

ется ли терм S атомом (константой), переменной или функтором соответственно. В наиболее сложном третьем случае структурированный терм S с помощью предиката functor (S, F, N) характеризуется именем старшей структуры F и числом аргументов N. Выделение остальных элементов структурированного терма реализуется списочным представлением терма вида

$$T=[\_|TL],$$

где TL — неунифицированные аргументы структурированного терма.

Алгоритмы выполнения команд унификации приведены при описании процессора унификации (гл. 5).

Команда unify_var_in используется для унификации с пустой переменной VAR.

Команда check_empty (PAR, Label) проверяет, является ли пустым параметр PAR, и, если да, осуществляет переход на метку Label.

Команда deref (V) выполняет полное дереференсирование переменной V (т. е. прохождение по адресной цепочке начиная с ячейки переменной V, содержащей первый адресный указатель, и до тех пор, пока «оконечная» ячейка не будет пустой или не будет представлять значение для терма).

Команда установки адресного указателя для молекулы begin_unify фиксирует адрес первого свободного элемента глобального стека, по которому будет записываться в глобальный стек молекула при унификации. Если среда не содержит глобальных переменных, то эти операции не выполняются, команда не генерируется, а Пролог-код утверждения начинается именно с команд унификации.

Создание локального и глобального фреймов для правила выполняется командой create_frame, а возврат из генерированного кода — командой exit или return. Команда create_frame генерируется сразу после блока команд унификации для утверждений, содержащих более одного литерала в теле, выполняет фиксацию среды текущей процедуры и устанавливает регистры для вызова новой. Команды exit генерируется как последняя команда утверждения, если в теле утверждения есть вызовы встроенных предикатов, освобождает локальный фрейм текущей процедуры, если она детерминирована, восстанавливает состояние регистров, передает управление по адресу в восстановленном регистре CP. Действие команды return аналогично последовательному выполнению enter и exit, она генерируется вместо команды enter, если утверждение является фактом.

4.3. МИКРОПРОГРАММНЫЙ ПРОЦЕССОР ЛОГИЧЕСКОГО ВЫВОДА

Микропрограммный ПЛВ работает в составе системы, включающей интерпретатор — программу, выполняемую под управлением операционной системы и использующую ее функции. Интерпретатор обеспечивает запуск и инициализацию ПЛВ, передачу управления от основного процессора (HOST) к ПЛВ. Таким образом, в этой системе интерпретатор выступает в качестве программной оболочки ПЛВ.

В структурной схеме ПЛВ (рис. 4.2) выделены макроблоки, которые можно рассматривать как отдельные устройства, выполняющие определенные, специфические операции. Совокупность таких операций и определяет функциональные возможности ПЛВ.

Процессор команд (ПК) предназначен для предварительной выборки команд из основной памяти HOST-процессора, выделения кода операции (КОП) объектной Пролог-команды, распаковки полей операндов команды. Эти функции реализуют блоки выборки и распаковки команд. Процессор команд считывает команды из основной памяти HOST-процессора в моменты, когда нет запросов на чтение (или запись) данных со стороны процессора памяти (ПП). Поля операндов распакованной команды по 32-разрядной шине поступают в процессор обработки (ПО) на один из входов АЛУ. Управление выдачей полей операндов из ПК осуществляет блок управления (БУ). Процессор команд связан с процессором ввода-вывода (ПВВ) двумя шинами адресов (АК) и команд (К).

Процессор ввода-вывода выполняет обработку запросов на ввод-вывод данных со стороны ПП и запросов на чтение команд со стороны ПК.

Процессор памяти имеет свою локальную память — кеш-память — для поддержки локального и глобального стеков, связан с ядром ПЛВ (БУ, ПО) и процессором унификации (ПУ) локальной шиной адреса (LA) и данных (LD). По локальной шине ПП принимает информацию и записывает в кеш-память, если адрес данных отображается в ее адресном пространстве, или читает информацию из кеш-памяти. Если адрес не отображается в адресном пространстве, ПП формирует запрос на ввод-вывод данных из основной памяти через ПВВ.

Для выполнения команд унификации предназначен ПУ. Он имеет свой собственный регистровый файл R1 ... Rn, некоторые регистры которого совпадают с регистрами ПО. Кроме этого, ПУ имеет блок анализа, блок сравнения, блок управления унификацией (БУУ).

Опишем более подробно ядро ПЛВ, включающее блок управления и процессор обработки. Назначение всех узлов БУ, кроме секвенсора, традиционное — название узла отражает его функцию. Секвенсор предназначен для формирования следующей микрокоманды микропрограммы при выполнении команды ПЛВ, позволяет реализовать последовательную выборку микрокоманд из памяти микропрограмм, выполнять безусловные и условные переходы к микропрограмме, анализировать четыре признака одновременно и формировать один из 16 возможных адресов перехода.

Арифметическо-логическое устройство ПО предназначено для выполнения арифметических и логических операций над адреса-

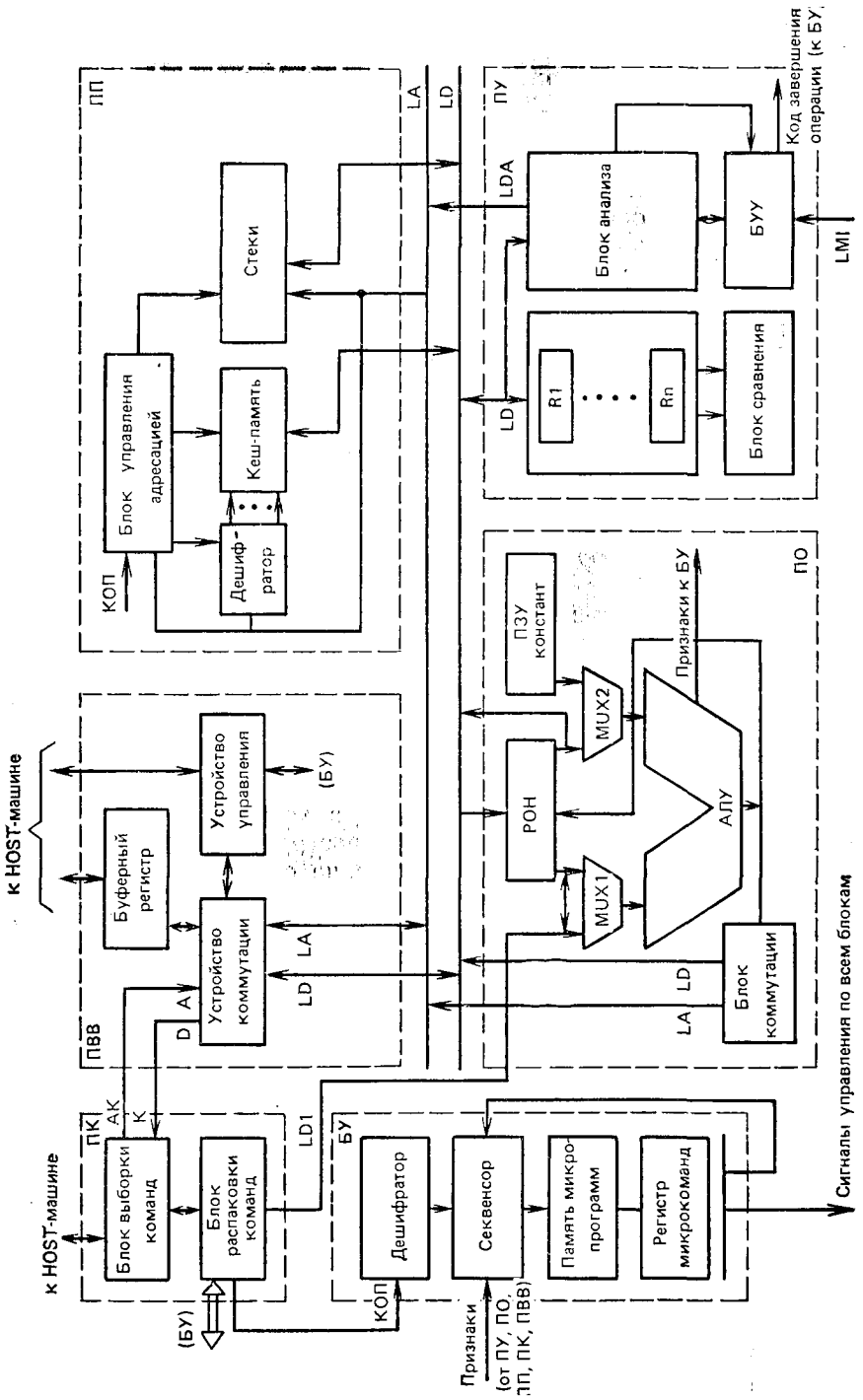


Рис. 4.2. Структурная схема микропрограммного ПЛВ

ми и данными: сложение; вычитание; инкремент +1, +2, +4; декремент —1, —2, —4; конкатенация полей двух операндов; логические операции над битовыми полями произвольной длины.

Блок РОН (регистровый файл) имеет два порта для записи данных и два порта для чтения данных. Потоки между шинами адреса и данных коммутируют мультиплексоры (MUX).

ПЗУ констант предназначено для хранения стандартных смещений, используемых в некоторых командах ПЛВ, а также значений тегов, которые используются в АЛУ для формирования данных для записи в стеки.

Процессор логического вывода работает в режиме прямого доступа к памяти системы, может прервать свою работу в связи с выполнением ввода-вывода в исполняемой скомпилированной программе и передать управление HOST-машине.

Чтобы HOST-машина выполнила требуемые действия, необходима следующая информация: содержимое всех регистров ПЛВ, отображающих текущее состояние вычислительного процесса; код операции, которую необходимо выполнить. С этой целью в памяти HOST-машины (основной памяти системы) выделяется область связи HOST-машины и ПЛВ, в которую при необходимости последний передает копии содержимого своих регистров и код операции. В эту область HOST-процессор по завершении выполнения требуемой операции помещает модифицированное содержимое регистров (HOST-машина поддерживает также регистры ПЛВ, но на программном уровне) и запускает спецпроцессор в требуемом режиме, записав в один из его регистров код режима пуска.

Выполнение базовых операций

Каждая Пролог-команда может быть выполнена с помощью набора базовых операций, реализуемых процессором. Общий формат базовых операций:

⟨*базовая операция*⟩ : : = ⟨*код операции*⟩[⟨операнд 1⟩] [⟨операнд 2⟩]

⟨код операции : : = ⟨ — — ¦ + ¦ — ¦ go to ¦ ⟨ = ¦ = ¦ ⟨ ¦ *shift_ left ¦ shift_right ¦ select [k, n]

⟨операнд i⟩ : : = регистр ¦ константа ¦ ячейка памяти

Здесь ⟨— — арифметическая операция присваивания операнду 1 значение операнда 2: ⟨операнд 1⟩ ⟨ — ⟨операнд 2⟩;

«+», «—», «*» — операции сложения, вычитания или умножения двух операндов;

go to — переход на новую Пролог-команду (т. е. изменение содержимого регистра адреса Пролог-команды), является внутренней операцией процессора команд;

«⟨ =», «⟨», «=» операции сравнения двух операндов с выработкой соответствующих признаков в АЛУ, которые могут проверяться блоком микропрограммного управления (БМУ) ядра с целью выработки действий по условию;

shift_left(n) и shift_right(n) — операции сдвига соответственно влево и вправо на n разрядов;

select [k, n] — выборка n байтов из операнда, начиная с байта k.

Рассмотрим выполнение тех базовых операций, которые не являются только внутренними операциями каких-либо блоков и позволяют описать выполнение любой Пролог-команды:

1) R1<—R2+(—) «constant»;
2) R1<—R2;
3) R1<—R2+(—)R3;
4) R1<—⟨операнд⟩ S(k);
5) R1>=R2;
6) [R1]<—R2;
7) R2<—[R1];
8) [R2]<—[R1];
9) go to адрес.

Здесь R1, R2 — регистры (соответствуют любым двум регистрам блока РОН); [R1] — содержимое ячейки памяти, адрес которой находится в R1.

Рассмотрим укрупненные алгоритмы выполнения базовых операций по структурной схеме на рис. 4.3 (назначение управляющих полей микрокоманды приведено в табл. 4.1).

1—3. R1 <— R2+(—) ... «constant», R1 <— R2, R1 <— R2 + (—) R3. В регистр R1 заносится содержимое регистра R2, сложенное с константой или содержимым регистра (рис. 4.4,*а,б*).

4. R1 <— ⟨операнд⟩ (S(k)). В качестве операнда выступает некоторое подмножество (до 4 байт) Пролог-команды. Байты, образующие ⟨операнд⟩, могут быть произвольно скоммутированы в АЛУ. Команда R1 <— ⟨APG_NO⟩*4, например, выполняется по схеме, приведенной на рис. 4.4,*в*.

5. R1 >= R2 (рис. 4.4,*г*). Эта операция связана с выработкой на АЛУ соответствующих признаков. Сформированные признаки анализируются БМУ.

6—8. [R1] <— R2, R2 <— [R1], [R1] <— [R2] (рис. 4.4,*д—ж*). БМУ анализирует признак выполнения операции в процессоре памяти — ПД (под-

Таблица 4.1

Поле микрокоманды	Назначение	Поле микрокоманды	Назначение
INI	Прерывание HOST-процессора	УПР. ПП	Управление ПП
		УПР. ввод-вывод	Управление ПВВ
УПР. ПК	Поле управления ПК		
АВХ1, АВХ2	Адреса записи в блок РОН	УПР. БМУ	Синхронизация операции в БМУ
АВЫХ1, АВЫХ2	Адреса чтения из блока РОН	АП	Адрес перехода в микропрограмму
ПК/БР	Управление коммутацией в MUX1	УПР. ПУ	Управление ПУ
		ВЫБ. УСЛ	Выбор проверяемого логического условия
ПЗУ/БР	Управление коммутацией в MUX2	УПР. БР	Управление блоком РОН
АДР. ПЗУ	Адрес чтения из ПЗУ	ЯДРО ПУ	Управление ядром ПУ
УПР. АЛУ	Управление АЛУ		

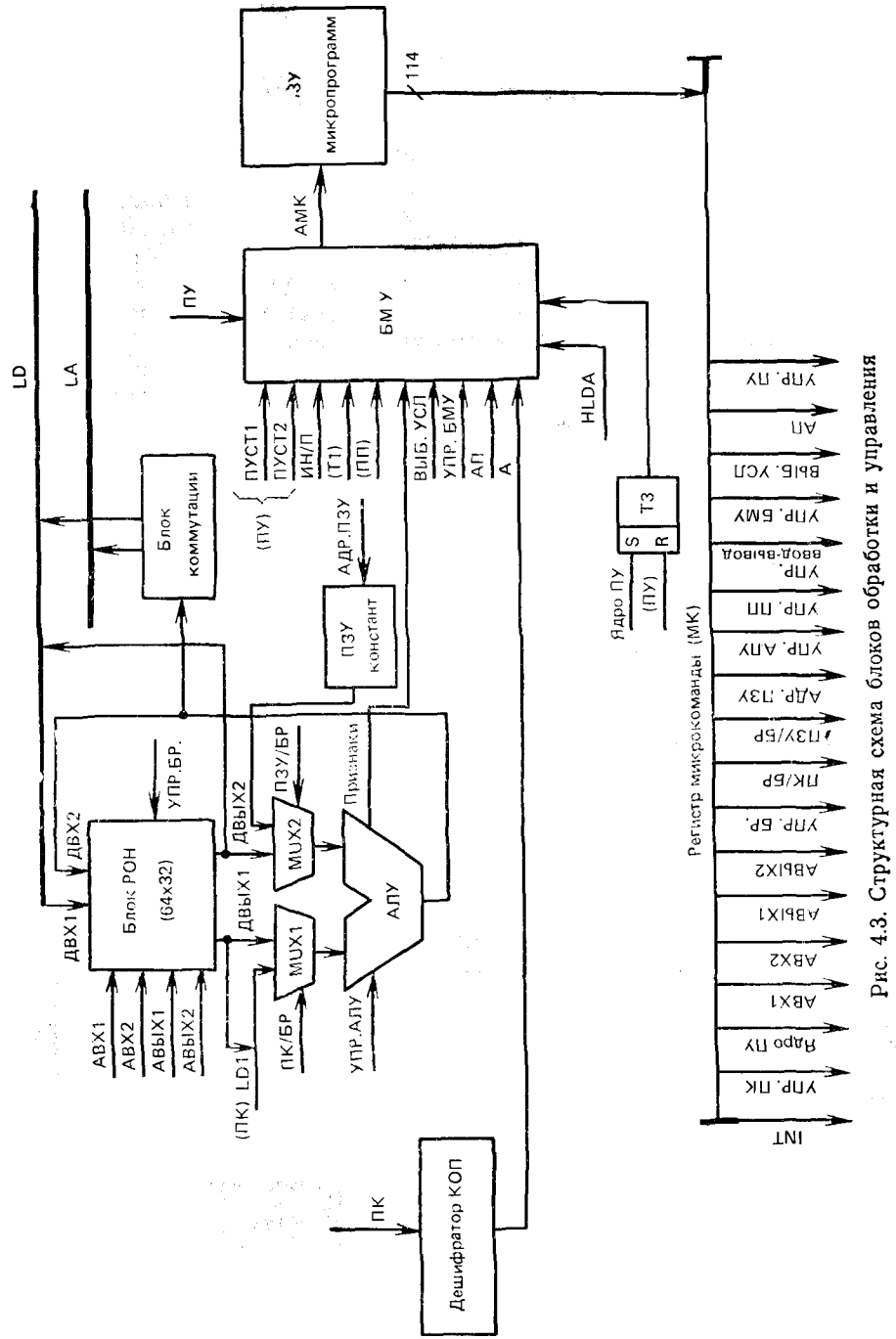


Рис. 4.3. Структурная схема блоков обработки и управления

тверждение данных). Если время ожидания исчерпано, а этот признак не выставлен, БМУ выдает сигнал ПВВ, подтверждающий прием данных. Если данные все еще не получены, БМУ переходит к обработке состояния ошибки. Описанные условия помечены на схеме буквой «У».

9. goto адрес (рис. 4.4,*з*).

Ниже приведены основные Пролог-команды и указаны типы базовых операций для них.

Команды стандартного режима адресации:
для VAR
VA<—L
VA<—G, операция 2,
VA<—CL
VA<—CG
для PAR

PA<—L+"admsize" PA<—CL+
+"admsize" R1<—R2(+)(—) "constant".

Команда регистрового режима адресации par_addr_args (ARG_N0) — операция 4.

Команды передачи и загрузки аргументов:
start_arg end_arg — операция 2;
load_cons (PAR, CONST) — операции 1, 4, 7;
load_str_cg (PAR, SKEL) — операции 4, 7;
load_str_g (PAR, SKEL) — операции 4, 7;
load_ref (PAR, VAR) — операции 4, 7, 6;
null_var (PAR, VAR) — операции 4, 6;
dear_var_arg (PAR, VAR) — операции 4, 6;
empty_out_var_arg (PAR, VAR) — операции 4, 6;
unsafe_var_arg (PAR, VAR) — операции 4, 6.

Команды управления предикатом:
proc_entry — операции 2, 1, 8, 5, 9;
choice_point — операция 2;
del_choice_point — операция 2;
exec (STATEMENT) — операции 1, 3;
trust_stmt (STATEMENT) — операция 4.

Команды управления утверждением:
begin_unify (GLOBAL FRAME SIZE) — операция 4;
create_frame_glob (LOCAL FRAME SIZE) — операция 4;
enter_body — операции 2, 3;
create_frame_noglob (LOCAL FRAME SIZE) — операции 2, 4;
retutn_glob (LOCAL FRAME SIZE) — операции 2, 4;

Команды вызова микропроцессора:
invoke (PRED_ADDR) — операции 3, 4;
call_prefix — операция 1.

Команды возврата:
fail (операции 5, 6, 9);
cut_def (LOCAL FRAME SIZE) — операции 4, 3, 5, 2;
cut_proceed_prefix_glob — операции 2, 5.

Команды унификации:
unify_var — операции 3, 4, 7, 8;
unify_var_in (PAR, VAR) — операции 5, 3, 4, 7, 8;
unify_var_out (PAR, VAR) —операции 3, 4, 7, 8;
unify_ref — операции 3, 4, 7, 8.

Система управления и синхронизации процессора

Микропрограммный принцип управления в ПЛВ позволяет модифицировать специализированный набор команд как в процессе наладки, так и в процессе серийного выпуска процессора, что в конечном итоге обеспечивает не только поддержку языка Пролог, но и сходных с ним языков программирования. Поскольку в состав ПЛВ (рис. 4.1) входят несколько специализированных процессоров, то для эффективной работы всего устройства необходимо децентрализованное управление. Однако это накладывает дополнительные ограничения на организацию синхронизации, обмен информацией между специализированными процессорами ПЛВ. Для квазипараллельной работы всего устройства целесообразно использовать синхронно-асинхронное их взаимодействие.

Когда необходимо обеспечить бесконфликтное взаимодействие HOST-процессора систем и ПЛВ, используется асинхронное распределение информации на системном уровне. В этом случае синхронизация доступа к системной шине обеспечивает разделение ресурса памяти ПЭВМ. Частота обмена по системной шине 16-разрядной ПЭВМ будет 5—8 МГц и 12—16 МГц для 32-разрядной.

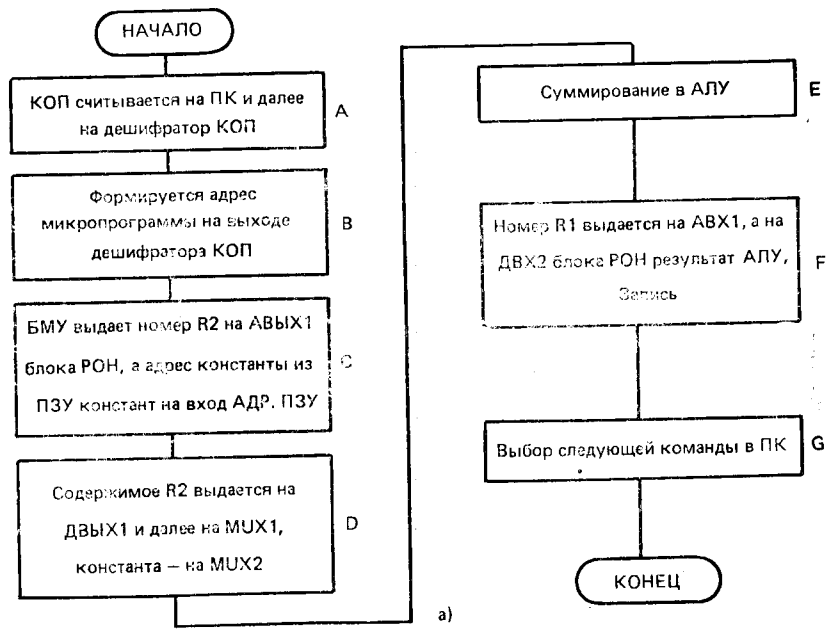
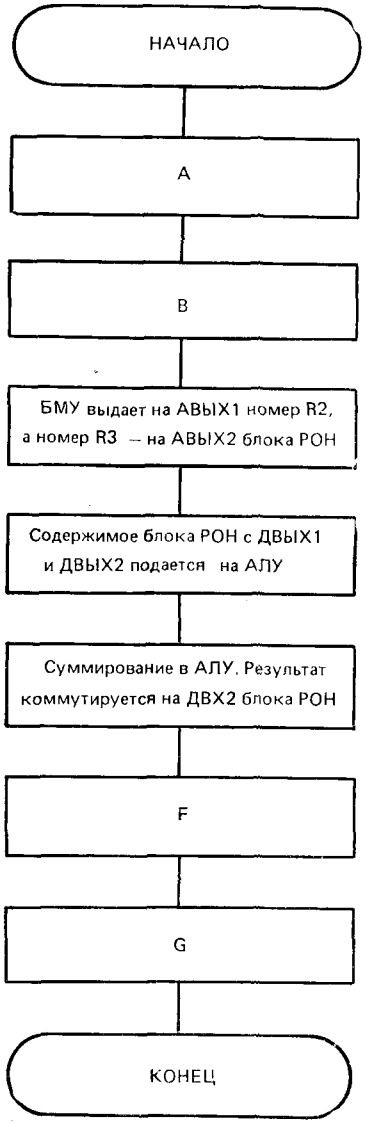


Рис. 4.4



б)

Рис. 4.4

Частота обмена по системной шине может быть снижена при организации по протоколу MULTIBUS из-за необходимости обслуживания низкоприоритетных устройств ПЭВМ. Таким образом, частота обмена по системной шине ПЭВМ во многом зависит от конкретной модели ПЭВМ, где будет использоваться ПЛВ.

Взаимодействие специализированных процессоров внутри ПЛВ осуществляется синхронно, т. е. подчиняется общему машинному циклу ПЛВ. Однако в связи с тем, что системные обмены информацией осуществляются асинхронно, в специализированных процессорах обеспечивается асинхронное взаимодействие как с системной шиной ПЭВМ, так и с сопрягаемыми блоками ПЛВ. К таким процессорам относится процессор команд, который выполняет опережающую выборку команд и их последующую предварительную распаковку. Опережающей выборкой команд из памяти ПЭВМ достигается максимальная загрузка ПЛВ при выполнении текущих операций, исключение простоев блоков управления для получения команды, а также уменьшение количества обращений к системной шине ПЭВМ при выполнении операций, требующих частых обращений к памяти ПЭВМ (например, для унификации).

Внутренняя синхронизация ПЛВ осуществляется от генератора синхроимпульсов БМУ. Блоки унификации и микропрограммного управления работают синхронно по отношению друг к другу. Это означает, что после начала операции унификации БМУ ожидает прихода сигнала завершения операции, после чего формирует следующую управляющую последовательность. Подобная организация работы объясняется тем, что алгоритмы выполнения команд ПЛВ основаны на алгоритмах виртуального Пролог-процессора, разработанного Уорреном, при такой синхронизации работы ос-

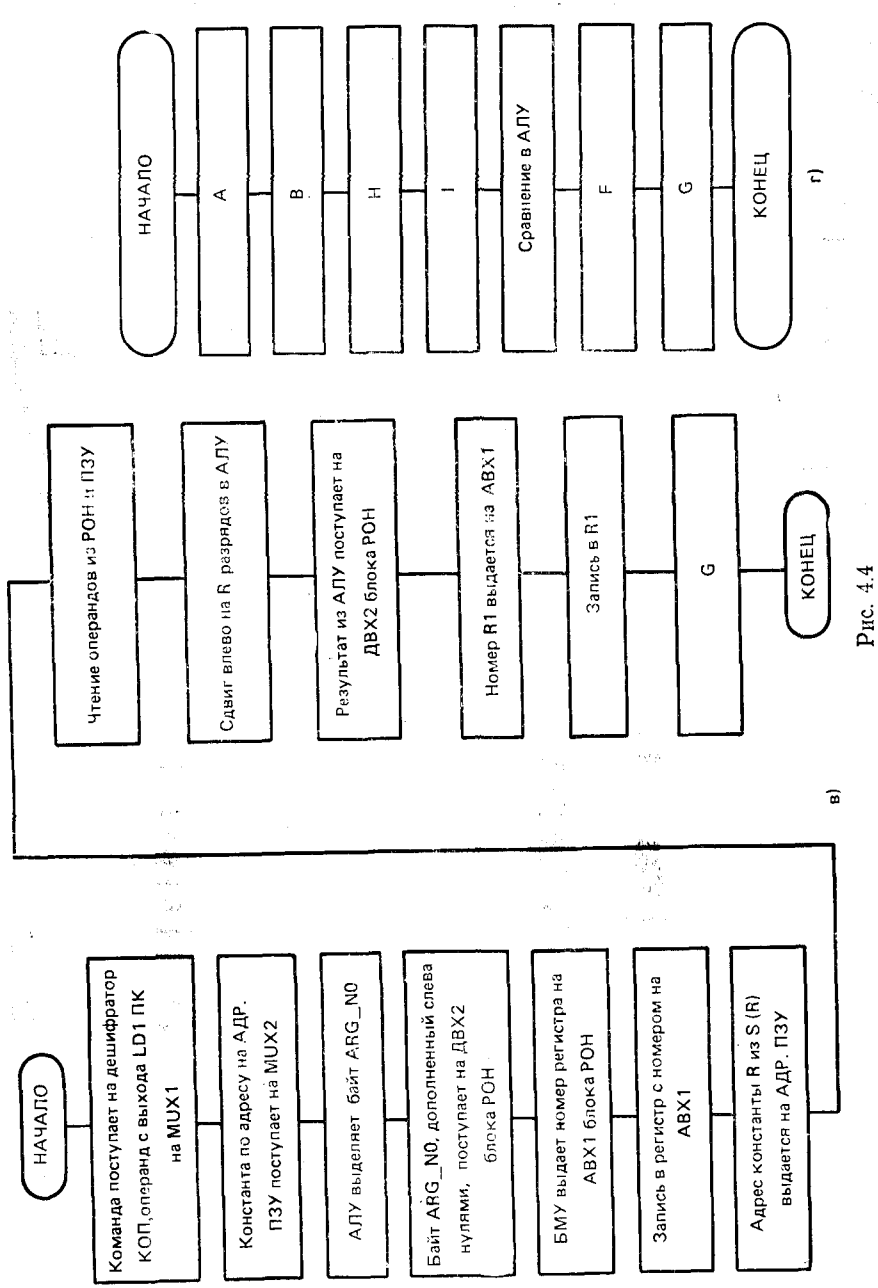


Рис. 4.4

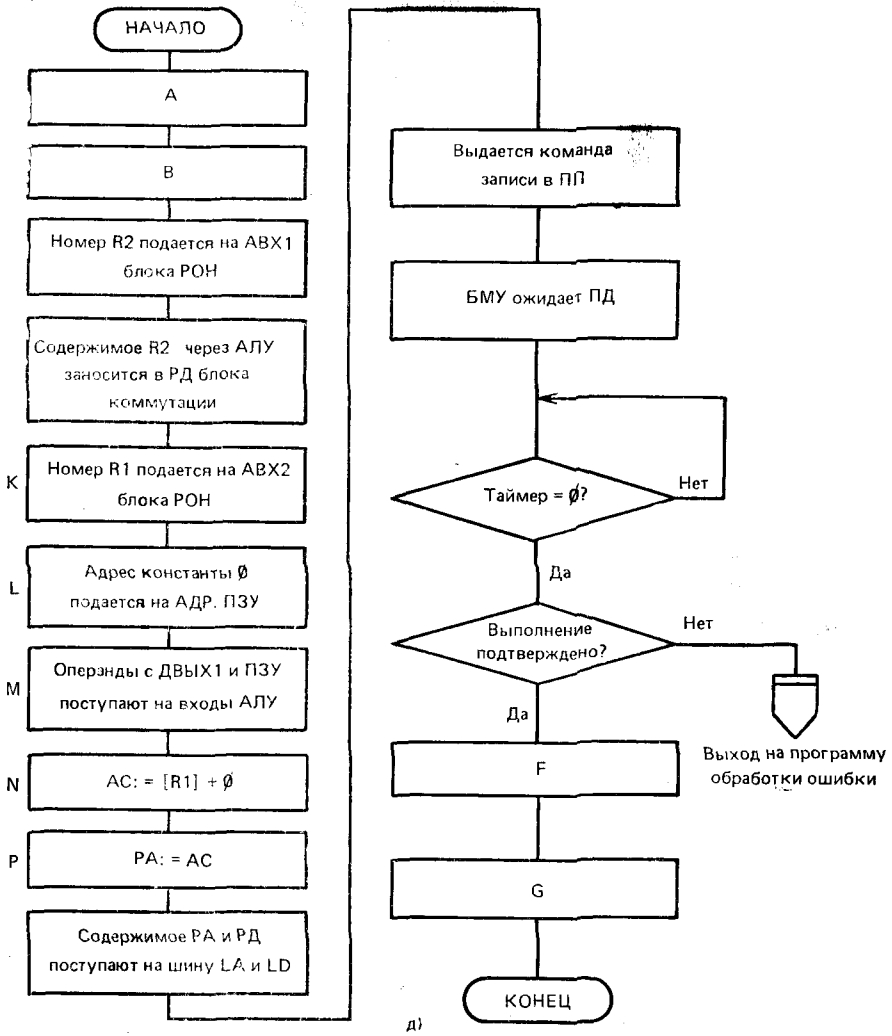


Рис. 4.4

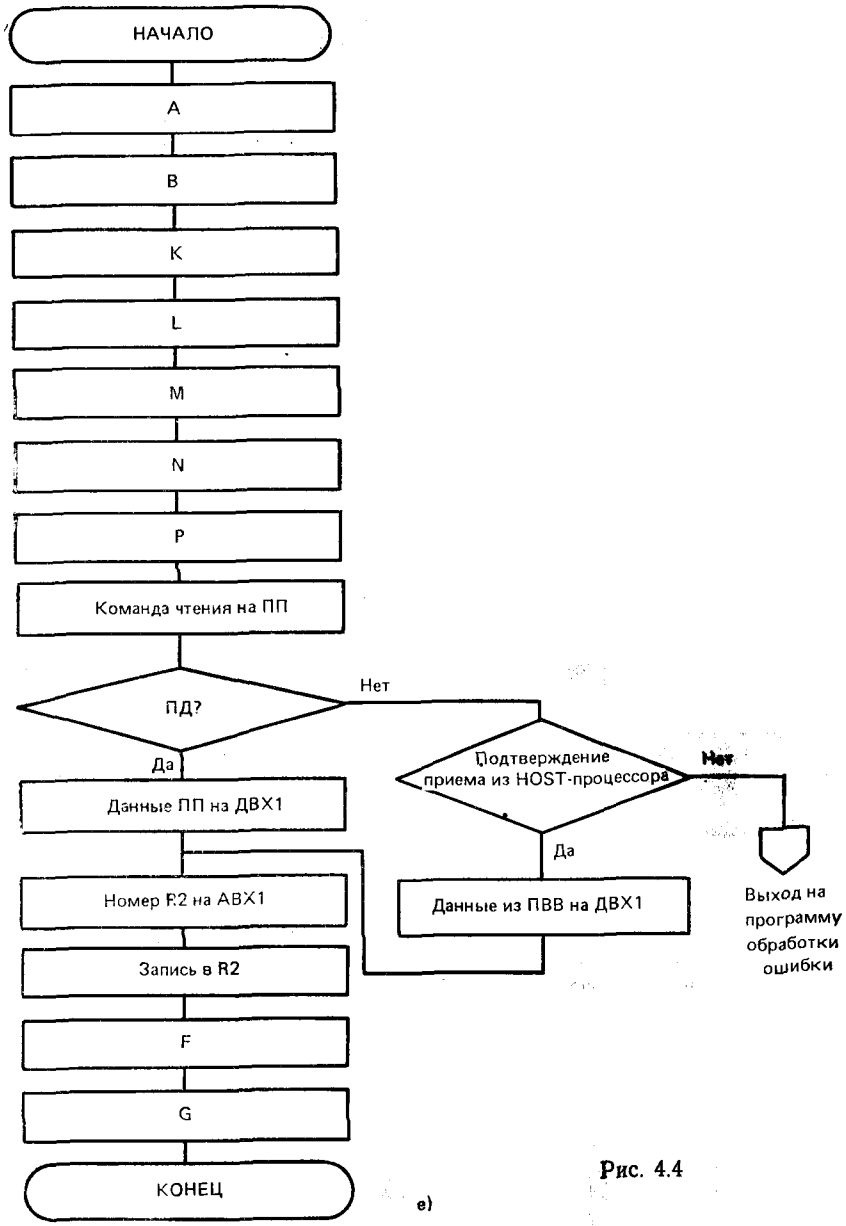


Рис. 4.4

новных блоков возможна автономная отладка как отдельной макрокоманды ПЛВ, так и их групп.

Синхронизация управления внутренними блоками ПЛВ осуществляется БМУ микрокомандами, имеющими горизонтальный способ микропрограммирования. Основные поля микрокоманды имеют следующее назначение:

управление БМУ для задания необходимых режимов работы;
управление контроллером кеш-памяти — чтение-запись блока;
управление процессором унификации;
управление схемой формирования чтения-записи;
управление мультиплексорами;
адрес ПЗУ команд;
код процессора команд;

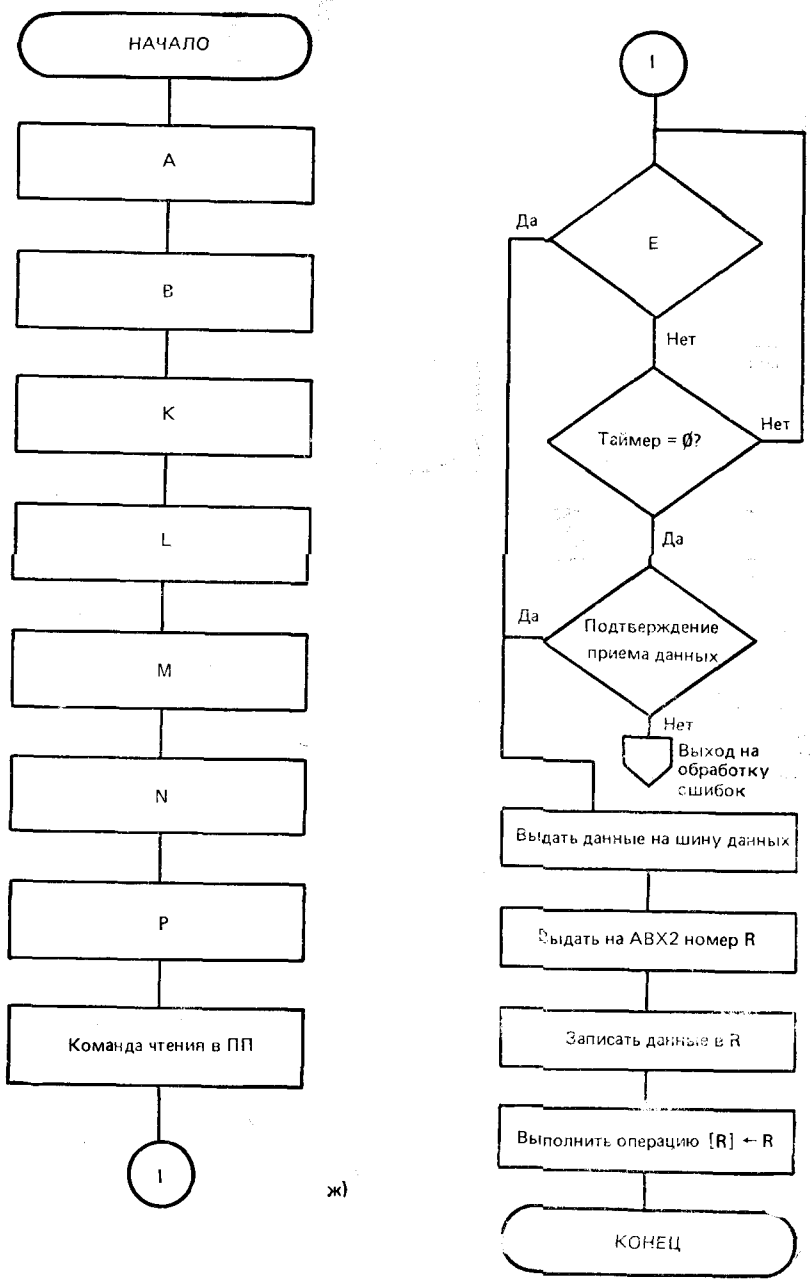


Рис. 4.4

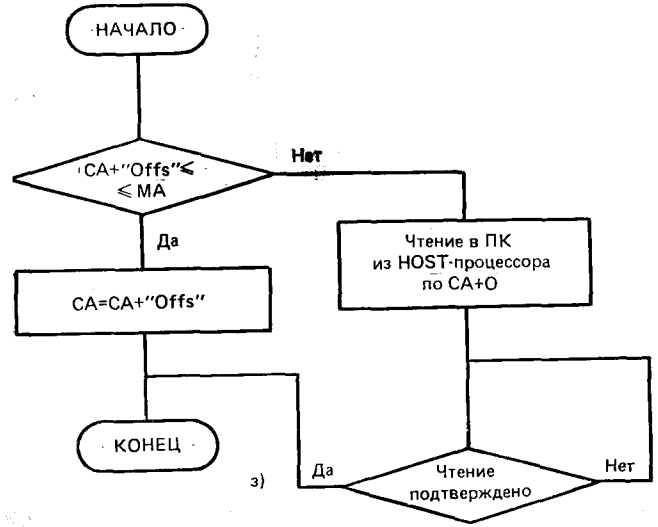


Рис. 4.4. Схемы алгоритмов базовых операций

управление регистрами адреса и данных при обращении к контроллеру ввода-вывода.

Процессор логического вывода работает в трех основных режимах:

инициализации — формирование начального состояния как рабочих регистров, так и регистров-указателей областей памяти;

выполнения Пролог-команд (на основании информации, содержащейся в рабочих регистрах и регистрах-указателях);

обработки исключительных ситуаций (при передаче управления интерпретатору языка Пролог).

Основное назначение режима инициализации — установка начального содержимого внутренних регистров ПЛВ. Режимы инициализации и обработки исключительных ситуаций имеют много общего — в обоих случаях процессору необходимо считать из заранее известной области памяти содержимое внутренних регистров ПЛВ после их модификации интерпретатором языка Пролог. Поэтому, получив сигнал аппаратного сброса, ПЛВ переходит в микропрограмму, обеспечивающую ожидание сигнала HOST-процессора на считывание известной области памяти. Поскольку считается, что системная шина ПЭВМ полностью «захвачена» HOST-процессором, ПЛВ находится в режиме ожидания до тех пор, пока по системной шине ему не будет передан специальный код, указывающий на переход во второй режим. В случае ожидания кода начала работы предварительная выборка команды в ПК отсутствует. В этом случае функции ПК сводятся к сканированию шины данных ПЭВМ и компарации приходящих кодов.

После получения команды инициализации процессора на системную шину ПЭВМ выставляется сигнал захвата шины и на-

чальное содержимое регистров ПЛВ считывается непосредственно в рабочие регистры. При обработке исключительной ситуации ПЛВ копирует содержимое внутренних регистров в определенную область памяти ПЭВМ и вырабатывает сигнал аппаратного прерывания HOST-процессора системы, передавая управление в область команд интерпретатора языка Пролог. После этого ПЛВ переходит в режим ожидания, аналогичный режиму после получения команды инициализации.

Во втором режиме работы ПК обеспечивает считывание и предварительную распаковку Пролог-команд, а также передачу микрокоманд процессором, входящим в состав ПЛВ. В этом случае управление блоками ПЛВ осуществляется теми полями микрокоманды, которые приведены выше.

Таким образом, использование синхронно-асинхронного управления ПЛВ позволяет обеспечить достаточно эффективную обработку информации и, кроме того, достаточно простое сопряжение ПЛВ с HOST-процессором на системном уровне.

4.4. RISC-ПРОЦЕССОР

На базе рассмотренной архитектуры микропрограммного процессора реализованы японские и американские Пролог-процессоры [17]. В основу другого варианта, не имеющего зарубежных аналогов, положены следующие соображения.

Команды машины Уоррена, в которые обычно компилируется исходная программа, содержат операции работы с памятью, в том числе вычисления адресов. Типовые процессоры при работе с памятью используют базовые, индексные и сегментные способы адресации, затрачивая в каждом случае от 6 до 12 тактов машинного времени. Включив в состав ЭВМ процессор, работающий со структурами данных и аппаратно поддерживающий вычисление адресов элементов этих структур, можно значительно повысить скорость выполнения откомпилированной Пролог-программы [3]. Использование микропроцессорных БИС для реализации блоков управления и обработки вносит избыточность и замедляет скорость выполнения операций.

Выполнение Пролог-команд можно реализовать не одним сопроцессором, а возложить на основной процессор, контроллер памяти и процессор унификации (рис. 4.5). В этой конфигурации набор откомпилированных команд будет реализован следующим образом. Основной процессор осуществляет общее управление, поддерживает работу интерпретатора и взаимодействует с контроллером памяти и унификационным процессором. Контроллер памяти построен на принципе аппаратного формирования сколь угодно сложного адреса. Для этого он содержит набор регистров, которые через мультиплексоры коммутируются на сумматор [6, 7]. Учитывая, что внутренний такт может быть в 4—6 раз меньше такта системной шины, вычисление любого адреса осуществляется за один такт контроллера. Процессор унификации, получая на

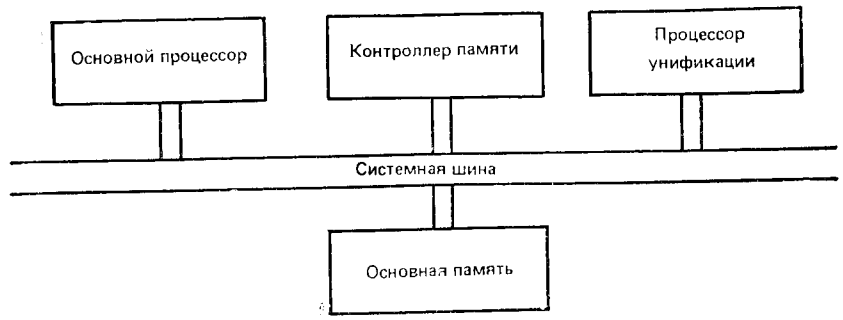


Рис. 4.5. Схема системы логического вывода

входе два терма в виде их адресов, полностью осуществляет унифицирование сколь угодно сложных данных (внутренняя организация и алгоритмы работы процессора унификации рассмотрены ниже).

Однако относительная сложность структуры (три процессора на одной шине) вынуждает искать более простые варианты. Одним из вариантов является совмещение функций контроллера памяти и процессора унификации в одном процессоре — RISC-процессоре (рис. 4.6). Этот процессор совместно с основным процессором производит обработку Пролог-команда следующим образом.

После окончания компиляции выполнение объекта Пролог-программы начинает основной процессор. Как только в объектной программе появляется последовательность откомпилированных команд (по префиксу ESC), управление передается RISC-процессору, который выбирает команды из памяти, по первому байту (коду операции) настраивает блок управления и начинает выполнение команды. При последовательном выполнении команд содержимое счетчика адреса команды наращивается. В случае перехода происходит передача управления, для чего в стек записывается адрес перехода, по которому передается управление командой RET.

Данный сопроцессор реализует упрощенную систему Пролог-команд, алгоритмы которых рассматривались в § 4.3. Входные регистры B0 ... B3 процессора (рис. 4.7) используются для хранения байтов команды (рис. 4.1), поступающих с шины данных DB. В регистр B0 записывается код операции. Этот же регистр исполь-

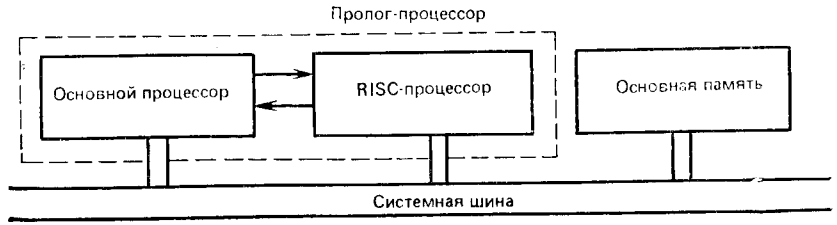


Рис. 4.6. Схема системы на основе RISC-процессора

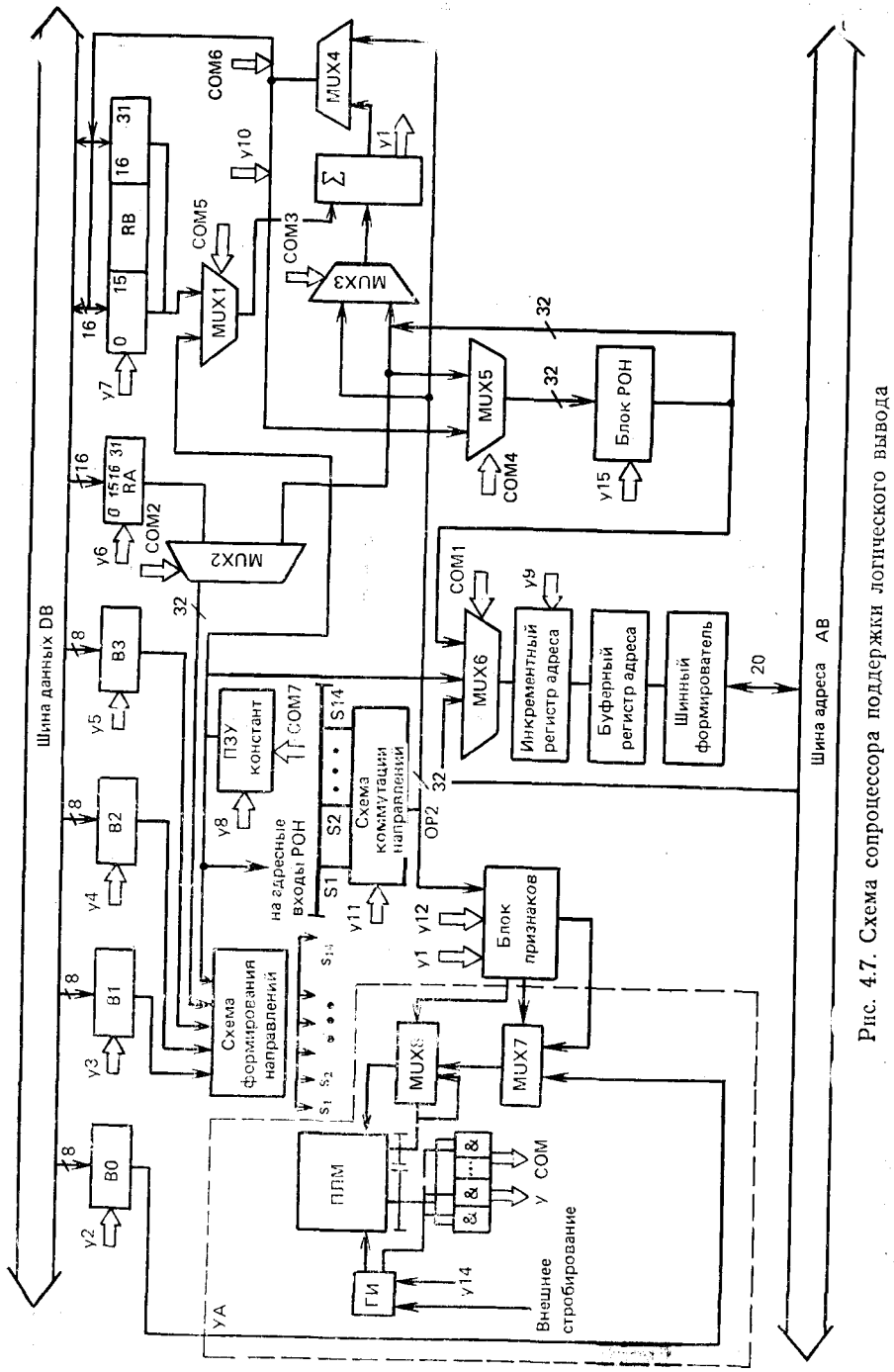


Рис. 4.7. Схема сопроцессора поддержки логического вывода

зуется в качестве регистра состояния управляющего автомата (УА), реализованного на программируемой логической матрице (ПЛМ) и генераторе синхроимпульсов (ГИ).

Необходимую для формирования операндов коммутацию байтов команд RISC-процессора обеспечивают схемы формирования и коммутации направлений. Такими направлениями являются следующие:

S1 — конкатенация байтов B2 и B3;

S2 — константа из ПЗУ констант;

S3 — учетверенный байт B1 (B1∗4);

S4 — дополнительный код содержимого регистра;

S5 — учетверенный результат конкатенации байтов B2 и B3 ((B2//B3))∗4, где // — операция конкатенации);

S6 — содержимое регистра данных RA;

S7 — конкатенация байтов B1//B2//B3;

S8 — учетверенный результат конкатенации байтов B1 и B2 ((B1//B2)∗4);

S9 — байт B1;

S10 — байт B2;

S11 — младшие три байта регистра дополняются нулями, старший байт сохраняется;

S12 — младшие три байта регистра сохраняются, старший дополняется нулями;

S13 — вместо старшего байта регистра подставляется константа из ПЗУ констант;

S14 — выбираются биты с 23-го по 16-й и дополняются тремя старшими нулевыми байтами.

Выбираемое направление определяется выполняемой командой за счет требуемой коммутации направлений Si на схеме коммутации направлений. Управление «раскруткой» команды осуществляется ПЛМ, которая вырабатывает управляющие сигналы на выходе Y и COM по тактам, синхронизируемым ГИ. Группа разрядов Y выполняет управление, а COM — коммутацию информации в ПЛВ. Условные переходы при выполнении алгоритмов команд выполняется в ПЛВ на основании признаков, формируемых в блоке признаков и на сумматоре при сравнении операндов.

Регистры RA и RB — 32-разрядные регистры данных, причем RA работает только на прием данных, на RB — на прием и передачу в шину данных.

Сопроцессор поддержки логического вывода реализует следующие операции: сложение, сравнение, формирование дополнительного кода, пересылку, сдвиг операнда (-ов).

Операнды поступают из основной памяти, блока РОН и ПЗУ констант. В блоке РОН 64 регистра, моделирующих регистры абстрактной машины Уоррена. Номера регистров, используемые в командах, выдаются на выход ПЛМ или хранятся в ПЗУ. Адресный вход блока РОН y15—12-разрядный: 6 для регистра-источника и 6 для регистра-приемника. Кроме того, каждому регистру

соответствует конкретная ячейка в основной памяти. Схема адресации этой ячейки и регистра в блоке РОН:

$$Aоп = \langle a0, \ldots, a7, a8, \ldots, a19 \rangle,$$

где разряды с a8 по a19 определяют номер регистра в блоке РОН, а разряды с a0 по a19 — адрес ячейки в памяти. На рис. 4.8 приведен формат группы разрядов управления РОН.

Схема функционирования ПЛВ следующая. HOST-процессор передает первые два байта объектной Пролог-команды по шине DB и внешним управляющим сигналом обеспечивает их запись в регистры B0 и B1, а также запуск ГИ. Байт кода операции поступает на вход ПЛМ, определяя ее выход как логическую функцию от входа, которая стробируется тактовыми импульсами ГИ. Параллельно обеспечивается прием адреса команды в инкрементный регистр (счетчик) адреса с шины адресов AB. Если в команде более двух байт, то следующие два будут считаны по адресу, на единицу большему, чем текущий адрес в счетчике адресов. Выполняемые команды могут быть безоперандными, одно- и двухоперандными. В первом случае выполнение команды сводится к модификации содержимого регистра B0; во втором — операнд формируется на выходе схемы коммутации направлений (OP2) и может коммутироваться или на вход комбинационного сумматора, или через мультиплексор на вход регистра RB, или на вход данных блока РОН, или в блок признаков.

Если команда двухоперандная, то второй операнд поступает из регистра либо RA, либо RB.

Блок признаков формирует признаки (биты), используемые при унификации:

первая унифицируемая переменная пуста;
вторая унифицируемая переменная пуста;
теги унифицированных переменных равны;
унификационный стек не пуст;
унификационный стек заполнен;
ссылка в сторону меньшего адреса;
выполнение команд унификации;
аргументы совпадают;
унифицируется молекула;
операция протоколирования в трейловом стеке;
первая и вторая унифицируемые переменные пусты;
унификация константы и функтора;

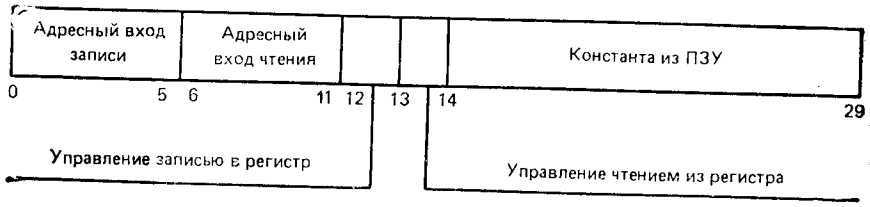


Рис. 4.8. Управляющие и адресные поля РОН

унификация константы и переменной;
унификация двух констант;
унификация двух переменных;
унификация переменной и функтора;
переполнение кучи;
недействительный тег;
недействительное значение;
завершение унификации неудачей;
завершение унификации удачей;
неверное число аргументов и др.

Коммутация направлений реализуется схемой на рис. 4.9. Выбор одного из направлений S1 ... S14 осуществляется внутренней ПЛМ, которая обеспечивает требуемую коммутацию на мультиплексорах. Входы этой ПЛМ соединены с четырьмя управляющими выходами ПЛМ (обозначенными y11).

Рассмотрим выполнение базовых операций в ПЛВ. Каждая операция записывается далее в условном формате, где используются следующие обозначения:

r — содержимое регистра r; может быть указано с индексом r1;

[r] — содержимое ячейки, адрес которой содержится в регистре r;

B0, B1, B2, B3 — байты объектной Пролог-команды, находящиеся в одноименных регистрах ПЛВ;

<— — операция пересылки;

goto(A) — переход по адресу A;

if(условие) ⟨оператор⟩ — условная операция, выполняемая, если указанное в скобках условие имеет место;

Ti — i-й внешний такт;

ti — i-й внутренний такт;

1. r1 <— r2 + B1*4

t1: выбор S3 и подача на вход сумматора; номера r1, r2 поступают на адресные входы записи и чтения блока РОН;

t2: содержимое r2 через мультиплексор коммутируется на вход сумматора; полученная сумма коммутируется на вход данных блока РОН и записывается по адресу r1, поступающему на адресный вход записи блока РОН.

2. r1 <— B2

t1: номер регистра r1 поступает на адресный вход записи блока РОН; t2: операнд, определяемый байтом B2 (направление S10), поступает через мультиплексор на вход данных блока РОН и производится запись.

3. r1 <— r2

t1: номера регистров r1 и r2 подаются из ПЗУ на адресные входы блока РОН — номер r2 на вход считывания, а номер r1 на вход записи; t2: содержимое r2 с выхода РОН записывается через мультиплексор на вход данных в регистр r1.

Отметим, что блок РОН позволяет одновременно производить чтение и запись для различных регистров.

4. [r1]<—r2

t1: чтение номера r2 из ПЗУ; t2: содержимое r2 с выхода РОН коммутируется по следующему тракту: мультиплексор — схема формирования направлений (направление S6) — схема коммутации направлений — мультиплексор —

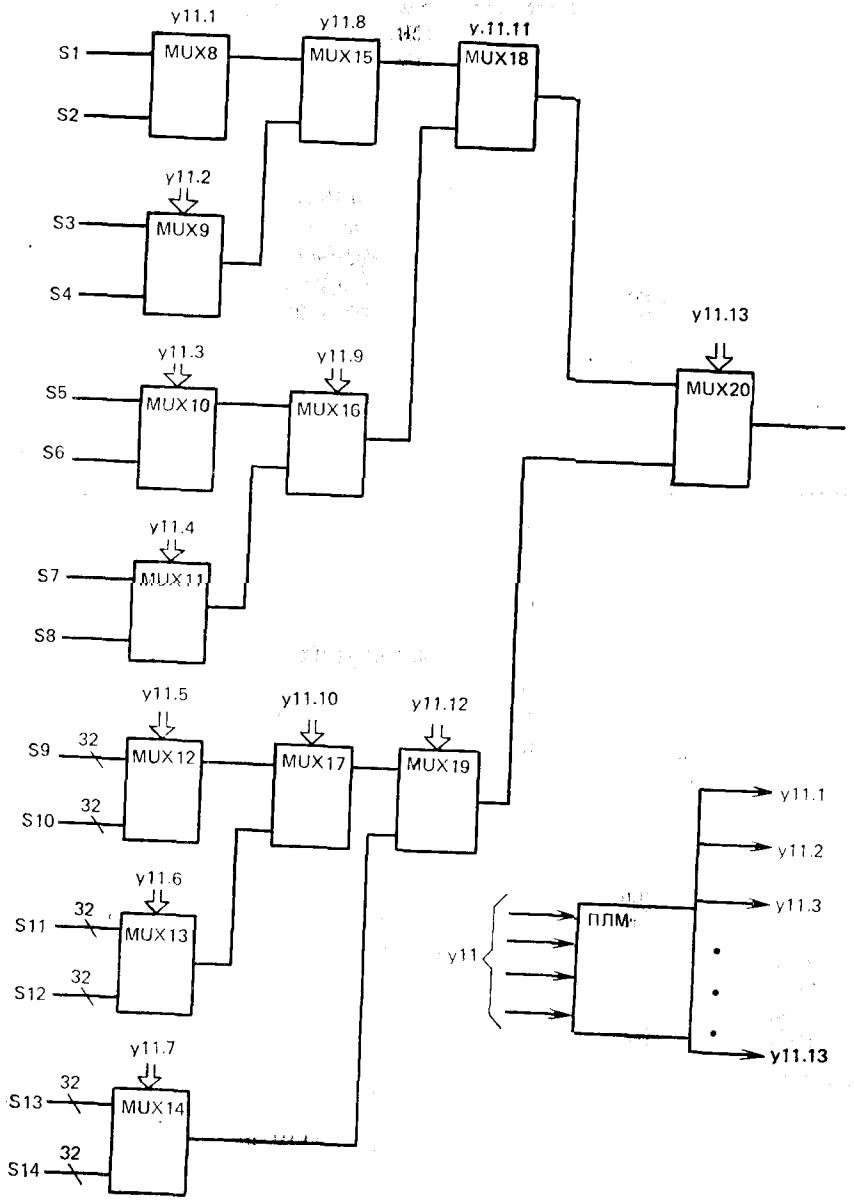


Рис. 4.9. Схема коммутации направлений

вход RB; t3: чтение номера r1 из ПЗУ; t4: запись содержимого r1 блока РОН в инкрементный регистр адреса; T1: запись в память по адресу в инкрементном регистре адреса данных из RB (описана в команде 15).

5. r1 <— r2+r3

t1: номер r2 поступает на адресный вход чтения РОН; t2: содержимое r2 с выхода блока РОН записывается в регистр RB по тракту: мультиплексор — схема формирования направлений — схема коммутации направлений — мультиплексор — RB (коммутируется направление S6); t3: номер r3 подается на адресный вход чтения РОН; t4: содержимое r3 поступает по направлению S6 на второй вход сумматора, а содержимое RB — на первый вход сумматора, результат записывается в РОН по адресу записи (r1), поданному вместе с номером r3.

6. r1 <— константа

t1: номер r1 поступает на вход адреса записи; константа коммутируется по направлению S2 из ПЗУ констант на вход данных РОН; t2: запись по адресу на адресном входе записи РОН.

7. goto(A+1)

Это команда перехода по адресу следующей команды программы. Исходный адрес A записывается в инкрементный регистр адреса. t1: к содержимому этого регистра добавляется 1; t2: адрес выдается на шину AB.

8. r1 <— [r2]

t1: чтение номера r2; t2: чтение содержимого r2 с выхода РОН в инкрементный регистр адреса (запись младших 20 разрядов); T1: запись в RA содержимого адресуемой ячейки из HOST-памяти; t3: выдача номера r1 на адресный вход записи блока РОН, содержимое RA коммутируется по направлению S6 через мультиплексоры на вход данных блока РОН, запись в r1.

9. [r1] <— [r2]

t1: чтение номера r2; t2: чтение содержимого r2 с выхода блока РОН в инкрементный регистр адреса (запись младших 20 разрядов); T1: запись в RB содержимого адресуемой ячейки из памяти HOST-процессора; t3: чтение номера r1 из ПЗУ; t2: запись содержимого r1 в регистр адреса; T2: запись содержимого RB по адресу в регистре адреса (описана в команде 15).

10. [r1] <—[r1]

t1: номер r1 из ПЗУ поступает на адресный вход блока РОН и в регистр адреса; t2: содержимое r1 с выхода блока РОН коммутируется по тракту: мультиплексор — схема формирования направления (направление S6) — схема коммутации направлений — мультиплексор — вход RB; T1: запись в память HOST-процессора содержимого регистра RB, выставляемого на шину данных, по адресу, находящемуся в адресном регистре; запись осуществляется за два такта (в первом такте старших двух байт RB, во втором — двух младших, см. команду 15).

11. r1 <— r2+/—const

t1: номера r2 и r1 поступают на адресные входы чтения и записи в РОН соответственно; t2: содержимое r1 с выхода данных блока РОН поступает на первый вход сумматора, константа через мультиплексор — на второй вход сумматора; результирующая сумма записывается в регистр r1 РОН, номер которого коммутируется на адресный вход записи.

12. goto — ⟨внутренняя метка⟩.
В отличие от базовой команды 7, внутренняя метка означает переход внутри выполняемой последовательности базовых операций, т. е. возврата управления в HOST-процессор не происходит. t1: требуемая установка байта B0.

13. if(r1>r2) [базовая операция]
t1: чтение номера r1 из ПЗУ на адресный вход чтения РОН; t2: передача содержимого r1 по тракту: мультиплексор — схема формирования направлений — схема коммутации направлений — мультиплексор — регистр RB; t3: номер r2 поступает на адресный вход чтения РОН; t4: содержимое r2 и r1 сравниваются на АЛУ. Условный переход в УА на основании результата сравнения.

14. r <— B +/— const
t1: номер регистра r считывается из ПЗУ и коммутируется на адресный вход записи блока РОН; коммутируемый байт B поступает по соответствующему направлению на второй вход сумматора, а константа из ПЗУ констант — на первый вход сумматора; t2: результат с выхода сумматора записывается по адресу на адресном входе записи блока РОН.

15. r1 <— r2 +/— ⟨константа⟩
Константа выбирается из полей команды (байты B2, B3, B4, B5). (Напомним, что с начала выполнения команды в B0 и B1 имеются два значащих байта. Если в команде используются еще два байта B2 и B3, то их необходимо считать из памяти HOST-процессора (требуется внешний такт T). Если в команде используются еще байты B4 и B5, то они поступают в RA. Так как RA — четырехбайтовый регистр, то в нем могут храниться байты B4, B5, B6 и B7.) t1: к содержимому регистра адреса добавляется 1; T1: содержимое регистра адреса выдается на шину адреса, чтение B2 и B3; t2: к содержимому регистра адреса добавляется 1; T2: содержимое регистра адреса выдается на шину адреса, чтение B4 и B5 в RA. (Действия аналогичны базовой операции 11, однако с тем отличием, что константа коммутируется из команды.)

16. [r1+Si] <— r2
Операнд одного из направлений Si складывается с содержимым r1, определяя тем самым адрес ячейки памяти, в которую записывается содержимое r2. t1: чтение адреса r1 из ПЗУ адресов; t2: содержимое r1 складывается на сумматоре с операндом направления Si, результат записывается в рабочий регистр блока РОН. Адрес рабочего регистра вместе с r1 формируется в такте t1; t3: чтение номера r2; t4: запись содержимого r2 в RB по тракту: выход РОН — мультиплексор — схема формирования направлений (направление S6) — схема коммутации направлений — мультиплексор — RB; t5: установка номера рабочего регистра на адресном входе чтения блока РОН; t6: содержимое рабочего регистра [r1+Si], усеченное до 20 младших разрядов, записывается в адресный регистр; T1: запись содержимого RB (старших двух байтов) по адресу в адресном регистре; t7: к регистру адреса добавляется 1; T2: запись содержимого RB (младших двух байтов) по адресу в адресном регистре.

17. r1 <— S13
t1: чтение из ПЗУ номеров r1 и r2, где r2 — регистр, три младших байта которого записываются в r1 (возможно, что r2=r1); t2: содержимое r2 поступает на схему формирования направлений, куда также поступает константа из ПЗУ констант; операнд с направления S13 коммутируется на вход блока РОН; t3: запись в r1. Если r1=r2, то схема несколько усложняется; t3: запись операнда OP2 в промежуточный рабочий регистр; t4: установка адреса r1 для записи и адреса рабочего регистра для чтения в РОН, далее выполняется команда 3, где вместо r2 используется рабочий регистр.

ГЛАВА 5.
РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНЫХ КОМПОНЕНТОВ МИКРОПРОГРАММНОГО ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

Рассматриваются особенности реализации отдельных функциональных блоков ПЛВ (см. рис. 4.2): процессора команд (ПК), процессора обработки (ПО), процессора памяти (ПП), процессора унификации (ПУ), процессора ввода-вывода (ПВВ) и блока управления (БУ), связанных через внутреннюю 32-разрядную шину данных LD и 24-разрядную шину адресов LA. Особое внимание уделено проектированию блока унификации, поскольку он определяет основную специфику аппаратной поддержки в ПЛВ. Подробно описана структурная схема этого блока, который по своей сложности является специализированным процессором со своим управлением, регистровой памятью и блоком обработки. Подробно рассмотрены алгоритмы унификации различных термов — констант, переменных и скелетонов (функторов), приведена система команд.

5.1. ПРОЦЕССОР КОМАНД

Процессор команд построен по конвейерной схеме и реализует следующие этапы обработки команд: выборку команд, буферизацию, распаковку (дереференсирование), очередь распакованных команд, исполнение (рис. 5.1).

Объектная Пролог-команда поступает по шине K в регистры RGH и RGL, куда помещаются соответственно старшие и младшие два байта команды, далее — в кольцевой буфер. Заполнением кольцевого буфера управляет устройство управления обменом с шиной (УУОШ). Кольцевой буфер содержит четыре 32-разрядных регистра, направление передачи информации между которыми показано на рис. 5.2. Устройство управления опережающей выборкой (УУОВ) использует один регистр для записи, а другой — для чтения. Поскольку обрабатываемые объектные Пролог-команды имеют различную длину, указатель записи и считывания в кольцевом буфере изменяется в процессе работы. Однозначно идентифицирует команду и определяет ее длину (число байт) и формат (см. рис. 4.1) код операции, который содержится в реги-

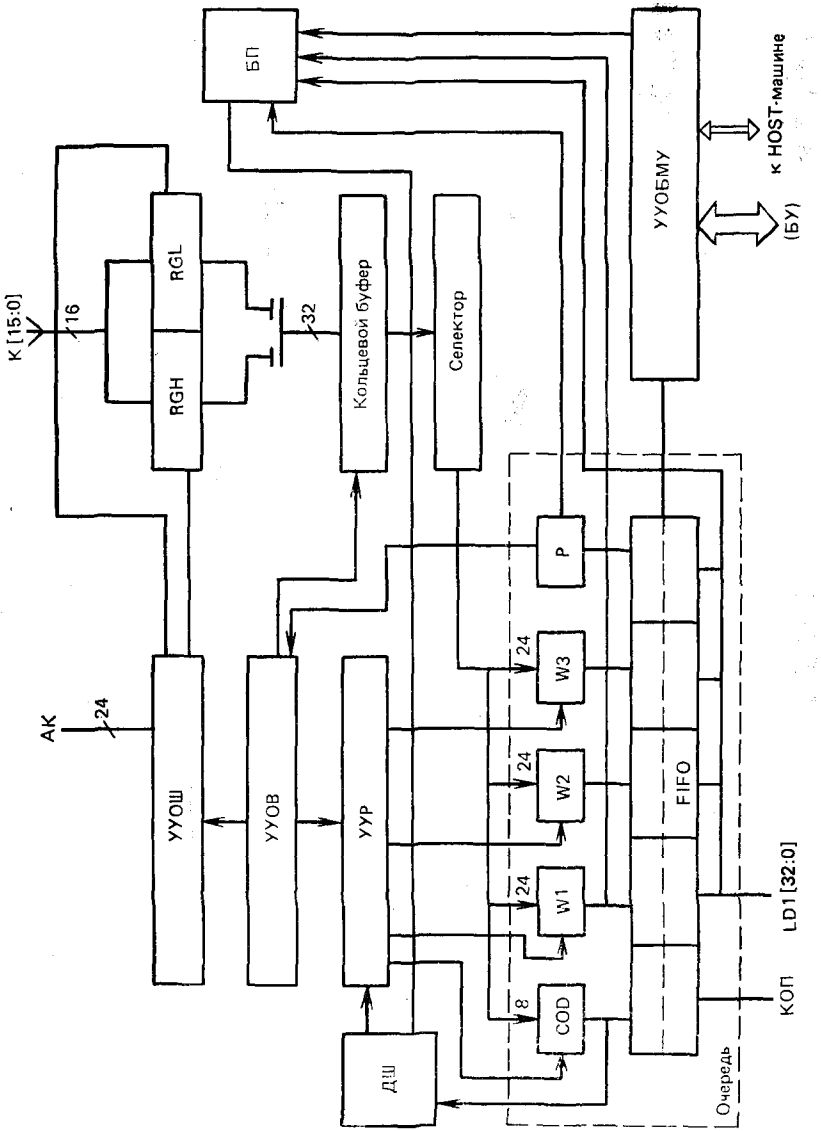


Рис. 5.1. Структурная схема процессора команд

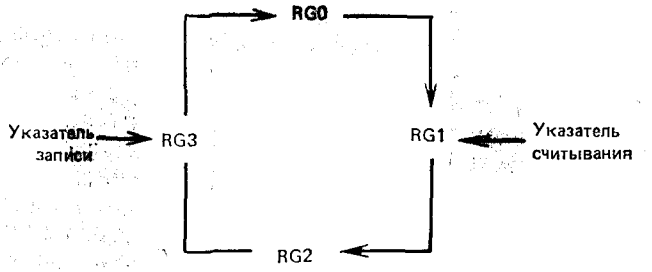


Рис. 5.2. Схема заполнения регистров кольцевого буфера

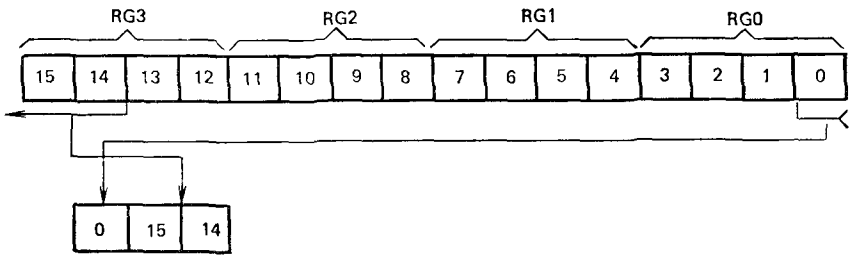


Рис. 5.3. Схема работы селектора

стре COD. Формирование длины команды, количества и размера операндов в байтах (регистры W1—W3), выделение команд по коду операции (КОП) выполняет дешифратор (ДШ).

Выборку кода операции и трех байтов данных из регистров кольцевого буфера выполняет селектор (рис. 5.3). Адрес, используемый для выборки команд из основной памяти HOST-процессора, содержит регистр P.

Устройство управления распаковкой (УУР) управляет очередью FIFO для распакованных команд.

Блок переходов предназначен для формирования значения счетчика команд, адресов безусловных переходов и загрузки счетчика адреса этими значениями.

Управление процессором команд осуществляет устройство управления обменом с блоком микропрограммного управления (УУОБМУ), который интерпретирует команды, выдаваемые блоком управления (БУ).

Операнды команды выдаются на шину LD1, а код операции на выход КОП, выход АК используется для адреса следующей команды.

5.2. ПРОЦЕССОР ПАМЯТИ

Процессор памяти (рис. 5.4) предназначен для управления памятью ПЛВ, включающей L- и G-стеки и кеш-память. Трейловый (TP) стек реализован в основной памяти HOST-машины, поскольку операции с ним выполняются значительно реже, чем операции с L- и G-стеками.

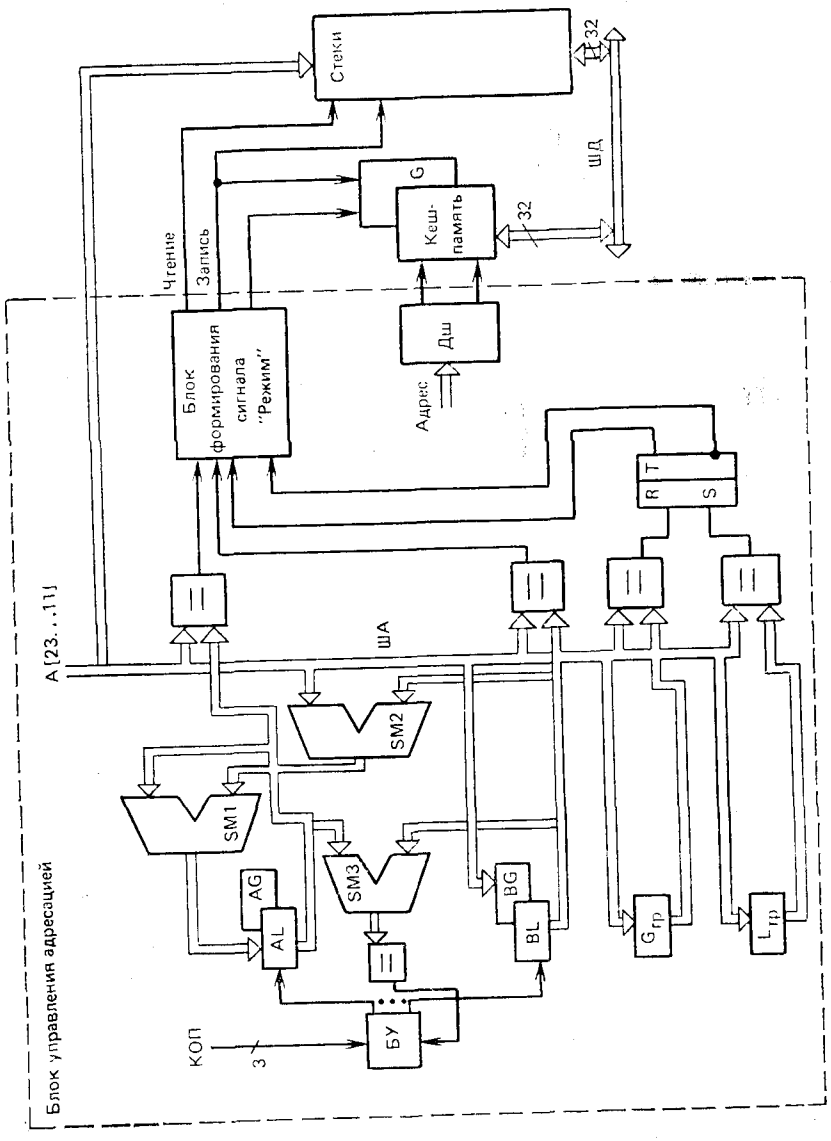


Рис. 5.4. Схема процессора памяти

Для оперативной выборки данных из стеков используются две кеш-памяти — одна для L-стека, вторая — для G-стека. В кеш-памяти отображается «верхушка» соответствующего стека, т. е. область, начальный адрес которой соответствует последней точке выбора в программе. Диапазон адресов, отображаемых в кеш-памяти в текущий момент, называется окном. Окно характеризуется максимальным и минимальным (верхними и нижними) граничными адресами. Процессор памяти реализует операции чтения-записи без перемещения адресного окна или с перемещением. В кеш-памяти отображается только информация текущего элемента трассы логического вывода, соответствующего самой младшей среде процедуры ENV_i. Запись в кеш-память и стеки производится параллельно. Адрес обращения в кеш-память формируется дешифратором.

После инициализации граничные адреса G- и L-стеков заносятся в регистры $G_{гр}$ и $L_{гр}$. Для хранения нижнего и верхнего адресов текущего окна предназначены регистры AL, AG, BL, BG. Сумматор SM1 вычисляет величину сдвига верхней границы окна,

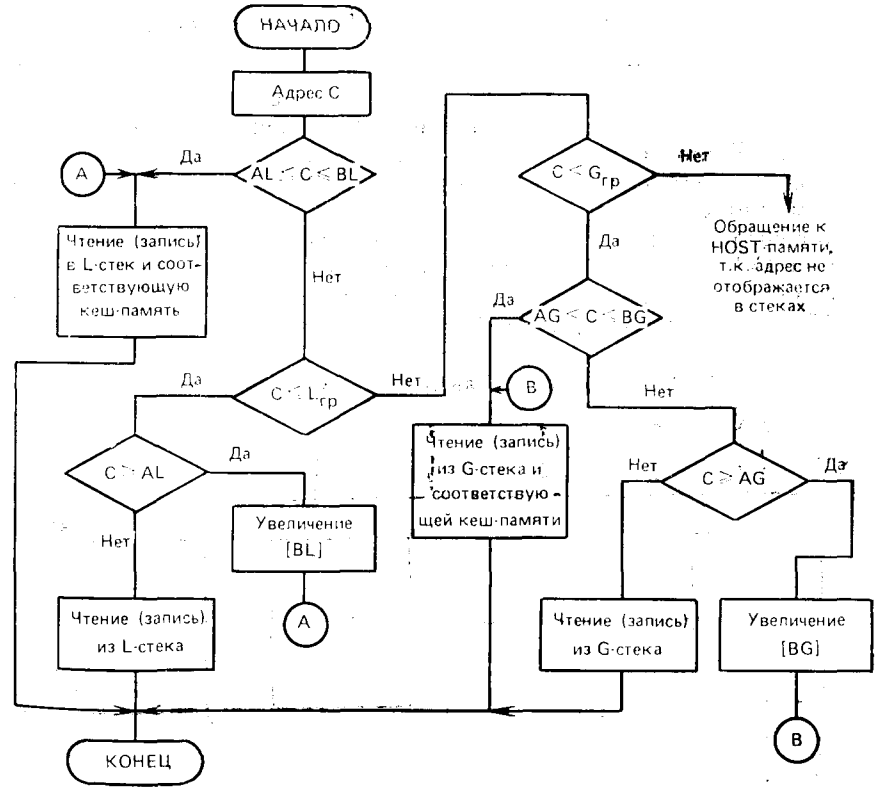


Рис. 5.5. Алгоритм выполнения операции чтения-записи в процессоре памяти

а сумматор SM2 — адреса, определяющие нижнюю границу окна, после каждого заполнения кеш-памяти. Сравнение верхних и нижних границ окна G- и L-стеков с текущим адресом и граничных адресов, разделяющих G-стек от L-стека и L-стек от TR-стека, с текущим адресом организуют четыре параллельно соединенных компаратора.

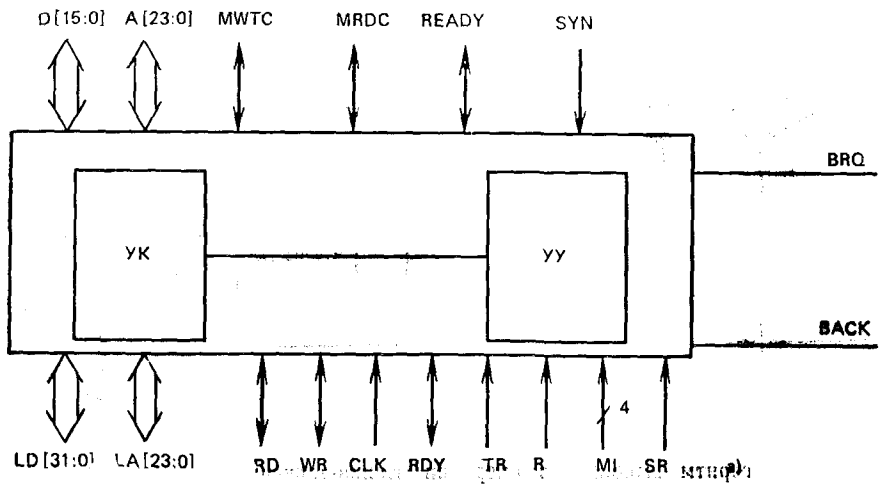
В зависимости от признаков, вырабатываемых компараторами при реализации алгоритма доступа к памяти (рис. 5.5), блок формирования сигнала «Режим» выполняет подачу сигналов чтения-записи. Выдаваемый на шину адрес C сравнивается с граничными адресами окна в кеш-памяти, соответствующей L-стеку. Если адрес принадлежит L-стеку (истинно условие $C \leqslant L_{гр}$, но не отображается в кеш-памяти), значение верхнего граничного адреса BL должно быть увеличено. Если условие $C \geqslant AL$ не выполняется, выданный адрес является ссылкой на старшую среду ENV_j логического вывода, поэтому выполняется только операция чтения-записи над L-стеком. Логика фрагмента схемы на рис. 5.5, относящегося к G-стеку, аналогична.

5.3. ПРОЦЕССОР ВВОДА-ВЫВОДА

Передачу данных с шины HOST-машины (типа MULTIBUS) на внутреннюю шину ПЛВ обеспечивает процессор ввода-вывода — когда ПЛВ пассивен, управляет передачей HOST-процессор, когда HOST-процессор пассивен, — ПЛВ. Таким образом, процессор ввода-вывода (ПВВ) образует единый блок памяти из памяти HOST-машины и памяти ПЛВ.

Процессор ввода-вывода (рис. 5.6,*а*) содержит устройство коммутации (УК) и устройство управления (УУ).

Устройство коммутации обеспечивает коммутацию данных с одной шины на другую, устройство управления осуществляет уп-



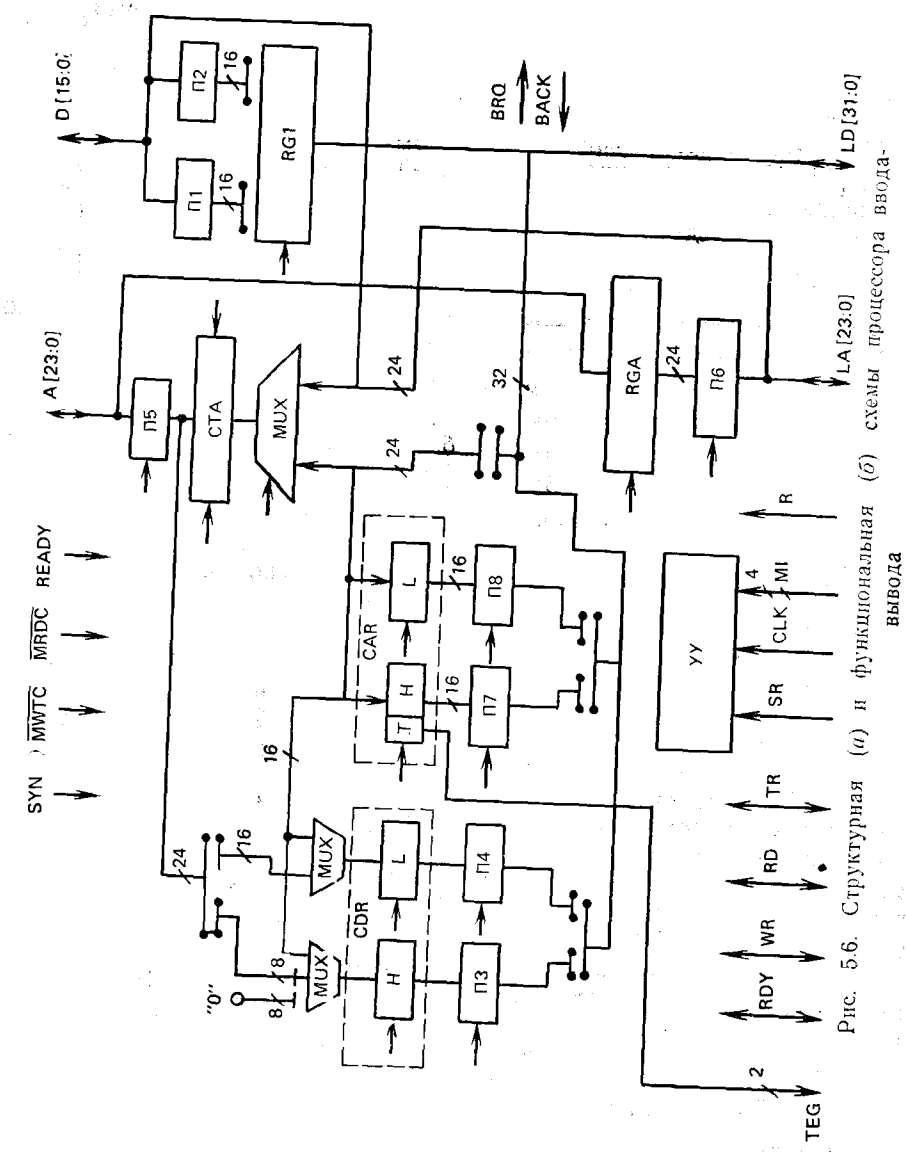


Рис. 5.6. Структурная (*а*) и функциональная (*б*) схемы процессора ввода-вывода

равление ПВВ в соответствии с поступающими управляющими сигналами. Назначение входов и выходов ПВВ следующее:

D[15:0] — двунаправленная шина данных, подключаемая к шине MULTIBUS;

A[23:0] — двунаправленная шина адреса, подключаемая к шине MULTIBUS;

MRDC, MWTC, READY — двунаправленные сигналы чтения, записи и готовности с шины MULTIBUS;

SYN — тактовый сигнал от шины MULTIBUS;

LD[31:0] — двунаправленная шина данных, подключаемая к внутренней шине ПЛВ;

LA[23:0] — двунаправленная шина адреса, подключаемая к внутренней шине ПЛВ;

RD, WR, RDY — двунаправленные сигналы чтения, записи и готовности с внутренней шины ПЛВ;

CLK — входной тактовый сигнал с внутренней шины ПЛВ;

TR — входной сигнал направления передачи;

R — входной сигнал сброса;

MI — входы микрокоманды из блока управления ПЛВ;

SR — вход запуска ПВВ;

BRQ — выходной сигнал запроса на освобождение машины MULTIBUS;

BACK — входной сигнал подтверждения освобождения шины MULTIBUS.

Сигнал TR определяет направление передачи управляющих сигналов (чтение, запись, готовность): 0 — активен HOST-процессор; 1 — активен ПЛВ. Если активен HOST-процессор, то передача информации через ПВВ идет под управлением HOST-процессора. Если активен ПЛВ, то передача идет под управление ПЛВ, УК коммутирует данные и адрес с одной шины на другую. При необходимости передачи информации УУ формирует сигнал BRQ для процессора команд, который освобождает шину и выдает сигнал BACK. Получив подтверждение, что шина HOST-машины свободна, ПВВ начинает обмен данными. После обмена УУ снимает сигнал BRQ, освобождая шину HOST-машины.

Функциональная схема ПВВ представлена на рис. 5.6,*б*. Регистр RG1 позволяет перейти от 32-разрядной шины к 16-разрядной, выдавая по очереди младшее и старшее полуслово. Регистры CDR и CAR позволяют перейти от 16-разрядной шины к 32-разрядной, записывая в них по очереди младшее (L) и старшее (H) полуслово. Из старшего полуслова регистра CAR выведены два разряда тега для анализа в блоке микропрограммного управления. Для фиксирования и инкрементирования адреса при передаче данных через ПВВ под управлением ПЛВ служит счетчик СТА.

Для хранения адреса при передаче данных через ПВВ под управлением HOST-процессора — регистр RGA. Передатчики Пi имеют три состояния на выходе.

Процессор обработки (ПО) предназначен для выполнения арифметических и логических операций, вычисления адреса данных и команд, а также для выполнения встроенных предикатов.

В процессе обработки (рис. 5.7) арифметические и логические операции над данными выполняет арифметическо-логическое устройство (АЛУ). Блок РОН (регистровый файл), представляющий собой двухпортовую память, позволяет одновременно считывать и записывать данные с двух направлений. Исключительной ситуацией является случай, когда данные записываются одновременно с двух направлений в одну и ту же ячейку памяти. В этом случае результат операции не определен. Блок РОН позволяет хранить 36-разрядные данные — 32 разряда информационных и 4

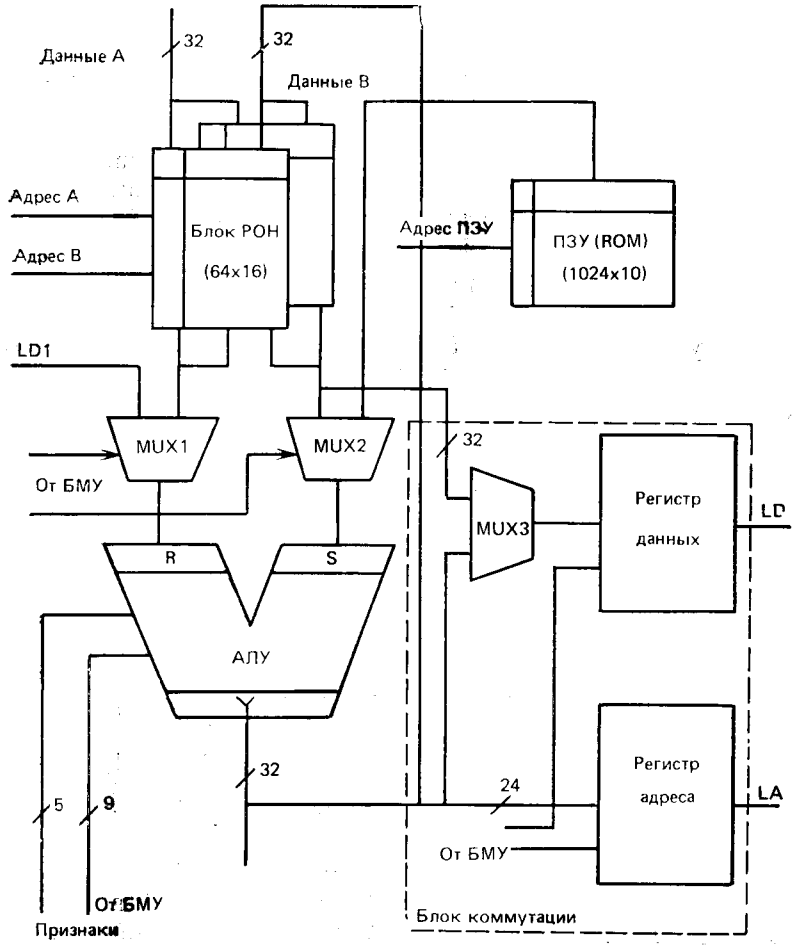


Рис. 5.7. Схема процессора обработки

разряда контроля на четность-нечетность каждого байта. Константы, которые применяются при вычислении адреса подпрограмм, а также при инкрементировании и декрементировании содержимого блока РОН хранятся в ПЗУ.

Данные для обработки в АЛУ могут поступать с регистрового файла или с внутренней шины данных LD ППВ, а также от процессора команд и ПЗУ констант. Результат операции записывается в один из регистров блока РОН или выдается на шины данных (LD) и адреса (LA). Код операции, которую необходимо выполнить, подается из регистра микрокоманды блока управления ПЛВ. После выполнения операции АЛУ вырабатывает признаки завершения операции, которые анализируются блоком управления ПЛВ при организации условных переходов. Кроме кода операции на АЛУ подаются сигналы синхронизации, выбора разрядности входных данных, разрешения выхода, выбора позиции, задающие режим работы АЛУ.

Выбор источника исходных данных для АЛУ выполняют мультиплексоры MUX1, MUX2, управляемые также блоком управления ПЛВ.

В процессоре обработки предусмотрена возможность непосредственно записывать данные с шины LD в регистровый файл и с регистрового файла на шину LD, что позволяет повысить быстродействие.

5.5. БЛОК УПРАВЛЕНИЯ

Блок управления (рис. 5.8) предназначен для выработки сигналов управления всеми ресурсами ПЛВ.

Дешифратор блока управления преобразует код операции в начальный адрес микропрограммы, реализующей эту операцию. Физически дешифратор представляет собой ПЗУ с 8 адресными входами и 16 выходами данных. В качестве адреса используется код операции.

Последовательностью выборки микрокоманд из памяти микропрограмм управляет секвенсор, формируя адрес микрокоманды в зависимости от входных условий и состояния управляющих входов. Секвенсор имеет два адресных 16-разрядных входа A и D, причем D может использоваться и для вывода информации из секвенсора. На вход A подается адрес из дешифратора КОП, вход D используется при организации безусловных либо условных переходов. Секвенсор анализирует множество входных признаков, вырабатываемых блоками ПЛВ и основным процессором. Внутренний стек секвенсора емкостью 33 16-разрядных слова используется при вызовах подпрограмм, организации повторений одного и того же фрагмента микропрограммы и обработке прерываний. Есть возможность увеличить глубину стека путем выгрузки старого содержимого стека через шину D с запоминанием во внешней памяти и заполнения стека новым содержимым.

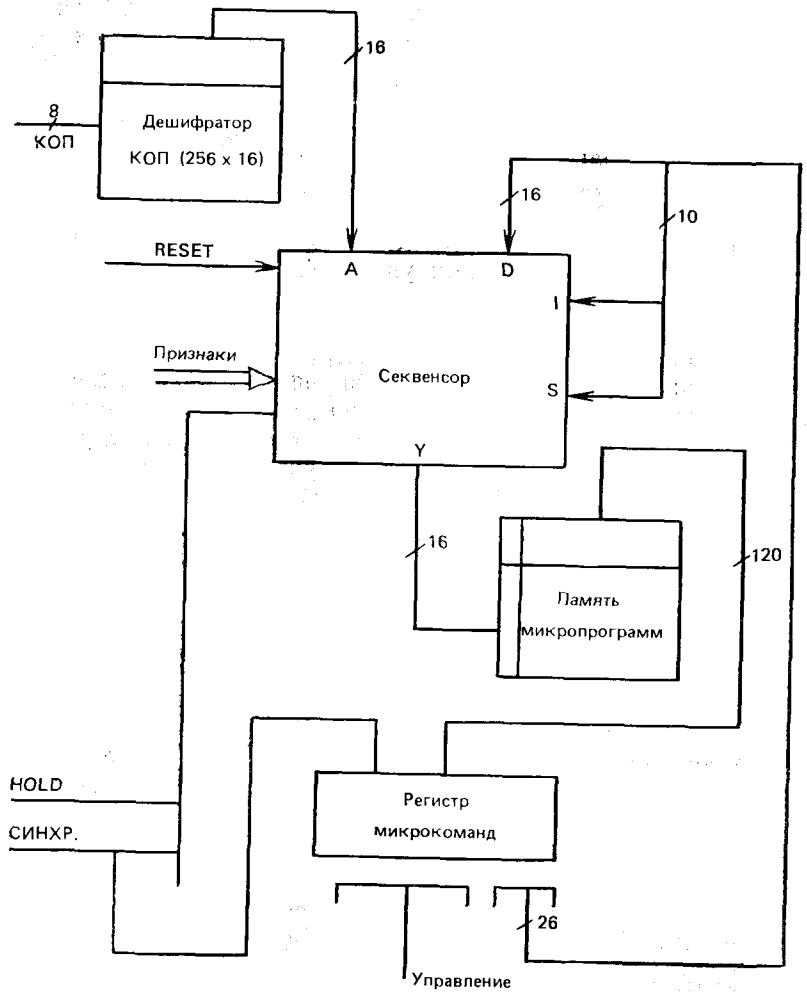


Рис. 5.8. Схема блока управления

Память микропрограмм — это ПЗУ с 16-разрядным адресным входом и 120-разрядным выходом данных. Максимальная ее емкость не должна превышать 64К.

Сигналы управления подаются на все блоки ПЛВ, в том числе и на секвенсор. Блок управления начинает работать по внешнему сигналу RESET, который приводит все блоки ПЛВ в исходное состояние. После сброса выдается нулевой адрес и считывается первая микрокоманда, которая обычно является микрокомандой микропрограммы начальной инициализации. Считанная микрокоманда содержит поле управления секвенсором, определяющее команду, которую он должен выполнить в следующем такте, а также поля, управляющие блоками ПЛВ.

При работе программы начальной инициализации производится настройка ресурсов ПЛВ на выполнение конкретных функций, после чего ПЛВ готов к приему Пролог-команды. Необходимые сигналы для ее считывания вырабатываются также блоком управления. Считанная команда поступает на вход дешифратора, в котором код операции преобразуется в начальный адрес микропрограммы, реализующей данную команду. Полученный адрес поступает на вход A секвенсора и выдается на выход Y секвенсора как адрес первой микрокоманды микропрограммы. После выполнения микропрограммы блок управления готов к приему следующей Пролог-команды, и процесс повторяется. При возникновении ситуации условного перехода секвенсор анализирует входные признаки.

Блок управления может синхронизировать работу с внешними активными устройствами, т. е. освобождать внешние шины данных и адреса по требованию извне. Для этих целей предусмотрен вход HOLD. При этом внутреннее состояние секвенсора сохраняется до тех пор, пока на входе HOLD сигнал высокого уровня. При сбросе сигнала HOLD секвенсор возобновляет свою работу.

Для выбора проверяемого условия предназначены входы секвенсора S; входы I являются управляющими входами секвенсора.

5.6. ПРОЦЕССОР УНИФИКАЦИИ *

Структурная схема

Процессор унификации (ПУ) в составе ПЛВ предназначен для аппаратной реализации команд унификации. Так как 70% логического вывода занимают операции по унификации двух термов, ПУ позволяет повысить эффективность (производительность) ПЛВ. Каждый процедурный вызов начинается с унификации аргументов цели с аргументами заголовка вызываемого утверждения.

Процессор унификации имеет следующие основные блоки (рис. 5.9):

REGISTERS — блок РОН;
SM — сумматор; COMP0—COMP3 — компараторы;
RGPSW — регистр состояния;
ROM — постоянное запоминающее устройство;
RGA и RGB — выходные буферные регистры;
RGI — входной регистр команд;
CU — устройство управления.
Внешние входы и выходы ПУ:
LDI — 32-разрядная входная шина данных;
LDO — 32-разрядная выходная шина данных;
LDA — 24-разрядная выходная шина адреса;
LMI — 6-разрядная входная шина команд;

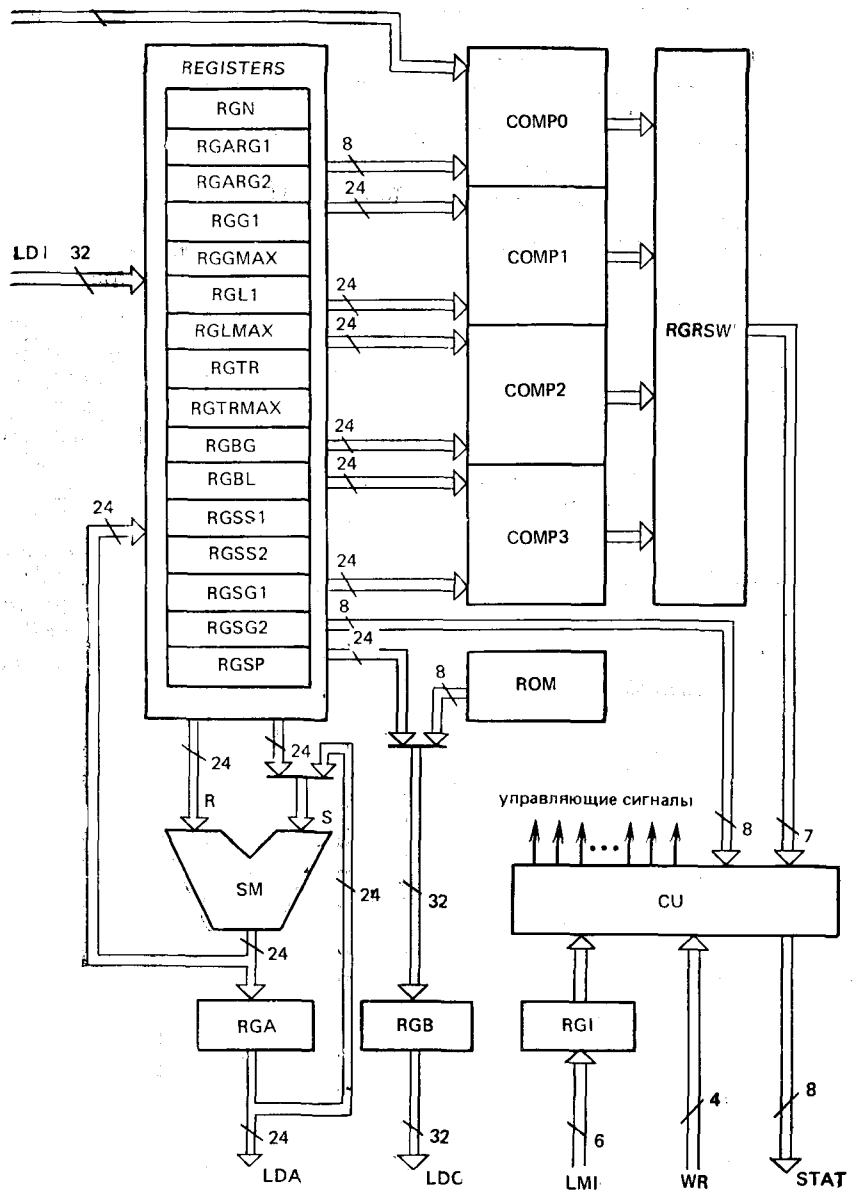


Рис. 5.9. Схема процессора унификации

* В написании этого раздела участвовал В. М. Щурко.

WR — 4-разрядная входная шина стробирования и синхронизации;

STAT — 8-разрядная выходная шина состояния.

Блок РОН содержит 16 регистров, каждый из которых имеет определенное назначение.

Регистры RGARG1 и RGARG2 — регистры унифицируемых аргументов, предназначены для приема и хранения тегированных значений (рис. 5.10), поступающих в ПУ с внешних входов LDI[0:31].

Регистры RGBL и RGBG — 24-разрядные, хранят копии содержимого одноименных регистров ПЛВ — ближайшую точку выбо-

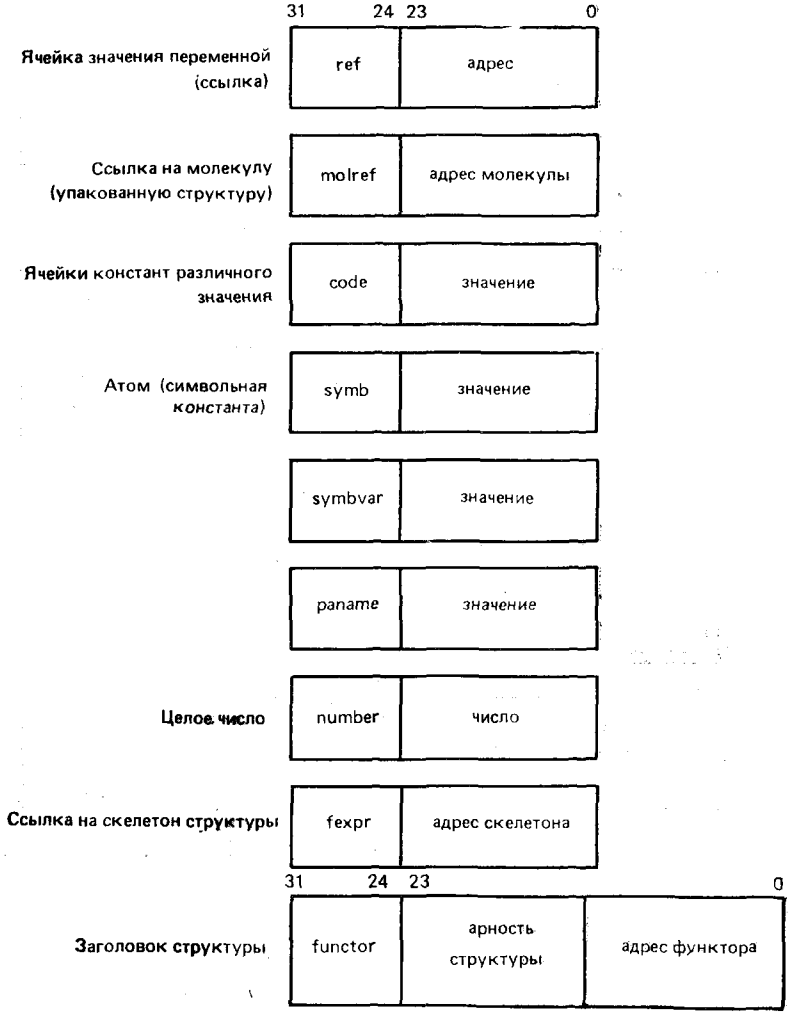


Рис. 5.10. Виды тегированных значений (symbvar соответствует global, local, undef)

ра в локальном и глобальном стеках соответственно. Эти регистры используются для выработки признаков записи в трейловый стек (поддерживается в основной памяти) адресов переменных, получивших значение при унификации.

Регистры RGGMAX, RGLMAX, RGTRMAX хранят значения максимального верхнего адреса глобального, локального и трейлового стеков соответственно (рис. 5.11).

Группа регистров RGL1, RGG1, RGTR хранит текущие указатели на верхушки локального, глобального и трейлового стеков соответственно.

Регистры RGSS1, RGSS2 хранят текущие указатели на скелетоны унифицируемых структур.

Регистры RGSG1, RGSG2 хранят адреса векторов переменных, входящих в скелетоны унифицируемых структур.

Регистр RGSP содержит указатель унификационного стека, который размещается в области локального стека.

Регистр RGN содержит текущий номер пары аргументов — относительный адрес в скелетоне структуры.

Регистр RGA содержит адрес тегированного значения, считанного из памяти в регистры RGARG1, RGARG2.

Регистры RGARG1 и RGARG2 — 32-разрядные, регистр RGN — 8-разрядный, все остальные — 24-разрядные.

Блок РОН является двухпортовым по записи. Во все регистры возможна запись информации входов LDI по командам загрузки

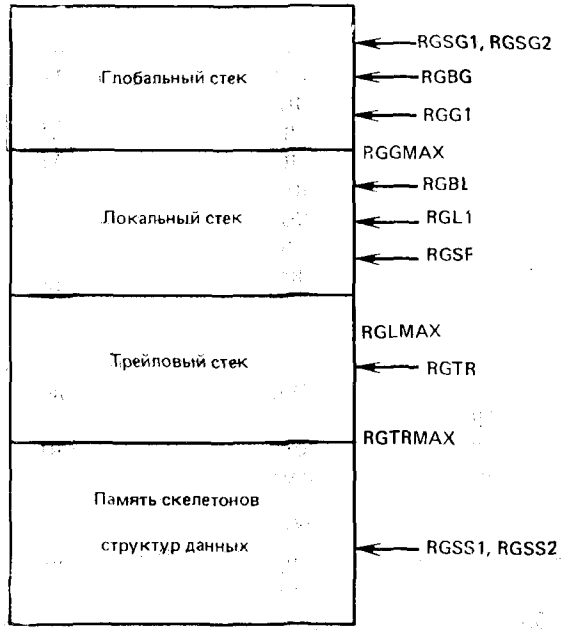


Рис. 5.11. Схема указателей в стеках

регистров, а в некоторые регистры — в процессе выполнения команд унификации. При этом информация в регистры RGN и RGSS1 принимается одновременно (для первого используются разряды [24 : 31] LDI, для второго — разряды [0 : 23]). При записи информации с шины LDI в остальные 24-разрядные регистры разряды [24 : 31] не используются. Возможна запись результата с выхода сумматора SM в регистры RGN, RGARG1, RGARG2, RGTR, RGG1 и RGSP в процессе выполнения команд унификации.

Выходы блока РОН коммутируются на двухвходовые компараторы COMP0—COMP3 и на сумматор SM. На компаратор COMP0 коммутируются теги унифицируемых термов, хранящиеся в регистрах RGARG1 и RGARG2, на COMP1 — выходные сигналы регистров RGARG1, RGARG2, RGG1 (первый вход) и RGGMAX (второй вход), на COMP2 — регистров RGBG, RGTR (первый вход) и RGARG1, RGARG2, RGTRMAX (второй вход), на COMP3 — регистров RGARG1, RGARG2, RGSP (первый вход) и RGBL, RGL1, RGLMAX (второй вход). На вход S сумматора SM коммутируются сигналы регистров RGSS1, RGSS2, RGSG1, RGSG2, на вход R — остальные. Сумматор SM используется для вычисления адресов при записи (чтении) информации в (из) памяти, а также для сравнения содержимого RGA с содержимым регистров указателей в стеках. Сравнение выполняется путем выполнения операции вычитания на сумматоре SM с формированием признака равенства нулю и знака результата.

Компараторы COMP1—COMP3 формируют признаки операндов (A>B, A=B, A<B), а COMP0 — признак совпадения (A==B) тегов. Признаки поступают в регистр состояния RGPSW. Регистр состояния ПУ включает следующие триггеры:

TEQT — признак совпадения тегов;
TEMP1 — признак пустой переменной в RGARG1;
TEMP2 — признак пустой переменной в RGARG2;
TLF — признак переполнения локального стека;
TGF — признак переполнения глобального стека;
TTF — признак переполнения трейлового стека;
TLE — признак очищения унификационного стека (при RGL1=RGSP), т. е. уровень вложенности структур равен нулю;
TTR1 — признак необходимости записи в трейловый стек пустой переменной RGARG1;
TTR2 — признак необходимости записи в трейловый стек пустой переменной RGARG2.

Признаки поступают на устройство управления CU и определяют переходы в микропрограммах выполняемых команд.

Базовые операции

Базовые операции представляют собой макрорасширения, используемые в алгоритмах команд ПУ. Их можно рассматривать как крупноблочные «куски», из которых компонуются команды унификации ПЛВ.

Операции с молекулами. Операция создания молекулы:

```
Макровызов:  molekule (RGARGi, RGSGi, RGG1, TEMPi),
             где RGARGi — регистр RGARG1 или RGARG2;
                RGSGi — регистр RGSG1 или RGSG2;
             TEMPi — триггер TEMP1 или TEMP2.
Макрорасширение:
   "mol' : RGSGi —> RGB, RGG1 —> RGA;
           RGB —> LD0, RGA —> LDA, RGARGi —> RGB,
           RGG1+4 —> RGA;
           RGB —> LD0, RGA —> LDA; 'molref': RGG1 — >
           RGARGi;
           RGG1+8 —> RGG1, '0' —> TEMPi;
             выход из макрорасширения.
```

До начала операции создания молекулы в RGARG1 или RGARG2 должны быть загружены тегированные значения вида fexpr[24 : 31]; адрес скелетона [0 : 23]. В регистре RGSG1 или RGSG2 содержится адрес вектора переменных, встречающихся в скелетоне структуры, а в регистре RGG1 — указание на свободное место в глобальном стеке.

Состояние глобального стека до и после выполнения операции molekule показано на рис. 5.12.

Операция распаковки молекулы:

```
Макровызов: decomp (RGARGi, RGSSi, RGSGi), где i=1, 2.
Макрорасширение:
   RGARGi[23: 0] —> RGA;
   RGA —> LDA, LDI —> RGSGi;
   RGARGi[23: 0]+4 —> RGA;
   RGA —> LDA, LDI —> RGSSi;
   выход из макрорасширения.
```

До начала операции регистр RGARG1 или RGARG2 должен содержать тегированное значение вида molref[31 : 24]; адрес молекулы [23 ... 0].

Схема, иллюстрирующая выполнение операции decomp, показана на рис. 5.13.

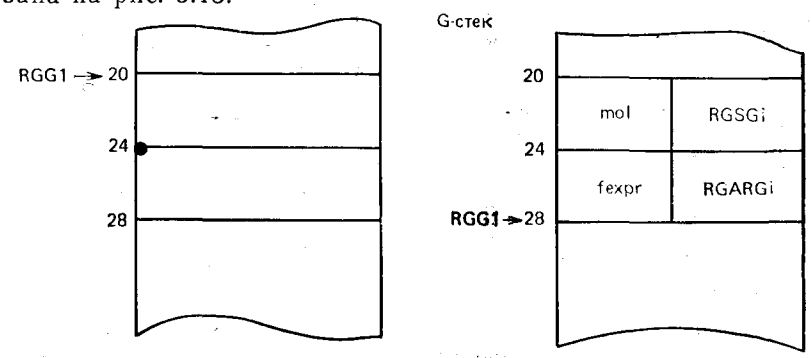


Рис. 5.12. Схема выполнения операции molekule

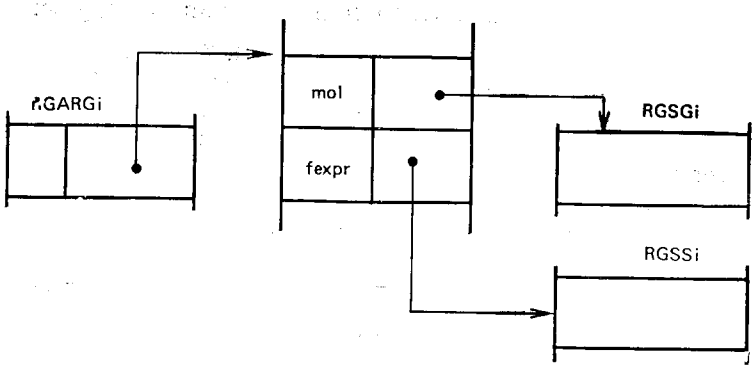


Рис. 5.13. Схема выполнения операции decomp

Операция проверки равенства функторов унифицируемых структур:

Макровызов: equal-functors (RGSS1, RGSS2).
Макрорасширение: RGSS1 —> RGA; RGA —> LDA, LDI —> RGARG1;
RGSS2 —> RGA; RGA —> LDA, LDI —> RGARG2;
если (RGARG1[31:0] = RGARG2[31:0])
то выход из макрорасширения;
иначе выработать код завершения "ПРОВАЛ".

До выполнения этой операции регистры RGSS1, RGSS2 должны быть загружены указателями на скелетоны унифицируемых структур (рис. 5.14).

Процессор унификации выдает код завершения выполнения команды (табл. 5.1) на выходы STAT.

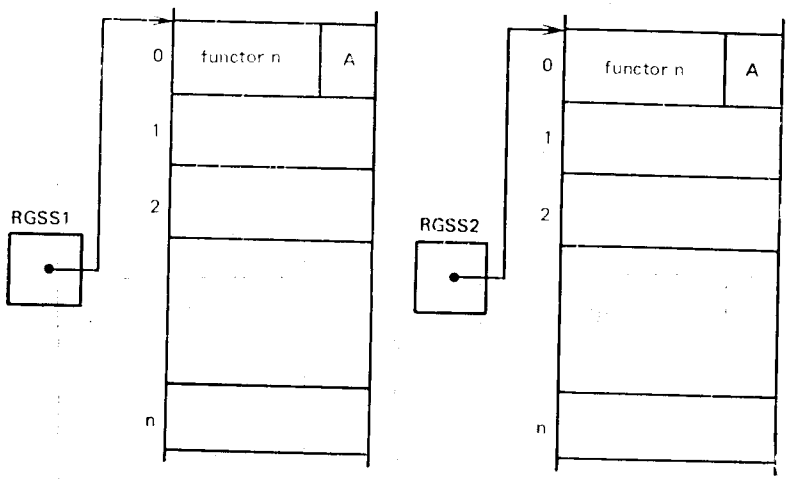


Рис. 5.14. Представление скелетонов унифицируемых структур

Таблица 5.1

STAT 2	STAT 1	STAT 0	Мнемоника	Содержание
0	0	0	УСПЕХ	Унификация завершилась успешно
1	1	1	ПРОВАЛ	Унификация «провалилась», т. к. термы не унифицируемы
0	0	1	ПЕР. ГС	Переполнение глобального стека
0	1	0	ПЕР. ЛС	Переполнение локального стека
0	1	1	ПЕР. ТС	Переполнение трейлового стека
1	0	0	ОШ. ТЕГ	Неправильный тег

Операция вычисления значения аргумента унифицируемой структуры:

Макровызов: SS_RGARGi (RGARGi, RGSSi, TEMPi, RGSGi), где i=1,2.
Макрорасширение:
RGSSi+RGN —> RGA;
RGA —> LDA; LDI —> RGARGi;
ЕСЛИ RGARGi[31 : 24]='global',
ТО RGSGi+RGARGi —> RGA;
RGA —> LDA: LDI —> RGARGi;
{ЕСЛИ RGARGi[31 : 24]='ref',
ТО DEREF(RGARGi, RGA, TEMPi);
ИНАЧЕ '0' —> TEMPi;}
выход из макрорасширения;
{ЕСЛИ RGARGi[31 : 24) = ('code'∨'symb'∨'number'∨
'symbvar'∨'paname'∨'fexpr'),
ТО '0'— —> TEMPi;
выход из макрорасширения;}
ИНАЧЕ 0 —> TEMPi;
выдать код завершения "ОШ.ТЕГ".

До начала выполнения этой операции в регистр RGSS1 или RGSS2 должен быть загружен указатель на скелетон структуры, а в регистр RGN — номер (относительный адрес) элемента структуры (рис. 5.15).

Макрорасширение заключается в выполнении чтения элемента скелетона (который определяется по номеру в регистре RGN) и проверке тега считанного элемента. Если тег соответствует глобальной переменной global, то выполняется чтение из вектора переменных в G-стеке значения этой глобальной переменной. Затем проверяется, является ли считанное значение адресной ссылкой (т. е. содержит ли оно тег ref). При положительном результате проверки определяется значение переменной с помощью операции дереференсирования (раскрутки) адресных ссылок. В противном случае выполняется выход из макрорасширения, поскольку в регистре RGARGi находится значение требуемой ячейки вектора глобальных переменных, т. е. им может быть константа, атом,

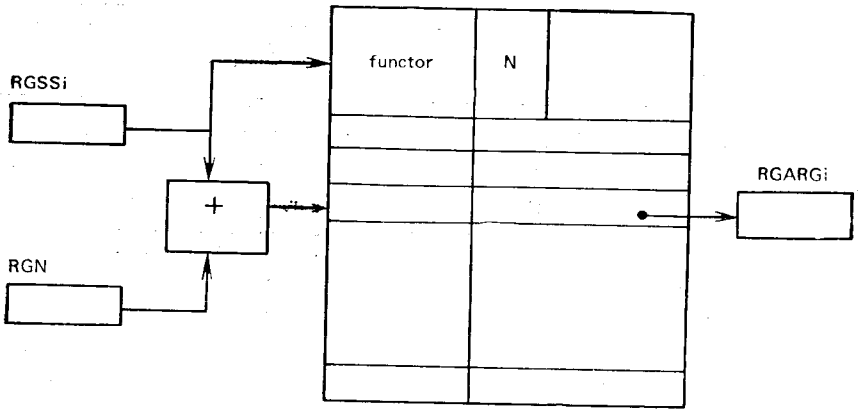


Рис. 5.15. Схема вычисления адреса аргумента структуры

ссылка на молекулу или структурированный терм в области данных.

Вложенные условные проверки в тексте макрорасширения выделены фигурными скобками.

Операция дереференсирования переменной:

```
Макровызов: DEREF(RGARGi, RGA, TEMPi, TTRi), где i=1,2.
Макрорасширение:
1: {ЕСЛИ (RGARGi[23 : 0]=RGA)&(RGARGi[23 : 0]<RGBL)
        ТО
   ЕСЛИ RGARGi[23 : 0]>RGGMAX
        ТО "1" —> TTRi;
   ИНАЧЕ
   ЕСЛИ RGARGi[23 : 0]<RGBG
        ТО "1"—>TTRi;
   ИНАЧЕ
        "0"—> TTRi;
   выход из макрорасширения;}
   ЕСЛИ (RGARGi[23 : 0]=RGA)&(RGARGi[23 : 0]>=RGBL)
        ТО "1"—> TEMPi, 0 —> TTRi;
   выход из макрорасширения;
   {ЕСЛИ RGARGi[23 : 0]<RGA
      ТО RGARGi[23 : 0]—>RGA;
        RGA—>LDA;
        LDI—>RGARGi;
   ЕСЛИ RGARGi[31 : 24]="ref"
        ТО переход на 1;
   ЕСЛИ (RGARGi[31 : 24]="molref"\/"code"\/"symb"\/"number"
\/"symbvar"\/"paname")
        ТО 0—>TEMPi;
           выход из макрорасширения;
   ЕСЛИ RGARGi[31 : 24]=/=(""molref"\/"code"\/"symb"\/"number"
\/"symbvar"\/"paname")
        ТО 0—> TEMPi;
   выдать код завершения "ОШИБКА ТЕГА";}
   ЕСЛИ RGARGi[23 : 0]>RGA
        ТО 0—> TEMPi;
   выдать код завершения "ОШИБКА ЗНАЧЕНИЯ".
```

Перед выполнением операции регистр RGARG1 или RGARG2 должен содержать аргумент с тегом 'ref', а регистр RGA — адрес этого аргумента. Суть операции дереференсирования переменной состоит в прохождении по цепочке ссылок до тех пор, пока считанное значение не будет иметь тег, отличный от 'ref', либо не окажется значением пустой переменной.

Если в результате дереференсирования будет получена пустая переменная, то производится также проверка того, лежит ли ячейка пустой переменной в области стека "выше" ближайшей точки возврата (рис. 5.16). Если ячейка пустой переменной лежит в заштрихованной области стека, то триггер TTRi устанавливается в состояние 1. Это означает следующее. Если при унификации эта переменная получит какое-либо значение, то адрес ее (ячейки в стеке) необходимо записать в трейловый стек. При возврате все переменные, получившие значения и не попавшие в область возврата, должны стать пустыми, адреса этих переменных находятся в трейловом стеке. Условие установки триггера TTRi:

$$(RGGMAX < RGARGi\ [23:0] < RGBL)$$

или

$$(RGARGi\ [23:0] < RGBG).$$

Условие (RGARGi[23:0]=RGA) означает, что адрес переменной совпадает с содержимым ячейки с этим адресом, а это значит, что переменная пустая. Заштрихованные участки на рис. 5.16 определяют текущий элемент трассы логического вывода ENV, отображаемый в L- и G-стеках. Определение попадания адресной ссылки в заштрихованную область выполняется в первом блоке, ограниченном фигурными скобками.

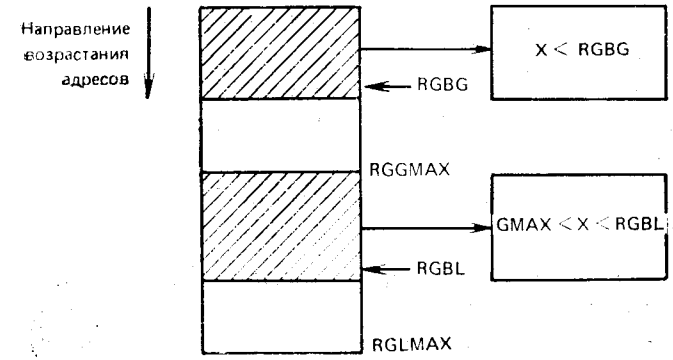


Рис. 5.16. Область стека выше ближайшей точки возврата

Условие (RGARGi[23 : 0] > RGA) означает адресную ссылку за пределы текущего элемента трассы в область больших адресов. Поскольку там находится «мусор», то выдается код «Ошибка значения». Ситуация, обрабатываемая во втором блоке, ограниченном фигурными скобками, соответствует допустимой адресной ссылке. Выполняется чтение содержимого ячейки с данным адресом. Если содержимое ячейки — снова адресная ссылка, то реализуется рекурсивный переход на метку 1. В противном случае анализируется тег прочитанного значения. При недопустимом теге выдается код «Ошибка тега».

Следующая операция дереференсирования аргумента использует предыдущую, однако выполняется при иных исходных условиях. Сначала анализируется тип аргумента путем выделения его тега. Если аргумент является адресной ссылкой, то выполняется чтение по этому адресу, а затем либо операция дереференсирования, либо выход из макрорасширения. Если аргумент не является адресной ссылкой, то выполняется блок макрорасширения, начинающийся с метки 1.

Операция дереференсирования аргумента:

Макровызов: DEREF_ARG(RGARGi, TEMPi, TTRi), где i=1,2.
Макрорасширение:
ЕСЛИ RGARGi[31 : 24] = 'ref'
ТО RGARGi[23 : 0] —> RGA;
RGA —> LDA; LDI —> RGARGi;
ЕСЛИ RGARGi[31 : 24] = 'ref'
ТО DEREF(RGARGi, RGA, TEMPi, TTRi);
выход из макрорасширения;
ИНАЧЕ переход к 1;
1: ЕСЛИ RGARGi[31 : 24] = ('molref'∨'code'∨'symb'∨'number'∨
'symbvar'∨'paname')
ТО '0' —> TEMPi;
выход из макрорасширения;
ЕСЛИ RGARGi[31 : 24] =/= ('ref'∨'molref'∨'code'∨'symb'∨'number'∨
'symbvar'∨'paname')
ТО '0' —> TEMPi;
выдать код завершения "ОШИБКА ТЕГА".

Перед выполнением операции регистр RGARG1 или RGARG2 должен содержать тегированное значение.

Операция записи в стек данных с протоколированием в трейловом стеке:

Макровызов: TR_WRITE(RGARGi, RGARGj), где i=1 и j=2 или i=2 и j=1.
Макрорасширение:
RGARGj[23 : 0] —> RGA, RGARGi —> RGB;
RGA —> LDA, RGB —> LD0;
RGTR —> RGA, RGARGj —> RGB;
RGA —> LDA, RGB —> LD0;
RGTR+4 —> RGTR
ЕСЛИ RGTR > RGTRMAX
ТО выдать код завершения "ПЕРЕПОЛНЕНИЕ ТРЕЙЛОВОГО СТЕКА"
ИНАЧЕ выход из макрорасширения.

Данная операция выполняет запись содержимого регистра RGARG1 (RGARG2) по адресу, находящемуся в RGARG2 (RGARG1), в трейловый стек.

Операция записи в стек данных без протоколирования в трейловом стеке:

Макровызов: WRITE(RGARGi, RGARGj), где i=1 и j=2 или i=2 и j=1.
Макрорасширение:
RGARGi[23 : 0] —> RGA, RGARGi —> RGB;
RGA —> LDA, RGB —> LDO;
выход из макрорасширения.

Данная операция аналогична предыдущей операции, за исключением фазы протоколирования в трейловом стеке.

Операция записи в унификационный стек:

Макровызов: PUSH_STACK.
Макрорасширение:
RGSP —> RGA, RGN‖RGSS1 —> RGB;
RGA —> LDA, RGB —> LDO, RGSP+4 —> RGSP;
RGSP —> RGA, RGN‖RGSG1 —> RGB;
RGA —> LDA, RGB —> LDO; RGSP+4 —> RGSP;
RGSP —> RGA, RGN‖RGSS2 —> RGB;
RGA —> LDA, RGB —> LDO, RGSP+4 —> RGSP;
RGSP —> RGA, RGB —> LDO;
ЕСЛИ RGSP ≥ RGLMAX
ТО выдать код завершения "ПЕРЕПОЛНЕНИЕ ЛОКАЛЬНОГО СТЕКА"
ИНАЧЕ выход из макрорасширения.

Данная операция выполняет запись текущего содержимого регистров RGSS1, RGSS2, RGSG1, RGSG2, RGN в унификационном стеке. Нижняя граница стека определяется содержимым регистра RGL1, указатель стека в регистре RGSP. Так как ячейка стека 32-разрядная, а регистр RGN 8-разрядный, то в стек записывается пара RGN‖RGSS1 (или RGSS2, RGSG1, RGSG2); ‖ — операция сцепления содержимого регистров.

Операция чтения из унификационного стека:

Макровызов: POP_STACK.
Макрорасширение:
RGSP—4 —> RGSP, RGA;
RGA —> LDA, LDI —> RGSG2;
RGSP—4 —> RGSP, RGA;
RGA —> LDA, LDI —> RGSS2;
RGSP—4 —> RGSP, RGA;

```
RGA —> LDA, LDI —> RGSG1;
RGSP—4 —> RGSP, RGA;
RGA —> LDA, LDI —> RGN‖RGSS1;
 ЕСЛИ RGSP=RGL1 ТО "1" —> TLE;
ИНАЧЕ "0" —> TLE;
выход из макрорасширения.
```

Данная операция выполняет чтение содержимого регистров RGSS1, RGSS2, RGSS2, RGSG1, RGSG2, RGN.

Алгоритм унификации

Приведенный далее алгоритм унификации рассматривается как операция унификации двух произвольных термов, которая называется операцией общей унификации (general_unify). Для его реализации необходимы следующие условия и признаки, вырабатываемые ПУ:

TEMP1, если RGARG1 содержит указатель ячейки пустой переменной (базовая операция DEREF(RGARG2, RGA, TEMP1, TTR1));

TEMP2, если регистр RGARG2 содержит указатель ячейки пустой переменной (базовая операция DEREF(RGARG2, RGA, TEMP2, TTR2));

TEQT, если теги значений RGARG1 и RGARG2 равны между собой (устанавливается компаратором COMPO);

TTR1, если необходима операция протоколирования в трейловом стеке RGARG1 (базовая операция DEREF(RGARG1, RGA, TEMP1, TTR1));

TTR2, если необходима операция протоколирования в трейловом стеке RGARG2 (базовая операция DEREF(RGARG2, RGA, TEMP2, TTR2));

TLE (триггер уровня вложения структур), если унификационный стек пуст (устанавливается компаратором при выполнении операции сравнения RGL1=RGSP);

Конец аргументов, если RGN=0.

В выполнении алгоритма участвуют следующие регистры блока РОН ПУ:

RGSS1 — адрес скелетона терма для регистра RGARG1;
RGSS2 — адрес скелетона терма для регистра RGARG2;
RGSG1 — адрес вектора глобальных переменных для регистра RGARG1;
RGSG2 — адрес вектора глобальных переменных для регистра RGARG2;
RGN — смещение (номер) для текущего унифицируемого аргумента скелтона.

Алгоритм общей унификации GENERAL_UNIFY

```
"0" —> RGN, RGL1 —> RGSP;
DEREF_ARG(RGARG1, TEMP1, TTR1);
DEREF_ARG(RGARG2, TEMP2, TTR2);
MATCH: {ЕСЛИ ((TEMP1=1)&(TEMP2=1)&(RGARG1[23 : 0]
```



```
RGARG2[23 : 0])&(TTR2=1)∨((TEMP1=0)&
(TEMP2=1)&(TTR2=1))
ТО TR_WRITE (RGARG1, RGARG2); переход к TERM:}
                                                1
{ЕСЛИ ((TEMP1=1)&(TEMP2=1)&(RGARG1[23 : 0]
RGARG2[23 : 0]&(TTR2=0)∨((TEMP1=0)&
(TEMP2=1)&(TTR2=0))
ТО WRITE(RGARG1, RGARG2); переход к TERM:}
                                                2
{ЕСЛИ ((TEMP1=1)&(TEMP2=1)&(RGARG1[23 : 0]
RGARG2[23 : 0])&(TTR1=1)∨((TEMP1=1)&
(TEMP2=0)&(TTR1=1))
ТО TR_WRITE(RGARG2, RGARG1); переход к TERM:}
                                                3
{ЕСЛИ ((TEMP1=1)&(TEMP2=1)&(RGARG1[23 : 0]
RGARG2[23 : 0]&(TTR1=0)∨((TEMP1=1)&
(TEMP2=0)&(TTR1=0))
ТО WRITE(RGARG2, RGARG1); переход к TERM:}
                                                4
{ЕСЛИ (TEQT=1)&(RGARG1[23 : 0]=RGARG2[23 : 0])&
(TEMP1=0)&(TEMP2=0)
ТО переход к TERM:}
                                                5
{ЕСЛИ (TEQT=1)&(RGARG1[23 : 0]=/=RGARG2[23 : 0])&
(RGARG1[31 : 24]='molref')&(TEMP1=0)&
(TEMP2=0)
ТО
DECOMP:
DECOMP(RGARG1, RGSS1, RGSG1);
DECOMP(RGARG2, RGSS2, RGSG2);
EQUAL_FUNCTORS(RGSS1, RGSS2);
ЕСЛИ (RGN=0)
ТО ВЫДАТЬ КОД ЗАВЕРШЕНИЯ "УСПЕХ"; КОНЕЦ;
ИНАЧЕ
PUSH_STACK; /* Сохранение RGSS1, RGSS2, RGSG1,
RGSG2 в унификационном стеке*/
SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
РЕКУРСИВНЫЙ ВЫЗОВ general_unify ДЛЯ АРГУМЕНТОВ ТЕРМОВ;}
                                                6
(ЕСЛИ (RGARG1[31 : 24]='fexpr')&(RGARG2[31 : 24]=
'molref')&(TEMP1=0)&(TEMP2)=0
ТО PUSH_STACK; RGARG1[23 : 0]—>RGSS1;
DECOMP(RGARG2, RGSS2, RGSG2);
EQUAL_FUNCTORS(RGSS1, RGSS2);
SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
```

```
РЕКУРСИВНЫЙ ВЫЗОВ general_unify ДЛЯ АРГУМЕНТОВ ТЕРМОВ}
                                                        7
      {ЕСЛИ (RGARG1[31 : 24] = 'molref') & (RGARG2[31 : 24] =
      'fexpr') & (TEMP1 = 0) & (TEMP2 = 0)
        ТО PUSH_STACK; RGARG2[23 : 0] —> RGSS2;
      DECOMP(RGARG1, RGSS2, RGSG2);
            EQUAL_FUNCTORS(RGSS1, RGSS2);
            SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
            SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
РЕКУРСИВНЫЙ ВЫЗОВ general_unify ДЛЯ АРГУМЕНТОВ ТЕРМОВ}
                                                        8
      {ЕСЛИ (RGARG1[31 : 24] = 'fexpr') & (RGARG2[31 : 24] =
            'fexpr') & (TEMP1 = 0) & (TEMP2 = 0)
ТО PUSH_STACK;
   RGARG1[31 : 24] —> RGSS1; RGARG2[31 : 24] —> RGSS2;
   EQUAL_FUNCTORS (RGSS1, RGSS2);
   SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
   SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
РЕКУРСИВНЫЙ ВЫЗОВ general_unify ДЛЯ АРГУМЕНТОВ ТЕРМОВ}
                                                        9
      {ЕСЛИ (TEMP1 = 1) & (RGARG2[31 : 24] = 'fexpr') & (TTR1 = 1)
      ТО MOLECULE(RGARG2, RGSG2, RGG1, TEMP2);
   TR_WRITE(RGARG2, RGARG1);
      переход к TERM:}
                       10
{ЕСЛИ (TEMP1 = 1) & (RGARG2[31 : 24] = 'fexpr') & (TTR1 = 0)
 ТО MOLECULE (RGARG2, RGSG2, RGG1, TEMP2);
      WRITE(RGARG2, RGARG1);
    переход к TERM:}
                       11
{ЕСЛИ (RGARG1[31 : 24] = 'fexpr') & (TEMP2 = 1) & (TTR2 = 1)
 ТО MOLECULE(RGARG1, RGSG1, RGG1, TEMP1);
    TR_WRITE(RGARG1, RGARG2);
     переход к TERM:}
                       12
      {ЕСЛИ (RGARG1[31 : 24] = 'fexpr') & (TEMP2 = 1) & (TTR2 = 0)
 ТО MOLECULE(RGARG1, RGSG1, RGG1, TEMP1);
    WRITE(RGARG2, RGARG1);
    переход к TERM:}
                       13
      ИНАЧЕ выдать код завершения "ПРОВАЛ"; КОНЕЦ;
{TERM: {ЕСЛИ (RGN = 1)
          ТО выдать код завершения "УСПЕХ"; КОНЕЦ;}
DECREM: RGN—4—>RGN;
        ЕСЛИ (TLE = 1) "Унификационный стек пуст"
          ТО ЕСЛИ (RGN = 1)
                ТО выдать код завершения "УСПЕХ";
```



```
                ИНАЧЕ SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
                      SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
  РЕКУРСИВНЫЙ ВЫЗОВ general_unify ДЛЯ АРГУМЕНТОВ ТЕРМОВ;
  ИНАЧЕ
    POP_STACK; "Формирование признака TLE"
                      переход к DECREM:
  КОНЕЦ;}
         14
```

В блоке $\{...\}_1$ проверяется условие, что унифицируется — либо две пустые переменные, либо одна из них пустая, а другая нет. Признак TTR2 = 1 указывает, что необходимо записать адреса переменных в трейловый стек, чтобы отменить их значения при возврате.

Блок $\{...\}_2$ аналогичен блоку $\{...\}_1$, с той разницей, что адреса переменных не заносятся в трейловый стек. Это может иметь место, если одна из переменных получила значение в более старшей среде логического вывода ENV.

Блоки $\{...\}_1$ и $\{...\}_3$ аналогичны друг другу; разница в том, что в блоке $\{...\}_1$ адресная ссылка направлена от второго аргумента к первому, а в блоке $\{...\}_3$ — наоборот. Сказанное относится также к паре блоков $\{...\}_2$ и $\{...\}_4$.

В блоке $\{...\}_5$ устанавливается факт равенства двух тегированных значений, имеющих одно и то же значение.

Блок $\{...\}_6$ выполняет унификацию двух структурированных термов, значения которых в момент унификации не определены. Оба терма представлены молекулами. Чтобы их проунифицировать, молекулы сначала распаковываются, а их функторы сравниваются. Если число аргументов термов равно 0, то этого достаточно. В противном случае информация о молекуле заносится в стек унификации и выполняется выборка аргументов и их унификация.

В блоке $\{...\}_7$ выполняется унификация двух структурированных термов, причем второй представлен ссылкой на молекулу, поэтому информация о втором терме заносится в унификационный стек, молекула распаковывается и после установки равенства имен термов извлекается пара первых аргументов обоих термов, для которых рекурсивно вызывается процедура general_unify.

Блок $\{...\}_8$ аналогичен предыдущему, только взаимно изменены роли регистров RGARG1 и RGARG2.

В блоке $\{...\}_9$ обе молекулы распакованы.

В блоке $\{...\}_{10}$ унифицируется свободная переменная (признак TEMP1 = 1) и структурированный терм. Для переменной создается молекула в G-стеке, а информация об аргументах протоколируется в трейловом стеке.

Блок $\{...\}_{11}$ аналогичен предыдущему, но протоколирование не требуется (TTR1 = 0).

Блок $\{...\}_{12}$ аналогичен блоку $\{...\}_{10}$, но со сменой ролей аргументов. Это же касается блоков $\{...\}_{11}$ и $\{...\}_{13}$.

Блок $\{...\}_{14}$ используется для перехода к следующей паре аргументов структурированных термов.

Система команд

GEN_U — выполняет алгоритм общей унификации GENERAL_UNIFY двух произвольных термов, загруженных в регистры RGARG1, RGARG2.

DEREF1 — выполняет дереференсирование аргумента, загруженного в регистр RGARG1:

```
DEREF_ARG(RGARG1, TEMP1, TTR1)
```

DEREF2 — выполняет дереференсирование аргумента, загруженного в регистр RGARG2:

```
DEREF_ARG(RGARG2, TEMP2, TTR2)
```

EMPT1 — создает в памяти пустую переменную по адресу, находящемуся в регистре RGARG1, и формирует признаки TEMP1, TTR1:

```
'ref'||RGARG1[23 : 0]—> RGB, RGARG1[23 : 0]—> RGA;
      RGA—>LDA, RGB—>LDO, "0"—>TEMP1;
ЕСЛИ (RGARG1[23 : 0]<RGBL)
     ТО "1"—>TTR1;
     ЕСЛИ (RGARG1[23 : 0]>RGGMAX)
          ТО "1"—>TTR1
     ИНАЧЕ
          ЕСЛИ (RGARG1[23 : 0]<RGBG)
            ТО "1"—> TTR1;
          ИНАЧЕ "0"—> TTR1;
            ИНАЧЕ "0"—> TTR1;
КОНЕЦ. (Выдать код завершения "УСПЕХ")
```

EMPT2 — создает в памяти пустую переменную по адресу, находящемуся в регистре RGARG2, и формирует признаки TEMP2, TTR2:

```
'ref'||RGARG2[23 : 0]—>RGB, RGARG2[23 : 0]—>RGA;
   RGA—>LDA, RGB—>LDO, "0"—>TEMP2;
   ЕСЛИ (RGARG2[23 : 0]<RGBL)
        ТО "1" —>TTR1;
          ЕСЛИ (RGARG2[23 : 0]>RGGMAX)
               ТО "1"—>TTR2;
          ИНАЧЕ
               ЕСЛИ (RGARG2[23 : 0]<RGBG)
                    ТО "1"—>TTR2;
               ИНАЧЕ "0"—>TTR2;
                 ИНАЧЕ "0"—>TTR2
                 КОНЕЦ. (Выдать код завершения "УСПЕХ")
```

MATCH — выполняет унификацию двух термов, загруженных в регистры RGARG1, RGARG2; аналогична команде GEN_U, только при ее выполнении пропускаются фазы дереференсирования регистров RGARG1 и RDARG2:

```
"0"—> RGN, RGL1—< RGSP;
переход в алгоритм GEN_U в точку входа MATCH:
```

SS_ARG1 — вычисляет значение аргумента структуры, указатель на которую содержит регистр RGSS1, а номер аргумента — регистр RGN:

```
SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
КОНЕЦ. (Выдать код завершения "УСПЕХ")
```

Если аргумент структуры является ячейкой переменной (тег = ='global'), то находится значение переменной, которое дереференсируется. Результат выполнения команды помещается в регистр RGARG1.

SS_ARG2 — аналогична предыдущей, но при ее выполнении используются регистры RGARG2, RGSS2, RGSG2, TEMP2, TTR2:

```
SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
КОНЕЦ. (Выдать код завершения "УСПЕХ")
```

LINKVAR — выполняет связывание двух пустых переменных, размещенных в регистрах RGARG1, RGARG2, используется после команд создания пустых переменных EMPT1 и EMPT2:

```
"0" —> RGN, RGL1 —> RGSP, "1" —> TEMP1, "1" —> TEMP2;
переход к точке входа в GEN_U MATCH:
```

DECOMP1 — выполняет декомпозицию (распаковку) молекулы, адрес которой находится в регистре RGARG1:

```
DECOMP(RGARG1, RGSS1, RGSG1);
КОНЕЦ. (Выдать код завершения "УСПЕХ")
```

DECOMP2 — выполняет декомпозицию (распаковку) молекулы, адрес которой находится в регистре RGARG2:

```
DECOMP(RGARG2, RGSS2, RGSG2);
КОНЕЦ. (Выдать код завершения "УСПЕХ")
```

MOL1 — создание (упаковка) молекулы, адрес скелетона структуры находится в регистре RGARG1, адрес вектора значений переменных — в RGSG1:

```
MOLECULE(RGARG1, RGSG1, RGG1, TEMP1);
КОНЕЦ.
```

MOL2 — создание (упаковка) молекулы, адрес скелетона структуры находится в регистре RGARG2, адрес вектора значений переменных — в RGSG2:

```
MOLECULE(RGARG2, RGSG2, RGG1, TEMP2);
КОНЕЦ.
```

UN_LIST — перед выполнением данной команды регистры RGSS1, RGSS2 содержат указатели начала унифицируемых струк-

тур, регистры RGSG1 и RGSG2 — адреса векторов значений переменных, регистр RGN — номер унифицируемого аргумента:

RGL1→RGSP;
SS_ARG(RGARG1, RGSS1, RGSG1, TEMP1);
SS_ARG(RGARG2, RGSS2, RGSG2, TEMP2);
переход к точке входа MATCH: команды GEN_U.

LOAD RGi — загрузка одного из регистров блока REGISTERS информацией с магистрали LDI, каждому регистру присвоен номер, который задается в поле команды:

LDI→RGi;
КОНЕЦ.

READ RGi — чтение содержимого одного из регистров блока REGISTERS на выходную магистраль LD0, номер регистра задается в поле команды:

RGi→LD0;
КОНЕЦ.

Примечание. Содержимое всех регистров (кроме RGARG1, RGARG2, RGSS1, RCSS2, RGSG1, RGSG2) выдается на шину LD0 с тегом 'ref' в разрядах [31 : 24], регистров RGARG1 и RGARG2 выдается со своими тегами, регистров RGSS1, RGSS2, RGSG1, RGSG2 с содержимым регистра RGN вместо тега. Команды ПУ 6-разрядные, формат приведен на рис. 5.17.

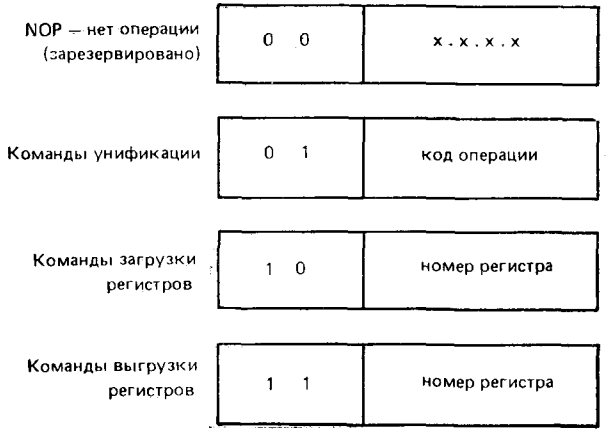


Рис. 5.17. Форматы команд процессора унификации

ГЛАВА 6.

МОДЕЛИРОВАНИЕ КОМПОНЕНТОВ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА

При моделировании работы ПЛВ решаются следующие основные задачи:

проверка полноты и корректности системы команд абстрактной машины и алгоритмов их выполнения;

определение наиболее часто используемого подмножества команд абстрактной машины;

определение длительности (или весовых эквивалентов) обработки команд;

оценка производительности ПЛВ и его компонентов.

Выбор оптимальной системы команд абстрактной машины, вообще говоря, плохо формализованная и весьма сложная задача.

Поэтому моделирование остается широко используемым механизмом тестирования и анализа систем.

Возможны два основных подхода к моделированию ПЛВ. Первый ориентирован на современные языковые средства моделирования (такие, как GPSS, Симула, Симскрипт и др.). Для их использования необходимо обеспечить интерпретацию процессов и механизмов языка Пролог в среде используемого языка моделирования. Второй подход связан с разработкой специализированной системы моделирования программно-аппаратных средств языка Пролог, которая должна обеспечивать настройку на заданную конфигурацию технических средств, систему команд Пролога и алгоритмов их выполнения.

Для трансляции исходной Пролог-программы в систему команд моделирующей программы удобно использовать процедурную интерпретацию Пролог-программ, при которой предикаты рассматриваются как частное подмножество процессов системы моделирования. В этой главе представлены модельные алгоритмы обработки правил и предикатов и приведены примеры спецификаций логических процессов в системе моделирования, в частности в среде GPSS. Второй подход рассмотрен на примере разработанного специализированного пакета, позволяющего вводить спецификации программно-аппаратной организации ПЛВ в удобной форме и интерпретировать результаты моделирования. Приведены результаты моделирования выполнения ряда Пролог-программ.

6.1. МОДЕЛИРОВАНИЕ ЛОГИЧЕСКОГО ВЫВОДА В СРЕДЕ УНИВЕРСАЛЬНОЙ МОДЕЛИРУЮЩЕЙ СИСТЕМЫ

Концепция моделирования логических процессов

Современные универсальные системы моделирования (GPSS, Симула, GASP, SLAM и др.) находят все большее применение при построении интеллектуальных систем, которое реализуется в двух главных направлениях. Во-первых, на базе языка моделирования реализуются механизмы логического вывода, обеспечивающие дедуктивный вывод, принятие решений и эвристический поиск, обработку реляционных запросов и др. Во-вторых, поддержка систем логического вывода обеспечивается функциями, которые от-

сутствуют в них или для реализации которых требуются дополнительные затраты вычислительных или программных ресурсов. Здесь речь идет об интеграции логического программирования и системы моделирования, обеспечивающей сочетание синтаксического и семантического аспектов исследуемой системы.

Применение языков моделирования для (или в) логического программирования основывается на следующем:

1) универсальности основных идеологических концепций, лежащих в основе механизмов моделирования;

2) наличии развитых средств для описания взаимодействий процессов во времени и пространстве;

3) реализации механизмов вероятностного и дифференциального аппарата, широкого диапазона математических функций, наличии сервисных процедур и др.

Универсальность позволяет воспроизводить логические процессы как некоторое подмножество абстрактных процессов системы моделирования. Универсальный абстрактный процесс описывается четверкой

$$P_i = \langle \tilde{S}_0,\ S_k,\ \tilde{S}_e,\ \Pi > i,\ i \in I. \qquad (6.1)$$

Здесь $\tilde{S}_0$, $\tilde{S}_e$ — выделенные подмножества начальных и конечных состояний; S_k — текущее состояние процесса P_i; Π — множество правил, задающих отображение

$$\Pi : S* \rightarrow Y(S*),$$

где $S*$ — пространство состояний процесса P_i; $Y(S*)$ — выделенное подмножество.

Правила Π могут задаваться следующим образом.

1. Через предикаты вида

а) $\Pi(S_k,\ S_{k+i})$, $k=0,\ 1,\ \ldots,\ K$; $i=0,\ \pm 1,\ \pm 2, \ldots,\ \pm K_k$;

б) $\Pi(S_k,\ S_{k+i},\ t)$, $t=1,\ 2, \ldots,\ T$;

в) $\Pi(S_{k-n},\ S_{k-n+1}, \ldots,\ S_k,\ S_{k+i},\ t)$,

связывающие два смежных во времени состояния. Случай б) предполагает наличие временной задержки t, разделяющей два смежных состояния. Задержка может задаваться в виде константы, функциональной или случайной величины с заданным распределением. В общем случае множество аргументов Π может быть больше 2 (правило в)).

2. Через вероятности q_{ij}, $j=i \pm n$, $n=0,\ 1,\ 2,\ \ldots$, связывающие состояния S_i и S_j.

3. Через эвристическую функцию оценки $C(.)$:

$$S_{k+i} = \arg\min\{C(S_e)\},\ e \in Y(S_k),$$

где S_{k+i} — состояние, определяющее минимальное значение эвристической функции $C(S_e)$, определенной на множестве состояний $\{S_e\}$.

4. Через выводимость

$$\{\tilde{S}_0,\ \tilde{S}_e,\ S_k\} \vdash S_{k+i},$$

$$\{\tilde{S}_0,\ \tilde{S}_e,\ S_k\} \vdash S_{k+i},\ t,$$

$$\{\tilde{S}_0,\ S_{k-n},\ \ldots,\ S_k\ \tilde{S}_e\} \vdash S_{k+i},\ t.$$

Реализация этого варианта связана с рекурсивным определением правил управления состоянием процесса логического вывода через сами эти правила.

Например, если при взаимодействии процессов P_i и P_j первый находился в состоянии S_{ki}, а второй — в состоянии S_{ej}, то в результате взаимодействия они перейдут в состояния $S_{k'i}$ и $S_{e'j}$ соответственно через время t_1 и t_2:

$$g_{ij}(S_{ki},\ S_{ej},\ S_{k'i},\ t_1,\ S_{e'j},\ t_2). \qquad (6.2)$$

При взаимодействии Пролог-процессов предполагается, что состояния S_{ki} и S_{ej}, кроме особых случаев, могут выбираться из исходных конечных выделенных множеств состояний. Схема функционирования логического процесса реализуется в двух вариантах A и B (рис. 6.1), в которых приняты следующие обозначения:

$\tilde{S}_0$	— список выделенных начальных состояний;
$\tilde{S}_e$	— конечное состояние;
$S^i_{01},\ S^i_{02},\ \ldots,\ S^i_{0\nu}$	— начальные состояния процесса;
$\hat{f}$	— флажок цикличности (0/1);
ν	— номер следующего начального состояния;
I	— номер процесса, передавшего управление процессу P_i;
J	— номер процесса, которому передает управление процесс P_i.

Варианту A соответствует вызов процесса P_i только по управлению, варианту B — с указанием начального состояния для P_i из $\tilde{S}_0$.

Взаимодействие логических процессов можно сравнить с передачей эстафетной палочки от одного процесса другому. Передача эстафетной палочки вызывающим процессом интерпретируется как факт его выполнения (истинности). Вызываемый процесс, получая эстафетную палочку, либо сам определяет выбор стартовой позиции (вариант A), либо передает эту возможность вызывающему процессу (вариант B). Если вызванный процесс успешно завершается, то он передает эстафету следующему процессу, в противном случае возвращает эстафету вызвавшему его процессу. Так, текущий процесс всегда определяет возможности, задаваемые схемами A и B для того процесса, которому он передает управление.

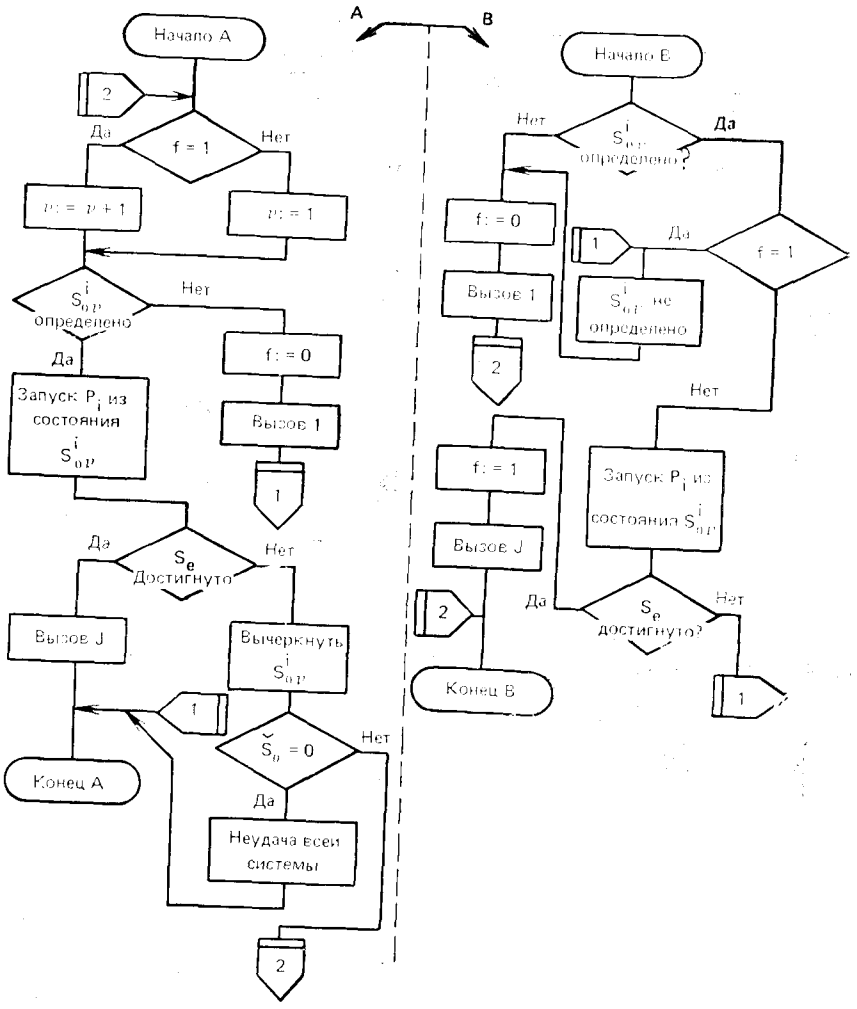


Рис. 6.1. Схема алгоритма логического процесса

Модельная интерпретация логических процессов

В языке Пролог основными механизмами логического вывода являются унификация и возврат.

В Пролог-программе атомные формулы интерпретируются как процессы, причем с формальной точки зрения они эквивалентны записи вида

$$P(\text{term 1, term 2, ..., term N}), \qquad (6.3)$$

где term i — переменная, константа или рекурсивно определяемое выражение вида $f(t_{i1}, \ldots, t_{iw})$ с функциональным символом f и символами $t_{i1}, \ldots, t_{iw}$ для термов. Формула является высказыванием, если не содержит переменных и все функциональные термы актуализированы (им присвоено значение).

Теорема в Пролог-программе имеет вид

$$P_0 \leftarrow P1\&P2\& \ldots \&PM. \qquad (6.4)$$

Модельной спецификацией для (6.3), (6.4) служит следующая схема:

PROGRAM;
* описание_предметной_области (A);
* исходное состояние (B);
* цель (C);
* ограничения (D);
* логика+эвристики (E);
END (PROGRAM).

Функционирование процесса приводит к изменению состояния моделируемой предметной области; достижение цели связывается с некоторым классом (или элементом множества) возможных состояний предметной области. Таким образом, доказательству теоремы в Прологе соответствует нахождение способа реализации цели.

Переменные в описании (6.3) назовем переменными состояния процесса P и обозначим V_i. Переменная V_i может принимать значения из некоторой области значений $\mathrm{Dom}(V_i)$. Присвоение значения переменной V называют актуализацией или связыванием. Если переменной не присвоено никакое значение из $\mathrm{Dom}(V_i)$, она рассматривается как деактуализированная.

Текущее состояние процесса определяется допустимой (имеющей смысл) комбинацией базовых статусов: Н — неактивный; И — инициированный; ОЖ — ожидающий; А — активный; t — выполнимый; f — невыполнимый.

Процесс неактивен тогда и только тогда, когда переменные, участвующие в его описании, деактуализированы и текущий список значений для их актуализации пуст (этот список, вообще говоря, не тождествен с $\mathrm{Dom}(V)$).

Процесс инициированный тогда и только тогда, когда его статус отличен от статуса Н. Инициированный процесс имеет статус (И-ОЖ), когда соответствующая ему формула является истинной (возможность I) или когда процесс содержит деактуализированные переменные, но не в состоянии их актуализировать, так как актуализация выполняется с участием других процессов (возможность II). Получение процессом статуса (И-А) осуществляется через модельный алгоритм синхронизации логических процессов.

Статус t соответствует процессу, который адекватен истинной формуле Пролог-программы (возможность III). Статус f, наоборот, определяет ложную формулу (возможность IV) или состояния процесса, характеризуемые наличием деактуализирован-

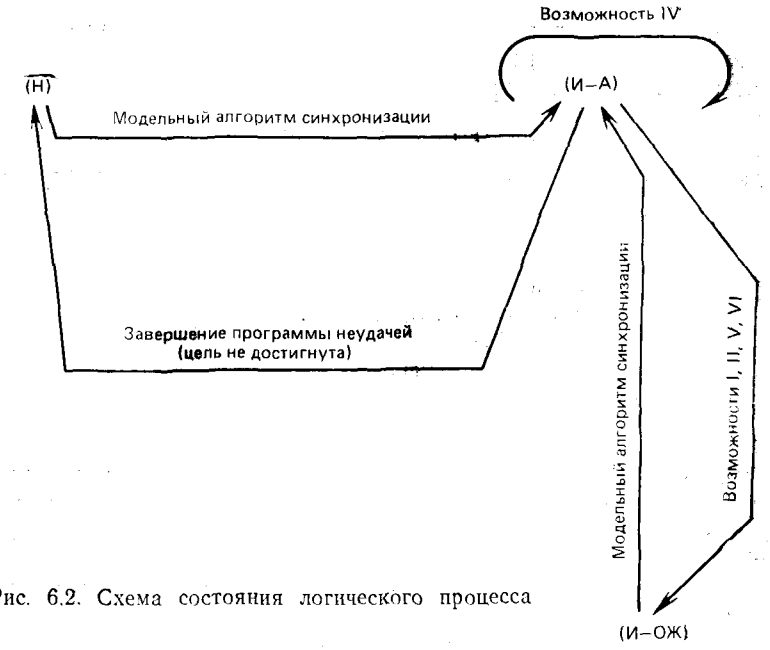


Рис. 6.2. Схема состояния логического процесса

ной(-ых) переменной(-ых) и пустым списком значений для ее (их) актуализации (возможность V).

Функционирование процесса иллюстрирует рис. 6.2.

Модель синхронизации процессов вывода

Для описания механизма синхронизации процессов вывода используем ориентированный граф $G(P, U)$ с множеством вершин-процессов P и дуг U, причем $(\overrightarrow{U_i, U_j}) \in U$, если, и только если процесс P_i является последователем P_j, а процесс P_j — предшественником P_i. Атомному процессу P_k соответствует висячая вершина в графе G. Ограничением на структуру графа G является недопустимость наличия в нем двух одинаковых вершин-процессов. Процессы считаются одинаковыми, если они имеют одинаковое определение типа (6.4) или имя (если процессы атомные) и если определяющие их термы совпадают, т. е. имеют один и тот же интерпретирующий их объект в моделируемой предметной области, а кроме того, принимают одинаковые значения при актуализации в любой момент времени. Последнее ограничение узаконивает рекурсивные определения для процессов.

Алгоритм синхронизации логических процессов:

1. Процессы P_i, P_j могут выполняться параллельно, если одновременно истинно, что они:

имеют статус (И-А);

не связаны некоторым путем в графе G;

не содержат совпадающих термов в ранее определенном смысле, если эти термы деактуализированные.

2. Любой процесс-последователь (сукцедент) P_i получает статус (И-А) только при условии, что всего его предшественники приобрели статус (И-ОЖ), реализовав для этого возможности I или II (рис. 6.2).

3. Если для некоторых P_i и P_j не выполняется п. 1, то первым выполняется процесс с бо́льшим приоритетом. Второй процесс получает статус (И-ОЖ) (возможность VI) и в зависимости от исхода выполнения первого (только в случае, если первый процесс получит статус t или реализует возможность II) может получить или не получить статус (И-А).

4. Если процесс P_i, имея статус (И-А), не способен актуализировать переменную, для которой список значений потенциально пополним, то он получает статус (И-ОЖ-f) и «возвращает» статус (И-А) всем своим предшественникам. В противном случае (список значений пуст и потенциально непополним) вся Пролог-программа завершается неудачей относительно достижения цели.

5. Если после п. 4 программа не завершена, как минимум один из процессов сохраняет статус (И-А) и выполняется. Если выполнение процесса связано с возможностями I или II, то процесс проверяет, есть ли процессы, ожидающие завершения в силу п. 3; если таковые есть, то активизирует их (устанавливает статус (И-А)), иначе переход к п. 2.

6.2. ТРАНСЛЯЦИЯ ПРОЛОГ-ПРОГРАММЫ В ПРОГРАММУ МОДЕЛИРУЮЩЕЙ СИСТЕМЫ

Для интерпретации логических процессов в моделирующей системе будем использовать два типа модельных процессов: процессы-предикаты и процессы-контексты. Последние имеют текущие значения для всех связанных (к данному моменту и месту выполнения исходной Пролог-программы) переменных. При возврате текущий процесс-контекст уничтожается, а из специально организованной очереди (цепи) извлекается предыдущий контекст, который становится текущим. Номер процесса-контекста свяжем с номером текущего выполняемого правила Пролог-программы. Процесс-предикат выполним в текущем контексте, если его аргументы унифицируются с аргументами процесса-контекста.

Язык абстрактной моделирующей системы

Для трансляции Пролог-программы в программу моделирования введем следующие команды (сегменты), интерпретируемые моделирующей системой (т. е. предполагается, что имеются модельные алгоритмы, реализующие эти команды):

backcontext — смена контекста при возврате;

testcontext(имя_предиката, номер_правила, признак) — проверка выполнимости процесса-предиката с выработкой приз-

нака 1 при положительном результате проверки и 0 в противном случае;

changecontext (имя_предиката, номер_правила, признак) — формирование нового контекста и сохранение предыдущего; при успешном завершении признак 1, иначе — 0;

change (имя_предиката, номер_правила) — смена активного процесса-предиката при возврате к предыдущей цели;

newsubgoal (имя_предиката, номер, признак) — выбор в теле активного правила предиката цели с заданным порядковым номером; если все предикаты-цели выбраны, признак 0, в противном случае 1;

call (имя_предиката, номер_правила, признак) — инициализация среды процесса-предиката с указанием номера правила для этого предиката; если выполнение предиката завершается успешно, признак получает значение 1, в противном случае 0;

activy_previous_process(имя_предиката, номер_правила) — возврат к предыдущему предикату в теле правила с указанием для него нового правила;

select (имя_предиката, номер_правила, признак) — выбор следующего правила для предиката; если правило выбрано, то признак 1, иначе 0;

choise_rule (имя_предиката_цели_программы, номер_правила, признак) — выбор правила и предиката цели с фиксацией во втором параметре команды номера правила для этого предиката; если предикат выбран, признак 1, иначе 0;

call_rule (имя_предиката, номер_правила, признак) — вызов процедуры обработки правила для указанного предиката; если правило завершается успешно, возвращаемый признак 1, в противном случае 0.

Процедура обработки цели программы:

```
AS: Num=1
A: choise_rule(Z,Num,S)/* Z связывается с именем следующего предиката в цели программы */
        if S=0 then do;  i
        write ("Программа завершена с контекстом: "[контекст]);
            od;
   else do;
call_rule (Z, Num, S); /* вызов правила с номером Num для предиката Z */
        if S=1 then goto AS;
        elf do;
      backcontext;
      change (Z, Num);
      goto A;
      od;
      od;
```

Процедура обработки правила:

```
C: i=0:
B: i=i+1;
   newsubgoal(Z, i, S); /*выбор следующего предиката в теле правила*/
   if S=0 then return(1); /*правило обработано успешно*/
E: select(Z, Num, SS); /*выбор правила для предиката Z*/
   if SS=0 then goto D; else
                        do;
   call(Z, Num SSS); /*инициализация среды процесса —предиката*/
   if SSS=1 then do;
   call_rule(Z, Num SS1); /* вызов правила для предиката Z с номером
                             Num*/
   if SS1=1 then goto B;
   else do;
        backcontext;
        goto E;
      od;
        od;
D: i=i—1;
   if i>0 then do;
                backcontext;
activy_previous_process(Z, Num); /*локальный возврат*/
                Num=Num+1;
                goto E;
                    od;
   else
                return(0);
                    od; /*неудача*/
```

Вызов предиката:

```
testcontext(Z, Num, S); /*проверка условия унифицируемости Z и текущей
                          среды*/
if S=0
      then return (0);
      else
          do;
changecontext (Z, Num, SS); /*установка значений переменных в результате
                              унификации*/
if SS=0
      then do;
            backcontext;
            return (0);
            od;
else return (1);
     od;
```

Управляющая структура модели основывается на представлении исходной структуры Пролог-программы в виде следующей таблицы:

Признак предиката	Имя предиката	Номера правила (списка)	Номер последнего использованного правила	Имя следующего предиката в правиле (или цели)	Имя предыдущего предиката в правиле (или цели)
1	2	3	4	5	6

Построим управляющую таблицу для следующей программы на языке Пролог:

```
goal
  pair(X, "Джейн"), likes(X, футбол).
clauses
(1) pair (X, Y):—
man (X),
girl (Y).
(2) man (бил).
(3) man (джон).
(4) man (питер).
(5) girl (джейн).
(6) girl (лина).
(7) likes (бил, бокс).
(8) likes (питер, футбол).
```

1	2	3	4	5	6
+	pair/2	[1]	нет	+likes/2	нет
+	likes/2	[7, 8]	нет	нет	pair/2
—	man/1	[2, 3, 4]	нет	[(1, girl/1)]	нет
—	girl/1	[5, 6]	нет	нет	[(1, man/1)]

В столбце 1 таблицы предикат цели помечен знаком «+», остальные — знаком «—». В столбце 5 знак «+» перед предикатом указывает, что это предикат цели, после косой черты указывается число аргументов предиката; запись (1, girl/1) указывает, что предикат girl/1 относится к правилу с номером 1 (в общем случае предикат может использоваться в нескольких правилах).

Значения доменов данной Пролог-программы формируются согласно следующей таблицы:

Предикат	Номер аргумента	Список значений аргумента	Имя домена
man	1	бил джон, питер	1
girl	1	джейн, лина	2
likes	1	бил, питер	3
likes	2	бокс, футбол	4

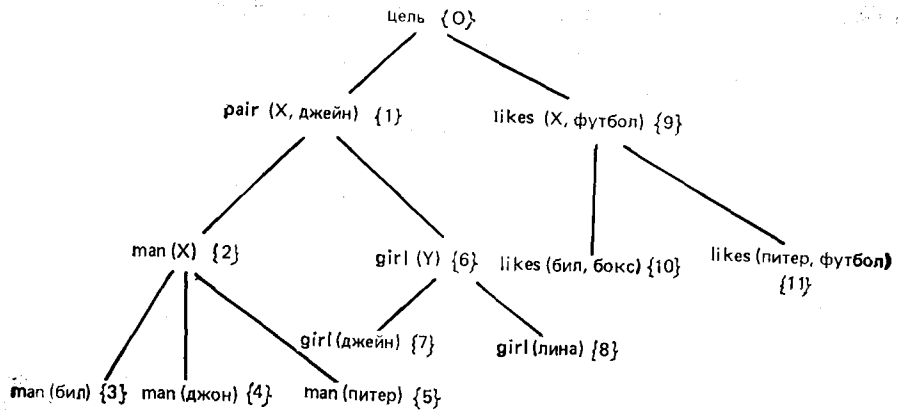


Рис. 6.3. Структура программного обеспечения системы моделирования

Структура G-графа, соответствующая этой программе, представлена на рис. 6.3. Узлы G-графа соответствуют процессам-предикатам, в фигурных скобках указаны приоритеты процессов (более приоритетным процессам соответствуют меньшие значения в фигурных скобках). Напомним, что уровни приоритетов используются для упорядочения вызовов «квазипараллельных» процессов-предикатов (см. алгоритм синхронизации логических процессов).

Пример моделирования логических процессов в среде GPSS

Покажем способ модельной реализации абстрактной схемы взаимодействия процессов, частным подмножеством которых являются логические процессы, на примере GPSS-модели.

В моделирующей системе GPSS процессы представляются динамическими объектами — транзактами. Транзакт характеризуется набором параметров: байтовых, полусловных и словных (4 байт), обозначенных соответственно PB, PH и PF. Установка значений параметров транзактов выполняется в блоке ASSIGN, который имеет следующий формат:

ASSIGN [параметр, которому присваивается значение][, значение, присваиваемое параметру][, тип]

«Тип» определяет тип параметра, являющегося первым операндом команды (PB, PH или PF). Параметры могут использоваться в качестве указателей на другие параметры. В этом случае они имеют следующее представление: «тип_параметра*параметр», например PB*PH3 представляет содержимое байтового параметра, номер которого содержится в параметре PH3. Статические объекты GPSS представляются сохраняемыми величинами: байтовыми (XB), полусловными (XH) и словными (XF). Значения сохраняемых параметров могут совместно использоваться различными транзактами. Для присваивания значений сохраняемым величинам используется команда SAVEVALUE, назначение операндов в формате которой аналогично ASSIGN, за исключением того, что первый и третий операнды относятся к сохраняемому параметру. Для про-

верки значений параметров и сохраняемых величин используется команда TEST, имеющая следующий формат:

TEST [отношение] [первый_операнд, второй_операнд] [метка]

Данная команда сравнивает первый операнд со вторым. В качестве отношения используется: G (больше), L (меньше), E (равно), GE (больше/равно), LE (меньше/равно). Если указанное отношение не выполняется, то осуществляется перевод транзакта на указанную метку в программе. Если метка не указана, то продвижение транзакта блокируется, пока указанное в команде отношение не будет выполнено. Удаление транзактов из модели осуществляет команда TERMINATE, а перевод транзактов в другое место программы, определяемое меткой, — команда TRANSFER.

Интерпретатор GPSS в каждый момент выполняет продвижение в модели транзакта с максимальным текущим приоритетом из числа незаблокированных транзактов. Это позволяет использовать приоритеты для управления очередностью обработки процессов, моделируемых транзактами.

Для установки и изменения приоритетов транзактов используется команда PRIORITY.

Для временного вывода транзактов из модели используется специальная конструкция — цепь. Занесение транзактов в цепь выполняет команда LINK, первый операнд которой указывает имя цепи, а второй — место в цепи транзакта: FIFO в конец цепи, LIFO в начало цепи, PB (PH, PF) согласно значению параметра PB (PH, PF). Вывод транзактов из цепи выполняет команда UNLINK, первый операнд которой определяет имя цепи, второй — число выводимых транзактов, третий — место в программе, с которого продолжает движение выведенный транзакт.

Команда SPLIT «расщепляет» транзакт на число транзактов, задаваемое первым параметром. Транзакт-родитель направляется на метку в программе, определяемую вторым операндом команды.

В приводимых ниже программах используются обозначения:

V $ (имя_переменной) — для переменных (вычисляемых величин GPSS),

FN $ (имя_функции) — для функций,

MB $ (имя_матрицы) (номер_строки, номер_столбца) (MH — матрица полусловных величин, MX — матрица однословных величин) — для матричных сохраняемых величин,

CH $ (имя_цепи) — для текущего числа транзактов в цепи. Подробные сведения о языке GPSS можно найти в [31].

Таким образом, процессы-предикаты будем моделировать транзактами, используя назначения параметров:

PB127 — имя предиката (кодируется числами);

PBi (i< > 127) — имя аргумента предиката кодируется числами — отрицательное число соответствует свободному аргументу);

PFi — значение аргумента;

PH1 — номер правила;

PH2 — номер альтернативного правила.

Отдельный класс транзактов образуют транзакты-контексты, которые хранят текущие значения всех связанных переменных. При возврате текущий транзакт-контекст уничтожается, а из специально организованной очереди (цепи) контекстов извлекается предыдущий контекст, который становится текущим.

Номер транзакта-контекста свяжем с номером текущего выполняемого правила. Процесс-предикат выполним в текущем контексте, если его аргументы унифицируются с аргументами транзакта-контекста.

Смену контекстов при возврате выполняет следующий фрагмент:

```
          TEST G      CH $CONTX, 0, FAIL     001
          UNLI NK     CONTX, 1, NEWCX        002
          TERMINATE                          003
NEWCX     ASSI GN     3, 1, PH               004
          ASSI GN     4, PB*PH3, PH          005
          SAVEVALUE   PH4, PF*PH4, XF        006
          ASSI GN     3+, 1, PH              007
          TEST E      PB*PH3, 0, NEWCX       008
          SPLI T      1, PROC                009
          LINK        CURCX, FIFO            010
PROC {выбор альтернативного процесса-предиката для восстановленного контекста}  011
FAIL {неудача всей программы}                012
```

Транзакты-контексты хранятся в цепи CH $ CONTX. Если при вызове предыдущего контекста эта цепь пуста (блок 1), то осуществляется выход на метку FAIL для фиксации неудачи всей программы. В противном случае восстанавливается предыдущий контекст (вывод транзакта-контекста из цепи в блоке 2). В блоках 4—7 для выбранного контекста создается база данных — параметры транзакта-контекста, хранящие значения длял переменных, заносятся в сохраняемые величины GPSS. Байтовые параметры PB транзакта-контекста — номера переменных программы, а полнословные PF — их значения. Если параметр PBi содержит нулевое значение, то это является признаком конца списка переменных (блок 8). Текущий (активный) контекст хранится в цепи с именем CURCX.

Формирование нового контекста путем копирования означенных переменных процесса в базу данных выполняет следующий фрагмент:

```
          ASSIGN      3, 1, PH               001
UNIF      TEST NE     PB*PH3, 0, SUCC        002
          ASSIGN      4, PB*PH3, PH          003
          TEST GE     PH4, 0, ENT1           004
          SAVEVALUE   PH4, PF*PH4, XF        005
ENT1      ASSIGN      3+, 1, PH              006
          TRANSFER    , UNIF                 007
SUCC      UNLINK      CURCX, 1, HERE         008
          TERMINATE                          009
HERE      SPLIT       1, MOD                 010
          LINK        CONTX, LIFO            011
MOD       {копирование базы данных в параметры транзакта_контекста}  012
          SPLI T      1, PROC                013
          LINK        CURCX, FIFO            014
```

Здесь устанавливается значение сохраняемой величины в базе данных для каждой связанной переменной процесса-предиката (блок 8). Проверка усло-

вия, что переменная является связанной, выполняется в блоке 4 (переменная свободна, если ее номер, содержащийся в PBi, отрицательный). Параметр PH3 используется как счетчик переменных. Если номер переменной равен 0 (блок 2), выполняется переход на метку SUCC для смены текущего контекста (содержится в цепи с именем CURCX) на обновленный.

Процесс может выполняться в текущем контексте, если значения связанных переменных процесса совпадают со значениями одноименных переменных контекста. Проверка выполнимости процесса реализуется следующим фрагментом:

```
AGAIN   ASSIGN     3, 1, PH                    001
        TEST NE    PB*PH3, 0, SUCC2            002
        ASSIGN     4, PB*PH1, PH               003
        TEST GE    PH4, 0, ENT1                004
        TEST NE    XF*PH4, PF*PH4, ENT1        005
        TRANSFER   ,FAIL2                      006
ENT1    ASSIGN     3+, 1, PH                   007
        TRANSFER   ,AGAIN                      008
SUCC2   {процесс удовлетворяет контексту}      009
FAIL2   {процесс не удовлетворяет контексту}   010
```

Данный фрагмент во многом схож с предыдущим (первые 8 блоков). Если процесс не выполним в текущем контексте, выполняется переход на метку FAIL2. Поэтому следует взять альтернативное правило для процесса в параметре PH2. Если такого правила нет, необходимо сменить контекст на его предыдущее значение.

Установка параметров процесса путем копирования их значений из базы данных реализуется в следующем фрагменте:

```
CYC     ASSIGN     3, 1, PH                          001
        TEST NE    PB*PH3, 0, SUCC3                  002
        ASSIGN     4, PB*PH3, PH                     003
        TEST L     PH4, 0, ENT2                      004
        TEST NE    XF*V$NUMPR, XF$FREE, ENT2         005
        ASSIGN     PB*PH3, V$NUMPR, PB               006
        ASSIGN     PB*PH3, XF*V$NUMPR, PF            007
ENT2    ASSIGN     3+, 1, PH                         008
        TRANSFER   ,CYC                              009
SUCC3   {вызов по завершении инициализации}          010
```

В этом фрагменте копирование значений производится из сохраняемых в базе данных для свободных переменных процесса (присваивание значений свободным переменным выполняется в блоке 7). Переменная считается свободной, если ее номер отрицательный (это условие проверяется в блоке 4). Если переменная как процесса, так и контекста свободна, ее значение не изменяется. Эта проверка выполняется сравнением с константой XF$FREE, кодирующей свободную переменную (в блоке 5). Переменная V$NUMPR преобразует отрицательный номер параметра в положительный.

Моделирование взаимодействий процессов-предикатов выполняется следующим образом. С каждым процессом связывается множество переменных состояния $p_i \in \{(0, 1, X)\}$, где $p_i = 0(1)$ интерпретируются как сброс (установка) условия p_i, а $p_i = X$ — как неопределенное состояние. Переменные состояния

процесса образуют состояние S_k. Функционирование абстрактного процесса может связываться с установкой и сбросом тех или иных переменных состояния. Кроме того, процесс может, изменяя состояние некоторого множества переменных (их сброс и установку), управлять активностью (состоянием) других процессов. Управление активизацией процессов связывается с вектором статуса MH $STAT (матрица с числом строк, равным 1). (Отметим, что переменные в MH $STAT могут принимать не обязательно три значения 0, 1, X (указанное ограничение, вообще говоря, не принципиально, так как, используя эти три значения, можно закодировать любой недвоичный объект).)

Общая схема моделирования, реализуемая GPSS-модулем:

1) i-е состояние системы;
2) активизация подходящего (требуемого) процесса;
3) выполнение процесса;
4) новое $(i+1)$-е состояние системы, переход к п. 1.

Этап 1 характеризуется состоянием MH $STAT, так же как и этап 4. Этап 2 выполняется с помощью матрицы MH $INITT, определяющей условия инициирования процессов. Каждая строка матрицы MH$ INITT определяет (если она целиком не содержит X, кодируемых как 2) некоторый вектор переменных статуса, в соответствии с которым определяются условия активизации одноименных процессов.

Пусть

$$\mathrm{INITT}\ (v^i_1, v^i_2, \ldots, v^i_m)$$

— строка матрицы MH $ INITT, соответствующая процессу 1. Тогда условие активизации процесса i будет вида

$$(\forall k \in \{1, \ldots, m\})\,(((v^i_k \neq X) \& (p_k \neq X)) \rightarrow (v^i_k = p_k)).$$

Полусловная матрица MH$COND выполняет функцию, противоположную MH $INITT, — запрещает исполнение процесса при срабатывании ограничительного условия, аналогичного вышеприведенному, но с $v_k \in$ MH $ COND.

Матрица MH $GOAL обеспечивает требуемую установку переменных статуса (состояния) в MH$ STAT, если таковая предусмотрена для соответствующего процесса. Полусловная матрица MH $FIN сходна с MH $STAT по формату; отличие между ними состоит в том, что MH $FIN задает целевое (конечное) состояние системы.

Матрицы MH $ INITT, MH $ GOAL, MH $ COND имеют следующий общий вид:

Номер транзакта требования на активацию процесса	Значения переменных состояния			
	p1	p2	...	pn

. . . . . .

Последняя строка матрицы нулевая.

GPSS-модуль имеет следующий вид:

```
        GENERATE   ,,1, 1, OPB, 6PH, OPF    01
        SAVEVALUE  TEK+, 1, XH              02
VYX     ASSIGN     5, 1, PH                 03
LISTN   SAVEVALUE  TABLE, 1, XH             04
```

```
          TEST NE    MH $INITT(PH5,1),0,FIN02                    05
MAX       TEST LE    XH $TABLE,XH $PSW,KOOG                      06
          SAVEVALUE  TABLE+,1,XH                                 07
          TEST NE    MH $INITT(PH5,XH $ TABLE),2                 08
                     MAX
          TEST E     MH $INITT(PH5,XH $TABLE),                   09
                     MH $STAT(1,XH $ TABLE),RAW
          TRANSFER   ,MAX                                        10
RAW       ASSIGN     5+,1,PH                                     11
          TRANSFER   ,LISTN                                      12
KOOG      ASSIGN     6,MH $INITT(PH5,1),PH                       13
          SPLIT      1,NEXT1                                     14
          PRIORITY   0,BUFFER                                    15
          ASSIGN     6,0,PH                                      16
          TRANSFER   ,RAW                                        17
FIN02     SAVEVALUE  TEK—,1,XF                                   18
          PRIORITY   0,BUFFER                                    19
AGAIN     TEST G     CH $CONDT,0,FIN03                           20
          UNLINK     CONDT,LNKG,1                                21
          TERMINATE                                              22
TRM       ASSIGN     1,MH $PRIOR(1,PH6),PH                       23
          LINK       CONDP,1PH                                   24
FIN03     TEST G     CH $CONDP,0,FIN04                           25
          UNLINK     CONDP,NEXT,ALL                              26
          PRIORITY   0,BUFFER                                    27
          TRANSFER   ,AGAIN                                      28
NEXT      ASSIGN     1,MH $PRIOR(1,PH6),PH                       29
NEXT1     LINK       CONDT,1PH                                   30
FIN04     TERMINATE  1                                           31
LNKG      PRIORITY   0,BUFFER                                    32
          SAVELUE    TEK+,1,XF                                   33
ROU       ASSIGN     5,0,PH                                      34
VARB      ASSIGN     5+,1,PH                                     35
          TEST NE    MH $COND(PH5,1),0,FIN02                     36
          TEST E     MH $COND(PH5,1),PH6,VARB                    37
          SAVEVALUE  CONTS,1,XH                                  38
ROBB      TEST LE    XH $CONTS,XH $PSW,ABORT                     39
          SAVEVALUE  CONTS+,1,XH                                 40
          TEST NE    MH $COND(PH5,XH $CONTS),2,                  41
                     ROBB
          TEST NE    MH $STAT(1,XH $CONTS),2,ROBB                42
          TEST NE    MH $STAT(1,XH $CONTS),                      43
                     MH $COND(PH5,XH $CONTS),ROBB
          SPLIT      1,FIN02                                     44
          ASSIGN     1—,MH $PRIOR(2,PH6)PH                       45
          TRANSFER   ,TRM                                        46
ABORT     PRIORITY   0,BUFFER                                    47
STATS     ASSIGN     5,0,PH                                      48
HELP      ASSIGN     5+,1,PH                                     49
```



```
          TEST NE    MH $GOAL(PH5,1)0,USER                       50
          TEST E     MH $GOAL(PH5,1),PH6,HELP                    51
          SAVEVALUE  CONTS,1,XH                                  52
CONSD     TEST LE    XH $CONTS,XH $PSW,VYX1                      53
          TEST NE    MH $GOAL(PH5,XH $CONTS),2,                  54
                     CONSD
          MSAVEVALUE STAT,1,XH $CONTS,MH $GOAL                   55
                     (PH5,XH $CONTS),MH
          TRANSFER   ,CONSD                                      56
VYX1      ASSIGN     5,0,PH                                      57
VYY       ASSIGN     5+,1,PH                                     58
          TEST LE    PH5,XH $PSW,FIN04                           59
          TEST NE    MHN $STAT(1,PH5),MH $FIN(1,                 60
                     PH5),VYY
          UNLINK     CONDP,NEXT,ALL                              61
          PRIORITY   0,BUFFER                                    62
          TRANSFER   ,VYX                                        63
          START      1                                           64
          END
```

Назначение используемых объектов, переменных и функций: XH $ CONTS и XH $ TABLE — полусловные сохраняемые величины, выполняющие роль счетчиков;

XH $PSW — число (константа), на единицу меньшее, чем число переменных состояния pi (столбцов в MH $STAT);

XH $TEK — сохраняемая величина, определяющая число активных процессов;

5PH — полусловный параметр-счетчик;

6PH — номер процесса (задается в столбце 1 таблицы выше); является аргументом матрицы приоритета MH $ PRIOR;

CH $ CONDT — цепь процессов, активных в текущем состоянии;

CH $CONDP — цепь процессов, для которых не выполняются входные условия из MH $ COND.

Рассмотрим работу GPSS-модуля.

В блоках 4—17 выполняется поиск всех тех процессов, которые должны быть инициированы (по матрице MH $ INITT) в текущем состоянии, определяемом матрицей MH $ STAT. Блоки 4—8 и 10 реализуют циклическую проверку для каждой переменной состояния (pi). Проверка совпадения по каждой переменной состояния в MH$ INITT и MH$ STAT осуществляется в блоке 9. Напомним, что значение pi=2 считается неопределенным. В блоке 13 параметру 6PH присваивается номер процесса, который может быть инициирован. Этот процесс из блока 14 подается в цепь в блоке 30. В эту же цепь с именем CONDT попадают все процессы, которые должны быть инициированы (получают статус И). Процессы в цепи располагаются согласно приоритету, устанавливаемому в блоке 29 по номеру процесса (параметр 6PH), задающему номер столбца в матрице MH $ PRIOR. Вывод инициированных процессов из цепи CONDT выполняется в блоке 21, причем выводится один процесс с максимальным приоритетом, он подается в блок 32. В блоках 34—41 выполняется проверка условий, задаваемых матрицей MH $ COND: может ли инициированный процесс выполняться или нет. Если процесс выполняться не мо-

жет, то в блоке 44 генерируется транзакт, выбирающий новый инициированный процесс из цепи CONDT, а «старый» процесс помещается в цепь CONDP, предварительно его приоритет снижается в блоке 45. Изменение приоритета позволяет изменить в последующем порядок выборки инициированных процессов для выполнения.

Если условия, определяемые для выбранного процесса по MH $ COND, выполняются, то этот процесс устанавливает значения переменных MH $ STAT (в блоках 47—56) в соответствии с матрицей MH $ GOAL. Собственно установка реализуется в блоке 55. Наконец, после того как состояние MH $ STAT изменено, осуществляется проверка MH $ STAT и MH$ FIN на совпадение (достигнуто ли целевое состояние) в блоках 57—63. Если целевое состояние не достигнуто, то осуществляется перевод процесса из цепи CONDP в CONDT (в блоке 61), а также возврат к процедуре поиска процессов, которые могут быть инициированы в новом состоянии MH$ STAT (т. е. переход к блоку 3).

В число процессов можно ввести так называемые процессы-демоны. Процесс-демон будет выполняться тогда, когда никакой «рабочий» процесс не может быть выполнен. В этом случае демон будет выполнять возврат к предыдущему или какому-либо другому состоянию MH$ STAT. По существу, запись процессов-демонов ничем не отличается от записи других процессов. Однако процессы-демоны можно наделить способностью изменять приоритеты (определяемые в параметре PR) рабочих процессов и тем самым влиять на порядок их выполнения. Эта возможность в приведенной программе не отражена. Здесь каждый рабочий процесс устанавливает себе исходный приоритет по значениям из строки 1 матрицы MH$ PRIOR, а затем (в случае неудачи) уменьшает его на величину, определяемую в соответствующем столбце строки 2 матрицы MH $ PRIOR.

Представленный подход к моделированию процессов логического вывода на основе их интерпретации в существующих моделирующих системах имеет следующие преимущества:

моделирующая система представляет «готовый» инструментарий для построения модели, а также всю необходимую выходную статистическую информацию, интересующую пользователя;

путем декомпозиции модели можно моделировать с интересующей степенью детализации отдельные функции, механизмы и команды, используемые в логическом выводе;

при моделировании аппаратуры и системных взаимодействий моделирующая система обеспечивает удобную интерпретацию для ресурсов, сигналов и правил их обработки;

предоставляется определенный сервис — редактирование, тестирование и графический вывод для программ моделирования.

Однако пользователю необходимо знать особенности обработки программ на Прологе, язык моделирующей системы, так как отсутствуют средства автоматической генерации моделирующих программ. Высокая трудоемкость разработки моделей является достаточно ограничительным фактором. Второй подход, основанный на разработке специализированной моделирующей системы, хотя частично лишен этого недостатка, однако связан с ограниченной предметной областью моделирования.

6.3. СТРУКТУРНО-АЛГОРИТМИЧЕСКАЯ МОДЕЛЬ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА*

Функциональный уровень моделирования

Моделирование работы ПЛВ основано на программной реализации его основной задачи — выполнении скомпилированной Пролог-программы. В соответствии с этой концепцией моделируются лишь те блоки ПЛВ и HOST-машины, которые существенны с точки зрения описания и выполнения алгоритмов команд:

HOST-память, содержащая доступные командам таблицы кодов Пролог-программы;

блок регистров;

кеш-память.

Структуры данных программной модели процессора (программный процессор) и ПЛВ совпадают с точностью до констант, т. е. значения кодов команд, флаги и т. п. выбраны произвольно. Алгоритмы выполнения базовых микроопераций и состав этих операций жестко заданные, изменению подлежит лишь время их выполнения (цены), которое определяется экспертным путем и зависит от конкретных аппаратных решений.

Таким образом, моделирование, основанное на этих принципах, позволяет:

выбрать рациональную систему команд исследуемого процессора;

разработать и отладить алгоритмы команд, найти (поиск) оптимальные решения по их реализации;

выбрать рациональный состав библиотеки микропрограмм;

собрать статистическую информацию в процессе моделирования.

При разработке программного обеспечения (ПО) использовались алгоритмические языки PL/1 и ассемблера в среде ОС ЕС. Диалоговое взаимодействие организовано с помощью средств ДСКП PRIMUS версии 2.5/M1, специальных программ форматирования экрана. Объем исходных текстов программ — более 1500 операторов.

Функционирование модели процессора

Рассмотрим организацию процесса моделирования ПЛВ, считая, что все входные данные созданы в соответствующих наборах. Основными компонентами ПО являются монитор (HOST) и пошаговый интерпретатор (STEP), остальные подпрограммы требуются лишь на этапе ввода исходных данных и получения статистических результатов. После запуска модели диалоговый монитор выводит на экран перечень возможных реализуемых им функций (главное меню):

1) запуск;
2) корректировка таблиц;

*В написании этого раздела принимал участие А. И. Попов.

3) просмотр трассы;
4) печать трассы;
5) HELP;
6) просмотр статистики;
7) печать статистики.

Выбор первой функции означает начало моделирования. При этом происходит инициализация программного процессора HOST-памяти, в которую загружается Пролог-программа, и начинается выполнение ее команд.

Диалоговый монитор представляет возможность выбора одного из трех режимов работы модели:

непрерывный,
пошаговый (покомандный),
операционный.

В первом режиме выполняется Пролог-программа без отображения команд и микроопераций на экране, после чего осуществляется возврат в главное меню. В пошаговом режиме на экран выводятся имя команды, которая будет выполняться, названия и значения параметров (полей), смещение в коде, задаваемое регистром команд. До и после выполнения команды возможен просмотр элементов архитектуры: содержимого отдельных регистров или участков HOST-памяти в символьном, десятичном или шестнадцатеричном форматах. В операционном режиме на экране дополнительно выводится текущая микрооперация алгоритма выполнения команды, а после нажатия клавиши «Ввод» выполняется не вся команда, а только микрооперация, представленная на экране. В пошаговом и операционном режимах можно изменить режим просмотра процесса моделирования.

В ходе инициализации программного процессора осуществляется заполнение таблиц, необходимых для интерпретации объектного кода Пролог-программы. Программа ZFGKOM производит загрузку таблицы описания форматов команд, программа ZFGARX — заполнение таблицы описания архитектуры программного процессора, программа ZAGMIC — ввод описания базовых микроопераций программного процессора, программа ZAGMP размещает в оперативной памяти библиотеку микропрограммы, содержащую описания алгоритмов команд; программа ZAGK на основании описания архитектуры строит таблицу описания кеш-памяти, необходимую для реализации алгоритма обмена «кеш — HOST-память», программа ZAGMP в HOST-памяти — бинарный код. По окончании интерпретации программы STAGR и STATK заполняют статистические таблицы.

В режиме корректировки возможна работа с таблицами, необходимыми для инициализации программного процессора.

Имеется также возможность просмотра данных, полученных в ходе трассировки, и вывод их на печать (режимы 3 и 4). Аналогичные возможности представляются в режимах 6 и 7 при работе со статистическими данными моделирования.

Функция HELP выводит на экране справочную информацию о системе моделирования.

Подготовка и ввод исходных данных

Для успешного функционирования программного процессора на его вход должны быть поданы соответствующим образом оформленные исходные данные:

код Пролог-программы;
описание форматов команд;
описание архитектуры процессора;
описание базовых микроопераций процессора;
библиотека микропрограмм.

В первую очередь необходимо сформировать код, который будет выполняться программным процессором. Поскольку моделируемый процессор имеет индивидуальную (уникальную) систему команд и не имеет компилятора, преобразующего Пролог-программу в код данной системы, то этот процесс приходится выполнять вручную. Вместе с тем прогонка полученных программ на модели позволяет оценить правильность «ручной» компиляции и может оказаться полезной при создании Пролог-компилятора. В процессе компиляции определяются адреса и содержимое таблиц в HOST-памяти, которые доступны ПЛВ и, соответственно, программному процессору.

Заполнение всех таблиц, находящихся в файле, осуществляется с помощью средств системы PRIMUS, позволяющих корректировать записи неопределенной длины. Программа ZAGMPL считывает все таблицы в массив, моделирующий HOST-память. Инициализация элементов этого массива возможна также и программным путем — с помощью программы на базовом языке программного процессора. На практике заполнение HOST-памяти осуществляется комбинацией двух этих способов.

На основании данных, полученных в процессе компиляции, формируется содержимое регистров программного процессора, которое используется при описании архитектуры. *Описание архитектуры* создается в режиме редактирования системы PRIMUS и хранится в одном из разделов библиотеки микроопераций в следующем формате:

Имя	Тип	Содержимое	Длина	Цена чтения	Цена записи

Здесь имя — имя элемента архитектуры длиной не более 16 символов, которое может использоваться в качестве операнда в базовых микрооперациях программного процессора. Имена регистров совпадают с именами регистров ПЛВ. Имя элемента, имитирующего HOST-память, — HOST. Имена кеш-памяти произвольные, но при этом имена регистров, фиксирующих начальное наложение кеш-памяти на HOST-память, должны иметь следующий вид:

X_MIN — регистр начального адреса кеш-X;
X_MAX — регистр последнего адреса кеш-X;
X_H — регистр текущего уровня кеш-X.

Тип — это символ, определяющий тип компонента ПЛВ, допустимы следующие значения типа:

R — регистр,
K — кеш-память,
P — память,
F — фиктивный массив.

Тип F введен для моделирования массивов, имеющих свойства памяти (например, ПЗУ), причем элементы такого массива располагаются в таблице описания архитектуры непосредственно за элементом типа F и обращение к ним в базовых микрооперациях может осуществляться как по этому имени, так и по имени фиктивного массива с указанием индекса, например PZU(1).

Содержимое указывает начальное содержимое регистра, которое загружается в него до начала выполнения кода, задается в десятичной форме.

Длина — значение, задаваемое в десятичной форме, определяет размерность компонента. Для компонентов типа R длина равна нулю, типа P или K — емкости памяти (в байтах), типа F — количеству следующих за ним ячеек.

В последних двух полях задаются цены (времена) чтения и записи информации при обращении к данному компоненту, что необходимо для сбора статистических данных в процессе выполнения программы.

Загрузка описания архитектуры выполняется программой ZAGARX. Работа программного процессора производится по обычной схеме, в два этапа:

выборка команды,
выполнение команды.

На первом этапе осуществляется считывание из массива, имитирующего HOST-память, очередного байта, адресуемого регистром P, после чего определяются код команды, ее длина, форматы полей (если они существуют) и имя раздела в библиотеке микропрограмм, где описан алгоритм данной команды. Все эти данные вычисляются в соответствии с *описанием команд*, имеющим следующий формат:

Имя	Код	Количество полей	Длина	Имя раздела	Тип

Здесь имя — символьное имя описываемой команды, длина имени не должна превышать 32 символов; код — десятичный код команды; длина — количество байт, которое занимает команда. Поле «Имя» содержит имя раздела библиотеки микропрограмм, где описан алгоритм выполнения команды. Если длина команды равняется одному байту, т. е. команда представлена только кодом операции, то количество полей команды равняется нулю. В противном случае количество определяет, сколько полей у команды. Тип команды анализируется в программах формирования статистических данных. Непосредственно за описанием команды следует *описание полей (параметров)* в следующем формате:

Имя	Тип	Смещение	Длина

Здесь имя — это имя параметра команды, которое используется в базовых микрооперациях в качестве операнда при описании алгоритма команды; символьные значения типа используются для внутренних целей моделирования; смещение определяет адрес параметра в поле команды, указывая количество предшествующих байт, и, возможно, бит, если поля команды не выравнены на границу байта; длина поля определяет количество байт и, возможно, бит, отводимых под него. Значения полей таблицы «Смещение» и «Длина» задаются в следующем формате:

BYTES[.BITS],

где BYTES — число байт в десятичной форме; BITS — число бит в десятичной форме.

На этапе выборки происходит выделение полей команды в соответствии с таблицей описания форматов команд и присвоение параметрам команды конкретных значений. После этого выполняется алгоритм команды, т. е. последовательная интерпретация базовых микроопераций, с помощью которых описан этот алгоритм. Каждой команде соответствует определенный раздел в библиотеке микропрограмм, который содержит программу на языке микроопераций программного процессора, реализующую алгоритм выполнения команды. Загрузка библиотеки осуществляется по подпрограмме STEP в соответствии с *описанием базовых микроопераций* следующего формата:

Имя	Тип	Количество операндов	Номер	Цена	Результат

Здесь имя — мнемоническое обозначение микрооперации, имеющее длину не более 8 символов; тип — символ, используемый для внутренних целей моделирования; цена — среднее время, необходимое для выполнения команды, используется при сборе статистической информации, задается в десятичном формате. Значение поля «Номер» в чем-то аналогично коду команды, определяется в процессе анализа микрооперации и используется для внутренних целей моделирования, форма записи десятичная. Количество операндов задается целым десятичным числом от 0 до 2. Значение поля «Результат» определяет направление действия команды: если оно равно 1, то результат записывается по адресу первого операнда, если 2, то по адресу второго.

Все базовые микрооперации программного процессора можно разбить на несколько групп.

1. *Операции пересылки данных*:

WORD — пересылка нескольких байт, количество байт определяется расширением, формат записи:

WORD/расширение] адрес1, адрес2

В операциях этой группы могут использоваться следующие виды адресации:

1) непосредственная; операнд в десятичной форме находится в поле команды;

2) регистровая, операнд находится в одном из регистров:

адресi := имя регистра;

3) косвенная, адрес операнда находится в одном из регистров:

адресi := [имя регистра];

4) прямая, адрес операнда определяется индексом массива, моделирующего память:

адресi := имя_массива (индекс),

где имя_массива — имя элемента архитектуры типа P, имеющего длину, отличную от нуля;

5) базовая, адрес операнда определяется выражением вида

адресi := [адр] (адр)

где адр — любой из предшествующих видов адреса операнда.

Количество байт, участвующих в пересылке, определяется расширением, которое может иметь следующие значения:

P1 — старшие два байта,
P2 — младшие два байта,
F — флаг (тег),
A — младшие три байта,
1 — первый байт,
2 — второй байт,
3 — третий байт,
4 — четвертый байт.

Нумерация начинается со старшего байта.

LOAD — загрузка нескольких байт (в зависимости от расширения) из слова, находящегося по адресу2, в слово по адресу1, формат записи:

LOAD/расширение] адрес1, адрес2

STOR — обратная LOAD — загрузка числа, находящегося по адресу1, в несколько байт (в зависимости от расширения) слова по адресу2, формат записи:

STOR/расширение] адрес1, адрес2

MOVE — длинная пересылка — пересылка группы байт с адреса1 по адресу2, количество пересылаемых байт определяется расширением операции, формат записи:

MOVE/количество] адрес1, адрес2

количество := имя регистра или число в десятичной форме.

Если расширение не указано, то по умолчанию во всех операциях пересылки участвует слово, т. е. 4 байт.

2. *Арифметические операции*:

+ — сложение,
— — вычитание,
* — умножение,
DIV — деление.

Формат записи:

операция адрес1, адрес2

Арифметические операции работают только со словами, результат заносится на место первого операнда.

3. *Операции побитовой обработки и логические операции*:

XR — реализация логической функции «Исключающее ИЛИ»,
AND — реализация логической функции И,
OR — реализация логической функции ИЛИ.

Формат записи:

операция [/расширение] адрес1, адрес2

NOT — реализация логической функции «инверсия», формат записи:

NOT[/расширение] адрес1

Расширение может принимать те же значения, что и в операциях пересылки, результат находится на месте первого операнда.

SL, SR — сдвиг влево и вправо, формат записи:

операции сдвига [/расширение] адрес1, адрес2

— сдвиг нескольких байт по адресу1 (в зависимости от расширения) на количество бит, определяемых словом по адресу2.

TM — сравнение по маске, формат записи:

TM[/расширение] адрес1, адрес2

— побитовое сравнение операнда, находящегося по адресу1, с маской по адресу2, после чего в регистр признаков заносится признак результата сравнения:

0 — у операндов нет совпадающих бит,
1 — у операндов есть совпадающие биты,
2 — у операндов все биты совпадают.
? — операция сравнения, формат записи:

?/расширение] адрес1, адрес2

4. *Операции управления:*

IF — переход по условию, формат записи:

IF операнд1, операнд2

Здесь операнд1 — знак одной из следующих логических операций:

= — равно, > = — больше или равно,
> — больше, < = — меньше или равно,
< — меньше, <> — не равно;

операнд2 — метка одной из операций микропрограммы, куда передается управление, если операнд1 совпадает с результатом сравнения.

GOTO — безусловный переход, формат записи:

GOTO операнд1

где операнд1 — метка из операций микропрограммы, куда передается управление.

CALL — вызов микропрограммы или переход в дополнительную точку входа команды или микропрограммы, формат записи:

CALL операнд1 [/точка входа], [параметры]

где операнд1 — это имя раздела библиотеки, где находится вызываемая микропрограмма; точка — метка, на которую передается управление; параметры — данные, адреса которых могут быть записаны в соответствии с существующими способами адресации.

RET — возврат из микропрограммы, параметров нет.

5. *Псевдооперации:*

PROC — заголовок микропрограммы, формат записи:

PROC операнд1, [параметры]

где операнд1 — имя раздела библиотеки, где хранится данная микроподпрограмма, параметры — формальные параметры микроподпрограммы;

END — конец микропрограммы;

STOP — завершение выполнения Пролог-программы.

Две последние операции операндов не имеют.

При составлении программ с использованием данных операций следует придерживаться следующих правил:

метка с двоеточием находится в первых восьми позициях строки;

код операции записывается в позициях 9—16;

операнды записываются с позиции 17, разделителем между операндами служит запятая;

признаком комментария служат два знака «—» в первых двух позициях строки.

6.4. ПАРАЛЛЕЛЬНОЕ МОДЕЛИРОВАНИЕ *

Рассмотрим системы моделирования, рассчитанные на исследование параллельных процессов и архитектур (см. § 6.1—6.3) ПЛВ. Будем различать простые и составные блоки.

*В написании параграфа принимал участие Н. И. Кузьмицкий.

Описание простого блока:

КЛАСС=⟨имя⟩, ТИП-ПРОСТОЙ[,ВРЕМЯ=⟨коэффициент⟩] [,MIC=YES]
⟨описание вычислительных средств⟩
⟨описание внешних выводов⟩
⟨описание автомата 1⟩
. . .
⟨описание автомата N⟩

Здесь ⟨имя⟩ — это имя, с использованием которого можно ссылаться на блок, например, при его включении в другой блок; ⟨коэффициент⟩ позволяет корректировать быстродействие блока.

Описание вычислительных средств:

РЕГИСТРЫ=⟨имя регистра⟩(⟨длина в битах⟩)[TIME=(⟨время чтения⟩,⟨время записи⟩)][=⟨имя поля⟩[(длина)]!!⟨имя поля⟩[(длина)] ...]

МАССИВЫ=⟨имя массива⟩(⟨размерность⟩)⟨описание элемента⟩ [TIME=(⟨время чтения⟩, ⟨время записи⟩)]

Здесь описание элемента приводится по правилам описания регистра. Массив может иметь характеристику CS (Control Storage — управляющая память):

CS=⟨имя микропрограммы⟩

Такой массив используется в программно (микропрограммно)-управляемом блоке (MIC=YES). В этот массив системой автоматически загружается управляющая микропрограмма из библиотеки. Имя раздела библиотеки, содержащего микропрограмму, указывается в характеристике CS.

СИГНАЛЫ: ⟨список имен⟩ [TIME=⟨время реакции⟩]

TIME может использоваться для указания быстродействия памяти и для оперативного изменения типа памяти.

Описание внешних выводов:

ВХОДЫ: ⟨список имен регистров и сигналов, являющихся внешними⟩

ВЫХОДЫ: ⟨список имен регистров и сигналов, являющихся внешними⟩

Описание автомата:

АВТОМАТ: ⟨имя автомата⟩⟨описание собственных данных⟩ ⟨имя состояния⟩: [⟨условие⟩:]⟨алгоритм⟩

Здесь описание собственных данных приводится по формату описания вычислительных средств блока; имя состояния может совпадать с именем внешнего сигнала, в этом случае переход в данное состояние может осуществляться и по прерыванию, условие — имя сигнала, может использоваться в частности для маскирования; алгоритм — описание алгоритма на базовом языке. Каждый автомат имеет системное состояние ожидания с именем WAIT.

Описание составного блока:

КЛАСС=⟨имя⟩, ТИП-СОСТАВНОЙ[, ВРЕМЯ=⟨коэффициент⟩]

ВЫВОДЫ=⟨список имен внешних выводов⟩ ⟨имя вывода⟩ ⟨имя подблока⟩

СВЯЗИ=⟨имя входа-выхода⟩⟨имя подблока⟩.⟨имя входа-выхода⟩[,TIME=⟨время прохождения сигнала⟩]

ШИНА=⟨имя подблока⟩.⟨имя входа-выхода⟩=⟨имя подблока⟩.⟨имя входа-выхода⟩=...[,TIME=⟨время прохождения сигнала⟩]

Выводам блока не нужно описания: они определяются связями. Списка блоков не нужно, он определяется из секции связей.

Расширения базового языка

Дополнительно к системе базовых команд описания последовательных алгоритмов используются следующие команды:

→ ⟨имя состояния⟩ — переход в новое состояние,

⇒ ⟨имя автомата⟩.⟨имя состояния⟩ — переход другого автомата данного блока в новое состояние, текущий автомат продолжает функционировать в текущем состоянии.

Для описания параллельности применяются следующие средства:

{ — начало списка параллельно выполняемых алгоритмов,

; — разделитель списка параллельно выполняемых алгоритмов,

} — конец списка параллельно выполняемых алгоритмов.

Скобки параллельности могут быть вложенными.

Для учета времени выполнения базовых операций используются временные скобки:

TIME(⟨время⟩) END TIME

Считается, что алгоритм, заключенный в эти скобки, реализуется за указанное время. Вложенность этих скобок не допускается.

Дополнительные команды обработки прерываний:

MASK — замаскировать прерывания,

DEMASK — размаскировать прерывания.

Дополнительные команды для процедур:

EX⟨имя внутренней процедуры⟩

PROCX — то же, что и в языке описания последовательных процедур, но используется для внутренних, по отношению к блоку процедур;

%INCLUDE⟨имя раздела библиотеки проекта⟩ — для подключения стандартных фрагментов на базовом языке.

Эти команды не могут быть вложенными.

Если в алгоритмическом моделировании базовый язык использовался для описания алгоритмов выполнения объектных Пролог-кодов, то при параллельном моделировании его функции значительно шире — он используется и для описания поведения структурного элемента Пролог-машины. Например, дешифрация кодов,

поддерживаемая на прежнем уровне системой, на параллельном уровне может быть описана на базовом языке как параллельный процесс.

Моделирование микропрограммного управления

Для поддержки микропрограммно-управляемого блока предусмотрены следующие средства:

1) массив с характеристикой CS, где указано имя соответствующего раздела библиотеки, в котором содержится управляющая микропрограмма;

2) таблица микрокоманд, аналогичная таблице описания команд в алгоритмическом моделировании, но расширенная рядом полей, в частности ценой декодирования, возможно, ценой выполнения (в этом случае подсчет времени выполнения на уровне команд базового языка не ведется).

Для формирования автоматов модели функционирования микропрограммно-управляемого блока используются следующие системные состояния:

выборка — связана с регистром-счетчиком, реализуется системно, может содержать описание времени;

декодирование — связано с таблицей описания команд (микрокоманд), реализуется системно, может содержать описание времени;

выполнение — это состояние, взаимодействующее с процессом декодирования через специальное системное поле (интрфейс); при декодировании определяется имя раздела библиотеки проекта, где на базовом языке описан алгоритм выполняемой микропрограммы.

Приведенные выше системные состояния при описании функционирования микропрограммно-управляемого блока можно вводить в соответствующие автоматы (даже работающие параллельно) в различных комбинациях, добавлять «свои» состояния. Более того, они могут попадать в автоматы различных подблоков данного микропрограммно-управляемого блока.

Возможно описание микропрограммного управления просто на базовом языке, без использования системных состояний.

Моделирование параллелизма

Моделирование параллелизма основано на понятии времени, учитываемом в системном регистре TIK. Каждому автомату соответствует блок, состоящий из двух частей: управляющей (PSW) фиксированной длины и части накопления статистической информации, длина которой зависит от количества состояний автомата и задания на моделирование.

В управляющей части учитывается текущее состояние, текущая команда, время работы автомата и выполнения команды базового языка либо интервал, указанный в скобках TIME/ENDTIME, определяющий более крупную единицу работы, чем команда базового языка. Единица работы является неразрывной относитель-

но прерываний. Время выполнения единицы работы определяется из цены базовых команд. Учитывается также время, необходимое для обращения к операндам, которое берется из описания вычислительных средств.

1. Регистр TIK доступен для чтения на уровне базовых команд. В каждом состоянии регистра TIK происходит обращение ко всем автоматам. При каждом обращении к автомату время единицы работы, учитываемое в PSW, уменьшается на 1. Когда это время становится равным нулю, единица работы считается выполненной.

Базовые команды «{», «;» «}» приводят к образованию новых (временных) PSW, в которых учитывается время выполнения каждого из параллельно выполняемых алгоритмов. По завершении всех алгоритмов выбирается максимальное время из временных PSW, и это время учитывается в постоянном PSW автомата. Базовые спецкоманды, например «СТАТИСТИК», «СОБЫТИЕ», «?TIK», «{», «;», «}», выполняется за нуль единиц времени.

Запрос на моделирование состоит из двух частей: инициализация и задание для сбора статистики и составление трассы.

Инициализация блока:

⟨имя подблока⟩ . ⟨имя автомата⟩ → ⟨состояние⟩

для всех автоматов каждого подблока

⟨имя регистра⟩ = ⟨значение⟩/* имя регистра может быть составным*/

⟨имя массива⟩ = ⟨значение⟩

⟨имя массива⟩(⟨индекс⟩) = ⟨значение⟩

⟨имя массива⟩(⟨диапазон индексов⟩) = ⟨значение⟩

⟨имя массива⟩} = ⟨имя раздела библиотеки проекта⟩

Для микропрограммно-управляемого блока управляющая микропрограмма загружается из библиотеки по описанию CS. Для программно-управляемого блока (спецпроцессора) имя объектной Пролог-программы задается в виде:

PROGR = ⟨имя раздела библиотеки проекта, содержащего программу⟩

Задание для сбора статистики блока:

⟨список автоматов и их состояний, для которых ведется учет времени нахождения в этом состоянии⟩

⟨список средств, для которых ведется учет числа обращений⟩

Задание для составления трассы:

⟨список событий, включаемых в трассу⟩ : ⟨действия⟩

. . .

Здесь событием может быть переход некоторого автомата в конкретное состояние (в частности, прерывание), или выполнение конкретной специальной команды базового языка, или истечение некоторого интервала времени. Действие — дамп памяти, уведомление о наступлении события на терминале с переходом в режим сервиса, предусматривающий просмотр и/или выдачу на печать накопленной статистики, просмотр и корректировку любого объекта процессора, интегрированного состояния любого автомата, выраженного в PSW, поддерживаемого системой. Трасса событий может быть выдана на печать.

Сбор статистики обеспечивается следующими средствами:

блоком сбора статистической информации автомата, который вместе с PSW составляет блок управления автоматом;

для каждого вычислительного средства счетчиком чтения-записи статистики обращений, совокупность которых обеспечивает статистику обращений блока;

базовыми спецкомандами.

СТАТИСТИК ⟨имя⟩ — содержимое программного счетчика с указанным именем наращивается на 1;

СОБЫТИЕ ⟨имя⟩ — выполняются действия, описанные в задании на моделирование.

Возможность одновременного отображения состояния нескольких параллельных процессов через определенный промежуток времени либо при наступлении определенных событий предоставляют сервисные средства. На экране отображаются блок, текущее состояние его автоматов и текущие базовые команды. Допускается корректировка состояний процессов, содержимого вычислительных средств, изменение интервала времени.

Библиотека проекта представляет собой текстовую базу данных, содержащую объекты следующих типов: Пролог-программы; описания простых и составных блоков; различные фрагменты на базовом языке самого разного назначения (алгоритмы микрокоманд, макросы; микропрограммы и другие тексты, например для инициализации; задания на моделирование; таблица базовых команд; таблица Пролог-команд; таблица микрокоманд). Наличие библиотеки проекта и использование абстрактных типов при описании блоков позволяет наглядно и компактно описывать сложные многоуровневые системы и проводить их комплексную отладку и оценку производительности.

Развитие программных средств алгоритмического моделирования удовлетворяет новым требованиям, предъявляемым к моделированию процессора. На этом уровне достаточно детально могут быть описаны архитектура ПЛВ (блоки и подблоки), учтен параллелизм при определении производительности, обеспечено моделирование прерываний при передаче сигнала с одного автомата на вход другого блока, который связан с некоторым состоянием своего автомата (некоторых своих автоматов).

6.5. РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ АБСТРАКТНОЙ МАШИНЫ ЛОГИЧЕСКОГО ВЫВОДА

Рассмотрим пример, иллюстрирующий создаваемые внутренние структуры для программ сцепления списков:

```
domains
listofinteger=integer*
predicates
append (listofinteger, listofinteger, listofinteger)
```

```
goal
append ([1, —2, 3], [4, 5], L),
write (L)
clauses
(1) append ([ ], L, L).
(2) append ([A Z], L, [A | X]):—
append (Z, L, X).
```

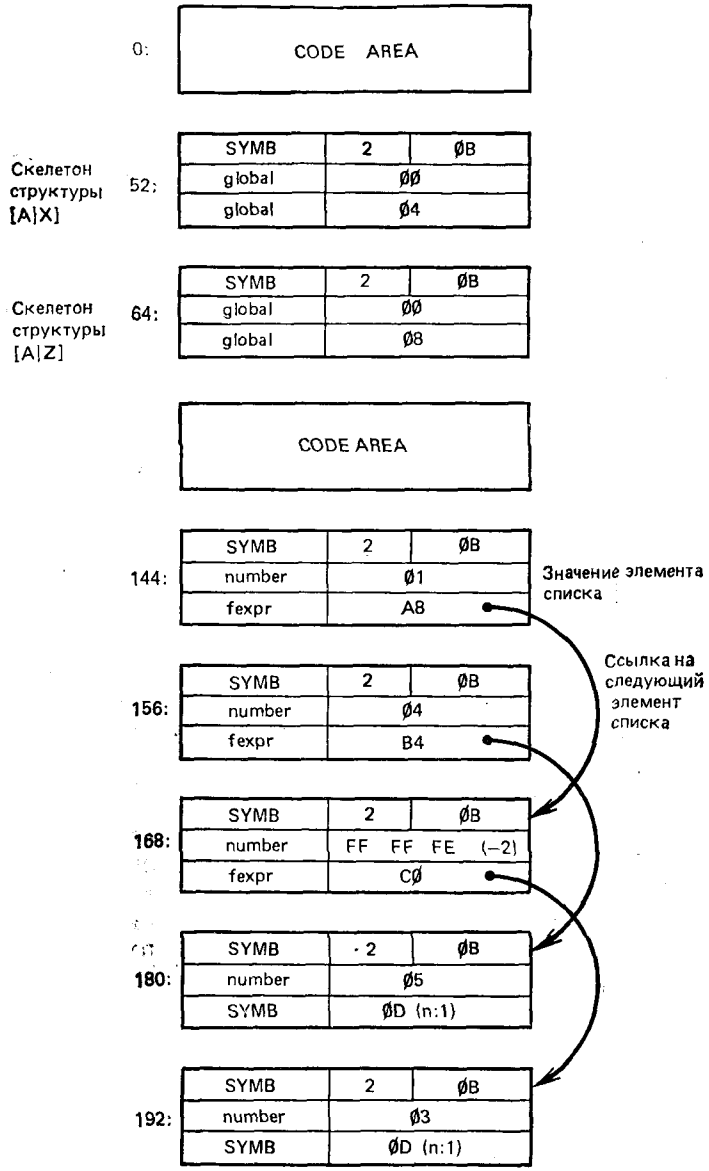


Рис. 6.4. Структуры программы сцепления списков

В результате компиляции получена программа в объектных Пролог-кодах, структура которой приведена на рис. 6.4. За объектным кодом CODE AREA располагаются сначала скелетоны структур [A|X], [A|Z], затем снова код и наконец данные запроса в виде списков [1, —2, 3], [4, 5]. По адресам 144, 168, 192 находятся элементы списка [1, — 2, 3], по адресам 156, 180 — элементы списка [4, 5], Ø В означает ссылку на таблицу символов.

В процессе моделирования инициализируются все таблицы, доступные ПЛВ: таблица символов и таблица утверждений, а также содержимого регистров L- и G-стеков, HOST-памяти процессора.

Состояния стека перед первой унификацией списков [A|Z]=[1, —2, 3] и L=[4, 5] показаны на рис. 6.5. Молекула занимает две ячейки в G-стеке. Ячейка с тегом mol используется для ссылки на скелетон структуры, а ячейка с тегом fexpr — для ссылки на значения элементов структуры. Скелетон создается для списка, представленного в формате ["голова"|"оставшаяся часть"]. Ячейка с тегом molref содержит указатель на молекулу: ячейка L+7 в L-стеке — указатель на молекулу терма [A|Z], а ячейка L+8 — указатель на молекулу терма L. Значения термов [A|Z] и L определены в области данных, куда ссылаются соответствующие им ячейки с тегом fexpr, т. е. [A|Z]=[1, —2, 3], L=[4, 5]. Ячейка L+9 содержит указатель на переменную X, которая пока не имеет значения: в G-стеке ей соответствует пустая ячейка с тегом empty.

Состояние стеков после первого выполнения правила 2 программы показано на рис. 6.6,*а*. Ячейки 10 и 11 G-стека представляют молекулу, построенную для терма [A|Z]; ячейка с тегом mol содержит ссылку на скелетон, описывающий структуру этого списка, ячейка с тегом fexpr — указатель вектора значений переменных A и Z. Значение для A есть 1 (хранится в ячейке 5), значение Z есть терм (хранится в ячейке 6) — содержит ссылку на молекулу (тег molref), занимающую ячейки 8 и 9. В ячейке 9 ссылка в область данных, определяющая, что значение Z есть список [—2, 3]. Ячейки 3 и 4 G-стека содержали молекулу, построенную для списка L. Значение L определяется содержимым ячейки 4 — указателем в область данных на список [4, 5]. В ячейках 1 и 2 содержится молекула, соответствующая всему списку [1, —2, 3].

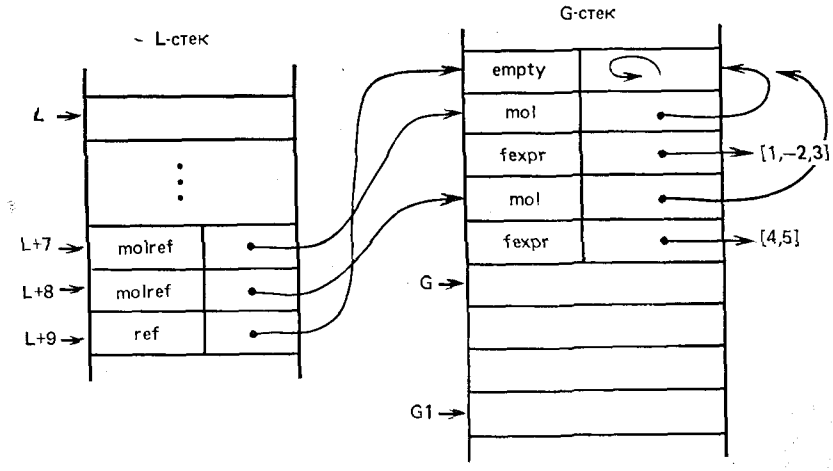


Рис. 6.5. Состояние стека перед унификацией списков

Ячейка 0 содержит ссылку на молекулу X, которая еще не создана (в ячейке 7, на которую она ссылается, нет значения, на что указывает тег empty).

Состояние стеков после второго выполнения правила 2 показано на рис. 6.6,*б*. Здесь молекуле, представляющей терм [A|Z], соответствуют ячейки 17 и 18, причем ячейки 17 содержит указатель на новый вектор глобальных переменных, начинающийся с ячейки 12, в которой хранится значение A=—2. Ячейка 13 содержит ссылку на молекулу в ячейках 15 и 16, представляющую терм Z. Значение Z определяется ссылкой в область данных на список [3].

Состояние стеков после третьего выполнения правила 2 показано на рис. 6.6,*в*, а результирующие состояния стеков — на рис. 6.6,*г*. Переменная X на рис. 6.6,*г* получает ссылку на ячейку 3, представляющую молекулу (см. рис. 6.6,*а*), значение которой есть [4, 5], поэтому X=[4, 5].

В процессе моделирования отлаживаются алгоритмы выполнения команд и уточняется архитектура процессора. При этом было выяснено, что введение кеш-памяти несущественно увеличивает быстродействие Пролог-процессора, но значительно усложняет его структуру. Поэтому было решено выполнять построение процессора с памятью в виде L- и G-стеков. (Статистика по результатам моделирования приведена ниже.)

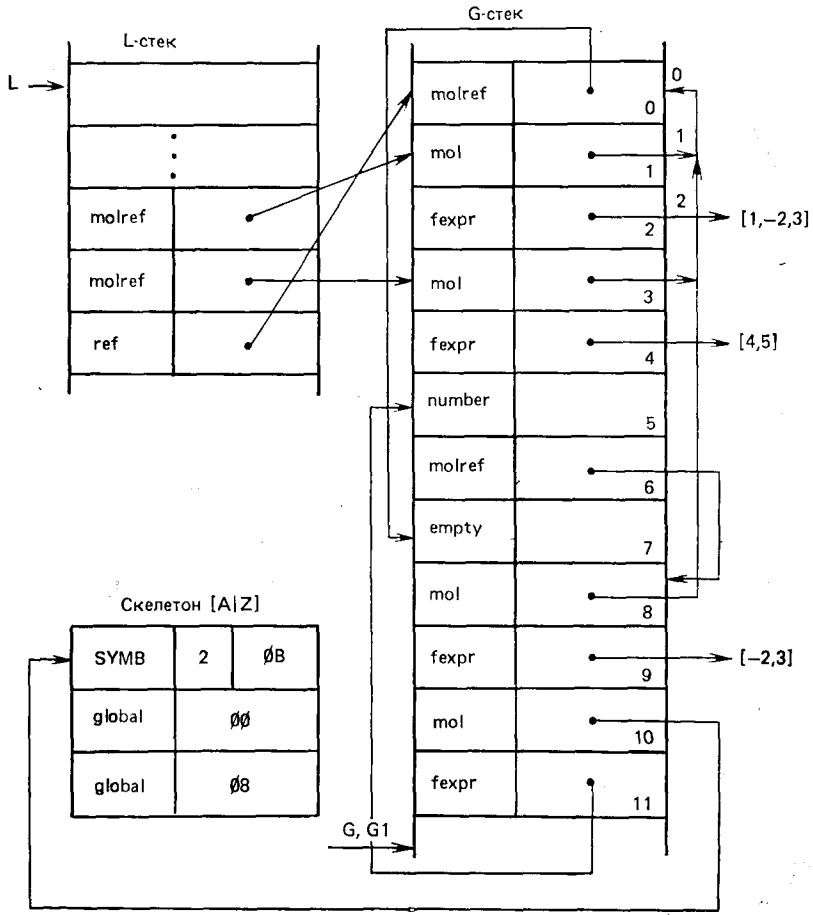


Рис. 6.6. Состояния стеков в процессе выполнения программы

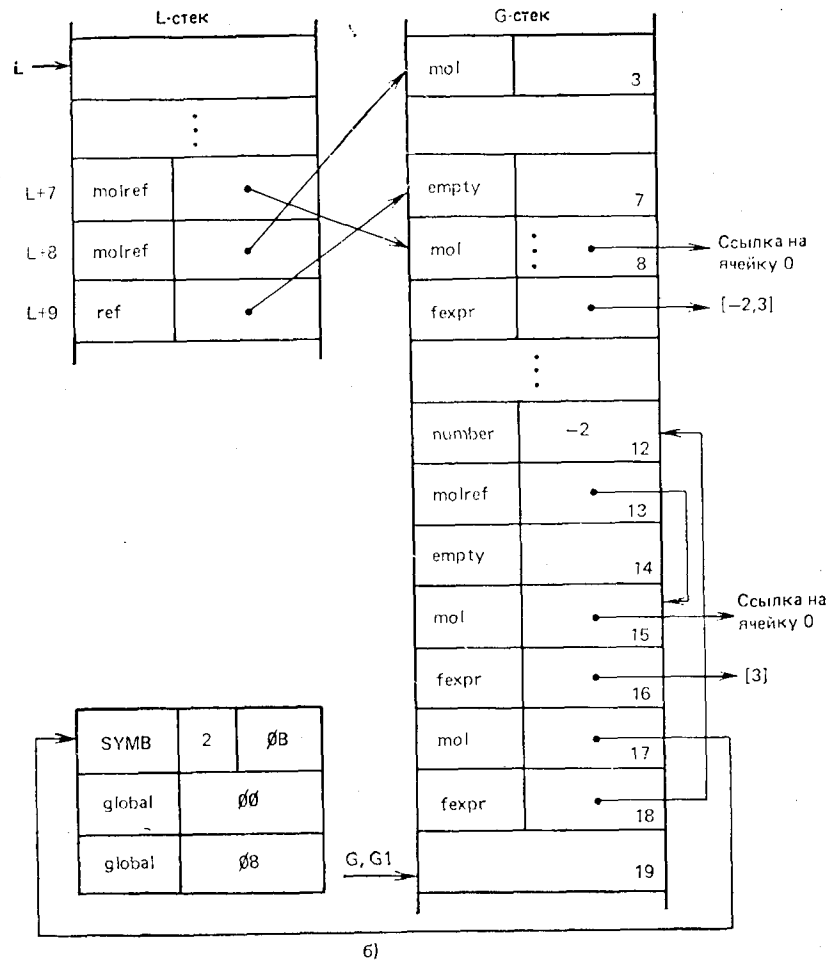


Рис. 6.6. Продолжение

Моделирование абстрактной машины логического вывода основано на анализе ее исполнения исходных Пролог-программ. Основная доля вычислительной «нагрузки» ложится на команды унификации (general_unify) и возврата (fail). Число команд унификации и возврата в скомпилированной программе определяется по следующему соотношению:

$$r = \frac{Vunify}{Vfail}$$

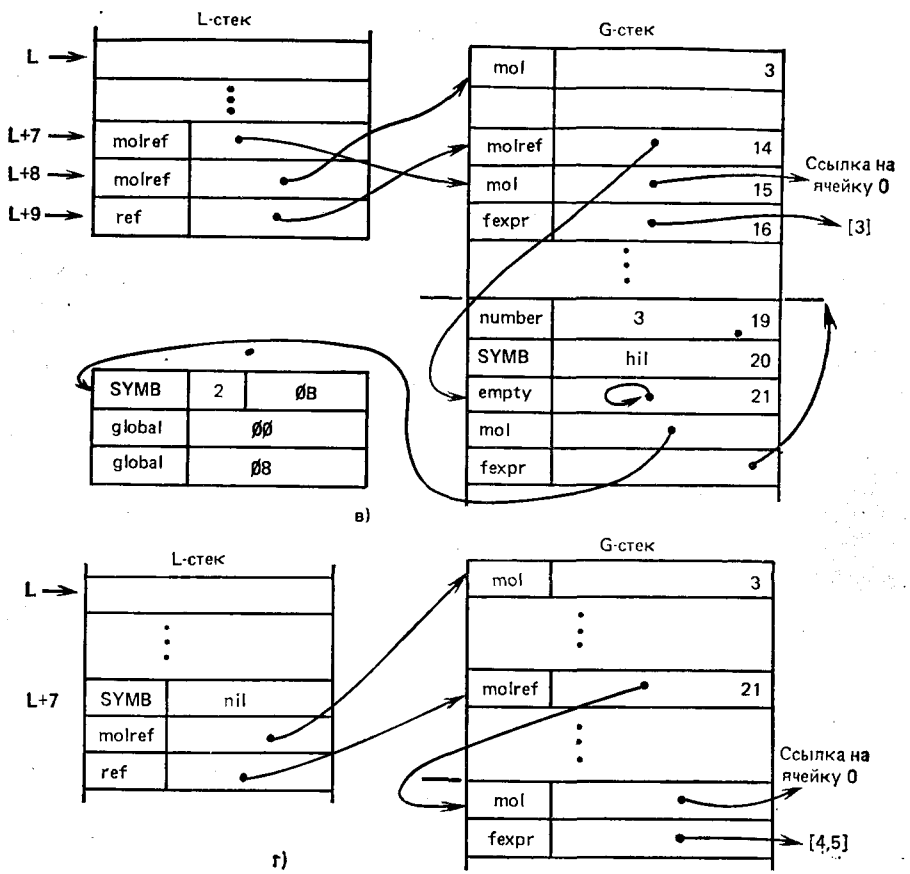


Рис. 6.6. Окончание

Результаты моделирования сведены в таблицу, где

r1 — отношение числа команд чтения-записи к общему числу команд;

r2 — отношение числа команд записи к общему числу команд;

r3 — отношение числа команд над регистрами к общему числу команд;

r4 — отношение числа команд чтения-записи из регистров к числу команд чтения-записи из памяти HOST-машины;

r5 — отношение числа команд над регистрами к числу команд чтения-записи из L- и G-стеков;

r6 — отношение числа команд чтения из HOST-памяти к числу команд записи в HOST-память;

r7 — отношение числа команд чтения из регистров к числу команд записи в регистры;

r8 — средний «вес» одной микрокоманды (по десятибальной шкале).

r	r1	r2	r3	r4	r5	r6	r7	r8
1	0,440	1,86	0,347	4,385	1,96	1,55	1,909	
1,5	0,444	1,878	0,351	4,308	2,04	1,33	2,00	
2	0,446	1,89	0,353	4,256	2,046	1,186	2,06	3,12
5	0,453	1,92	0,360	4,128	2,128	0,816	2,22	
10	0,456	1,93	0,363	4,07	2,1675	0,724	2,2	
20	0,458	1,94	0,365	4,03	2,23	0,712	2,33	

Из полученных данных можно сделать следующие выводы:

1. Примерно половина команд модельного эквивалента скомпилированной программы — это команды чтения-записи, причем команды чтения относятся к числу команд записи примерно в отношении 2 : 1, т. е. примерно каждая третья команда — это команда чтения.

2. Около трети команд приходится на регистры абстрактной машины.

3. Чтение-запись в регистры выполняются примерно в 2 раза чаще, чем чтение-запись из стеков.

4. Целесообразно вынесение регистров ПЛВ в отдельный регистровый файл, интенсивность работы с которым в 2 раза выше интенсивности работы с памятью. При этом выигрыш в быстродействии связан с оптимизацией трассы передачи информации между памятью и регистрами, а также между самими регистрами.

5. Примерно каждая пятая команда — это команда работы со стеком, что ставит под сомнение целесообразность выделения стеков отдельно от HOST-памяти, поскольку использование предварительного чтения данных из памяти позволяет сбалансировать время доступа к стеку и HOST-памяти.

6. Целесообразна опережающая выборка данных из регистров в силу п. 3.4.

7. Так как для кодирования микрокоманд предназначен принцип горизонтального кодирования, организация микропрограммного конвейера нецелесообразна: она не позволяет получить существенный выигрыш в быстродействии.

8. Основной выигрыш в быстродействии может быть получен за счет глобального распараллеливания трассы команд и данных в пределах ПЛВ с опережающей выборкой данных из HOST-памяти.

ГЛАВА 7.
РАЗРАБОТКА ПРОГРАММНО-АППАРАТНОГО ЭМУЛЯТОРА НА БАЗЕ ПЕРСОНАЛЬНЫХ ЭВМ

Аппаратная реализация системы логического вывода связана с проблемами организации системного интерфейса между специализированным ПЛВ и

основным процессором, выбором и отладкой системы команд Пролог-машины, работой с памятью, а также определением структуры программного обеспечения, организующего совместное функционирование процессоров. Кроме того, возникают проблемы тестирования и отладки узлов ПЛВ, отработки микропрограмм, для чего необходимо эмулировать выполнение набора команд абстрактной Пролог-машины в различных режимах.

В данной главе рассматриваются вопросы построения программно-аппаратного эмулятора Пролог-машины с набором команд, описанным в гл. 4. Представлены технические решения контроллера шины и памяти, обеспечивающие расширение адресного пространства памяти и ускорение доступа к памяти. Показаны особенности построения блоков управления и обработки на основе 32-разрядных СБИС.

7.1. ПРИНЦИПЫ ПОСТРОЕНИЯ ЭМУЛЯТОРА ПРОЛОГ-МАШИНЫ

Программно-аппаратный эмулятор Пролог-машины может быть реализован на базе персональной ЭВМ (ПЭВМ) класса IBM PC и ее отечественных аналогов ЕС1841 и ЕС1842. Эти ПЭВМ позволяют реализовать двухпроцессорную конфигурацию, содержащую основной процессор и сопроцессор, вместо которого используется эмулятор Пролог-процессора. При этом соглашения по интерфейсу с основным процессором ПЭВМ сохраняются. Здесь и далее вся система на основе ПЭВМ называется программно-аппаратным эмулятором Пролог-машины, а сопроцессор основного процессора ПЭВМ — аппаратным эмулятором Пролог-процессора.

Основная цель разработки эмулятора Пролог-процессора — обеспечить последовательную разработку и отладку микропрограмм, реализующих команды ПЛВ, а также наладку и тестирование блоков ПЛВ. При этом эмулятор последовательно усложняется за счет включения готовых отлаженных узлов. Для этого необходимо создать аппаратно-программный комплекс, обеспечивающий взаимодействие с пакетом программ, реализованным в среде ПЭВМ, совместимых с ПЭВМ IBM PC, и аппаратные средства, обеспечивающие эмуляцию аппаратных средств Пролог-процессора, а также разработать программные средства, обеспечивающие аппаратно-программный интерфейс с пользователем и регистрацию полученных результатов.

Для решения этих задач необходимо:

расширить непосредственно адресуемую процессором область системной памяти, чтобы обеспечить возможность моделирования Пролог-программ большого объема:

реализовать аппаратную поддержку доступа и обработки данных Пролог-системы;

реализовать аппаратно-программную поддержку унификации и указателей областей памяти Пролог-системы.

Чтобы обеспечить возможность эмуляции аппаратных средств Пролог-системы в режиме работы, максимально отображающем реальные условия в вычислительной системе, времена доступа к

системной памяти управляющей машины и Пролог-процессоров должны быть соизмеримы.

Система с аппаратным эмулятором Пролог-процессора может быть выполнена по схемам, приведенным на рис. 7.1. Подключение эмулятора Пролог-процессора по схеме на рис. 7.1,*а* сложно из-за необходимости дополнительных затрат на организацию взаимодействия пользователя с Пролог-системой в случае реконфигурации системы. Схема на рис. 7.1,*б* обеспечивает минимизацию аппаратной части эмулятора и максимальное использование ресурсов серийно выпускаемых ПЭВМ. Взаимодействие основного

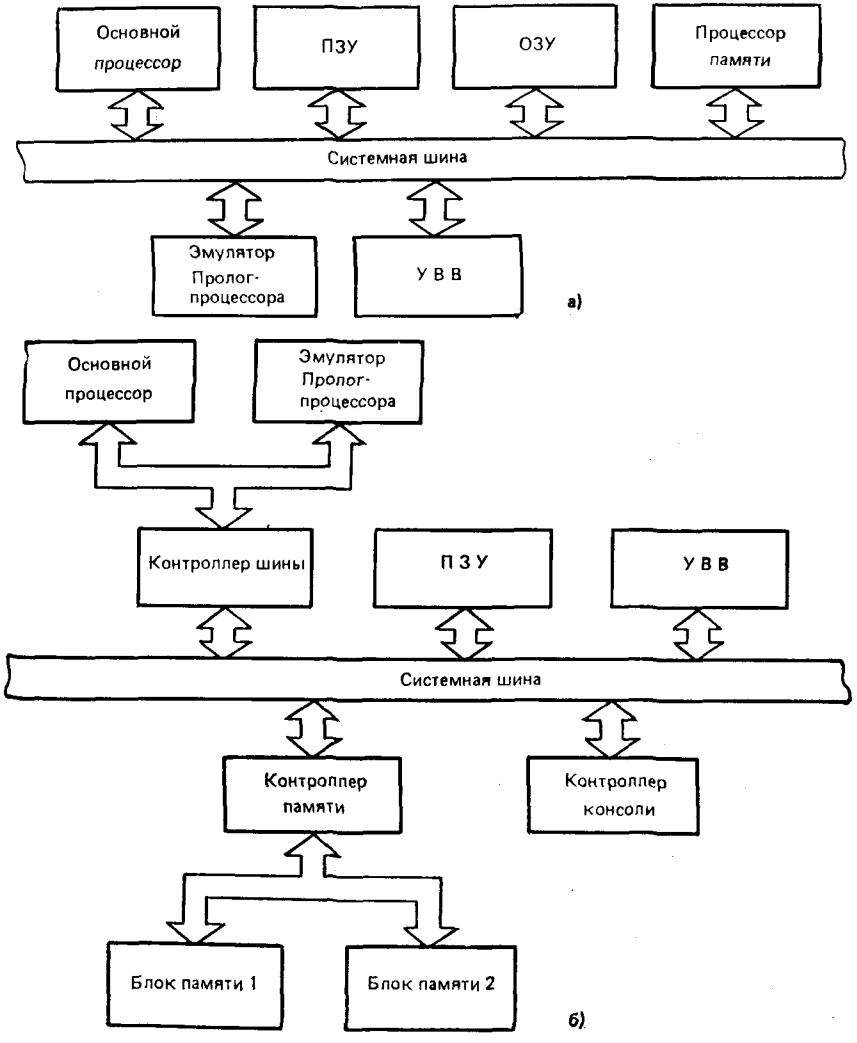


Рис. 7.1. Схемы подключения Пролог-процессора

...ществляется по внутренней шине микропроцессора 1810ВМ86 как сопроцессоров 1810ВМ87 и 1810ВМ89. Структура информационных потоков для этой схемы представлена на рис. 7.2.

Мультиплексируемая во времени шина данных используется всеми аппаратными устройствами, обеспечивающими эмуляцию Пролог-системы.

Контроллер шины соединяет мультиплексированную шину обоих процессоров с системной шиной ПЭВМ и отдельными шинами адреса и данных. Быстрый доступ к структурам данных ПЛВ и одновременно расширение пространства непосредственно адресуемой системной памяти обеспечивает контроллер памяти. При работе с основным процессором это устройство пропускает информацию шин адреса и данных через себя, полностью сохраняя логику работы механизма адресации микропроцессора 1810ВМ86. При работе с ПЛВ контроллер памяти обеспечивает быстрый доступ к структурированным термам Пролога.

Эмулятор Пролог-процессора включает только блок управления и блок обработки (см. описание микропрограммного ПЛВ в гл. 4), моделирующие взаимодействие и работу внутренних регистров ПЛВ. Область микропрограммной памяти блока управления в эмуляторе резервируется как область системной памяти, что обеспечивает эффективную коррекцию алгоритмов управления аппаратной частью эмулятора. Кроме того, внутренние регистры Пролог-машины имеют отображение на область системной памяти ПЭВМ. Подобная организация позволяет достаточно просто прослеживать трассу выполнения Пролог-программы. Передача содержимого регистров от (к) эмулятора Пролог-процессора (у) осуществляется через определенный в системе вектор прерывания, позволяющий инициировать в основном процессоре необходимую часть программного ядра поддержки модели.

Программное обеспечение эмулятора включает:

базовую систему ввода-вывода, совместимую с операционной системой MS-DOS;

пакет программного обеспечения для взаимодействия аппаратных средств эмулятора с ядром DOS;

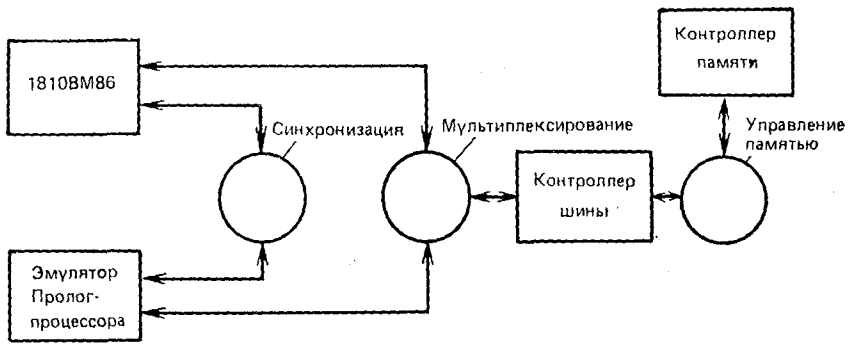


Рис. 7.2. Потоки информации в схеме на рис. 7.1,б

пакет программного обеспечения эмулятора Пролога, адаптированный к ПЭВМ, совместимым с IBM PC;

пакет программного обеспечения сервисных средств.

7.2. КОНТРОЛЛЕРЫ ШИНЫ И ПАМЯТИ*

Контроллер шины обеспечивает синхронизацию и подключение к системной шине основного процессора и аппаратного эмулятора Пролог-процессора. Контроллер памяти выполняет адресацию и управление блоками памяти для хранения информации, используемой в процессе логического вывода, а также микропрограмм, эмулирующих систему команд Пролог-процессора.

Контроллеры шины и памяти со стороны процессоров, получая информацию с мультиплексированной двунаправленной шины данных и адреса, слово состояния процессора (S0—S2), состояния сегментов S3, S4 и сигналы синхронизации доступа к системной шине ПЭВМ (RQ/GT0, RQ/GT1), формирует запросы шин адреса и данных, сигналы записи и чтения (RD\, WR\), а также сигналы B0—B3 выбора блока памяти. В качестве формирователей шин адреса и данных можно использовать серийно выпускаемые микросхемы 1810ИР82 и 1810ВВ86, которые позволяют обеспечить простой переход от мультиплексируемой шины 16-разрядного процессора 1810ВМ86 к системной шине ПЭВМ. Причем разрядность шины данных в зависимости от аппаратной конфигурации может меняться от 16 до 32 разрядов, а шины адреса — от 20 (при 16-разрядной конфигурации) до 32 разрядов. Подобная организация системного контроллера позволяет обеспечить простой переход к 32-разрядной шине.

Передача данных через контроллер шины (рис. 7.3) осуществляется за два такта обращения к основной памяти. Контроллер шины получает 32-разрядные слова данных либо от основного процессора (по входу МПД1), либо от эмулятора ПЛВ (по входу МПД2). Коммутацию требуемого направления выполняет мультиплексор MUX1 сигналом от схемы И, определяемым конъюнкцией сигналов $\overline{RQ/GT0}$&RQ/GT1. Мультиплексор MUX2 коммутирует адресные выходы основного процессора и эмулятора Пролог-процессора. Селектор Sel2 выделяет старшее (биты 16—31) и младшее (биты 0—15) полуслова слова данных, которые коммутируются на шину данных (ШД), входящую в состав системной шины наряду с шиной адреса (ША) и управления (ШУ), через выходные формирователи ВФД. Коммутацию старшего и младшего полуслов данных выполняет мультиплексор MUX3, управляющим входом которого является выход Т-триггера, переключаемого тактовым сигналом CLK от основного процессора. Адресный сумматор SM используется для увеличения на 1 выданного адреса для записи (чтения) старшего полуслова данных.

*В пам...

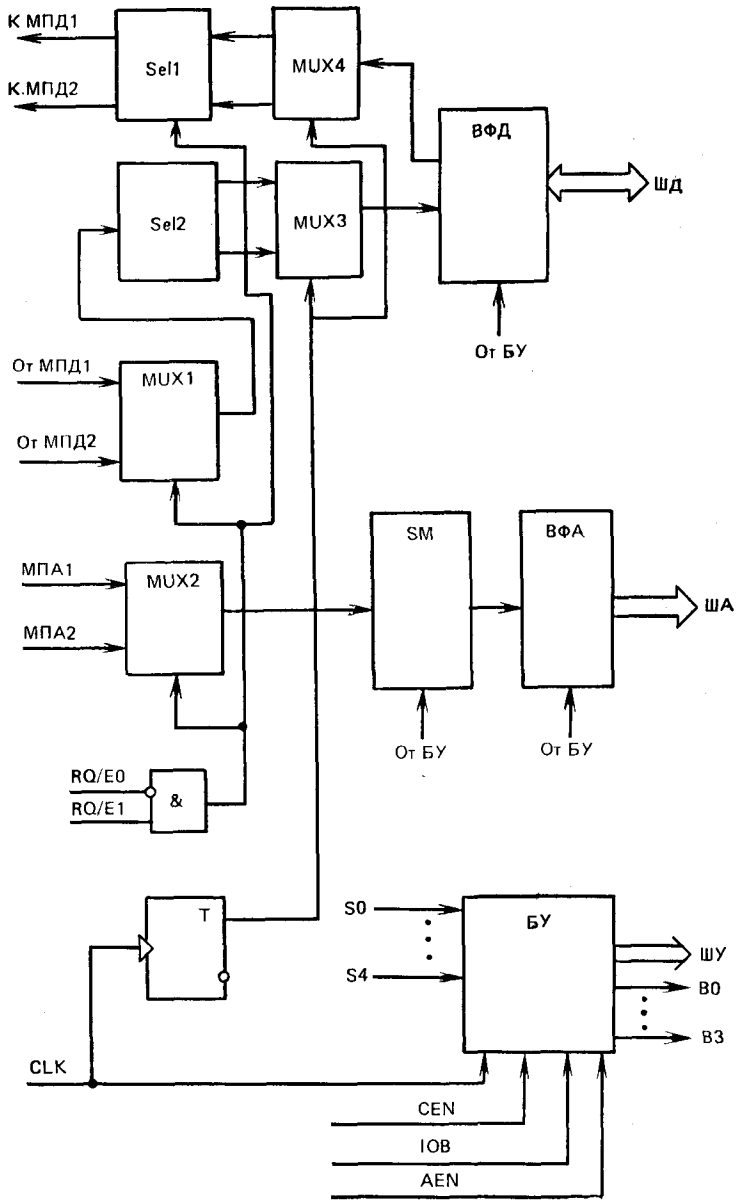


Рис. 7.3. Схема контроллера шины

Блок управления (БУ) выдает сигналы на ШУ и контроллер шины и может быть реализован на базе микросхемы К1810ВГ88, структурная схема которой представлена на рис. 7.4. Дешифратор состояний вырабатывает следующие сигналы состояний: подтверждение прерывания, ввод из устройства ввода-вывода (УВВ), вывод на УВВ, останов, выборка команды, чтение из основной памяти, запись в основную память, пассивное. Вход $\overline{AEN}$ управляет выдачей сигналов управления с выхода схемы формирования управляющих сигналов, сигнал IOB определяет либо работу с основной памятью, либо с УВВ. Сигнал CEN синхронизирует выдачу управляющих сигналов. Дешифратор блока памяти используется для выбора одного из четырех модулей памяти, которые можно включать в систему, расширяя ее адресное пространство в 4 раза.

Контроллер памяти (рис. 7.5) позволяет адресовать элементы структурированных термов с произвольной степенью вложенности. Напомним, что структуры представляются молекулами, для каждого элемента которой известно смещение относительно соответствующего вектора глобальных переменных. Если элемент структуры, в свою очередь, является структурированным термом, то для адресации элементов этого терма определяется адрес нового вектора глобальных переменных и соответствующие смеще-

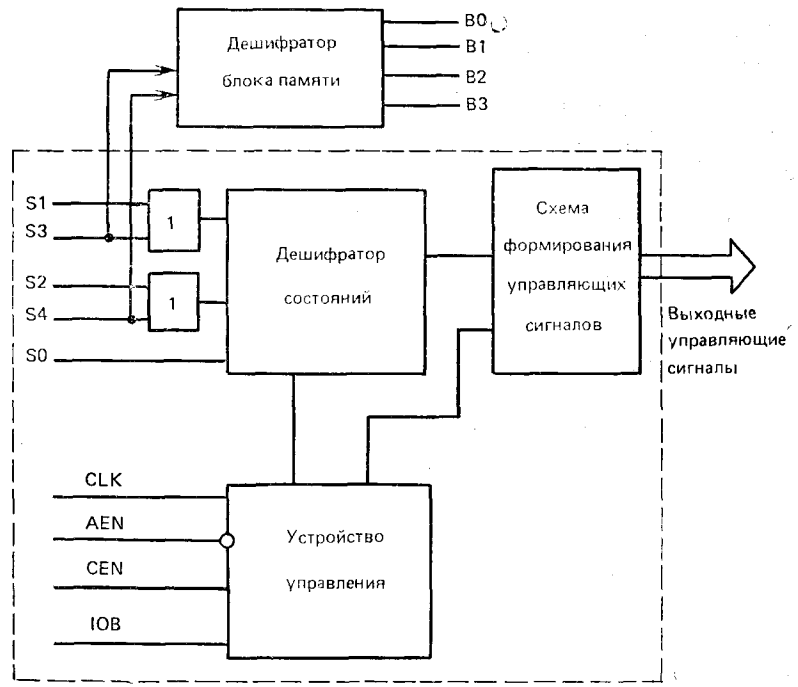


Рис. 7.4. Схема блока управления контроллера шины

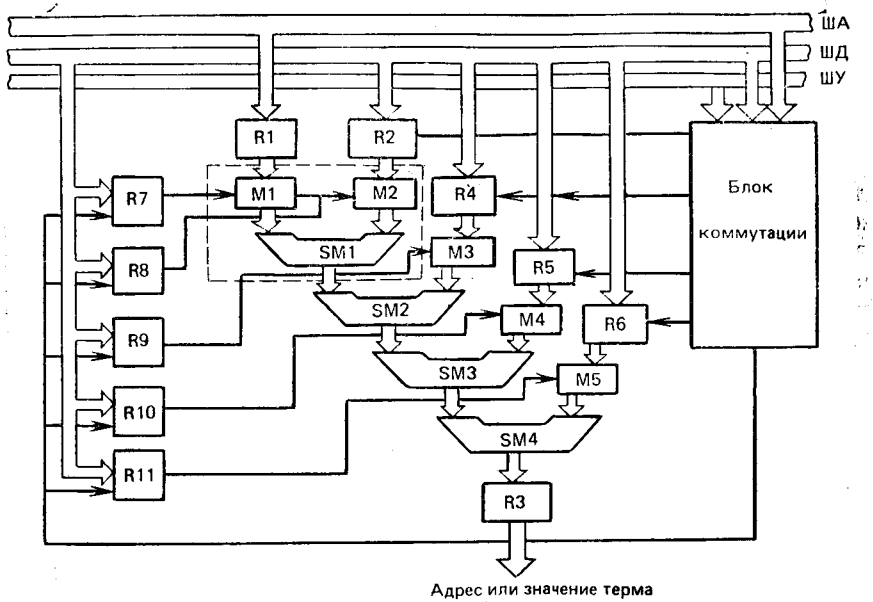


Рис. 7.5. Схема контроллера памяти

ния. Таким образом, общим выражением адреса элемента на k-м уровне вложенности в структуре является следующее:

$$[...[[A_0+Off_i]_0+Off_j]_1+ ... +Off_m]_k,$$

где A_0 — начальный адрес вектора глобальных переменных структуры с уровнем вложенности 0; Off_i — смещение для i-го элемента.

Мультиплексоры M1—M5 выполняют коммутацию нужных байтов операндов при вычислении адресов. Сумматоры SM1—SM4 используются для вычисления частных сумм вида A+Off. Смещения для элементов структурированных термов поступают по шине данных в регистры данных R2—R6. Схема на рис. 7.5 может быть наращена присоединением дополнительных каскадов, выделенных штриховой линией. Адрес A_0 заносится в регистр R1.

Контроллер памяти работает в режимах прямой, косвенной и непосредственной адресации. В первом случае адрес, подаваемый на входные регистры, пропускается на прямую в регистр R3. Во втором случае выполняется вычисление адреса по указанной выше формуле. Третий режим применяется для доступа к элементам списка. Управление записью осуществляет блок коммутации, который представляет собой дешифратор на 8 направлений.

Доступ к системной шине ПЭВМ осуществляется с помощью сигналов управления локальной шиной основного процессора ПЭВМ. При захвате шины основным процессором аппаратный эмулятор Пролог-процессора находится в режиме ожидания до тех пор, пока по внутренней шине процессоров не будет передан

известный эмулятору код префикса, указывающий, что следующая команда является командой ПЛВ. Для основного процессора этот код является несуществующим. По получении кода префикса эмулятор по локальной шине передает сигнал запроса шин адреса и данных ПЭВМ. Получив ответный сигнал (системные шины ПЭВМ свободны), аппаратный эмулятор Пролог-процессора «захватывает» их до окончания предписанных ему операций. По окончании работы он по той же локальной шине генерирует еще один сигнал, разрешая доступ к системной шине ПЭВМ основному процессору. Алгоритм работы повторяется.

Если по возникновению ошибки при выполнении программы необходим выход из Пролог-программы с последующим обращением к интерпретатору Пролог-системы, эмулятор генерирует аппаратное прерывание, которое после передачи управления шинами ПЭВМ центральному процессору обеспечит обработку необходимой информации средствами интерпретатора Пролог-системы.

7.3. ОСОБЕННОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭМУЛЯТОРА ПРОЛОГ-МАШИНЫ

Проектирование программного обеспечения эмулятора Пролог-машины включает два этапа:

разработка и адаптация системного программного обеспечения работы Пролог-системы на верхнем уровне;

разработка пакета программного обеспечения, позволяющего проводить реконфигурацию и отладку аппаратной части Пролог-системы как комплексно, так и отдельными частями с целью отработки микропрограмм Пролог-процессора.

Разработка системного программного обеспечения включает модификацию BIOS с целью введения системных прерываний, обеспечивающих взаимодействие основного процессора ПЭВМ и эмулятора, а также отладку системы. Системное программное обеспечение, располагаемое в резидентном ПЗУ, является неизменным в течение всего времени работы. При такой организации модификации ядра BDOS не потребуется, так как можно использовать системные функции операционной системы, начиная с версий MS-DOS 2.10 и выше, где осуществляется управление распределением системной памяти. Необходимо отметить, что для резидентного программного обеспечения MS-DOS дополнительные области памяти, вводимые в структуру эмулятора Пролог-машины, недоступны.

Система программной эмуляции Пролога реализуется на базе двух программных подсистем: интерпретатора и RTS (Run Time System), под управлением которой выполняется скомпилированный код Пролог-программы. Интерпретатор выполняет обработку встроенных предикатов, команды ввода-вывода, инициализацию регистров абстрактной Пролог-машины, обработку ошибок, создает управляющие таблицы, используемые в логическом выводе. RTS получает управление от интерпретатора всякий раз, ког-

да происходит обращение в область скомпилированного кода Пролог-программы. В этом случае код скомпилированной команды «перехватывается» программой монитора и передается на выполнение аппаратному эмулятору Пролог-процессора, который реализует ее, используя текст соответствующей микропрограммы в основной памяти. Кроме того, монитор должен обеспечивать выполнение следующих команд:

команды отображения и модификации областей памяти, формирующих содержимое регистров эмулятора Пролог-процессора;

команды отображения и модификации содержимого регистров эмулятора Пролог-процессора;

команды управления режимами работы (выполнение одного шага программы, трассировка программы и т. п.);

команды передачи управления интерпретатору Пролог-системы и возврата в режим управления монитором;

команды инициализации и модификации аппаратной части эмулятора.

Команды отображения и модификации областей памяти и регистров через монитор имеют отображение на собственный раздел файловой системы, что позволяет сохранять варианты отлаживаемого программного обеспечения в процессе работы во внешних файлах. Команды управления режимами работы позволяют выполнять пошаговый и непрерывный прогон программы, собирать статистику о частоте использования команд объектной Пролог-программы, выводить на экран содержимое областей памяти и регистров.

Модификация BIOS заключается во введении в нее множества команд инициализации аппаратной части эмулятора. В остальном структура BIOS сохраняется. Адреса векторов прерываний Пролог-процессора выбираются из тех, которые не задействованы при работе MS-DOS и BIOS. Векторы прерываний, обслуживающие аппаратную часть эмулятора, жестко определены структурой аппаратной части блока контроллера прерываний 1810ВМ59.

При инициализации программных блоков монитора векторы прерываний от дисплей-консоли оператора, содержат адреса точек входа в подпрограммы монитора для переключения от программ интерпретатора Пролога к программам монитора. Это позволяет при нажатии определенной комбинации клавиш переходить из программы интерпретатора Пролога в программу монитора, что обеспечивает независимую работу интерпретатора и Пролог-системы, а также регистрацию результатов работы интерпретатора Пролога и эмулятора.

7.4. ОРГАНИЗАЦИЯ АППАРАТНОГО ЭМУЛЯТОРА ПРОЛОГ-ПРОЦЕССОРА

Начальная структура эмулятора соответствует рис. 7.6. Обмен данными с эмулятором ведется по шине адреса, данных и

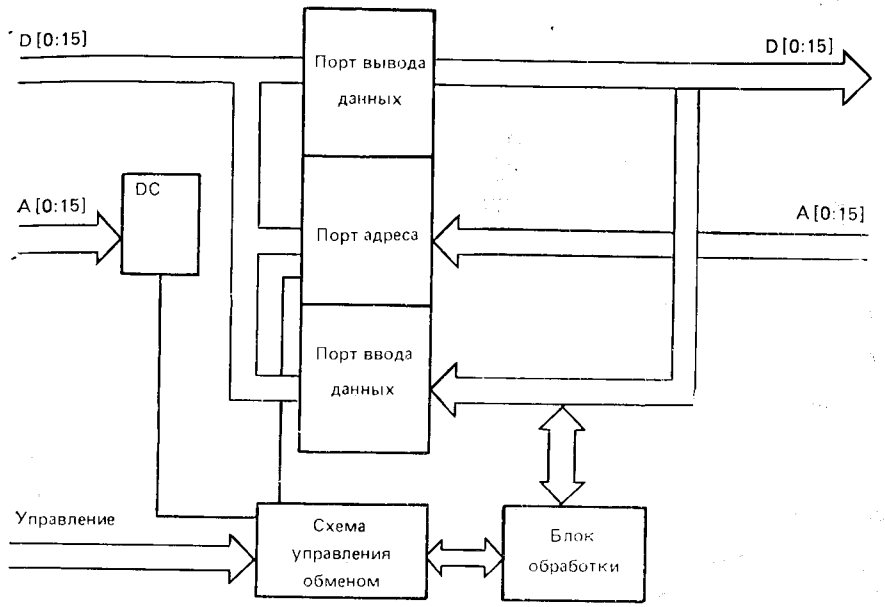


Рис. 7.6. Начальная схема эмулятора

управления с задержкой сигнала готовности о выдаче команды. Состояние основного процессора передается по шине управления в схему управления обменом. Как только встречается команда Пролога, основной процессор передает ее адрес и байты команды в соответствующие порты. Принятый адрес соответствует началу последовательности слов в ОЗУ, представляющих микрокоманды, образующие микропрограмму принятой команды. Далее выполняется считывание этих микрокоманд и их обработка в блоке обработки.

Схема, изображенная на рис. 7.7, является промежуточной между изображенной на рис. 7.6 и RISC-процессором, описанным в гл. 4. Аппаратный эмулятор Пролог-процессора моделирует работу RISC-процессора логического вывода. Для его реализации по схеме на рис. 7.7 использованы два функциональных блока — блок управления (БУ) и блок обработки (БО). Блок управления предназначен для выработки сигналов управления БО и обработки управляющих сигналов с ШУ.

Дешифратор команд предназначен для преобразования кода операции, получаемого с шины, в адрес микропрограммы, реализующей данную команду.

Секвенсор управляет последовательностью выборки микрокоманд и выполнен на СБИС — аналоге Am29331. Выбор этой СБИС обусловлен тем, что 16-разрядная шина адресов микрокоманд, внутренний стек (33 слова по 16 разрядов) и 12-разрядная шина данных, а также входы запроса и разрешения прерывания позволяют создавать эффективные микропрограммы.

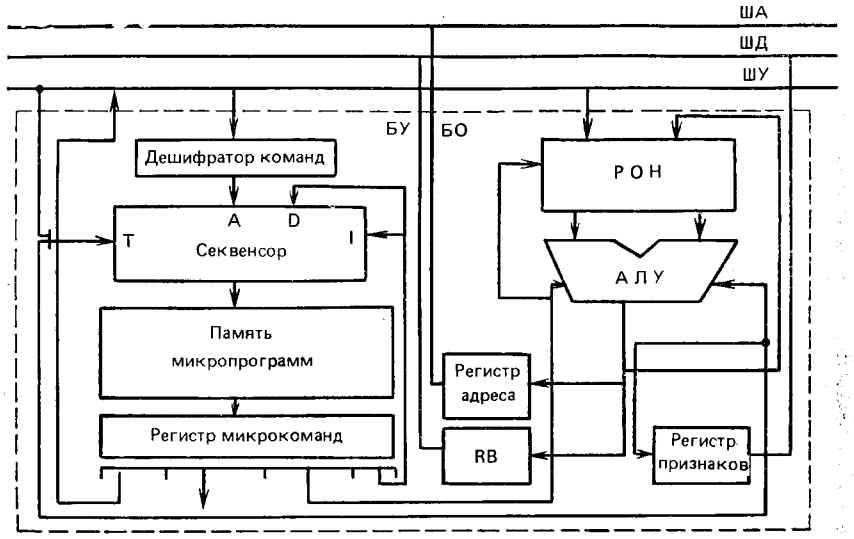


Рис. 7.7. Промежуточная организация эмулятора Пролог-процессора

Память микропрограмм состоит из ОЗУ и ПЗУ. В ПЗУ хранятся микропрограммы начальной инициализации, а ОЗУ позволяет быстро менять микропрограммы выполнения той или иной Пролог-команды. Микропрограммы загружаются в ОЗУ основным процессором по 16-разрядной шине данных, что требует введения дополнительных средств для развязки внутренних шин блока управления и внешних шин, а также для выбора нужной ОЗУ памяти микропрограмм.

Регистр микрокоманд предназначен для хранения текущей выполняемой микрокоманды.

Весь блок управления выполнен на базе серийно выпускаемых микросхем. Принципиальная схема блока управления в составе эмулятора ПЛВ приведена на рис. 7.8.

Выходной регистр микрокоманды собран на 13 8-разрядных регистрах R1 ... R13, информация в которые поступает из модулей ОЗУ W1 ... W13 и ПЗУ U1 ... U13. Адрес подается на входы A1 ... ... A12, а сигналы управления чтением-записью — на входы R/W, сигнал разрешения операции — на вход CS.

Дешифратор команд реализован на регистре R14 и двух микросхемах ПЗУ M1 и M2. Слово данных, считываемое из ПЗУ, коммутируется на адресные входы A[0 : 15] секвенсора SG, на входы данных D[0 : 15] поступает часть микрокоманд выходного регистра. Назначения входов секвенсора приведены в табл. 7.1.

Выходы секвенсора Y[0 : 15] используются для адресации модулей ПЗУ U1 ... U13 и ОЗУ W1 ... W13. Кроме того, часть выходов секвенсора поступает на ПЗУ C1, которое используется для возврата адреса продолжения через формирователи F1 и F2 на шину адреса в основной процессор. Формирователи выполняют

Таблица 7.1

Обозначение	Назначение
A0—A15	Входная шина адреса микрокоманды
D0—D15	Двунаправленная шина данных
RST	Начальная установка
CP	Синхросигнал
HOLD	Перевод шин D и Y в состояние 3 и Cin в низкий уровень
INTA	Двунаправленный сигнал подтверждения прерывания
INTEL	Разрешение обработки внешнего прерывания
INTR	Запрос внешнего прерывания
M0—M3	Шины многонаправленного ветвления в микропрограмме
OED	Разрешение выдачи информации на шину D
S0—S3	Шина выбора условия
SLAVE	Задание режима «ведомый» (аналогично СБИС АЛУ)
T0—T7	Входная шина проверяемых условий
T8—T11	Шина арифметических признаков (C—T8, N—T9, V—T10, Z—T11)
Cin	Входной перенос
FC	Внешний сигнал принудительного продолжения
I0—I5	Шина микрокоманд
ERROR	Сигнал ошибки формирования адреса в режиме «ведущий-ведомый»
EQUAL	Двунаправленный сигнал совпадения текущего адреса с адресом находящегося в регистре сравнения
AFULL	Двунаправленный сигнал заполнения 28 и более уровней стека
Y0—Y15	Двунаправленная шина

двунаправленный обмен шиной данных. Выбор направления определяется сигналами на входа OE$\varnothing$ и OE1. D-триггер T1 используется для разрешения работы БУ. Вырабатываемый на его выходе сигнал WRITE управляет записью в регистр R14. Сигнал WD на входе C D-триггера является внешним стробирующим сигналом.

Блок обработки предназначен для хранения содержимого регистров ПЛВ и выполнения различных операций с ними. Блок РОН содержит набор регистров машины Уоррена, а также регистры B1, B2, B3, RA, служащие для хранения байт команды, и регистр RB, используемый для выдачи данных на ШД. Код операции (байт B0) поступает непосредственно на дешифратор команд (регистр R14 на рис. 7.8). Функции коммутации и формирования направлений выполняет АЛУ, обеспечивая выборку байтов, вставку байтов в слово данных и сдвиги. Регистр RB может использоваться как порт вывода. По шине данных информация заносится в регистр RA через формирователи (не показаны) либо в другие регистры РОН. Информация в регистры B1, B3 записывается с ШД [8 : 15], а в регистр B2 — с [0 : 7] ШД.

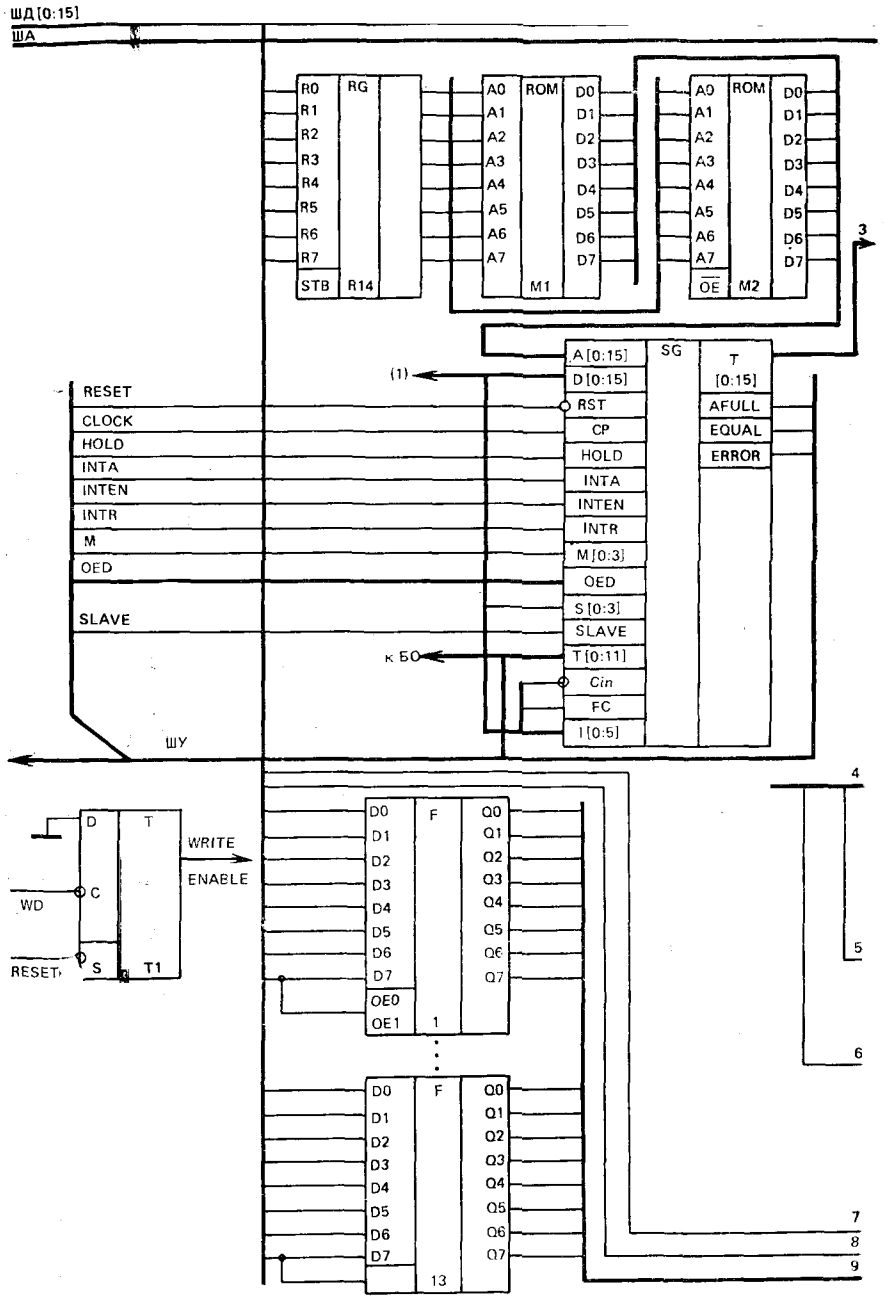
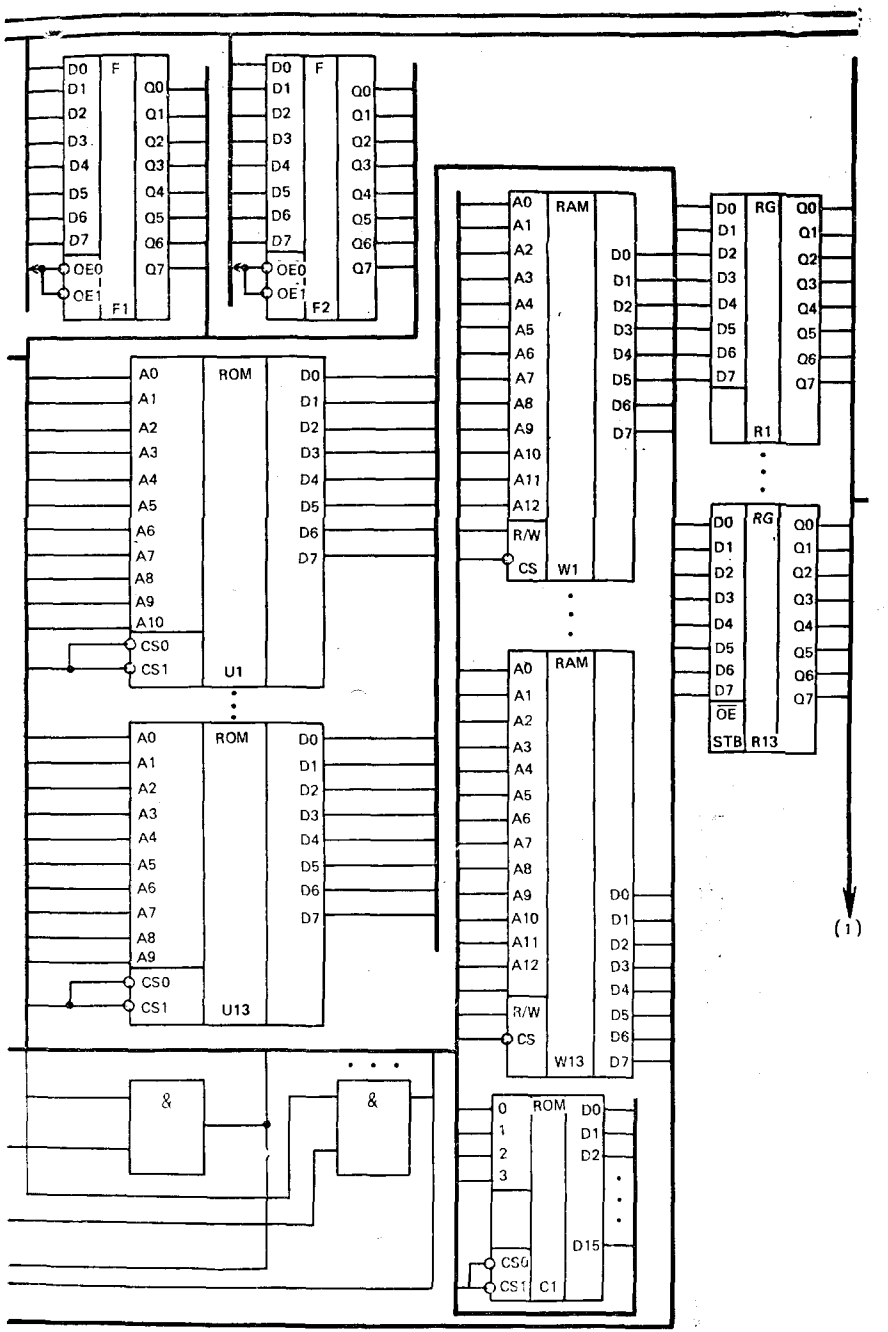


Рис. 7.8. Принципиальная схема эмулятора ПЛВ



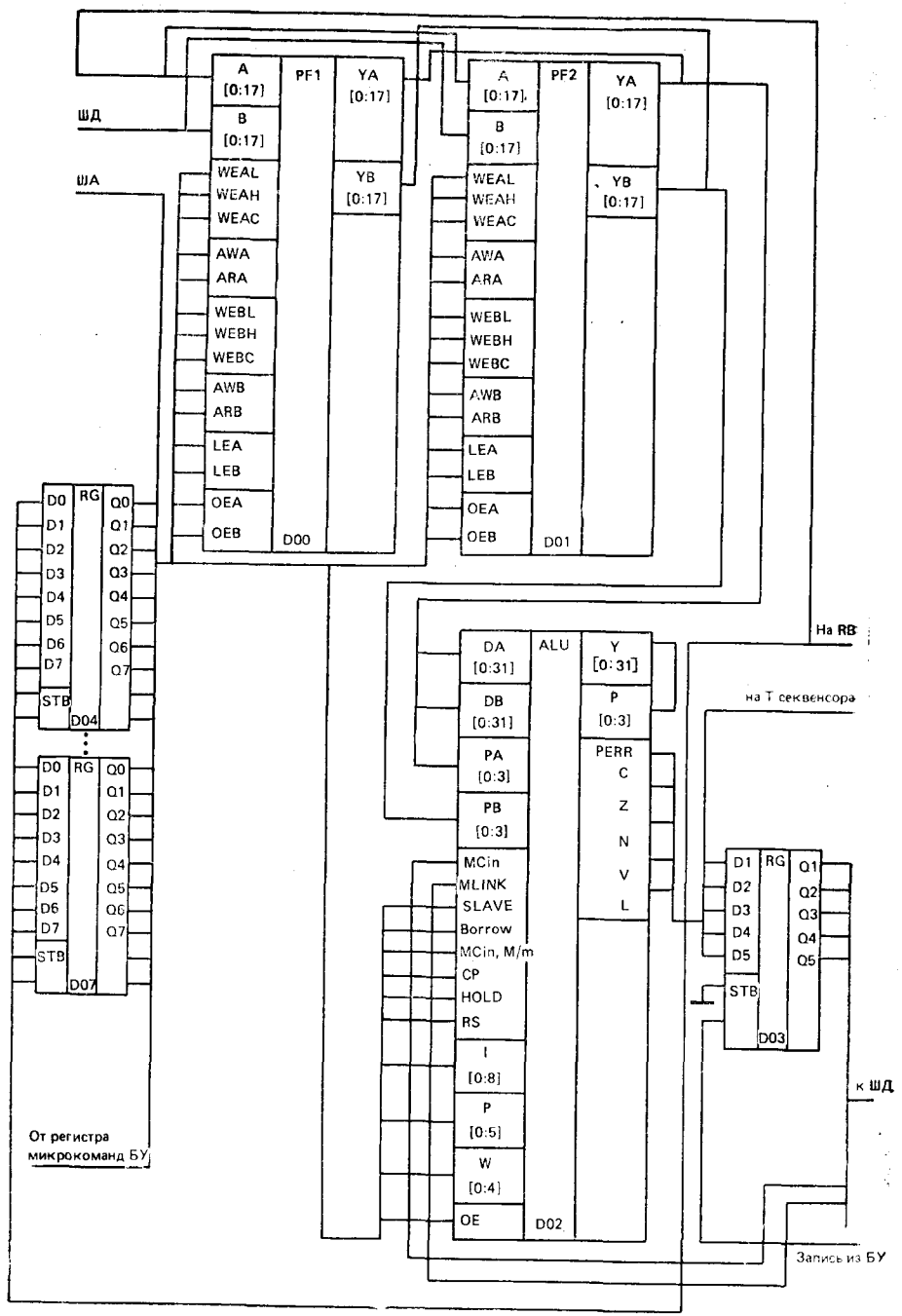


Рис. 7.9. Принципиальная схема блока обработки ПЛВ

Принципиальная схема блока обработки приведена в составе соответствующих схем ПЭВМ на рис. 7.9.

В состав блока обработки входят следующие микросхемы: D00, D01 — регистровые файлы, хранящие содержимое регистров ПЛВ; D02 — АЛУ; D03 — регистр признаков (флагов); D04—D07 — регистры адресов данных.

На входы B регистровых файлов подаются данные с внешней шины данных, которые записываются по адресу, выдаваемому блоком управления, входы A служат для записи данных в регистровые файлы с входа Y АЛУ по адресу из блока управления. Все необходимые сигналы для записи (считывания) в регистровые файлы формируются микропрограммно. Информация с выходов YA и YB регистровых файлов поступает на входы DA, DB АЛУ, на входы PA, PB АЛУ поступают по два старших разряда данных A и B каждого регистрового файла — биты контроля на четность-нечетность.

Из регистра микрокоманд на АЛУ подаются сигналы управления I[0 : 8] выбора поля P[0 : 5] и разрядности обрабатываемых данных W[0 : 4], сигнал разрешения выдачи OE=Y, а также некоторые управляющие сигналы. Все эти сигналы позволяют задать нужный режим работы АЛУ.

После выполнения заданной операции результат поступает на шину Y АЛУ, далее может выдаваться на внешнюю шину данных через буферные регистры-формирователи и на внешнюю шину адреса через регистры D04 ... D07 либо записываться в регистровый файл.

Таблица 7.2

Обозначение	Назначение
A0—A17	Шина порта A
B0—B17	Шина порта B
WEAL	Сигнал разрешения записи младшего байта порта A
WEAH	Сигнал разрешения записи старшего байта порта A
WEAC	Общий сигнал разрешения записи с порта A
WEBL	Сигнал разрешения записи младшего байта порта B
WEBH	Сигнал разрешения записи старшего байта порта B
WEBC	Общий сигнал разрешения записи с порта B
AWA0—AWA5	Адрес записи и считывания порта A
AWB0—AWB5	Адрес записи и считывания порта B
LEA	Сигнал записи в выходной буферный регистр порта A
LEB	Сигнал записи в выходной буферный регистр порта B
OEA	Сигнал разрешения выдачи информации на шину YA (при OEA=1 YA — в состоянии 3)
OEB	Сигнал разрешения выдачи информации на шину YB (при OEB=1 YB — в состоянии 3)
YA0—YA17	Шина порта A
YB0—YB17	Шина порта B

Таблица 7.3

Обозначение	Назначение
DA[0 : 31]	Шина данных
DB[0 : 31]	Шина данных
PA[0 : 3]	Биты отчетности шины DA
PB[0 : 3]	Биты четности шины DB
MCin **	Внешний входной перенос
MLINC **	Внешний сдвигаемый разряд
SLAVE **	Сигнал задания режима «ведомый» (SLAVE=1)
BOROW	Сигнал «заем» (BOROW=1) входных и выходных переносов в командах вычитания
M/m **	Сигнал управления входным переносом и сдвигаемым разрядом при сдвигах
CP	Сихросигнал (фронт при HOLD=0)
HOLD **	Сигнал разрешения записи в триггеры (при HOLD==1 содержимое рабочего регистра и регистров состояния сохраняется)
RS	Сигнал управления выдачей признаков состояния АЛУ (C, Z, N, V, L): C — выходной перенос; Z — признак нулевого результата; N — знак; V — переполнение; L — переполнение
I[0 : 8]	Шина микрокоманд
P[0 : 5]	Шина кода позиции *
W[0 : 4]	Шина кода ширины *
OE	Сигнал разрешения выдачи данных (OE=1 — шина Y в состоянии 3)
Y[0 : 31]	Шина результата при OE=0 работает на выход, при OE=1 в состоянии 3, в режиме «ведомый» на вход
PY[0 : 3]	Шина бит четности шины Y
PERR	Флаг ошибки паритетного контроля шин DA и DB
MSER	Флаг ошибки паритетного контроля выходной шины Y

* Задают операнды переменной длины.
** Для СБИС АЛУ ряд выводов запаяны жестко: M/m=1; MLINIC=0; MCin=0; SLAVE=0; BOROW=0; HOLD=0.

Признаки, формируемые АЛУ при выполнении операции, подаются на входы условий Т секвенсора и одновременно записываются в регистр признаков D03, после чего могут использоваться внешними устройствами либо при отладке. Выход PERR АЛУ сообщает о нарушении паритета. Назначения входов/выходов регистрового файла и АЛУ из МПК Am29300 приведены в табл. 7.2, 7.3.

7.5. ОПИСАНИЕ МИКРОКОМАНД

Назначение бит микрокоманды RISC-процессора, описанного в гл. 4, приведено в табл. 7.4. Для выполнения микропрограмм RISC-процессора на аппаратном эмуляторе разряды управления коммутацией на мультиплексорах и схемой коммутации и форми-

Таблица 7.4

Биты	Назначение
0	Управление записью в байт B1
1	Управление записью в байт B2
2	Управление записью в RA (разряды 0 ... 15)
3	Управление записью в RA (разряды 16 ... 31)
4	Управление записью в RB (разряды 0 ... 15)
5	Управление записью в RB (разряды 16 ... 31)
6	Управление шиной данных
7	Признак направления обмена по шине данных
8 ... 11	Управление блоком коммутации направлений. Номер направления задается двоичным кодом
12 ... 19	Адреса констант и адресов в ПЗУ
20	Управление коммутацией на MUX4
21	Управление коммутацией на MUX1
23	Управление коммутацией на MUX3
24, 25	Управление АЛУ
26 ... 33	Номер регистра в блоке РОН для записи
34 ... 39	Номер регистра в блоке РОН для чтения
40	Управление MUX6
41	Разрешение выдачи адреса на шину
42/43	Управление приемом-выдачей адреса на шине
44	Обнуление инкрементного регистра
45	Разрешение счета в инкрементном регистре
46 ... 48	Выбор типа проверяемого логического условия: 000 — равно, 100 — a<b, 010 — a—b. Остальные комбинации зарезервированы
49	Признак арифметических операций в АЛУ
50	Управление выходом MUX4
51	Управление коммутацией в MUX5
52	Сигнал запуска основного процессора
53	Запрос шины сопроцессора
54, 55	Управление портами ввода-вывода
56	Признак 32-разрядного слова
57	Проверяемое условие равно 1
58	Инкрементный переход
59	Признак перехода в микропрограмме (условный-безусловный)
60 ... 62	Номер проверяемого условия
63 ... 75	Адрес микрокоманды
76	Чтение из памяти
77	Запись в памяти
78	Запро на регенерацию
79	Сигнал запроса шины для связи с основным процессором (RQ/GT1)
80	Условие обращения к ПЛВ
81	Запись в РОН
82	Чтение содержимого регистра RB(0), RB(1)
83	Чтение содержимого регистра RB(2), RB(3)
84	Чтение РОН
85 ... 88	Адрес бита проверяемого условия (USW)
89	Значение проверяемого условия USW
90	Запись-чтение USW
91	Разрешение работы с USW
92	Значение устанавливаемого бита в USW
93	Запись в регистр B3
94	Входной перенос в АЛУ

Окончание табл. 7.4

Биты	Назначение
95	Запись в регистр адреса микрокоманды из регистра адреса возврата
96	Запись в регистр адреса возврата из регистра адреса команды

рования направлений не используются; функции инкрементного регистра выполняет регистр адреса блока обработки (рис. 7.6).

Для написания микропрограмм использован мнемокод, формат микроопераций в котором следующий:

⟨тип_объекта⟩[индекс]⟨операция⟩⟨параметры⟩

Здесь тип_объекта соответствует: REG — регистр; MUX — мультиплексор; ALU — АЛУ; MEM — динамическая память; ROM — постоянная память; COM_DIR — блок коммутации направлений; CT — счетчик; T — триггер и т. д.; [индекс] определяет порядковый номер объекта на схеме; ⟨операция⟩ задает операцию, выполняемую на данном объекте, например:

REG [1] read — операция чтение в регистр 1;

MUX [2] first — коммутация первых входов на втором мультиплексоре;

TR[1] set — установка триггера 1 в единицу;

ALU add — сложение на сумматоре.

Имеется особая операция, не связанная с конкретным элементом схемы, — операция условного и безусловного переходов в микропрограмме.

Описание микрокоманды условного перехода:

COND (X, 1, адрес)

Здесь X — проверяемое логическое условие: если X=0, выполняется переход на следующую микрокоманду; адрес — адрес (номер микрокоманды) передачи управления, если X=1.

Описание безусловного перехода:

COND (,, адрес)

Запись

X := ⟨выражение⟩

присваивает X значение выражения (0, если выражение ложно, и 1, если истинно).

Комментарии записываются между ограничителями

/*.....*/

Вызов микропрограмм обозначается так:

> имя микропрограммы

Общая структура микропрограммы имеет следующий вид:

```
1: микрооперация 1.1,
   микрооперация 1.2,
. . . . . .
   микрооперация 1.n 1
2: микрооперация 2.1,
   микрооперация 2.2,
. . . . . .
   микрооперация 2.n 2
   .
   .
   .
K: микрооперация k.1,
   микрооперация k.2,
   . . . . . .
   микрооперация k. nk.
```

Каждая микрокоманда имеет метку, отделяемую от остальной части числом с двоеточием, например N:, где N определяет относительный адрес микрокоманды. Каждая микрокоманда состоит из параллельно выполняемых микроопераций, записываемых в произвольном порядке и разделяемых запятыми.

Примеры базовых микропрограмм

1. Пересылка типа регистр — регистр:

```
1: REG[Ri] read,
   MUX[5] second.
   REG[Rj] write
```

2. Пересылка типа регистр — память:

```
1: REG[Ri] read,
   MUX[2] second,
          COM_DIR 6, /*коммутация направления 6*/
   MUX[4] second,
   REG[RB] write
2: REG[Rj] read,
   MUX[6] second,
   REG[Addr] write /*запись адреса в регистр адреса */
3. REG[RB] read_first_half /*чтение разрядов 0 ... 15 регистра RB*/
   REG[Addr] read, MEM write
4: REG[Addr] increment /*увеличение адреса на 1*/
5: REG[RB] read_second_half /*чтение разрядов 16 ... 31 регистра RB*/
   REG[Addr] read, MEM write
```

3. Операция REG i<—REG j+constant,
   где constant — константа из ROM.

```
1: REG[j] read,
   MUX[2] second,
   COM DIR 2,
   ROM read (constant),
   MUX[1] first,
```

MUX[3] first,
ALU add
2: MUX[4] first,
MUX[5] first,
REG[i] write

В качестве примеров кодов команд рассмотрим следующие микропрограммы:

Start match — команда генерируется как первая команда для Пролог-кода утверждения, выполняет запись в регистр G1 адреса первого свободного элемента глобального стека, по которому будут записываться молекулы во время унификации. Но если среда не содержит глобальных переменных, то эти операции выполняться не будут, эта команда не генерируется и Пролог-код утверждения начинается непосредственно с блока команд унификации.

```
1: REG[G] read
   COM DIR dir /*коммутация B1 ||B2*/
   MUX[3] second
   ALU add
2: MUX[3] second
   ALU add
   MUX[4] first
   MUX[5] first
   REG[G1] write
```

Invoke — команда вызова скомпилированных предикатов, определенных пользователем:

```
1: REG[P] read
   COM DIR /* коммутация направления, соответствующего RA */
   MUX[2] second
   MUX[4] second
   REG[R6] write
2: COM DIR dir /* коммутация направления, соответствующего B1|| B2||
     B3 */
   REG[R3] read
   MUX[1] second
   MUX[3] first
   ALU add
3: ALU add
   MUX[4] first
   MUX[5] first
   REG[P] write
   End /* переход по адресу в регистре P */
```

7.6. СЕРВИСНЫЕ СРЕДСТВА МИКРОПРОГРАММИРОВАНИЯ

Технология отладки аппаратного и микропрограммного обеспечения эмулятора Пролог-процессора предусматривает наличие программных инструментальных средств, позволяющих создавать наборы (файлы) микропрограмм и выполнять различные функции по их ведению (печать, корректировка и т. п.). С этой целью разработана программа, выполняющая указанные функции, с использованием средств системы программирования Turbo Pascal 5.0.

Для трансляции микропрограмм, написанных на мнемокоде, используется транслятор, который формирует специальный файл. Микропрограммы находятся в файле с последовательным доступом, длина записи фиксирована — 40 байт. Заголовок файла находится в первой записи и содержит следующую информацию:

количество микрокоманд в файле (1-е поле, 2 байта),
количество микропрограмм в файле (2-е поле, 2 байта),
резервное поле (38 байт).

Записи, начиная со второй, содержат следующие поля:

бинарная часть или собственно микрокоманда (1-е поле, 13 байт),
мнемоника или комментарий микрокоманды (2-е поле, 27 байт).

Наличие комментария облегчает работу по ведению файла, так как отражает целевое назначение микрокоманды.

После микрокоманд файл содержит таблицу микропрограмм, количество элементов которой равно количеству микропрограмм, находящихся в файле. Каждая запись таблицы имеет следующие поля:

имя микропрограммы (1-е поле, 30 байт);
адрес (номер записи в файле) первой микрокоманды (2-е поле, 2 байта);
адрес (номер записи в файле) последней микрокоманды (3-е поле, 2 байта), предполагается, что микрокоманды расположены последовательно.

Таблица микропрограмм позволяет осуществлять быстрый доступ к той или иной микропрограмме. Размер первых двух полей заголовка и последних двух полей записи таблицы микропрограмм фиксированный, поскольку ни длина микрокоманды, ни количество микрокоманд не могут иметь значений, превышающих 32 767 (максимальное число, представляемое двумя байтами). Размер остальных полей зависит от длины микрокоманды и записи файла, которые представлены в программе поименованными константами и могут легко модифицироваться, алгоритмы обработки файла не изменяются. В ходе развития интерфейса между программой и пользователем длина микрокоманды и записи может задаваться в процессе диалога, если пользователя, конечно, не устраивают значения, принимаемые по умолчанию.

При создании файла предполагается, что разряд с номером 0 — самый младший в микрокоманде, расположенный в самом младшем байте микрокоманды.

Программа обеспечивает следующие режимы работы:

4) создание файла микропрограмм;
2) просмотр микропрограмм;
3) печать файла с микропрограммами;
4) корректировку.

В режиме 1 пользователь указывает имя создаваемого файла,

после чего у пользователя запрашивается имя вводимой микропрограммы (пустой ввод является признаком конца файла) и мнемоника вводимой микрокоманды (пустой ввод является признаком конца микропрограммы).

Пользователь вводит через пробел позиции разрядов, в которых данная микропрограмма имеет 0 (пустой ввод является признаком того, что таких нулей нет) или 1 (пустой ввод является признаком того, что таких единиц нет) в отличие от нейтральной микрокоманды. Если строки ввода (127 символов) не хватает, в последней позиции вводимой строки следует набрать символ '&' и нажать 'ВВОД', после этого будет представлена дополнительная строка ввода. Если количество введенных микрокоманд равно 0, файл не создается.

В режимах 2 и 3 файл вводится на экран (режим 2) или принтер (режим 3), возможны два варианта вывода:

16-ричного представления бинарной части микрокоманд;

позиций разрядов, в которых данная микрокоманда имеет отличия от нейтральной.

На экран выводится имя микрокоманды, номер (адрес) микрокоманды внутри микропрограммы, содержимое бинарной части в одном из двух форматов и комментарий, если пользователем был указан соответствующий признак.

В режиме корректировки на экран выводится символ приглашения '>', после которого пользователь может дать одну из следующих команд (регистр произвольный):

для работы с файлами

P[>] — вывод списка микропрограмм файла;

W — выход из режима корректировки с сохранением изменений;

N — выход из режима корректировки без сохранения изменений.

Для работы с микропрограммами

K prog_name — корректировка микропрограммы с именем prog_name,

S — завершение корректировки программы с сохранением изменений,

Q — завершение корректировки программы без сохранения изменений,

U[prog_name] — удаление микропрограммы с именем prog_name,

L[>] — вывод корректируемой микропрограммы;

для работы с микрокомандами

0 n, n1, n2,... — установить 0 n-й строки в n1, n2, ... разрядах,

1 n, n1, n2,... — установить 1 n-й строки в n1, n2, ... разрядах,

I n — вставить нейтральную микрокоманду после строки n,

D n — удалить микрокоманду n,

Z n — вставить нейтральную микрокоманду вместо строки n,

T n[> — вывести микрокоманду n.

Команда P выводит имена из таблицы микропрограмм и размер микропрограмм в записях. Команда W переписывает существующий файл, внося в него сделанные изменения; если изменений не было, команда не выполняется. Команда K фиксирует корректируемую микрокоманду, создавая рабочую область для нее. Если микропрограмма зафиксирована, команду U можно давать без параметров. Команда L осуществляет вывод рабочей области (на экран или принтер). Параметр '>' указывает на необходимость вывода на принтер.

Смена режимов возможна без повторной загрузки, пока пользователь не даст положительного ответа на запрос программы о завершении работы.

7.7. СРЕДСТВА ОТЛАДКИ

В зарубежной практике для отладки спецпроцессоров используется станция на основе микропроцессора 68020 фирмы Motorola, имеющего быстродействие на порядок выше, чем микропроцессор фирмы Intel. Для таких станций наработано значительное количество отладочного программного и аппаратного обеспечения. Все это говорит о высокой обеспеченности разработчиков спецпроцессоров. Ввиду практического отсутствия в СССР станций на основе микропроцессора Motorola, недостатка опыта работы с ним, отсутствия других 16- и 32-разрядных систем опишем отладку с использованием ПЭВМ ЕС1840.

На начальном этапе проверяется прохождение сигналов, обеспечивающих минимальную работоспособность схемы, путем зацикливания управляющих программ, индивидуальных для каждого сигнала, в отладчике (DEBUG).

На следующем этапе производится отладка алгоритмов функционирования, выявляются как аппаратные, так и программные ошибки.

Более подробно этапы наладки процессорного модуля показаны на рис. 7.10.

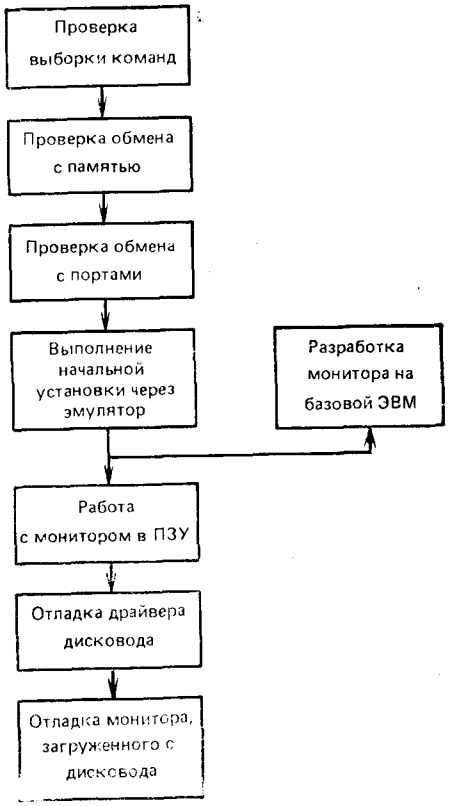


Рис. 7.10. Схема алгоритма наладки ПЛВ

Проверка основных операций по обмену с памятью, с портами ввода-вывода осуществляется загрузкой программ в ПЗУ по адресу начала работы микропроцессора по сбросу (FFFF0 для К1810ВМ86). Сброс подается либо от генератора, либо вручную от схемы сброса. Для проверки программы начальной установки используется уже эмулятор.

Станция отладки обеспечивает отладку ПЛВ на базе RISC-процессора для проверки алгоритмов команд машины Уоррена; предусмотрены хранение микропрограмм в памяти основного процессора, занесение микрокоманд в порты по шине MULTIBUS, выдача микрокоманды на исполнение по специальному сигналу, задержка исполнения микрокоманды, перевод основного процессора в состояние ожидания (WAIT), возврат управления основному процессору по сигналу TEST, анализ условий и переход в соответствии с данной микрокомандой. Здесь реализуется пошаговый режим выполнения микрокоманды — останов по заданному адресу микрокоманды, заданному условию, с переходом на нужный адрес микрокоманды по заданному адресу микрокоманды и заданному условию, а также режим выполнения команд — останов по заданному адресу команды, по заданному условию, с переходом на нужный адрес команды по заданному адресу команды и заданному условию, сервисные функции отображения памяти, пересылки и изменения данных.

Технология отладки

Для отладки аппаратного и программного обеспечения сопроцессора разработан специализированный отладчик, который выполняет:

просмотр и изменение областей памяти, включая и регистры спецпроцессора;

пошаговую обработку микрокоманд;

микрокоманды с остановом по адресу микрокоманды;

микрокоманды с остановом по условию;

микрокоманды с остановом по адресу и переходом по определенному адресу микрокоманды;

микрокоманды с остановом по условию и переходом по определенному адресу микрокоманды;

Пролог-команды с остановом по адресу и переходом по определенному адресу команды;

Пролог-команды с остановом по условию и переходом по определенному адресу команды;

пересылку областей памяти, включая регистры Пролог-процессора.

При отладке ПЛВ последовательно реализуются следующие этапы:

1. Проверка записи в порты эмулятора Пролог-процессора, перезаписи из них в регистр микрокоманд.

2. Проверка отдельных полей микрокоманды. В первую очередь проверяется запись в регистры — производится обращение к внешним регистрам B1, B2, RA, RB как к портам ввода-вывода, а запись в РОН — через внешние регистры заданием соответствующей микрокоманды. Затем проверяются поля микрокоманды, управляющие селектором при предварительной инициализации регистров-источников, а также работа мультиплексоров при проверке регистров.

3. Выполнение базовых микропрограмм (BAS):

проверка записи в память выполнением микропрограммы, считывающей содержимое регистра в регистр RB, откуда оно вводится в память (ОЗУ) по команде IN, а затем считывается содержимое соответствующего адресного регистра и производится программная запись HOST-процессором;

проверка чтения из памяти производится путем микропрограммного считывания адреса из ПЛВ, программной выборки нужной ячейки, записи прочитанных данных в RB, микропрограммной перезаписи в требуемый регистр.

Вызов программно-микропрограммной процедуры чтения (записи) памяти инициируется переходом с возвратом в тексте микропрограммы. В теле этих процедур имеются и соответствующие команды задания адреса регистра.

Анализ в микрокоманде необходимости записи в регистр осуществляется перед выдачей каждой микрокоманды. После анализа выполняется запись через соответствующий порт и переход к следующей микрокоманде.

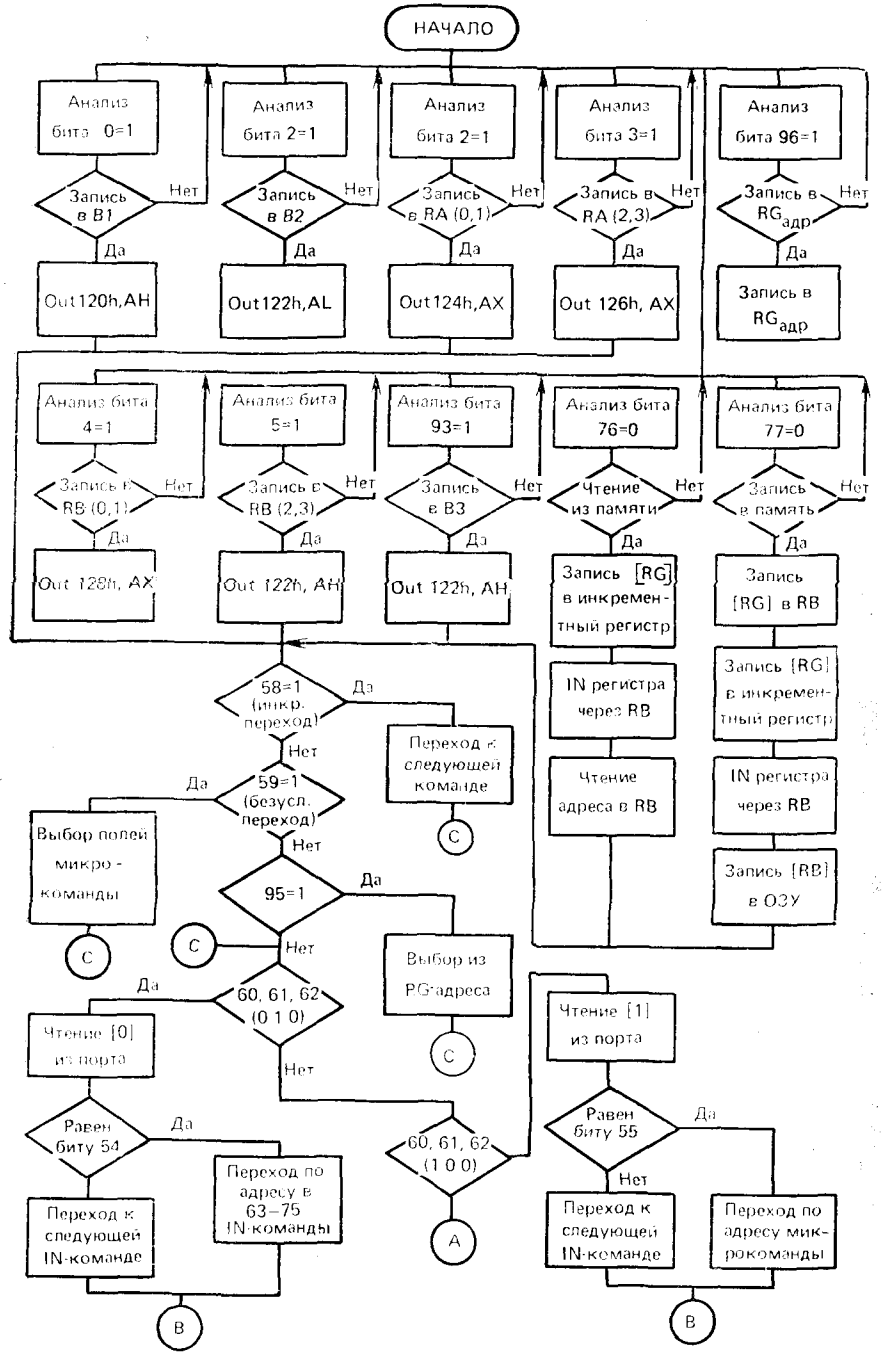
Алгоритм программы исполнения микрокоманды приведен на рис. 7.11. Имеется следующие соответствие адресов портов и регистров B1 — 120h, B2 — 121h, B3 — 122h, RA (0,1) — 124h, RA (2, 3) — 126h, RP (0, 1) — 128h, RB (2, 3) — 120h.

С целью исключения нарушения содержимого регистра RB при операциях чтения-записи памяти оно предварительно сохраняется, а затем в конце операции восстанавливается. Программа предназначена для имитации исполнения микрокоманды и заменяет работу блока управления в реальном автономном ПЛВ, помогает при обработке алгоритмов Пролог-команд.

Программное обеспечение отладки

Для отладки программно-аппаратного обеспечения ПЛВ разработан монитор моделирующей системы, обеспечивающий возможность выполнения следующих команд.

d [адр1] [,число] — просмотр содержимого памяти, где адр1 — адрес начала отображаемой области; число — количество отображаемых байт, по умолчанию 100H. Все параметры команды могут быть опущены, при этом осуществляется вывод очередных 256 байт, следующих после области памяти, просмотренной предыдущей командой d. На экран выводятся как 16-ричное представление очередного байта, так и в ASCII-кодах, соответствую-



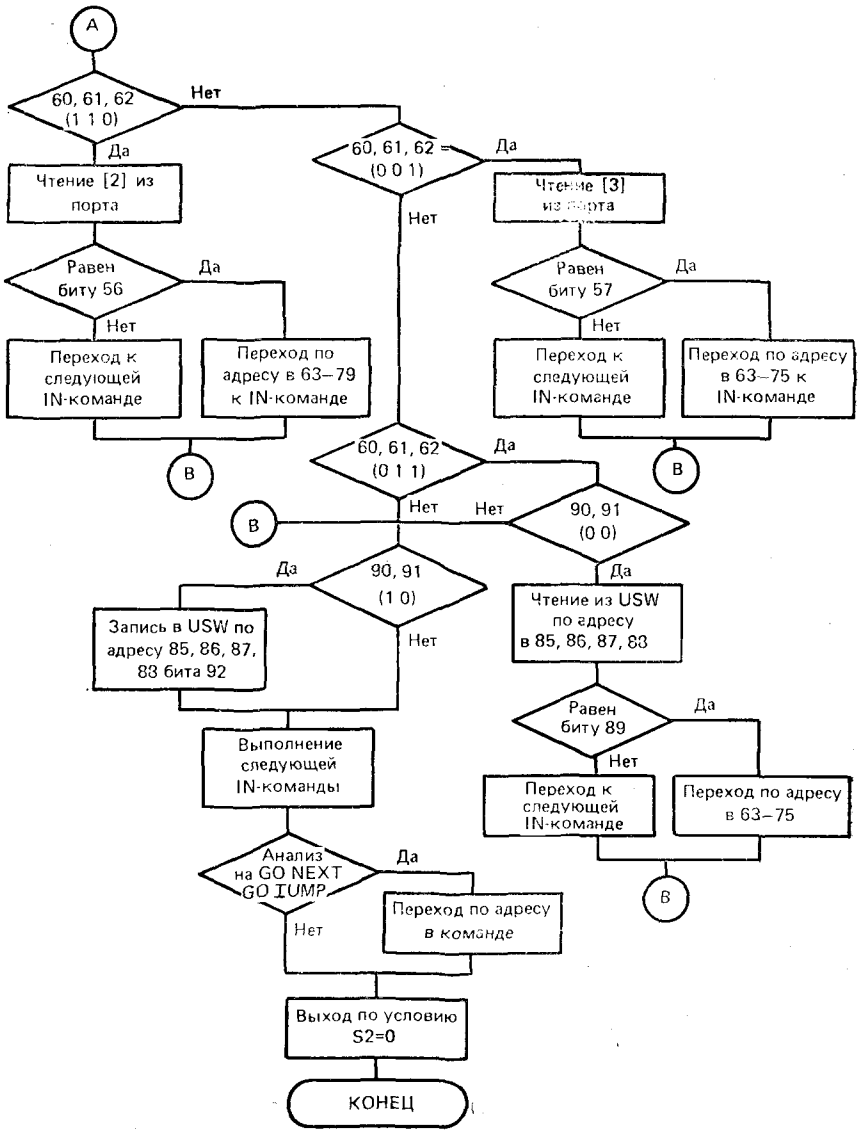


Рис. 7.11. Схема алгоритма тестирования

щих данному значению; символы, значение кода которых менее 32, выводятся в виде точки.

m адр1, адр2, число — пересылка содержимого памяти начиная с адреса адр1 по адресу адр2; количество пересылаемых байт определяется параметром «число».

r [имя регистра-число] — вывод содержимого регистров ПЛВ на экран в 16-ричной форме; указав в качестве параметра имя

регистра и его новое значение, можно по ходу выполнения программы изменять содержимое регистров.

g адр1 [, адр2] — выполнение команд начиная с адреса, определяемого значением параметра адр1. Если указан второй параметр, то выполнение продолжается до команды, находящейся по адресу адр2. Если второй параметр не указан, то выполнение продолжается, пока не будет выполнена команда RETF. После выполнения заданного участка программы на экран выводится содержимое регистров.

с адр1, код [число] — модификация содержимого памяти — заполнение памяти начиная с адреса, определяемого значением первого параметра, 16-ричным кодом, который является вторым параметром команды; третий параметр по умолчанию равен 1. Произведение длины второго параметра (в байтах) на число повторений не может быть больше 65 535.

t — пошаговое выполнение программы — выполнение очередной инструкции, после чего на экран выводится содержимое регистров.

p адр 1 [=число] — вывод содержимого порта ввода-вывода на экран в 16-ричной форме; если указан второй параметр, то в порт выводится заданное значение.

При задании команды параметры представляются в виде

$$\text{адр} := \text{XXXX} : \text{YYYY},$$

где XXXX — адрес сегмента; максимальное значение OFFFFH.

Поскольку существующие отладчики предназначены в основном для работы с HOST-процессором или в лучшем случае с имеющимися для них арифметическими сопроцессорами, т. е. не могут обеспечить доступ к регистрам разрабатываемого сопроцессора, необходимо разработать специализированный отладчик, ориентированный на уникальную архитектуру сопроцессора. Следует учесть, что отладчик будет работать с файлами микропрограмм, имеющими специфическую структуру. Отладчик выполняет следующие функции:

1) инициализация областей оперативной памяти, содержащих скомпилированный код и данные, необходимые для работы сопроцессора, т. е. на отладчик возлагаются функции, выполняемые интерпретатором Пролога в случае программной поддержки скомпилированного кода; причем скомпилированный код для сопроцессора несколько отличается от программно поддерживаемого прототипа;

2) просмотр и корректировка структур данных, находящихся в оперативной памяти, при этом возможна интерпретация содержимого памяти не только двоичная, шестнадцатеричная или ASCII, но и в терминах структур данных Пролога;

3) загрузка с диска и размещение в оперативной памяти файла, содержащего микропрограммы Пролог-команд, т. е. программное моделирование ПЗУ микрокоманд, что позволяет на этапе отладки микропрограмм осуществлять их корректировку, а также создавать внутри них точки останова;

4) запись на диск области оперативной памяти, что позволяет сохранить, например, скорректированный файл микропрограмм;

5) выполнение одной очередной микрокоманды;

6) выполнение одной очередной Пролог-команды;

7) создание точки останова внутри микропрограммы;

8) создание точки останова между Пролог-командами;

9) просмотр и корректировка содержимого регистров сопроцессора, причем возможен просмотр всех или группы регистров.

На этапе отладки отладчиком могут программно поддерживаться некоторые базовые микропрограммы, например, связанные с передачей управления, адрес следующей выполняемой микрокоманды может определяться программно на основе анализа соответствующих полей текущей микрокоманды. Это позволяет сосредоточить внимание на отладке микропрограмм основных алгоритмов логического вывода (унификации и возврата).

ГЛАВА 8.

РЕАЛИЗАЦИЯ ПРОЦЕССОРА ЛОГИЧЕСКОГО ВЫВОДА В ВИДЕ КОМПЛЕКТА СБИС

Отдельные блоки ПЛВ могут быть выполнены на основе существующей элементной базы: процессор обработки на 32-разрядном комплекте СБИС AMD 29300; регистровый файл на двух микросхемах Am29334; АЛУ на микросхеме Am29332; L- и G-стеки на микросхемах К537РУ10 (РУ17); блок управления на микросхемах ППЗУ 1623РТ1 и секвенсоре Am29331. Для реализации этих блоков можно использовать и другие существующие комплекты СБИС. Специализированные устройства ПЛВ — процессор команд (ПК) и процессор унификации (ПУ) — более эффективно реализовать в виде отдельных СБИС.

8.1. СПЕЦИФИКАЦИЯ СБИС «ПРОЦЕССОР КОМАНД»

СБИС «Процессор команд» (СБИС ПК) предназначена для предварительной выборки и распаковки команд ПЛВ. Структурная схема СБИС соответствует структурной схеме процессора команд на рис. 5.1. Условное графическое обозначение СБИС приведено на рис. 8.1,*а*, функциональное назначение выводов — в табл. 8.1.

Приведение СБИС в исходное состояние осуществляется сигналом низкого уровня на входе SR. Начальная установка и фиксация информации во внутренних элементах памяти СБИС (регистрах и триггерах) осуществляется по фронту сигнала синхронизации SYN. Временна́я диаграмма начальной установки совпадает с соответствующей временно́й диаграммой СБИС процессора унификации.

Таблица 8.1

Обозначение	Назначение
LINS0—LINS31	Шина команд (входы К на рис. 5.1)
LMI0—LMI7	Шина микрокоманд (входы УУОБМУ на рис. 5.1)
SR	Начальная установка
WRI	Сигнал приема команд
WRD	Сигнал приема данных
WRMI	Сигнал приема микрокоманд
SYN	Сигнал синхронизации
LD0—LD31	Двухканальная шина данных
LA0—LA23	Шина адреса (выходы АК на рис. 5.1)
LOP0—LOP7	Шина кода операции (КОП на рис. 5.1)
AKA	Сигнал сопровождения выдачи адреса
AKD	Сигнал сопровождения выдачи данных
RDY	Сигнал готовности (флаг окончания выполнения микрокоманды)

Сформированный адрес СБИС выставляет на шину LA, сопровождая его сигналом AKA (рис. 8.1,*б*). Со входной шины LINS считанная команда (через контроллер шины) при наличии сигнала WRI записывается в буфер команд. Время от момента выдачи адреса до приема команды определяется временем распространения сигналов по шине быстродействием контроллера, загруженностью шины и т. д., после приема команды СБИС снимает адрес и сигнал сопровождения AKA.

Микрокоманда от блока управления (БУ) ПЛВ принимается при наличии сигнала WRMI (рис. 8.1,*в*). Через два такта (прием и расшифровка) в зависимости от принятой макрокоманды возможны три случая:

выборка поля;

безусловный переход по полю;

безусловный переход по адресу извне.

В первом случае информация выдается на шину LOP, во втором — на шину LD. Сигнал готовности RDV сбрасывается после приема новой микрокоманды и устанавливается после ее окончания. В претьем случае на шине LD должен подаваться адрес прехода при наличии сигнала WRD. Адрес и сигнал можно выставить одновременно с записью микрокоманды.

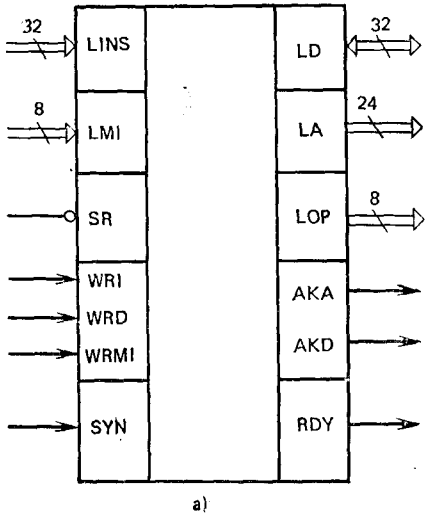
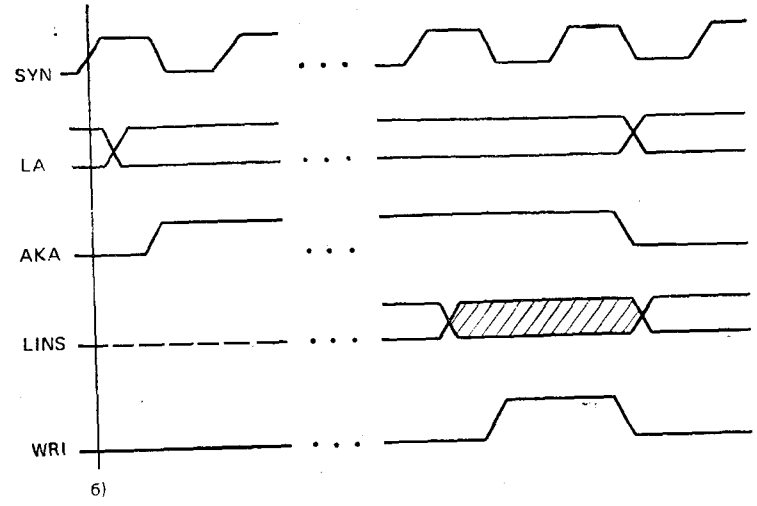
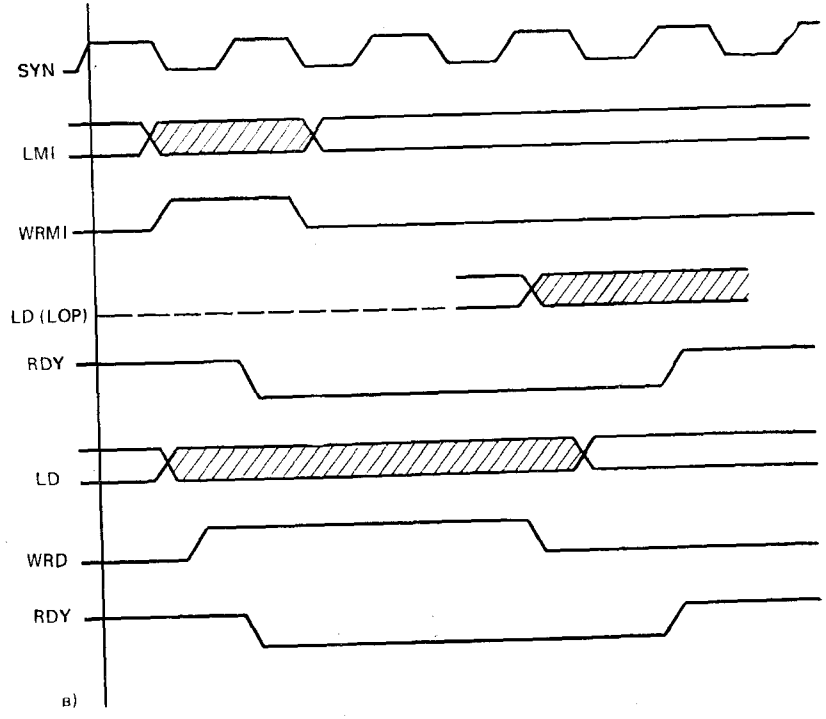


Рис. 8.1. СБИС процессора команд





Формат микрокоманды СБИС:

7	6	5	4	3	2	1
КОП			ОАП			ДП

где КОП — код операции, задает одну из четырех возможных микрокоманд (табл. 8.2); ОАП — относительный адрес поля операнда текущей команды ПЛВ или кода операции следующей команды в буфере команд; ДП — длина выбираемого поля в байтах.

Таблица 8.2

Мнемоника микрокоманды	Назначение
ВПК	Выбор поля кода операции. Длина поля 1 байт. ОАП содержит длину текущей команды в байтах
ВП	Выбор поля в микрокоманде. ОАП содержит номер последнего байта выбираемого поля команды (0, 1, 2, 3)
БП1	Безусловный переход — адрес перехода содержится в команде. Длина поля 3 байт, так как адрес 24-разрядный
БП2	Безусловный переход — адрес перехода поступает извне, поля ОАП и ДП не используются

Ориентировочная степень интеграции СБИС ПК — 7200 вентилей или 60 тыс. транзисторов (табл. 8.3). Распределение вентилей и транзисторов по основным узлам СБИС.

Таблица 8.3

Наименование элемента	Количество	
	вентилей	транзисторов
Мультиплексор выбора поля команд (MUX1)	4000 (1000)	
Элементы памяти (регистры и триггеры)	1750	20 000
Сумматор формирования адреса поля операнда	150	750
Счетчик адреса команд	400	2000
Дешифраторы	500	1600
Всего:	7200	60 000

Наиболее сложным элементом является мультиплексор выбора поля команд, представляющий 32 одноразрядных коммутатора с организацией «16 в 1». Для его реализации в «чистом» виде требуется 1000 вентилей (или 7800 транзисторов). Однако он имеет сложую топологическую разработку, поэтому для оценки степени интеграции взято 4-кратное превышение расчетных данных мультиплексора (в таблице указаны в скобках).

Требования к быстродействию СБИС «Процессор команд» определяются исходя из того, чтобы СБИС должна обеспечить прием из памяти HOST-машины новой Пролог-команды за время исполнения текущей. Команда выполняется за минимальное количество циклов (четыре): прием команды и передача кода операции в БУ, формирование первой микрокоманды, непосредственное исполнение (минимум два цикла). Исходя из этого, время цикла СБИС ПК должно совпадать с временем цикла СБИС ПУ (быстрее не требуется) и составлять 120—130 мс, что вполне достижимо для современного уровня технологии.

8.2. СПЕЦИФИКАЦИЯ СБИС «ПРОЦЕССОР УНИФИКАЦИИ»

СБИС «Процессор унификации» предназначен для унификации двух произвольных термов, может быть использована в составе ПЛВ, процессоров баз данных.

Структурная схема СБИС ПУ приведена на рис. 5.9, условное графическое обозначение — на рис. 8.2,*а*, функциональное назначение выводов указано в табл. 8.4.

Для приведения СБИС в исходное состояние на вход SR подается сигнал начальной установки низкого уровня. Установка осуществляется по фронту синхросигнала SYN. В этом состоянии на вывод RDY поступает сигнал высокого уровня, свидетельствующий о готовности. Прием сигнала команды с шины LMI в регистр RGI осуществляется при наличии сигнала высокого уровня на входе WRI. Его отсутствие показывает, что СБИС находится в исходном состоянии (режим ожидания). Временная диаграмма начальной установки и приема команд приведена на рис. 8.12,*б*.

После приема команды СБИС снимает признак RDY (сигнал низкого уровня), приступает к исполнению команды, в процессе выполнения команды возможны обращения СБИС по чтению (за-

Таблица 8.4

Обозначение	Назначение
LDI0 ... LDI31	Шина данных входная
LMI0 ... LMI5	Шина команд
LT0 ... LT7	Шина тегов
WRI	Сигнал приема команд
WRD	Сигнал приема данных
SR	Начальная установка
SYN	Сигнал синхронизации
LDO0 ... LDO31	Шина данных входная
LDA0 ... LDA23	Шина адреса
STAT0 ... STAT2	Состояние (признак) унифицируемости термов
AKA, AKD	Сигнал сопровождения выдачи адреса и данных
RDY	Сигнал готовности (флаг окончания операции)
EMP1, EMP2	Признак пустой переменной

Таблица 8.5

Код	Мнемоника	Содержание
000	READY	Нормальное завершение команды
001	ERR TAG	Ошибка тега
010	ERR REF	Ошибка значения операнда
011	ERR GLOB	Переполнение глобального стека
100	ERR TR	Переполнение трейлового стека
101	ERR STACK	Переполнение унификационного стека
110	ERR LOC	Переполнение локального стека
111	FAIL	Неуспешное завершение унификации

писи) к локальному и глобальному стекам данных, унификационному стеку, к памяти HOST-машины (в случае непопадания адреса в кеш-память). Выдача адреса на шину LDA сопровождается сигналом на выходе AKA, а выдача данных на шину LDO сопровождается сигналом на выводе AKD. Если производится запись из ПУ в блоки кеш-памяти или стеки, присутствуют сигналы на выводах и AKA, и AKD. Если выполняется чтение из блоков кеш-памяти или стеков, то выдается сигнал только на вывод AKA. Запись считываемых данных с входной шины LDI в регистры (приемник определяется текущей микрокомандой) осуществляется при наличии сигнала высокого уровня на входе WRD, т. е. аналогично приему команды. Запись информации во все регистры и триггеры СБИС ПУ производится по фронту синхросигнала SYN. Временна́я диаграмма циклов записи и чтения приведена на рис. 8.2,*в*.

После завершения выполнения команды СБИС выставляет флаг (сигнал высокого уровня), а на выход STAT — код состояния ПУ. Возможные случаи завершения выполнения команды приведены в табл. 8.5.

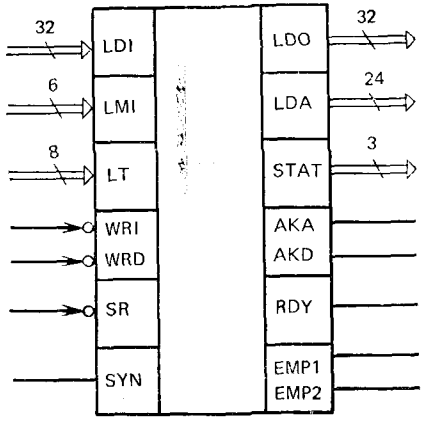


Рис. 8.2. СБИС процессора унификации

Система команд СБИС ПУ включает 21 команду (табл. 8.6).

Все команды СБИС, кроме команд загрузки-выгрузки регистров, специализированные, ориентированы на интерпретацию механизма унификации. Команды загрузки-выгрузки устанавливают содержимое тех регистров, которые определяют границы стеков HOST-машины (RGLT, RGGI, RGTR, RGGMAX, RGLMAX, RGTRMAX) и начальные значения указателей стеков (RGSP, RGBC, RGBL). Для

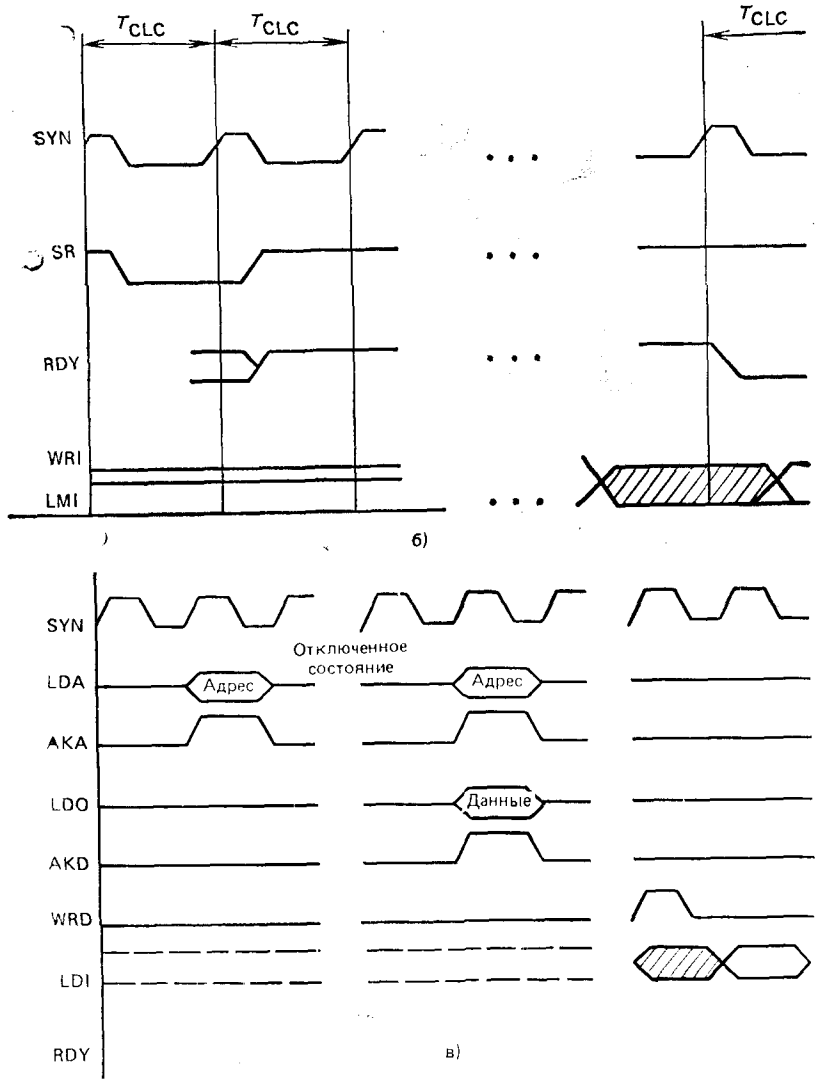


Рис. 8.2 (Окончание)

ряда регистров СБИС начальная загрузка регистров не требуется.

Ориентировочная степень интеграции СБИС ПУ без внутреннего управления 10 000 вентилей или 80 000 транзисторов (табл. 8.7).

Для получения максимальной производительности ПЛВ необходимо обеспечить минимально возможное время цикла, которое определяется быстродействием используемых СБИС памяти и блоков. Цикл записи (считывания) микросхем памяти 565РУ5 составляет 230 нс, время цикла микросхемы АЛУ (функциональный

Таблица 8.6

Мнемоника	Назначение
LOAD RG	Загрузка RGI с шины LDI[0 : 31]
READ RG	Чтение содержимого RGI на выходную шину LDO [0 : 31]
DEREF1	Дереференсирование содержимого RGARG1 с выработкой признака EMP1
DEREF2	Дереференсирование содержимого RGARG2 с выработкой признака EMP2
DECOMP1	Декомпозиция молекулы, ссылка на которую находится в RGARG1
DECOMP2	Декомпозиция молекулы, ссылка на которую находится в RGARG2
MOLECUL1	Создание молекулы в глобальном стеке и загрузка в RGARG1
MOLECUL2	Создание молекулы в глобальном стеке и загрузка в RGARG2
SS ARG1	Создание элемента структуры в RGSS1
SS ARG2	Создание элемента структуры в RGSS2
LINK VARS	Связывание двух пустых переменных
MAKE EMP1	Создание в стеке пустой переменной по адресу в RGARG1
MAKE EMP2	Создание в стеке пустой переменной по адресу в RGARG2
MATCH	Сопоставление двух термов
UNIF LISTS	Унификация двух списков
GEN UNIF	Унификация двух произвольных термов
FLAG1	Сравнение флагов в RGARG1 и на шине LT
FLAG2	*Сравнение флагов в RGARG2 и на шине LT*
NOP	Нет операции
LOAD ARG1	Загрузка RGARG1
LOAD ARG2	Загрузка RGARG2

аналог Am29332) 180 нс, секвенсора (Am29331) 100 нс, регистрового файла (Am29334) 60 нс. Исходя из этих данных необходимо обеспечить время цикла СБИС ПУ 120—150 нс.

Таблица 8.7

Наименование блока	Количество	
	вентилей	транзисторов
Элементы памяти (регистры, триггеры)	4150	47 000
АЛУ	450	2 200
Компараторы COMP1, COMP2, COMP3	480	2 700
Мультиплексоры и коммутаторы	2500	25 850
Схемы сравнения L1, L2 и логика признаков	50	350
Дешифраторы	300	2 000
Всего:	10 000	80 000

8.3. СПЕЦИФИКАЦИЯ СБИС RISC-ПРОЦЕССОРА

СБИС комбинационного процессора предназначена для интерпретации Пролог-программы и поддержки работы со структурами данных и памятью в системе логического вывода.

Структурная схема СБИС RISC-процессора приведена на рис. 4.7. Графическое обозначение этой СБИС на рис. 8.3,*а*, функциональное назначение выводов — в табл. 8.8.

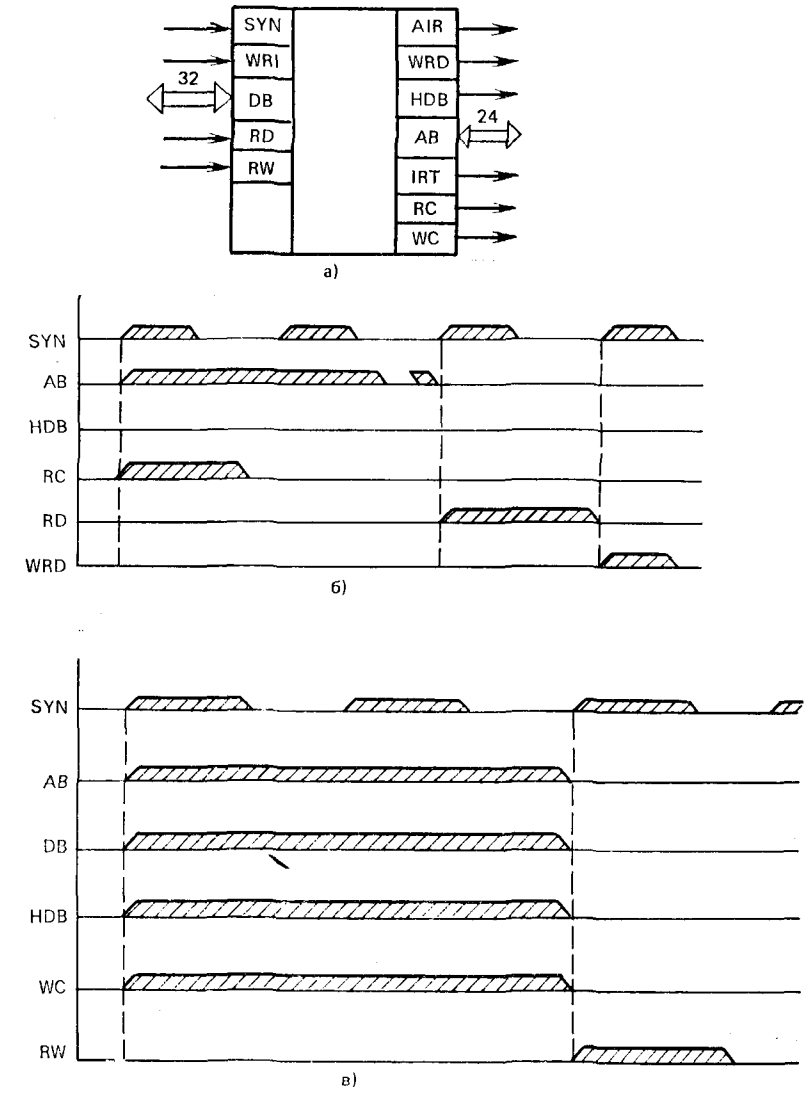


Рис. 8.3. СБИС RISC-процессора

Таблица 8.8

Обозначение	Назначение	Обозначение	Назначение
SYN	Синхросигнал	WRI	Сигнал подтверждения приема данных
WRI	Сигнал приема команды	HDB	
DB	Шина данных	AB	Сигнал захвата шины данных
RD	Наличие данных на шине		
RW	Подтверждение записи	IRT	Шина адреса
AIR	Подтверждение приема	RC	Возврат управления
		WC	Управление чтением
			Управление записью

Прием первых двух байт команды осуществляется по сигналу WRI, в ответ на который сопроцессор выставляет сигнал AIR, отключая основной процессор от шины на время выполнения Пролог-команд. По сигналу WRI также обеспечивается запись адреса первого байта команды, принятой с входа AB.

При обращении к памяти для чтения (рис. 8.3,*б*) сопроцессор выставляет на шине AB адрес, а на HDB — сигнал захвата шины данных и выдает сигнал RC. Появление сигнала RD указывает на поступление данных. После приема данных в регистрах RA или RB формируется сигнал WRD.

При записи данных (рис. 8.3,*в*) сопроцессор проверяет наличие сигнала HDB, если шина свободна, выставляет сигнал HDB, данные — на шину DB, а адрес — на шину AB, а затем формирует сигнал на выходе WC. Сигнал на входе RW подтверждает завершение записи в память.

Возврат управления в основной процессор выполняется сигналом IRT.

ГЛАВА 9.

НАПРАВЛЕНИЯ РЕАЛИЗАЦИИ ЛОГИЧЕСКОГО ВЫВОДА

Одним из перспективных направлений реализации аппаратной поддержки Пролог-систем является распараллеливание вычислений.

Описанные методы логического вывода не связаны ограничениями относительно вида доказываемой логической формулы, допускают эффективное распараллеливание, поскольку частные интерпретации могут находиться и обрабатываться независимо друг от друга. Кроме того, аппаратная реализация этих методов проще, поскольку в них не предусматривается операция возврата, логический вывод на реляционных моделях эффективно реализован для поиска информации в базе данных. Можно сказать, что логический вывод в Прологе ведется "узким фронтом" по длинной траектории. В описанных методах "фронт широкий", но при этом сокращается длина вывода.

9.1. СХЕМА ПАРАЛЛЕЛИЗМА В ПРОГРАММАХ ЛОГИЧЕСКОГО ВЫВОДА

Формализация подхода к распараллеливанию программ. Схемы параллелизма строятся исходя из предположения, что предикат, содержащий в процессе вывода несвязанные переменные, рассматривается как предположительно истинный. Такое допущение позволяет независимо строить цепочку предикатов, соответствующих последовательности их вызовов, с разных мест программы, после чего соединить построенные цепочки для получения завершенной трассы логического вывода. Понятие цепочки предикатов установим из примера

```
goal
   a(Y).
clauses
 a(X):—b(X, Z), c(Z, 1).
a(X):—b(X, 2).
b(U, V):—U=2,
         V=3.
b(u, v): —U=3,
         V=2
c(0, 0).
c(1, 2).
```

Порядок вызовов предикатов и присваивания значений переменным для этой программы следующий:

$$\mathbf{a(Y)}\ \text{unif}\ a(X) \longmapsto b(X, Z)\ \text{unif}\ b(U, V) \longmapsto U=2,$$
$$V=3,\ X=2,\ Z=3, \longmapsto c(Z, 1)\ \text{unif}\ c(0, 0) \longrightarrow \text{fail} \ldots,$$

где unif — обозначение операции унификации; $\longmapsto$ — вызов предиката; fail — оператор возврата. Опуская для простоты символы операций и группируя операции присваивания, получаем пример цепочки:

$$\{a(Y), a(X), b(X, Z), b(U, V),$$
$$(U=2, V=3, X=2, Z=3), c(Z, 1), c(0, 0)\}.$$

Цепочка C называется *основной*, если порядок записи в ней предикатов соответствует допустимой трассе логического вывода, т. е. трассе, которая может быть воспроизведена при выполнении исходной Пролог-программы. Частными примерами цепочек являются пустая цепочка (обозначаемая □) и вырожденная (■), которая содержит неунифицируемую пару одноименных предикатов. Если требуется указать, что цепочка C или какая-то ее часть является вырожденной, то также будем использовать обозначение {C, ■}. Видно, что одноименные предикаты группируются парами; предикат, стоящий в паре слева, — *вызывающий*, справа — *вызываемый*.

Пусть C — основная цепочка, C′ — *хвостовая часть* цепочки C, если C′ начинается с имени вызываемого предиката, для кото-

рого представленное в цепочке правило для этого предиката не последнее в исходной программе, например {c(0, 0)}. Хвостовая часть, не содержащая других хвостовых частей, называется *собственной*, а цепочка, полученная отбрасыванием из основной цепочки C ее собственной хвостовой части, называется *основной* (для C), обозначим C″=ker(C).

Единственная операция, вводимая для цепочки, есть операция (логического) сцепления (&&), которая удовлетворяет следующим правилам:

1. C1&&(C2&&C3)=(C1&&C2)&&C3 (ассоциативность)
2. C1&&□=□&&C1=C1 (сложение с нулем)
3. ■ && ■ = ■ &&C1= ■ ,
4. C1&&C2≠C2&&C1 (если C1 и C2 непусты)
5. C&& ■ = ker(C)
   ker(□)=□
   ker(■)=□

5a: Если C=C1&&C2 и C2= ■ , то C=ker(C1)

6. Если C — основная цепочка и C=C1&&C2, то C1 и C2 — также основные цепочки

7. Если C=C1&&C2, то C&&C2=C и C&&C1=C (поглощение)

Цепочки, первые члены которых совпадают, назовем *однокорневыми.*

Успешный логический вывод может быть представлен в виде основной цепочки C=C1&&C2&& ... Cn. Для его эффективного распараллеливания необходимо решить следующие задачи:

генерация семейств S_i однокорневых цепочек;

выбор из каждого семейства S_i по одной цепочке C_{ik}, такой, что применение к ним операции && дает невырожденную основную цепочку.

Эта вторая задача может быть тривиальной, если сгенерированные семейства цепочек содержат не более одной цепочки. В общем случае эффективный алгоритм ее решения описан в § 9.5.

Решение первой задачи связано с определением семейства цепочек, генерируемых каждым процессором в системе. В схеме с И-параллелизмом основные цепочки строятся для предикатов тела текущего обрабатываемого правила. Для рассмотренного выше примера эта схема реализуется следующим образом.

Предположим, что имеется два процессора, первый генерирует цепочку

C1={a(Y), a(X), b(X, Z), b(U, V), (U=2, V=3, X=2, Z=3)},

второй процессор — две вырожденные цепочки

C2={c(Z, 1), c (0, 0)}= ■,

C3={c(Z, 1), c(1, 2)}= ■.

Поскольку в И-схеме второй процессор строил цепочку независимо от первого процессора, то выполнять операцию C1&& ■ не имеет смысла: новое правило для b(X, Z) также не приведет к

построению основной результирующей цепочки. Поэтому результат, полученный вторым процессором, является указанием для первого процессора отбросить хвостовую часть до вызова предиката a(X) : {a(Y)}. После этого первый процессор строит результирующую цепочку вида

{a(Y), a(X), b(X, 2), b (U, V),

(U=2, X=2, V=2, V=3), ■ }={a(Y), a(X), b(X, 2)}

(правило 5a).

Теперь выполняется последний неиспользованный вызов для b(X, 2):

{a(Y), a(X), b(X, 2) b(U, V), (Y=X=U=3, V=2)},

в результате получается окончательное решение.

В схеме с ИЛИ-параллелизмом все процессоры строят одно и то же семейство однокорневых цепочек. В нашем примере будет построено три цепочки, из которых первые две — первым процессором:

C0={a(Y)},

C1={a(X), b(X, Z), b(U, V), (U=2, V=3, X=2, Z=3),

c(Z, 1), c(3, 1) ■ },

C2={a(X), b(X, 2), b(U, V), (V=2, U=3, X=3)}.

Результирующая цепочка программы находится как C0&&C2.

При использовании И-схемы процессоры не выполняют «лишнюю» работу, однако необходимость синхронизации процессоров и неравномерная вычислительная «нагрузка» на процессоры являются ее недостатком. В ИЛИ-схеме более простое управление процессорами, но значительно большая доля лишней работы, чем в И-схеме. В некоторых случаях порядок расположения альтернатив делает бессмысленным их раздельную обработку процессорами. Так, в примере вычисления суммы

```
count(0, R, R).
count(X, CurrentSum,Result):—
      X1=X—1,
       NewSum=CurrentSum+X,
      count(X1, NewSum,Result).
```

Логика вычисления суммы требует указанного расположения правил, которые по существу не являются альтернативными, а используются последовательно для организации рекурсивного накопления суммы.

Общим недостатком приведенных схем является то, что они не обеспечивают максимальной скорости обработки программ логического вывода. В идеале каждый процессор системы должен построить одну из цепочек C1, C2, ..., Cn, такую, чтобы C= =C1&&C2&& ... &&Cn была основной результирующей цепочкой.

Обозначив через t_i время построения цепочки Ci, получим соотношение для времен t_i и t_j произвольных цепочек Ci и Cj в виде примерного равенства $t_i \approx t_j = t$. В этом случае время, затрачиваемое на всю программу, будет функцией $0(t)$, тогда как при чисто последовательной обработке $0(nt)$, а при реализации И-схемы $0(\frac{n}{m}t)$, где $m = \min(k, n)$, k — среднее число предикатов в теле правила.

Рассмотрим процедуру, которая представляет решение в указанном направлении. Исходными посылками являются следующие:

1. Каждый процессор строит однокорневые цепочки с определенного места программы до определенного места программы.

2. Программа разбивается на уровни, на каждом из которых представлены одноименные правила:

```
        goal
          a(Y).
              clauses
a(X):—b(X, Z), c(X, T), d(Z).        уровень 1 процессора 1
a(R):—c(1, R), d(R), e(R).
a(T):—c(2, Z), e(Z).
```

```
b(U, V):—U=1, V=2.                   уровень 1 процессора 2
b(U, V):—U=2, V=3.
b(U, V):—U=3, V=4.
```

```
d(0).                                уровень 2 процессора 2
d(2).
d(4).
d(3).
```

```
c(4,2).                              уровень 3 процессора 2
c(1,3).
c(2,4).
```

```
e(3).                                уровень 2 процессора 1
e(4).
```

Уровни закрепляются за отдельными процессорами (в идеале такое закрепление должно предусматривать примерное равенство вычислительной нагрузки для процессоров системы). Каждый процессор обрабатывает правила на одном и том же уровне в порядке их записи. Обработка правила заключается в построении основной цепочки. Если при доказательстве возникает ссылка на уровень другого процессора, то построение цепочки завершается, а процессор переходит на следующий закрепленный за ним уровень. Если обрабатываемый уровень последний, то процессор возвращается на самый верхний свой уровень, выбирает еще не обработанное правило и выполняет построение следующей цепочки на этом уровне.

3. Один из процессоров (например, первый) является основным. Он синхронизирует работу других процессоров и выполняет операции && для построенных цепочек. Этот процессор может установить флажок возврата для другого процессора на некотором уровне. Установленный для некоторого уровня флажок возврата вынуждает соответствующий процессор вернуться на данный уровень для построения новой цепочки. Такой порядок действий моделирует механизм возврата в программе логического вывода. Выполнение вывода в системе с двумя процессорами (I и II) выполняется согласно программе.

1. C0 = {a(Y), a(X), b(X, Z)} (процессор 1; уровень 1)
2. C1 = {b(U, V), (U=1, V=2)} (процессор 2; уровень 1)
3. C2 = {C0&&C1} = {a(Y), a(X), b(X, Z), b(U,V), (X=U=1, Z=V=2)} (процессор 1)
4. C3 = {d(0)} (2; 2)
5. C4 = {a(Y), a(X), b(X, Z), b(, U, V), c(X, T), c(1, T)} (X=U=1, Z=V=2), (1; 1)
6. C5 = {c(4, 2)} (2; 3)
7. C6 = C4&&S5 = = {a(Y), a(x), b(X, Z), b(U, V), (X=U=1, Z=V=2), c(X, T), c(1, T), c(4, 2), ■ }

   C6 = ker(C6). (Процессор 1 устанавливает флажок возврата для процессора 2 на уровень 3)
8. C7 = {c(1, 3)} (2; 3)
9. C8 = C6&&C7 (процессор 1)
10. C9 = {b(U, V), (U=2, V=3)} (2; 1)
11. C10 = C8&&{d(Z), d(2)} (1; 1)
12. C11 = {d(2)} (2; 2)
13. C12 = C10&&C3 = {C8&&{d(Z), d(2), d(0)}, ■ }

    C12 = ker(C12) = C8&&{d(Z), d(2)} (процессор 1)
14. C13 = {c(2, 4)} (2; 3)
15. C14 = C12&&C11 = = {a(Y), a(X), b(X, Z), b(U, V), (X=Y=U=1, Z=V=2), c(X, T), c(1, T), (T=3), d(Z), d(2)}

цепочка C4 — результирующая цепочка программы

Порядок построения цепочек является условным: соблюден лишь порядок использования правил на каждом уровне. Запись типа (II.1) означает, что цепочка построена процессором II на уровне 1. Флажок возврата устанавливается, только если нет цепочек на некотором уровне, к которому производится обращение, или присоединение единственной цепочки этого уровня привело к образованию вырожденной цепочки.

Модель и устройство распараллеливания вывода. Пусть исходная Пролог-программа

```
goal
    a(X), b(X), c(X).
clauses
    a(m). a(m2). a(m1).
    b(m). b(n1). b(m1).
    c(q). c(q1). c(m1).
```

имеет единственное решение: X=m1. При выводе X последовательно получает значения m, m2, m1. Таким образом, идея распараллеливания может заключаться в параллельной унификации на всех трех доменах. Такая схема реализуется с помощью вычислительных элементов. На вход каждого поступают фреймы Pi и Pj, аргументы которых унифицируются. Один из этих фреймов соответствует фрейму аргументов предиката, а другой фрейму-контексту. Фрейм-контекст — это список переменных с теми значениями, которые получены к данному моменту в данном месте программы. На начальном этапе будут сформированы три фрейма-контекста P1, P2, P3, соответствующие предикатам a(m1), a(m2) и a(m).

При указании вычислительного элемента для участия в процессе вывода ему назначается предикат P, который необходимо доказать, точка возврата A при неудаче доказательства по И-схеме тела клоза (т. е. следующая альтернатива), точка возврата B при неудаче доказательства по всем альтернативам. При неудаче унификации происходит возврат либо в A, либо в B. При удаче унификации с предикатом P1 запоминается новая точка возврата B, а также точка A первой альтернативы для первого недоказанного предиката из тела правила с заголовком P1. Передача производится на тот же вычислительный элемент, на котором велось доказательство предиката P. Если имеется альтернатива у предиката P1, она передается на свободный вычислительный элемент.

Для одноименных переменных используется разделение структур данных, т. е. собственно переменная указывается один раз в клозе. В роли ссылок используются виртуальные адреса, которые преобразуются в физические в вычислительном элементе, т. е. каждый такой элемент имеет свою копию исходной структуры.

Используются следующие структуры данных:

PT-элемент списка предикатов программы, CT-элемент списка клозов с одноименными предикатами в заголовке, HT-элемент списка аргументов предиката заголовка клоза, BT-элемент списка предикатов тела клоза, AT-элемент списка аргументов предиката тела клоза;

D HAME — имя предиката, ARG — имя (или ссылка) на аргумент;

CTR — ссылка на начало списка CT, BTR — ссылка на начало списка BT, HTR — ссылка на начало списка HT, ATR — ссылка на начало списка AT.

Применительно к этим структурам данных алгоритм работы одного вычислительного элемента вывода, за исключением обработки случаев удачи и неудачи унификации, описанных выше, без учета унификации функторов будет следующим. Под формальными параметрами понимаются аргументы запроса (или предиката тела клоза), под фактическими — аргументы предикатов заголовков альтернатив.

Вывод начинается с разбора вопросительного клоза, для каждого его предиката находится начало списка описателей клозов. Начинается унификация аргументов предикатов запроса с аргументами заголовка. При удаче унификации в свободное поле HT записывается ссылка на следующий унифицируемый клоз (на CT), а в поле AT — ссылка на клоз, предикат тела которого унифицируется по ИЛИ-схеме. Определение следующего унифицированного клоза ведется путем унификации следующего заголовка клоза после данного со ссылкой на последний унифицированный клоз. Этот процесс может быть распараллелен и путем одновременной унификации всех аргументов предикатов заголовка. Основным препятствием этому явится однопортовая память. Поэтому данный процесс может быть выполнен последовательно. Так можно избежать присвоения различных значений одной переменной. При неудаче унификации по И-схеме аргументов предикатов тела некоторого клоза выбираются на основании ссылки в свободном поле HT описатель следующего клоза с унифицированным заголовком и его собственное начало списка HT записывается по адресу начала HT неунифицированного клоза, ссылка на начало списка HT в списке CT меняется на адрес предыдущего HT. Если в результате таких подтягиваний приходим в последнее CT, возможен возврат при неудаче унификации. Возврат в предикат тела, унифицируемого по ИЛИ-схеме, происходит на основании null в поле LINK последнего CT и по наличию адреса возврата в свободном поле AT. Процесс унификации всех предикатов данного вопросительного клоза прекращается, если хотя бы один его предикат не унифицирован.

Аппаратная поддержка параллельного вывода по данному алгоритму реализуется устройством, содержащим N блоков унификации, подключенных к блоку памяти через схему доступа. В основу работы устройства положены следующие принципы, использующие ранее введенные структуры данных. До начала работы устройства текст исходной Пролог-программы должен быть преобразован в списковую форму представления и занесен в блок памяти. При поступлении запроса к Пролог-программам имя предиката из этого запроса обрабатывается одновременно несколькими блоками унификации. Каждый из блоков унификации пытается выполнить унификацию запроса с одним из тех клозов, где встречается такое же имя предиката, как и в запросе. Распределение таких клозов между блоками унификации происходит путем передачи адресов через регистр ссылок. Эта возможность обеспечивается самим способом организации списков: каждый клоз содержит ссылку на адрес следующего клоза, имеющего такое же имя предиката. При успешной унификации, выполненной любым из блоков унификации, полученный в результате клоз запоминается и обрабатывается в дальнейшем таким же образом, как и первоначальный запрос. Если при очередной унификации получен пустой клоз, то это означает, что найдено решение. В зависимости от требований к запросу (нужно ли найти хотя бы одно

решение или же все решения) работа устройства может быть остановлена или продолжена. Во втором случае работа устройства заканчивается лишь после обработки всех получившихся модулей.

9.2. КОНВЕЙЕРНЫЙ ПРОЛОГ-ПРОЦЕССОР

При разработке модели конвейерного процессора необходимо выполнить следующие требования: она должна быть совместима с последовательной организацией вычислений в текущих версиях Пролога, обеспечивать последовательную загрузку элементов при реализации трассы вывода.

Пусть исходная Пролог-программа имеет вид

```
goal
    P(X, Y, Z, X1, Y1, Z1).
clauses
      P(X, Y, Z, X1, Y1, Z1):—R(X, Y, Z), T(X1, Y1, Z1).
 R(a, b, c).
 R(d, e, f).
 T(a, b, c).
```

При последовательной реализации процессор для цели P выбирает первую подцель R, ищет для нее соответствующий клоз и выполняет унификацию. Если она успешна, обрабатывается литерал T и т. д. Если унификация первого клоза с литералом R не успешна, выполняется второй и т. д. При конвейерной реализации после обработки текущей цели R одновременно с поиском первой подцели для литерала R можно вести подготовку второй подцели T и т. д.

Структуры данных, локальный, глобальный и трейловые стеки и скомпилированный код располагаются в системной памяти. Каждый автомат конвейерного логического процессора (рис. 9.1) имеет регистровую память для хранения альтернативного решения, которое содержится после первой цели в основной памяти. Появляющиеся в процессе решения цели динамически распределяются по конвейеру. В данном случае не требуется сложной синхронизации. Рассмотрим работу конвейерного процессора на примере следующей Пролог-программы:

```
        A <-- B(...)
C1  B (...) <-- P1(...), P2(...)
C2  B (...) <-- R1(...)
C3  P1(...) <-- T1(...)
C4  P1(...) <-- V1(...)
C5  T1(...) <-- V1(...)
C6  V1(...) <-- ...
```

Пусть в состав процессора входят три автомата. Тогда выполнение программы будет проходить в следующей последовательности:

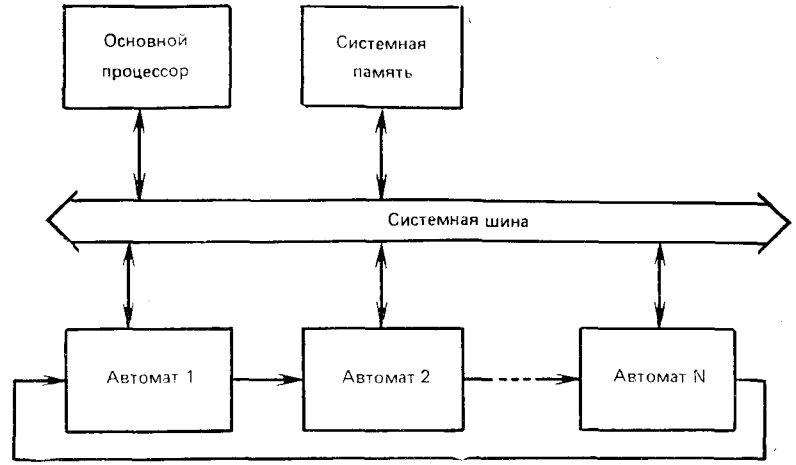


Рис. 9.1. Схема конвейерного процессора логического вывода

1. Автомат 1 обрабатывает цель — находит клоз C1, головной литерал которого совпадает с целью, для него создаются структуры данных, проводится унификация. До ее завершения подцель P1 передается в соседний автомат, который настраивается на правило C3.

2. После успешной унификации текущей цели B(...) управление передается автомату 2, который работает с новой подцелью P1 — находит клоз C3, передает адрес новой подцели T1(...) соседнему автомату для инициализации. Одновременно автомат 1 инициализируется для основной цели B, занимаясь литералом P2.

3. После успешной унификации термов литералов P1 управление передается автомату 3, который начинает выполнять унификацию термов T1(...). Первая подцель клоза C5 передается автомату 1, который готовит для нее термы. Если унификация в автомате 3 завершается неудачно, то ищется следующая альтернатива, она имеется в автомате 2 для клоза C4. Тогда подцель для P1 в виде литерала V1 передается автомату 3, а автомат 2 унифицирует новую цель. Это обеспечивает естественную синхронизацию и распределение по ветвям дерева поиска. Автомат 2 будет продолжать поиск для альтернативного решения цели P1(...).

Для синхронизации работы в систему команд машины Уоррена вводятся новые команды. Команда check_right осуществляет передачу адреса подцели правому автомату для предвыборки команды, установки регистров аргументов. По команде start_right выдается прерывание правому автомату, указывающее, что все конфликты данных разрешены и выполнение может начаться без учета зависимости данных.

9.3. ИСПОЛЬЗОВАНИЕ АССОЦИАТИВНОЙ ПАМЯТИ ДЛЯ ЛОГИЧЕСКОГО ВЫВОДА

Рассмотрим основные идеи развития концепции программного окружения на базе подхода, использующего ассоциативные структуры данных. Основное достоинство этого подхода — обеспечение максимального распараллеливания в первую очередь процедуры унификации как наиболее частой и трудоемкой процедуры логического вывода. При этом подходе отпадает необходимость аппаратной реализации стеков абстрактной машины, наиболее узких мест для распараллеливания. Наконец, этот подход автоматически предполагает использование массивов памятей и структурированных адресов, т. е. позволяет расширить узкие места, связанные с линейной организацией памяти.

Одна из очень серьезных проблем — представление предикатов и термов. Пусть предикат

нравится(X, фрукт(апельсин, яблоко))

представлен в виде структуры, приведенной на рис. 9.2. Если вызываемый предикат

нравится (джон, Z)

то результат унификации термов этих предикатов также представляется в виде структуры.

Для эффективной реализации унификации требуется обработка структурированных термов и предикатов одновременно.

Идея одновременной унификации всей структуры базируется на следующем. Структура «погружается» в память так, что ее вершины «наводятся» на определенные адреса по ассоциативным связям. В нашем примере роль ассоциативных связей выполняют указатели (теги) — имена «нравится» и «фрукт». Значением тега любой вершины в структуре является последовательность имен старших подструктур (начиная с корневой вершины) в маршруте, соединяющем корневую вершину с данной, дополненная порядковым номером данной вершины относительно числа потомков ее родителя. Сказанное иллюстрирует следующая таблица:

Имя вершины	Значение ассоциативного указателя (тега)
нравится	нравится
X	нравится. 1
фрукт	нравится. 2
апельсин	нравится. фрукт. 1-нравится. 2.1
яблоко	нравится. фрукт. 2-нравится. 2.2

Таким образом, ячейка памяти, в которую должно быть погружено слово «апельсин», определяется тегом «нравится. 2.1» или «нравится. фрукт. 1». Для обеспечения параллельного доступа в ячейки необходима ассоциативная многопортовая память или линейная память. Ассоциативная унификация обеспечивает экономию времени выполнения унификации за счет параллельной унификации термов.

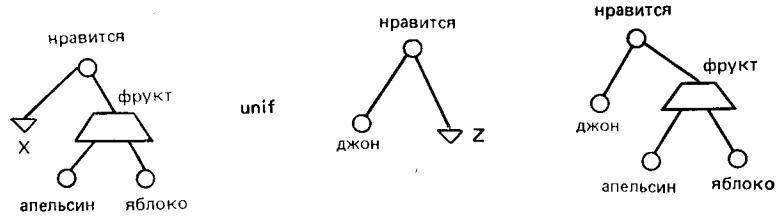


Рис. 9.2. Структура предиката «нравится»

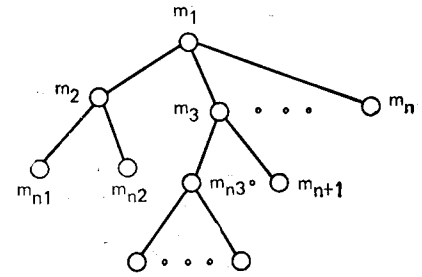


Рис. 9.3. Представление ассоциативной структуры дерева вывода

Второе важное преимущество ассоциативного представления термов и предикатов — отсутствие необходимости в стеках. Если характерной особенностью машины Уоррена является наличие глобального, локального и трейлового стеков, обеспечивающих хранение информации по трассе логического вывода, то в данном подходе трасса логического вывода может быть представлена текущим состоянием унифицированных ассоциированных структур (рис. 9.3):

$m_{n\,i}$ — кортеж вида ⟨тег, значение, имя⟩

где тег∈(O, V, o, ▽ , ✳), O — предикат, o — константа, V — переменная, ▽ — структура, ✳ — анонимная переменная.

Такая структура в процессе вывода модифицируется. Назначение компилятора — создать исходную структуру из предикатов цели и построить алгоритм ее обработки, используя две базовые операции унификации и деунификации (развязывания термов).

Реализовать такой подход можно, используя представление предикатов как процессов, записывающих и стирающих информацию в ячейках ассоциативной памяти, которые им доступны по тегам, и активизацию и деактивизацию предикатов.

Аппаратная поддержка ассоциативных структур базируется на идее аппаратной реализации хеш-таблиц и многопортовой памяти. Преимущество хеширования состоит в том, что число физических адресов значительно больше числа тегов, однако сами теги хранятся в памяти и их предварительно нужно считать. Не прямое решение этой проблемы заключается в использовании только одного корневого тега, кодирующего структуру. Таким тегом может служить список, например, вида

[нравится, нравится. 1, нравится. 2, нравится. 2.1, нравится. 2.2]

Рис. 9.4. Аппаратная реализация функции хеширования

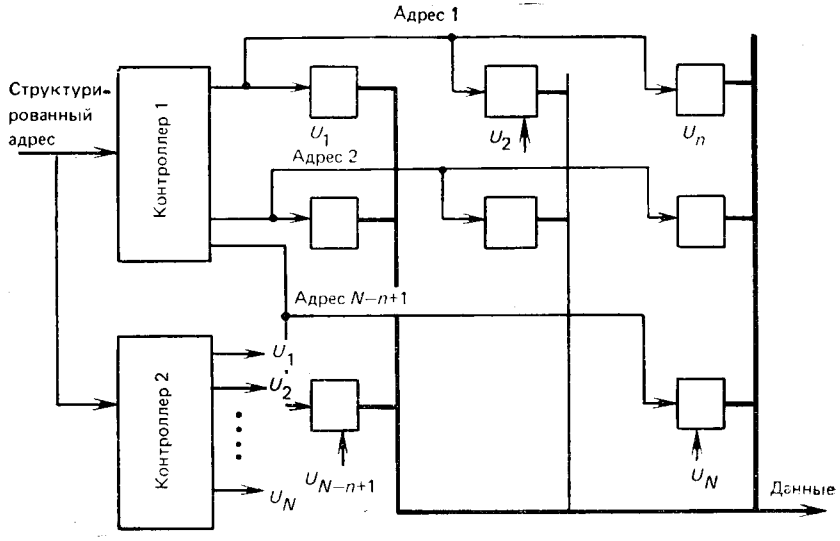


Рис. 9.5. Аппаратная поддержка

Если учесть, что на число аргументов накладывается ограничение, то можно проще решить этот вопрос (рис. 9.4) — программирование хеш-памяти должно выполняться компилятором.

Одним из подходов к решению проблемы организации линейной памяти для хранения нелинейных структур является организация массива модулей памяти с поддержкой сложных структурированных адресов (рис. 9.5). Контроллер 1 выполняет распаковку структурированного адреса, т. е. формирует адреса-термы, контроллеры 2 — подачу команды чтения-записи по требуемым модулям U_i.

9.4. ЛОГИЧЕСКИЙ ВЫВОД НА ОСНОВЕ ИНТЕРПРЕТАЦИЙ

Здесь рассмотрена модификация процедуры логического вывода, предложенная Бетом. В этой процедуре используется понятие семантической выводимости формул логики предикатов. Пусть

φ и ψ — две (правильно построенные) формулы, причем ψ семантически следует из φ, т. е. $\varphi \models \psi$, если во всякой интерпретации, в которой истинна φ, ложна ψ. Интерпретация называется контрпримером, если в ней $\varphi \& \bar{\psi} = 1$. Процедура Бета строит контрпример (если существует) для φ и ψ. Пусть надо доказать, что, например, формула

$$\psi = \forall x P(x) \vee \forall x Q(x)$$

является следствием формулы

$$\varphi = \forall x (P(x) \vee Q(x)).$$

Для этого нужно найти такую интерпретацию (подстановку для x), при которой φ истинно, а ψ ложно, или показать, что такой интерпретации нет. Ложность ψ означает, что ложна подформула как $\forall x Q(x)$, так и $\forall x P(x)$. В свою очередь, ложность $\forall x P(x)$ означает, что $\exists a \bar{P}(a)$, а ложность $\forall x Q(x)$ означает, что $\exists b \bar{Q}(b)$. Итак, мы ввели интерпретацию, для которой ψ ложно. В этой интерпретации

$$\varphi_1 = (\bar{P}(a) \vee Q(a)),$$
$$\varphi_2 = (P(b) \vee \bar{Q}(b)).$$

Чтобы φ_1 и φ_2 были истинными, достаточно положить, что $\bar{Q}(a)$ и $\bar{P}(b)$ истинны.

Таким образом, мы доказали, что $\varphi \not\models \psi$, т. е. интерпретация

x:	a	b
$P(\cdot)$	Ложно	Истинно
$Q(\cdot)$	Истинно	Ложно
$\bar{P}(\cdot)$	Истинно	Ложно
$\bar{Q}(\cdot)$	Ложно	Истинно

удовлетворяет φ и опровергает ψ.

Пусть теперь

$$\psi = \forall x (P(x) \vee Q(x)) \text{ и } \varphi = \forall x (P(x)) \vee \forall x (Q(x)).$$

Для ложности ψ необходимо, чтобы $\exists x (\bar{P}(x) \vee \bar{Q}(x))$. Пусть, например, $x = a$. В этом случае имеет место $\bar{P}(a)$ и $\bar{Q}(a)$. В этой интерпретации

$$\varphi = \bar{P}(a) \vee \bar{Q}(a).$$

Чтобы φ была истинна, нужно, чтобы была истинна подформула либо $\bar{P}(a)$, либо $\bar{Q}(a)$, либо обе вместе, а это противоречит допущению о ложности ψ. Таким образом, построить контрпример нельзя и, стало быть, $\varphi \models \psi$.

Чтобы построить регулярную процедуру нахождения контрпримера, рассмотрим запись

$$F \models \psi,$$

где $F=\{\varphi_1, \varphi_2, ..., \varphi_N\}$ и φ_i — правильно построенные формулы (ППФ) логики предикатов. В каждой интерпретации любая из формул превращается в высказывание, которое представляет собой выражение из атомарных формул (высказываний), связанных логическими связками &, $\vee$, $\to$, $\leftrightarrow$, а также операцией отрицания и вспомогательными символами. Например, запись $\varphi=\bar{P}(a)\vee\bar{Q}(a)$, где a — некоторое значение для предметной переменной x.

Пусть $f_{i_1}(x^{(i)}_1, ..., x^{(i)}_{n_1}), ..., f_{i_m}(x^{(i)}_1, ..., x^{(i)}_{n_m})$ — атомарные формулы в записи ППФ φ_i. Пусть предметные переменные x_{ij} получили некоторые значения из соответствующих областей определения. Наша задача — найти такое распределение значений атомарных формул $f_{i_k}(\cdot)$, $i=1, 2, ..., N$, $k=1, ..., n_m$, $m=1, 2, ..., M$, на которых сложные высказывания φ_i истинны, а ψ ложно. При этом мы допускаем, что различные ППФ из F и ψ могут содержать общие атомарные формулы или их отрицания.

Эта задача может быть переформулирована: найти интерпретацию Ip (или доказать, что такая интерпретация невозможна), на которой истинна конъюнкция $\varphi_1\&\varphi_2\&...\&\bar{\psi}$. Используем следующие преобразования. Заменим переменные при кванторах существования так, чтобы разные кванторы связывали разные переменные. Чтобы не прибегать к сколемизации и упростить унификацию формул, рассмотрим далее такие формулы, в которых квантор всеобщности не предшествует квантору существования, что не влияет на общность результата. Избавимся от кванторов существования, введя специальные обозначения $P(\check{x})$ для $\exists x(P(x))$ и $P(\square)$ для $\bar{\exists}x(P(x))$, где $\square$ — символ пустого элемента; $\check{x}$ — некоторое фиксированное значение x, например

$$\exists x\exists y(P(x, y))=\exists x(P(x, \check{y}))=P(\check{x}, \check{y}).$$

(Обозначение $\check{y}$ имеет тот же смысл, что и связанная переменная y).

Избавиться от кванторов всеобщности позволяет эквивалентное преобразование

$$\forall xP(x)=\overline{\bar{\exists}x\overline{P(x)}}=\overline{P(\square)}.$$

Поэтому $\forall x(P(x, \check{y}))=\overline{\bar{\exists}x\overline{P(x, \check{y})}}=\overline{P(\square, \check{y})}$. Преобразование кванторов выполняем, начиная с самого внутреннего квантора, например

$$\forall x\exists y\exists z(P(x, y, z))=\forall xP(x, \check{y}, \check{z})=$$
$$=\overline{\bar{\exists}x\overline{P(x, \check{y}, \check{z})}}=\overline{P(\square, \check{y}, \check{z})}.$$

Кроме того, потребуется формула

$$\exists x(P(x)\vee Q(x))\sim\exists x(P(x))\vee\exists x(Q(x))=P(\check{x})\vee Q(\check{x}),$$

а также формулы де Моргана

$$\overline{P(x)\vee Q(x)}=\overline{P(x)}\&\overline{Q(x)} \text{ и } \overline{P(x)\&Q(x)}=\overline{P(x)}\vee\overline{Q(x)}.$$

Для формулы $\exists x(P(x)\&Q(x))$ используем специальное представление

$$\exists x(P(x)\&Q(x))=\exists x(P\&Q\{x\})=P\&Q\{\check{x}\},$$

чтобы противопоставить его $P(\check{x})\&Q(\check{x})$, так как $P(\check{x})\&Q(\check{x})\neq P\&Q\{\check{x}\}$. В правой части последнего неравенства x одинаково инициализируется для $P(x)$ и $Q(x)$, а в левой это условие не обязательно.

Запись $(P(—, \check{y})\&Q(—))\{\check{x}\}$ следует понимать как $\exists x\exists y\times(P(x, y)\&Q(x))$, в которой x одинаково инициализируется для P и Q.

Пусть в результате преобразования получена формула

$$F: P(\check{x}, \square, \square, \check{y}),$$

означающая, что существуют такие x и y, для которых нет подходящих значений второго и третьего аргументов, для которых P истинно. Обозначим аргументы этой формулы $\widetilde{P}1$, $\widetilde{P}2$, $\widetilde{P}3$ и $\widetilde{P}4$ соответственно (буква P сохраняет наименование предикатного символа, а индекс соответствует номеру аргумента).

Таким образом, для истинности P необходимо, чтобы имело место $Ip=\{\widetilde{P}1\neq\check{x}$ или $\widetilde{P}4\neq\check{y}\}$, где Ip — символ интерпретации для $P(.)$.

Общее правило: пусть в интерпретации для любой формулы указана переменная $\square$, например $P(\check{x}_1, \check{x}_2, ..., \check{x}_n; \square, ..., \square)$. Тогда интерпретация для P имеет следующий вид: $Ip=(\widetilde{P}1\neq\check{x}_1\vee\widetilde{P}2\neq\check{x}_2\vee ...\vee\widetilde{P}n\neq\check{x}_n)$, например $P(\check{x}, \square)\to Ip=\{\widetilde{P}1\neq\check{x}\}$.

Остаются два крайних случая:

1) $P(\check{x}, \check{y}, \check{z})\to Ip=(\widetilde{P}1=\check{x}, \widetilde{P}2=\check{y}, \widetilde{P}3=\check{z})$ соответствует варианту, когда нет аргумента со значением $\square$;

2) $P(\square, \square, ..., \square)\to Ip=\square$ — пустая интерпретация.

Пусть дано множество формул $f_1, f_2, ..., f_N$. Интерпретация Ip, которая удовлетворяет всем формулам, получается как конъюнкция интерпретаций I1&I2&...&IN. Конъюнкция интерпретаций I1 и I2 есть $I1=\square$, если I1 и I2 содержат контрарную пару аргументов, например $\widetilde{P}1=\check{x}$ и $P1\neq\check{x}$. Отметим, что пара аргументов $\widetilde{P}1=\check{x}$ и $\widetilde{P}1=\check{y}$ не является контрарной.

Правило резолюции для интерпретаций: если $\widetilde{P}1=\check{x}$ и $\widetilde{P}1\neq\check{y}$, то $\check{x}\neq\check{y}$. Это следствие включается в виде новой интерпретации It.

Рассмотрим, наконец, запись $P\&M\{\square\}$. Если верно, например, $P(\check{x})$, то должно быть ложно $M(\check{x})$, т. е. верно $\bar{M}(\check{x})$, и наоборот. Пусть, например, имеется интерпретация, в которой $M1=\check{x}$. Тогда из $P\&M\{\square\}$ следует $(\widetilde{M}1=\check{x}\&\widetilde{P}1\neq\check{x})\vee(\widetilde{M}1\neq\check{x}\&\widetilde{P}1=\check{x})$. И вообще, если имеются интерпретации

$$I1: \widetilde{M}1=\check{x}$$

$$I2 : \widetilde{M}1 = \check{y}$$
$$IN : M1 = \check{z},$$

то для P&M{□} строится интерпретация

$$IN+1 : ((\widetilde{M}1 = \check{x} \& \widetilde{P}1 \neq \check{x}) \vee (\widetilde{M}1 \neq \check{x} \& \widetilde{P}1 = \check{x})) \&$$
$$\& ((\widetilde{M}1 = \check{y} \& \widetilde{P}1 \neq \check{y}) \vee (\widetilde{M}1 \neq \check{y} \& \widetilde{P}1 = \check{y})) \& \ldots$$
$$\& ((\widetilde{M}1 = \check{z} \& \widetilde{P}1 \neq \check{z}) \vee (\widetilde{M}1 \neq \check{z} \& \widetilde{P}1 = \check{z})).$$

Для выражения P1&P2&...Pk{□} при $\widetilde{P}k1 = \check{x}$ строится интерпретация

$$\widetilde{P}11 \neq \check{x} \vee \widetilde{P}21 \neq \check{x} \ldots \vee \widetilde{P}k - 1, \quad 1 \neq \check{x}.$$

Кроме того, используем следующее преобразование:

$(*)$ $P \vee Q\{I\} = P(I) \vee Q(I1)$, $I \neq \square$, I — общая интерпретация

$(**)$ $P \vee Q\{\square\} = P(\square) \& Q(\square)$

$(***)$ $P \& Q\{I\} \rightarrow p \& Iq$

$(****)$ $P \& Q\{\square\}$ порождает интерпретацию по следующим правилам:

1. Порождены все интерпретации для $(*)$, $(**)$, $(***)$, только после этого можно строить интерпретацию для $(****)$.
2. Если для Q или P уже найдена некоторая интерпретация после шага 1, то для P&Q{□} строится результирующая интерпретация, как описано выше.
3. Если ни для Q, ни для P не получено интерпретации, то выражение P&Q{□} заменяется формулой логики высказываний $\overline{P} \vee \overline{Q}$.
4. Если для некоторой формулы нет интерпретации, то она заменяется на соответствующее высказывание, которое включается в интерпретацию этой формулы.
5. Если $\widetilde{P}1 = \check{x}$ есть интерпретация для $P(\check{x})$, то $\widetilde{P}1 \neq \check{x}$ — для $\overline{P}(\check{x})$.
6. Интерпретация для дизъюнкта D общего вида

$$D = P(I'p) \vee Q(I'g) \vee \ldots \vee R(I'r)\{I\}$$

есть $Id = I'p \& I \vee I'g \& I \vee \ldots \vee I'r \& I$.

7. Правило де Моргана для интерпретаций:

Если $Ip = I1 \vee I2$, то $I\overline{p} = \overline{I1 \vee I2} = \overline{I1} \& \overline{I2}$;

если $Ip = I1 \& I2$, то $I\overline{p} = \overline{I1} \vee \overline{I2}$.

Например:

$$Ip = \{\widetilde{P}1 \neq \check{x},\ \widetilde{P}2 = \check{y}\} / * I1 = \{\widetilde{P}1 \neq \check{x}\};\ \ I2 = \{\widetilde{P}2 = \check{y}\}\ * /$$
$$I\overline{p} = \overline{I1} \vee \overline{I2} = \{\widetilde{P}1 = \check{x} \vee \widetilde{P}2 \neq \check{y}\}.$$

8. Интерпретации строятся только для формул или только для их отрицаний.

Пример 1. Пусть даны формулы

$$\exists y(S(y) \& M(y)) = S \& M\{y\},$$
$$\forall x(\overline{P}(x) \vee \overline{M}(x)) = \overline{\exists x(P(x) \vee M(x))} = P \& M\{\square\}.$$

Покажем, что из них логически следует формула

$$\exists z(S(z) \& \overline{P}(z)),$$

для чего последнюю заменяем ее отрицанием:

$$\forall z(\overline{S}(z) \vee P(z)) = S \& \overline{P}\{\square\}$$

и находим интерпретацию для трех формул, которая должна быть пустой.

Получаем следующие интерпретации:

$$I1 = \{\widetilde{S}1 = \check{y} \& \widetilde{M}1 = \check{y}\} \text{ для } S \& M\{\check{y}\},$$
$$I2 = \{(\widetilde{M}1 = \check{y} \& \widetilde{P}1 \neq \check{y}) \vee (\widetilde{M}1 \neq \check{y} \& \widetilde{P}1 = \check{y})\} \text{ для } P \& M\{\square\},$$
$$I3 = \{(\widetilde{S}1 = \check{y} \& \widetilde{P}1 = \check{y}) \vee (\widetilde{S}1 \neq \check{y} \& \widetilde{P}1 \neq \check{y})\} \text{ для } S \& \overline{P}\{\square\}.$$

Отметим, что I3 построено с учетом правил 5 и 7 для P и $\overline{P}$. Нетрудно увидеть, что конъюнкция I1&I2&I3=□.

Пример 2. Доказать, что из F1 и F2 выводится G:

$$F1 : \forall x(C(x) \rightarrow (W(x) \& R(x))),$$
$$F2 : (\exists x)(C(x) \& O(x)),$$
$$G : (\exists x)(O(x) \& R(x)).$$

После преобразования получим

$$F1 : C \& (\overline{W} \vee \overline{R})\{\square\},$$
$$F2 : C \& O\{\check{x}\},$$
$$\overline{G} : O \& R\{\square\}.$$

Имеем следующие интерпретации:

$$IF2 = \{\widetilde{C}1 = \check{x} \& \widetilde{O} = \check{x}\},$$
$$I\overline{G} = \{(\widetilde{O}1 = \check{x} \& \widetilde{R}1 \neq \check{x}) \vee (\widetilde{O}1 \neq \check{x} \& \widetilde{R}1 = \check{x})\},$$
$$IF1 = \{(\widetilde{C}1 = \check{x} \& \widetilde{W}1 = \check{x} \& \widetilde{R}1 = \check{x}) \vee (\widetilde{C}1 \neq \check{x} \& \widetilde{R}1 \neq \check{x} \& \widetilde{W}1 \neq \check{x})\}.$$

После чего убеждаемся, что их конъюнкция пуста, а это доказывает выводимость $F1 \& F2 \models G$.

9.5. ЛОГИЧЕСКИЙ ВЫВОД НА РЕЛЯЦИОННЫХ МОДЕЛЯХ

Пусть задано множество предикатов $P_j(X_j)$, $j = 1, \ldots, j$; $X_j = \{x_{j_1}, x_{j_2}, x_{j_{nj}}\}$. Каждый предикат $P_j(.)$ определяет некоторое конечное множество кортежей $\{b_j\}$ вида $\langle \overset{\circ}{x}_{j_1}, \overset{\circ}{x}_{j_2}, \ldots, \overset{\circ}{x}_{j_{nj}} \rangle$, где $\overset{\circ}{x}_{j_l} \in \mathrm{Dom}(x_j)$, на которых $P_j(X_j)$ истинно, обозначим его $B_j =$

$=(b_{1_j}, b_{2_j}, \ldots, b_{z_j})$. Считаем множества $B_1, B_2, \ldots, B_J$ заданными. Пусть $b_\alpha \in B_t$, $b_\beta \in B_r$ $(t \langle \rangle r)$. Предположим, что $b_\alpha \# b_\beta$ $(b_\beta \# \# b_\alpha)$, т. е. b_α совместим с b_β, если и только если имеет место одно из двух:

b_α и b_β не содержат общих переменных;

b_α и b_β содержат общие переменные и значения этих переменных совпадают.

Если $b_\alpha, b_\beta \in B_q$, то $b_\alpha \# b_\beta$ не выполняется.

Пример 1.

$$b_\alpha = \langle x_1 = 1,\ x_2 = 7,\ x_3 = \text{«а»} \rangle,$$

$$b_\beta = \langle x_1 = 1,\ x_2 = 7,\ x_4 = \text{«*»},\ x_6 = \text{«*»} \rangle.$$

Общие переменные для b_α и b_β здесь x_1 и x_2, причем их значения одинаковы, поэтому $b_\alpha \# b_\beta$. Наша задача сводится к выбору из каждого множества $B_j (j = 1 \ldots J)$ представителя $b_{j\mu_j}$, т. е. нахождению такой системы представителей $\bar{B} = \{b_{1\mu_1}, b_{2\mu_2}, \ldots, b_{J\mu_J}\}$, в которой любые два связаны отношением совместимости (#).

Пример 2. Пусть $B_1 = \{b_1, b_2, b_3\}$, $B_2 = \{b_4, b_5\}$, $B_3 = \{b_6, b_7\}$. Причем отношение совместимости кодируется симметричной относительно диагонали двоичной матрицей вида

b	b_1	b_2	b_3	b_4	b_5	b_6	b_7
b_1	1	0	0	0	1	1	0
b_2	0	1	0	0	1	0	0
b_3	0	0	1	1	0	0	1
b_4	0	0	1	1	0	1	1
b_5	1	1	0	0	1	0	1
b_6	1	0	0	1	0	1	0
b_7	0	0	1	1	1	0	1

где $b_{ij} = 1$, если $b_i \# b_j$, и $b_{ij} = 0$ в противном случае. Тогда решением будет $B = \langle b_3, b_4, b_7 \rangle$ и недопустимо, например, $\bar{B} = \langle b_1, b_5, b_7 \rangle$, так как условие $b_1 \# b_7$ не выполняется. По определению отношение совместимости симметрично, но в общем случае нетранзитивно.

Сформулированная задача не предполагает, что B обязательно существует или единственность B. Ответ на задачу должен предусматривать и отсутствие допустимого решения.

Лемма. Пусть дана матрица $[b_e, m]$, кодирующая $\langle \#, B \rangle$. Тогда максимальная единичная подматрица матрицы $[b_e, m]$ содержит не более J строк и столбцов (J — число исходных предикатов P_j).

Доказательство. Если число строк (столбцов) больше J, то какие-то две строки должны обязательно входить в одно множество B_j, а это невозможно: любая пара кортежей из B_j $(j = I, \ldots, J)$ не связана отношением совместимости.

Максимальная единичная подматрица матрицы $[b_{ij}]$ из примера 2:

b	b_3	b_4	b_7
b_3	1	1	1
b_4	1	1	1
b_7	1	1	1

Таким образом, интересующий нас алгоритм должен находить максимальную единичную подматрицу исходной матрицы $[b_{ij}]$. Если при этом окажется, что число строк (столбцов) этой подматрицы меньше J, то исходная задача не имеет решения. В противном случае решение будет представлено всеми строками найденной подматрицы (строка соответствует некоторому кортежу b_t). Кроме того, если возможных решений несколько, то нас устраивает любое.

Опишем всю процедуру, которая содержит три субпроцедуры: A, B, C. Процедуры B и C могут образовывать рекурсивно повторяющуюся последовательность.

Субпроцедура A. Последовательно вычеркиваем строки (и одноименные столбцы) из матрицы $[b_{ij}]$, содержащие на шаге вычеркивания максимальное число нулей, до тех пор, пока не получим единичную подматрицу. Если кандидатов на вычеркивание несколько, выбираем любой (например, с меньшим номером). Субпроцедура A дает в худшем случае некоторое отправное решение, в лучшем случае находит сразу максимальную единичную подматрицу, что проверяется сравнением числа строк в ней с J. Субпроцедура A использует эвристическое правило: для удаления из исходной матрицы всех нулей за минимальное число шагов на каждом шаге удаления выбрасывается строка (столбец) с максимальным текущим числом нулей (основа так называемых «жадных» алгоритмов).

Пример 3. Пусть $[b_{ij}]$ имеет вид

b_j \ b_i	1	2	3	4	5	6
1	1	1	1	0	0	0
2	1	1	0	0	0	0
3	1	0	1	0	1	0
4	0	0	0	1	1	0
5	0	0	1	1	1	0
6	0	0	0	0	0	1

Вычеркиваем строку 6 (столбец 6), затем строку 2 (столбец 2), строку 1 (столбец 1), затем строку 3 (столбец 3). Получим:

b_j \ b_i	4	5
4	1	1
5	1	1

Субпроцедура B. Пусть найдено отправное (опорное) решение $\tilde{b}$, число строк в $\tilde{b}$, есть N. Субпроцедура B пытается найти новое решение с бóльшим числом строк. Для этого вычеркиваем все те строки и столбцы из $[b_e; m]$, которые содержат N или менее единиц (так как такие строки наверняка не входят в единичные подматрицы большего размера). Вычеркивание продолжаем до тех пор, пока одновременно выполняются следующие условия:

1) есть кандидат на вычеркивание;

2) есть строка, содержащая более N единиц.

Возможны следующие случаи:

1) выполняются оба условия. Производим вычеркивание кандидата;

2) имеет место условие 1 и не имеет места условие 2, что означает, что текущее опорное решение нельзя улучшить; конец всего алгоритма;

3) не имеет места условие 1, имеет место условие 2, здесь происходит переход к процедуре C;

4) не имеет места ни условие 1, ни условие 2. Этот случай невозможен, так как он определяется противоречивыми условиями (исключающими друг друга).

Для примера 3 $N=2$, поэтому вычеркиваем все строки по порядку с числом единиц, меньшим или равным 2. Так, вычеркнув сначала строку 2 (столбец 2), затем строки (столбцы) 1 и 3, мы «попадаем» в случай 2. Таким образом, для примера 3 максимальная единичная подматрица содержит две строки (4 и 5).

Субпроцедура B существенно ускоряется, если число строк в матрице, содержащих более N единиц (где N — число строк в текущем опорном решении), меньше либо равно N, так как нет смысла проводить дальнейшие вычеркивания: нельзя построить единичную подматрицу с числом строк, большим N. Это значит, что из рассмотрения матрицы $[b_{e,m}]$, получаемой в ходе выполнения субпроцедуры B, можно всякий раз определить, стоит ли продолжать или прекратить вычеркивание строк (столбцов).

Субпроцедура C. Это последняя из рассматриваемых в итерационном цикле процедур, до которой дело чаще всего не доходит или ограничивается вариантом 1 этой субпроцедуры. Данная субпроцедура выделяет два варианта.

Вариант 1. В результате выполнения субпроцедуры B получили некоторую текущую несокращаемую матрицу $[b_{e,m}]'$, кроме того, есть текущее опорное решение $\tilde{b}$ с числом строк N. Для этого варианта предполагается, что в матрице $[b_{e,m}]'$ есть хотя бы одна строка с числом единиц, равным $N+1$. Пусть, например, при $N=2$ это строка 3:

	1	3	4	5	6
3	1	1	0	1	0

Тогда легко проверить, является ли множество строк 1, 3 и 5 (т. е. тех строк, где стоят единицы в строке 3) новым опорным решением. Для этого достаточно построить квадратную подматрицу со строками 1, 3, 5 и столбцами 1, 3, 5 из исходной матрицы $[b_{e,m}]$. Если построенная подматрица единичная, то нам удалось улучшить решение (нашли $\tilde{\tilde{b}}$). Поэтому снова возвращаемся к субпроцедуре B, приняв $\tilde{b}=\tilde{\tilde{b}}$ и $N=N+1$. Если построенная подматрица не единичная, то удалим строку (столбец) 3 из $[b_{e,m}]$ и снова вернемся к субпроцедуре B со старым опорным решением $\tilde{b}$ и N.

Вариант 2. Текущая несокращаемая матрица, как и в варианте 1, $[b_{e,m}]'$, текущее опорное решение B с числом строк N. Любая строка матрицы $[b_{e,m}]'$ содержит более $N+1$ единиц. Определим строку с минимальным числом единиц. Пусть эта строка имеет вид

	i_0	i_1	i_2	i_3	i_z
i_0	1	1	0	1 . . .	0

Построим квадратную подматрицу матрицы $[b_{e,m}]'$ со строками и столбцами, определяемыми, как и в варианте 1, единичными элементами строки i_0. Обозначим эту матрицу $\beta(i_0, i_1, i_3, \ldots)$, где в скобках указаны номера строк (столбцов), которые ее образуют. Нужно определить, содержит ли матрица $\beta(i_0, i_1, i_3, \ldots)$ единичную подматрицу с числом строк (столбцов), большим N (N — текущий рекорд). Если матрица не содержит такой подматрицы, то удалим строку (столбец) i_0 из $[b_{e,m}]'$ и возвратимся к субпроцедуре B, так как такое удаление могло создать условия для ее выполнения. Если найдем в β новое опорное решение $\tilde{\tilde{b}}$ и новое значение N (новый рекорд), то снова перейдем к субпроцедуре B.

Поставленная задача — это задача, решаемая субпроцедурой B. Ясно, что подматрица $\beta(\cdot)$ имеет меньший размер, чем исходная матрица $[b_{e,m}]'$ (текущая матрица $[b_{e,m}]'$), так как строка i_0 не может состоять только из единиц (i_0 по предположению содержит минимальное число единиц, иначе вся текущая матрица была бы единичной, и необходимость выполнения дальнейших действий отпадает). Таким образом, поставленная задача требует рекурсивного обращения к субпроцедуре B с опорным решением b, содержащим N строк, и исходной матрицей β вместо $[b_{e,m}]'$, причем размер $\beta(\cdot)$ всегда меньше размера $[b_{e,m}]'$. Это гарантирует общую сходимость процесса.

Итак, мы описали общий процесс решения в виде схемы взаимодействия субпроцедур A, B, C. Чтобы обеспечить наибольшую ясность, была принята неформальная форма описания. В этом плане наиболее сложным оказался вариант 2 субпроцедуры C, который состоит в рекурсивном обращении к субпроцедуре B для

матрицы меньшего размера. Теоретически число таких обращений ограничено.

Пример. Известно [9], что рассмотренная задача NP-полна. Ее решение в системе Пролог могло бы потребовать выполнения переборной процедуры с генерацией большого числа бесполезных комбинаций свойств. NP-полнота рассмотренной задачи проявилась в варианте 2 субпроцедуры С, который рекурсивно генерирует подзадачу меньшей размерности. Однако, чтобы такая генерация имела место, нужна матрица $[b_{e,m}]$ специального вида. Для подавляющего большинства (22 из 24) примеров вообще не требуется выполнять вариант 2 субпроцедуры С. Таким образом, основным достоинством алгоритма является его узкая область поиска с весьма высокой априорной вероятностью быстрой (полиномиальной) сходимости к результату.

Ниже описан общий пример выполнения алгоритма. Исходная матрица $[b_{e,m}]$ имеет следующий вид:

b_j \ b_i	1	2	3	4	5	6
1	1	0	0	1	0	1
2	0	1	1	1	0	0
3	0	1	1	1	0	1
4	1	1	1	1	1	0
5	0	0	0	1	1	1
6	1	0	1	0	1	1

1. Выполнение субпроцедуры А: $\bar{b}=\{5, 6\}$, $N=2$.

2. Субпроцедура С (вариант 1) — вычеркиваем строку 1 (столбец 1) из $[b_{e,m}]$:

	1	4	6
1	1	1	1
4	1	1	0
6	1	0	1

что дает

	2	3	4	5	6
2	1	1	1	0	0
3	1	1	1	0	1
4	1	1	1	1	0
5	0	0	1	1	1
6	0	1	0	1	1

$\bar{b}=\{5, 6\}$, $N=2$.

3. Субпроцедура С:

	2	3	4
2	1	1	1
3	1	1	1
4	1	1	1

Новый рекорд $\bar{b}=\{2, 3, 4\}$, $N=3$.

4. Субпроцедура В для нового рекорда (вариант 2):

	2	3	4	5	6
2	1	1	1	0	0
3	1	1	1	0	1
4	1	1	1	1	0
5	0	0	1	1	1
6	0	1	0	1	1

Итак, найдена максимальная единичная подматрица

	2	3	4
2	1	1	1
3	1	1	1
4	1	1	1

9.6. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ИНТЕРПРЕТАЦИЙ И ДЕДУКЦИИ В ЛОГИЧЕСКОМ ВЫВОДЕ

Рассмотрим доказательство формулы $F \to G$, в которой $G = g_1 \vee g_2 \vee \ldots \vee g_n$, F и g_i, $i=1, \ldots, n$, — ППФ формальной системы. При доказательстве устанавливается факт тождественной истинности или опровержения формулы. Если формула опровержима, то в логике предикатов находится интерпретация, в которой это имеет место.

Предлагается следующий алгоритм, суть которого заключается в последовательном умножении обеих частей $F \to G$ на $\bar{g}_i$ ($i=1, \ldots, n$), например, в порядке возрастания индекса i, пока не будет получена одна из следующих ситуаций:

1) $F^* \to \square$,

2) $F^* \to F^*$ ($\square \to \square$ или $F^* \to 1$),

3) $\square \to F^*$,

где $\square$ — символ пустой формулы ($\square = X\&\bar{X}$); F^* — непустая

формула формальной системы. Ситуация 1 означает невыводимость G; ситуации 2, 3 — выводимость G из F $(1=X\vee\bar{X})$.

Пример. Выполнение алгоритма для логики высказываний. Доказать ложность

$$XY\vee\bar{X}T\vee YT\to\bar{X}T\vee\bar{Y}.$$

Умножаем на Y

$$XY\vee\bar{X}TY\vee YT\to\bar{X}TY,$$

умножаем на $\bar{X}TY=X\vee\bar{T}\vee\bar{Y}$:

$$XY\bar{T}\vee XY\vee XYT\to\square$$

(ситуация 1).

При реализации алгоритма для логики предикатов находим интерпретацию I, которая в ситуации 1 означает опровержение, а в ситуации 2, 3 — выполнимость формулы $F\to G$. Интерпретация I определяется в процессе умножения на $\bar{g}_i$ с помощью устанавливаемых ниже правил для различных случаев умножения одноместных предикатов.

Как и ранее, заглавными буквами A, B, C, ... обозначим переменные, а строчными a, b, c, ... — константы; $\check{a}$, $\check{b}$, $\check{c}$, ... — некоторые значения, присвоенные переменным A, B, C, ..., причем необязательно, чтобы $a=\check{a}$. Выделим следующие случаи.

Случай 1. Аргументы обоих участвующих в произведении предикатов являются константами, например $P(a)\&P(b)$. Это произведение никак не преобразуется, за исключением следующих частных случаев:

$$P(a)\&P(a)=P(a),\quad P(A)\&P(\check{a})=P(\check{a}),$$

$$P(a)\&\bar{P}(a)=\square,\quad P(\check{a})\&P(a)=P(a)_{A=a}.$$

Случай 2. Аргумент одного предиката — константа, второго — переменная:

$$P(a)\&P(X)=P(a)_{\check{x}=a}\vee P(a)\&P(\check{x})_{X\neq a},$$

$$P(a)\&\bar{P}(x)=P(a)\&\bar{P}(\check{x})_{X\neq a}.$$

Здесь введена частная интерпретация I для произведения, например $I=(X\neq a)$. Если оказывается, что интерпретация противоречива (т. е. содержит контрарную пару аргументов, например $x=a$ и $x\neq a$), то соответствующая ей формула заменяется на $\square$.

Случай 3. Оба аргумента — переменные:

$$P(X)\&P(Y)=P(\check{x})_{\check{x}=\check{y}}\vee P(\check{x})\&P(\check{y})_{\check{x}\neq\check{y}},$$

$$P(X)\&\bar{P}(Y)=P(\check{x})\&\bar{P}(\check{y})_{\check{x}\neq\check{y}}.$$

Обобщение приведенных правил для k-местных ($k>1$) предикатов реализуется заменой предиката $P(X_1, X_2, \ldots, X_k)$ произведением $P_1(X_1)\&P_2(X_2)\&\ldots\&P_k(X_k)$, где $P_i(X_i)$ — одноместный предикат, утверждающий, что i-м аргументом предиката P является X_i.

Рассмотрим использование такого представления на примере

$$P(A, c)\&P(H, B)=[P1(A)\&P1(H)]!![P2(c)\&P2(B)],$$

где операция !! удовлетворяет следующим правилам:

$$P1(\alpha)!!P2(\beta)=P(\alpha, \beta)$$

$$P1(\alpha)!!\overline{P2}(\beta)=\square$$

$$\overline{P1}(\alpha)!!P2(\beta)=\square$$

$$\overline{P1}(\alpha)!!\overline{P2}(\beta)=\overline{P}(\alpha, \beta)$$

$$[P1(\alpha)\vee P1(\beta)]!![P2(\xi)\vee P2(\nu)]=$$
$$=P(\alpha, \xi)\vee P(\alpha, \nu)\vee P(\beta, \xi)\vee P(\beta, \nu)$$

$$[P1(\alpha)\&P1(\beta)]!![P2(\xi)\&P2(\nu)]=$$
$$=P(\alpha, \xi)\&P(\beta, \nu)\vee P(\alpha, \nu)\&P(\beta, \xi)$$

$$[P1(\alpha)\&\overline{P1}(\beta)]!![P2(\gamma)\&\overline{P2}(\gamma)]=P(\alpha, \gamma)\&\overline{P}(\beta, \gamma)$$

Здесь α, β, ..., γ — константы или переменные. В отличие от правила для одноместных предикатов $[P2(\gamma)\&\overline{P2}(\gamma)]$ недопустимо приравнивать $\square$.

Правило обратной замены:

$$P(\alpha, \beta, \ldots, \gamma)=P1(\alpha)!!P2(\beta)\ldots!!Pz(\gamma).$$

Для нашего примера получаем

$$[P1(\check{a})_{a=\check{h}}\vee P1(\check{a})\&P1(\check{h})_{a\neq\check{h}}]!![P2(c)\&P2(\check{b})_{c\neq\check{b}}\vee P2(c)_{c=\check{b}}]=$$

$$=P(\check{a}, c)\&P(\check{a}, \check{b})_{\substack{a=\check{h}\\ c\neq\check{b}}}\vee P(\check{a}, c)_{\substack{a=\check{h}\\ c=\check{b}}}\vee P(\check{a}, c)\&P(\check{h}, \check{b})_{\substack{a\neq\check{h}\\ c\neq\check{b}}}\vee$$

$$\vee P(\check{a}, \check{b})\&P(\check{h}, \check{c})_{\substack{a=\check{h}\\ c=\check{b}}}\vee P(\check{a}, c)\&P(\check{h}, c)_{\substack{a\neq\check{h}\\ c=\check{b}}}.$$

Как видно, порождены четыре частные интерпретации, которые взаимно исключают друг друга. Это обстоятельство можно использовать для параллельной реализации независимых трасс логического вывода.

Следующий пример демонстрирует поиск интерпретации, в которой выполняется формула

$$P(A, b)\vee T(C, E)\to P(f, b)\vee\bar{P}(A, b)\&T(e, d).$$

Умножаем левую и правую части формулы на $\bar{P}(f, b)$ и $\overline{P(A, b)\&T(e, d)}$.

Результаты произведений представим в следующем виде:

1. $P(A, b)\&\bar{P}(f, b)=[\overline{P1}(f)\&P1(A)]!![\overline{P2}(b)\&P2(b)]=$

$$=[P1(\check{a})\&\overline{P1}(f)_{A=f}]!![\overline{P2}(b)\&P2(b)]=P(\check{a}, b)\&\overline{P}(f, b)_{A\neq f}.$$

2. $\overline{\bar{P}(A, b)\&T(e, d)}=P(A, b)\vee\overline{T}(e, d).$

3. $(P(\check{a}, b)\&\overline{P}(f, b)_{A\neq f})\&P(A, b)=$

$$=(P(A, b)\&P(\check{a}, b))\&\overline{P}(f, b)_{A\neq f}=$$

$$=P(\check{a}, b)\&\overline{P}(f, b)_{A\neq f}.$$

4. $T(C, E)\&\overline{T}(e, d) =$

$= [T1(C)\&\overline{T1}(e)]!![T2(E)\&\overline{T2}(d)] =$

$= [T1(\check{c})\&\overline{T}1(e)_{C\neq e}]!![T2(\check{e})\&\overline{T}2(d)]_{E\neq d} =$

$= T(\check{c}, \check{e})\&\overline{T}(e, d)_{\substack{C\neq e\\ E\neq d}}.$

Используя результаты 1—4, получаем

$$P(\check{a}, b)\&\overline{P}(f, b)_{A\neq f}\&\overline{T}(e, d)\vee T(\check{c}, \check{e})\&\overline{T}(e, d)_{\substack{C\neq e\\ E\neq d}}\&\overline{P}(f, b)\&P(\check{a}, b)_{A\neq f}\rightarrow\square.$$

Ясно, что формула выполнима в интерпретации $I1=\{A=f, C=e, E=d\}$ и не выполнима в интерпретации $I2=\{A\neq f, C\neq e, E\neq d\}$.

ЗАКЛЮЧЕНИЕ

Остановимся на конспективном изложении основных теоретико-прикладных аспектов развития систем логического программирования, которые остались вне рассмотрения в этой книге.

Развитие концептуальной схемы логического программирования связано с обобщением хорновских клозов для представления знаний о проблеме и с расширением соответствующих механизмов вывода на их основе.

Обобщенный хорновский клоз связывается со следующими допущениями, которые далее последовательно учитываются в записи таких клозов.

1. Кроме чисто логических предикатов могут указываться метапроцедуры, например моделирующие процессы, определенные в гл. 6. Для записи таких процессов будем использовать префикс &, например:

P :— Q, R, &T, &W, Z,

где в дополнение к литералам Q, R, Z указаны моделирующие процессы (программы) T и W. Выполнение головного литерала связано с выполнением литералов Q, R, Z и процессов T и W. Выполнимость T и W понимается в смысле данного в § 6.1 определения абстрактного процесса: процесс T или W выполним (истинен) для указанного начального состояния, если, начав выполнение из этого состояния, он завершит свое выполнение в какомлибо из выделенных конечных состояний. Если процесс завершает свое выполнение в состоянии, не принадлежащем выделенному множеству конечных состояний, то это интерпретируется как неудача. Особое место занимают бесконечно длящиеся процессы, играющие роль, аналогичную невыводимым формулам логики предикатов. Поскольку процессы T и W развиваются во времени, то до тех пор, пока они не выполнены, состояние литерала P не определено. Это значит, что все зависящие от P процессы не могут быть запущены.

Таким образом, бесконечно длящиеся процессы приводят к неопределенным результатам, интерпретируемым так, что система

не знает к моменту завершения моделирования, истинно или ложно доказываемое утверждение.

2. Для управления процессами в обобщенном временно́м клозе введем временны́е отношения, имея в виду, что литералы интерпретируются как процессы с нулевой длительностью:

а) P1, P2 — процесс P_2 запускается в момент завершения P_1;

б) P1=P2 — моменты запуска P_1 и P_2 совпадают;

в) P1>P2 — P_1 запускается раньше P_2, при этом возможно квазипараллельное протекание процессов;

г) P1>>P2 — P_1 запускается и завершается раньше, чем запускается P_2;

д) P1<>P2 — моменты запуска P_1 и P_2 не связаны.

Для представления отношений а-д введем следующие функции:

$LAST^{max}$(P1, ..., Pn) — максимальный момент завершения процессов P_1, ..., P_n,

$LAST^{min}$(P1, ..., Pn) — минимальный момент завершения процессов P_1, ..., P_n,

$EARL^{max}$(P1, ..., Pn) — максимальный момент запуска процессов P_1, ..., P_n,

$EARL^{min}$(P1, ..., Pn) — минимальный момент запуска процессов P_1, ..., P_n.

Чтобы отличать временны́е аргументы от других аргументов предикатов, будем заключать их в квадратные скобки. Запись P[term] означает, что момент запуска для P есть term. Имеют место соответствия

P1=P2 и $P1[EARL^{min}(P2)]$ или $P1[EARL^{max}(P2)]$,

P1, P2 и $P2[LAST^{max}(P1)]$,

P1>>P2 и $P2[LAST^{max}(P1)+const]$.

Пример обобщенного временного клоза:

$$\begin{aligned}&Q(X, Y, Z, [LAST^{max}(\&P1, P2, P3, \&P4)] :—\\ &\quad \&P1(X, W, [EARL^{min}(P2)+const])\\ &\quad P2(Y, V, T, [\varnothing])\\ &\quad P3(Y, W, [LAST^{max}(\&P1, P2)])\\ &\quad \&P4(T, [LAST^{max}(P3)])\end{aligned}$$

Укажем, что порядок, в котором перечислены процессы, не обязательно совпадает с порядком их запуска. Поскольку порядок запуска процессов заранее неизвестен, усложняется сам механизм возврата, однако это связано с дополнительными техническими, а не принципиальными трудностями.

3. Логика управления. Дополнительно к временны́м отношениям на множестве предикатов можно ввести следующие отношения:

а) отношение несовместности — предикаты P1 и P2 несовместны (P1#P2) если в любой интерпретации, в которой истинен P1, ложен P2, и наоборот;

б) отношение частичной несовместности — предикаты P1 и P2 несовместны в интерпретации I, если в этой интерпретации один из них истинен, а другой ложен;

в) отношение запрещения — запись P1 —o P2 означает, что истинность P1 требует ложности P2 (обратное может не иметь места); очевидно, что P1#P2 ~ (P1 —o P2)&(P2 —oP1);

г) отношение альтернативного выбора — запись P1 : P2 : ... : Pn означает, что истинен один из альтернативных предикатов;

д) отношение предпочтительности — P1 ▷ P2, если предпочтительнее сначала доказывать P1, а затем P2;

е) отношение эквивалентности — эквивалентность P1 ~ P2 означает, что доказав истинность предиката P1, докажем истинность P2, и наоборот;

ж) отношение смежности P1|← P2, если сразу за доказательством P1 следует доказывать P2.

Множество управляющих отношений образует нелогическую часть программы, которая используется для планирования процесса доказательства цели. Это связано с модификацией традиционного прологовского порядка вызова предикатов, но позволяет выполнить программу более эффективно.

4. Использование нечеткой логики. Истинность или ложность предиката оценивается числом в интервале [0, 1]. Запись (0.3 P) интерпретируем как оценку истинности предиката P, равную 0,3.

Пусть даны $\mu_p P$, $\mu_q Q$, где μ_p, μ_q — оценки истинности предикатов P и Q соответственно. Тогда

$$\text{а)}\ \mu_z Z = \mu_p P \& \mu_q Q,\ \mu_z = \min(\mu_p, \mu_q);$$

$$\text{б)}\ \mu_z Z = \mu_p P \vee \mu_q Q,\ \mu_z = \max(\mu_p, \mu_q);$$

$$\text{в)}\ \mu_p = 1 - \mu_p;$$

$$\text{г)}\ \mu_z Z = \mu_p P \to \mu_q Q = \mu_{\bar{p}} \bar{P} \vee \mu_q Q,\ \mu_z = \max(1 - \mu_p, \mu_q).$$

Вывод в системе нечеткой логики строится с учетом операций а) — г) и правил вывода логики предикатов.

Развитие языка Пролог связано со следующими аспектами:

интеграция с языками логического и функционального программирования и имитационного моделирования;

обеспечение диалога с пользователем для участия его в управлении процессом вывода, а также поиска компромиссных решений при наличии взаимоисключающих целей и возможности построения сложных (неконъюнктивных) целей, которые могли бы противоречить одна другой;

включение внутренних решающих процедур, обеспечивающих «прохождение» неоднозначных, незаконченных или неверных участков программы, и операторов-демонов для изменения программы;

использование операторов для описания индуктивных рассуждений, эвристик и средств поддержки и генерации гипотез;

введение понятия частичного решения, которое, хотя и связано с состоянием fail, имеет самостоятельную ценность, программирование состояний, отличных от fail и success;

моделирование семантики (наличие средств для описания целей вывода, прагматики и контекста).

Развитие аппаратного окружения языка Пролог в повышении скорости обработки программы. Необходима архитектурная концепция базовой логической машины, являющейся физической реализацией абстрактной машины (например, Уоррена как одной из наиболее известных). В отличие от универсальной ЭВМ фон Неймана в аппаратном окружении Пролога должна быть учтена специфика алгоритмов логического вывода, используемых структур данных и параллелизм. Практические аспекты развития аппаратного окружения состоят в следующем:

необходимость в первую очередь аппаратной поддержки унификации и возврата, занимающих основную часть времени обработки;

использование внешнего параллелизма — предвыборки и предобработки команд, параллелизма пересылок данных и микропрограмм, реализацию многопортовой памяти;

построение универсальной логической машины на идеях, изложенных выше, с определением принципов ее работы и организации.

СПИСОК ЛИТЕРАТУРЫ

1. **Агафонов В. Н., Борщев В. Б., Воронков А. А.** Логическое программирование в широком смысле//Логическое программирование. — М.: Мир, 1987.— С. 298—366.
2. **Бенерджи Р.** Теория решения задач. — М.: Мир, 1978. — 345 с.
3. **Бранохе М.** Управление памятью в реализациях Пролога//Логическое программирование. — М.: Мир, 1988. — С. 193—210.
4. **Вишняков В. А. и др.** Возможности языков моделирования для логического программирования//Тез. докл. 1-й Всесоюз. науч.-техн. конф. по искусственному интеллекту. — М.: Наука, 1988. — Т. 3. — С. 215—220.
5. **Вишняков В. А. и др.** Аппаратная поддержка логического вывода Пролога в ПЭВМ//Искусственный интеллект в автоматическом управлении технологическими процессами. Тез. докл. Всесоюзн. науч-техн. конф. — М.: ВНИИИиЭ, 1989. — С. 83—84.
6. **А. с.** 1575188 СССР. Устройство адресации памяти/В. А. Вишняков и др. — Опубл. 1990, Бюл. № 24.
7. **Вишняков В. А., Дементьев И. В., Папков А. С.** Аппаратное обеспечение объектно-ориентированной ЭВМ//Передовой опыт. — 1989. — № 3. — С. 26—29.
8. **Вишняков В. А., Хведчук В. И.** Подход к параллельной реализации функций логического программирования//Электронное моделирование. — 1990. — № 2. — С. 25—29.
9. **Гэри М., Джонсон Т.** Вычислительные машины и труднорешаемые задачи. — М.: Наука, 1982. — 480 с.
10. **Ефимов Е. И.** Решатели интеллектуальных задач. — М.: Наука, 1982. — 317 с.

11. **Заявка** Японии № 61-194537.
12. **Заявка** Японии № 61-194538.
13. **Кахро М. И., Калья А. П., Тыугу Э. Х.** Инструментальная система программирования ЕС ЭВМ (ПРИЗ). — М.: Финансы и статистика, 1987. — 158 с.
14. **Кларк К. Л., Маккейб Ф. Г.** Микро-Пролог: Введение в логическое программирование. — М.: Радио и связь, 1987.
15. **Клоксин У., Меллиш К.** Программирование на языке Пролог. — М.: Мир, 1987. — 234 с.
16. **Ковальский Р.** Логическое программирование//Логическое программирование. — М.: Мир, 1988. — С. 298—366.
17. **Компьютеры** на СБИС/Т. Мотоока и др. — М.: Мир, 1988. — 324 с.
18. **Мануэль Т.** Проблемы внедрения искусственного интеллекта//Электроника.— 1985. — Вып. 3. — С. 30—40.
19. **Мануэль Т.** Система разработки ПО, повышающая производительность труда программистов//Электроника. — 1985. — Вып. 3. — С. 39—46.
20. **Маслов С. Ю.** Теория дедуктивных систем и ее применение. — М.: Радио и связь, 1986. — 132 с.
21. **Нильсон Н.** Принципы искусственного интеллекта. — М.: Радио и связь, 1985. — 372 с.
22. **Пат.** ЕПВ № 0104487.
23. **Попов Э. В.** Экспертные системы. — М.: Наука, 1987. — 285 с.
24. **Построение** экспертных систем/Под ред. Ф. Хейс-Рота. — М.: Мир, 1987. — 440 с.
25. **Слейгл Дж.** Искусственный интеллект. — М.: Мир, 1973. — 313 с.
26. **Тамура Н. и др.** Последовательная Пролог-машина РЕК//Язык Пролог в пятом поколении ЭВМ. — М.: Мир, 1988. — С. 310—325.
27. **Тик Э., Уоррен Д. Г. Д.** Конвейерный Пролог-процессор//Язык Пролог в пятом поколении ЭВМ. — М.: Мир, 1988. — С. 248—261.
28. **Тыугу Э. Х.** Концептуальное программирование. — М.: Наука, 1984.— 252 с.
29. **Хоггер К.** Введение в логическое программирование. — М.: Мир, 1988. — 348 с.
30. **Чень Ч., Ли Р.** Математическая логика и автоматическое доказательство теорем. — М.: Наука, 1983. — 360 с.
31. **Шрайбер Т. Дж.** Моделирование на GPSS. — М.: Машиностроение, 1980.
32. **Экспертные** системы/Под ред. Р. Форсайта. — М.: Радио и связь, 1987. — 220 с.
33. **Элти Дж., Кулебс М.** Экспертные системы концепции и примеры. — М.: Финансы и статистика, 1987. — 19 с.
34. **Despain A. M., Patt Y. N., Dobry T. P.** High Performance PROLOG the Multiplicative Effect of Several Levels of Implemention//Comput. Soc. Internat. Conf., Compcom, 1984.
35. **Dobry T. P., Despain A. M., Patt Y. N.** Performance Studies of a Prolog Machine Architecture//IEEE Trans. — 1985. — Vol. 2. — P. 188—190.
36. **Goto A., Tanaka H., Moto-oka T.** Highly Parallel Inference Engine PIE-goal Rewriting Model and Machine Architecture//New Generation Computing. — 1984. — Vol. 2, N 1. — P. 37—58.
37. **Levi G. A.** GHL Abstract Machine and Instruction Set// Конференция МПЛ-86. — С. 157—171.
38. **Lloyd J. W.** Foundations of Logic Programming. — Berlin: Springer-Verlag, 1984. — 215 p.
39. **Nakazaki R. et al.** Design of a High-Speed Prolog Machine (NPM)//Proc. 12th Annual Internat. Symp. Computer. — 1985. — P. 191—196.
40. **Nakazaki R. et al.** Design of a CoOperative High Peformance Sequential Inference Machiner (CHI)//NEC Research and Development. — 1986. — Vol. 1, № 4.
41. **Nishikawa H.** The Personal Sequential Inference Machine (PSI): its Design Philosophy and Machine Architecture//Proc. Logic Programming Workshop. — 1983. — P. 53—73.
42. **Ona R., Also M., Shumizu H. et al.** Architecture of a Rediction based Parallel Inference Machine PIM-R//New Generation Computer. — 1983. — N 3. — P. 197—228.
43. **Robinson I.** A PROLOG Processor Based on a Pattern Matching Device// Конференция МПЛ-86. — С. 172—179.
44. **Taki K., Nakajima K., Hakajima H., Ikedd H.** Performance and Architectural Evolution of the PSI Mashine//ACM. — 1987. — Vol 7. — P. 128—134.
45. **Uchida S. et al.** Outline of the Personal Sequential Inference Machine: PSI// New Generation Computing. — 1983. — Vol. 1, N1.
46. **Uchida S.** Inference Machine: From Sequential to Parallel//Proc. 10th Internat. SYMP. Computer Architecture. — 1983. — P. 410—416.
47. **Vlahavas I., Hatatsis C.** RISC PROLOG Machine Architecture//Microprocessing and Microprogramming. — 1987. — N 21. — P. 259—266.
48. **Wada K., Myhamota V., Kio S.** Intermedialecode for Sequantial Prolog Machine PEK//Microprocessing and Microprogramming. — 1987. — N 21. — P. 275—282.
49. **Warren D. H. D.** Implementing Prolog — Compiling Predicate Logic Programs//DAI Research Report N 39—40/University Edinburg. — 1977. — Vol. 1, 2.
50. **Warren D. H. D.** An Abstract Prolog Instruction Set. Tech. Note 309, AI Research Center, SRI International, 1983.
51. **Wise G.** EPILOG-reinterpreting and Extending PROLOG for Multiprocessor Environment// Реализация-84. — С. 252—268.
52. **Yokota M. et al.** The Design and Implementation of a Personal Sequential Inference Machine: PSI//New Generation Computing. — 1983. — Vol. 1, N2.
53. **Yokota M. et al.** A Microprogrammed Interpreter for the Personal Sequential Inference Machine//Proc. Internat. Conf. ON FGCS, ICOT. — 1984.

ОГЛАВЛЕНИЕ

Научное издание

Вишняков Владимир Анатольевич, **Буланже** Дмитрий Юрьевич, **Герман** Олег Витольдович

АППАРАТНО-ПРОГРАММНЫЕ СРЕДСТВА ПРОЦЕССОРОВ ЛОГИЧЕСКОГО ВЫВОДА

Заведующий редакцией Ю. Г. Ивашов
Редактор Т. М. Толмачева
Переплет художника В. А. Алексеева
Художественный редактор Н. С. Шеин
Технический радектор А. Н. Золотарева
Корректор Н. Л. Жукова

ИБ № 2256

Сдано в набор 21.03.91 Подписано в печать 18.09.91
Формат 60×90¹/₁₆ Бумага тип. № 1 Гарнитура литературная
Печать высокая Усл. печ. л. 16,5 Усл. кр.-отт. 16,5
Уч.-изд. л. 18,45 Тираж 6000 экз. Изд. № 23154
Зак. № 39 Цена 5 р.

Издательство «Радио и связь». 101000 Москва, Почтамт, а/я 693

Типография издательства «Радио и связь». 101000 Москва, ул. Кирова, д. 40