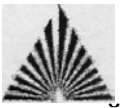


ЮЖНЫЙ
ФЕДЕРАЛЬНЫЙ
УНИВЕРСИТЕТ

И.И. Левин
А.И. Дордопуло
В.А. Гудков

ПРОГРАММИРОВАНИЕ РЕКОНФИГУРИРУЕМЫХ ВЫЧИСЛИТЕЛЬНЫХ УЗЛОВ НА ЯЗЫКЕ COLAMO

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Технологический институт
Федерального государственного образовательного
учреждения высшего профессионального образования
«Южный федеральный университет»

Научно-образовательный центр
«Многопроцессорные вычислительные
и управляющие системы»
Южного федерального университета

Базовая кафедра Южного научного центра РАН
«Интеллектуальные и многопроцессорные системы»
Технологического института Южного федерального университета в г. Таганроге

И.И. Левин
А.И. Дордопуло
В.А. Гудков

Программирование реконфигурируемых вычислительных узлов на языке COLAMO

Учебное пособие

Таганрог 2011

УДК 004.432

Рецензенты:
д.т.н., зав. кафедрой информатики Таганрогского государственного педагогического института Я.Е. Ромм
д.т.н., зав. лабораторией машиностроения и высоких технологий Южного научного центра РАН С.Н. Шевцов

Левин И.И., Дордопуло А.И., Гудков В.А. Программирование реконфигурируемых вычислительных узлов на языке COLAMO: Учебное пособие. – Таганрог: Изд-во ТТИ ЮФУ, 2011. - с.

В данном пособии дается анализ существующих языков параллельного программирования различных архитектур вычислительных систем. Приводятся их особенности и недостатки. Представлен язык высокого уровня COLAMO, используемый для программирования реконфигурируемых систем. Рассмотрены основные особенности данного языка. Показаны различные варианты параллельных программ на языке COLAMO и соответствующие им информационные графы. Рассматриваются интегрированная среда разработки параллельных программ и основные параметры ее настройки.

Учебное пособие предназначено для магистрантов, аспирантов и студентов, обучающихся по направлению «Информатика и вычислительная техника», а также для специалистов, занимающихся разработкой параллельных прикладных программ для реконфигурируемых вычислительных систем при изучении новых информационных технологий в науке, технике и образовании.

Табл. 0. Ил. 57. Библиогр.: 39 назв.

ISBN

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Настоящее учебное пособие ориентировано на студентов и аспирантов специальностей «Информатика и вычислительная техника», а также на студентов и аспирантов других специальностей при проведении практических и лекционных занятий.

Для наглядности излагаемого материала в тексте приведены программы и эквивалентные им информационные графы.

В первой главе представлен анализ современных средств параллельного программирования традиционных многопроцессорных вычислительных систем (системы с кластерной архитектурой) и реконфигурируемых вычислительных систем (РВС). Показаны преимущества и недостатки языков параллельного программирования при разработке параллельных программ для реконфигурируемых вычислительных систем. Рассмотрен язык высокого уровня COLAMO, используемый для разработки параллельных прикладных программ, выполняющихся на РВС.

Во второй главе рассмотрены методы программирования на языке COLAMO. Показаны правила доступа к внутренней и внешней памяти. Рассмотрены правила «однократного присваивания» и «единственной подстановки». Представлены примеры организации параллельной и конвейерной обработки данных.

В третьей главе приведены особенности использования вычислительных конструкций языка COLAMO. Показаны различные варианты взаимодействия условных операторов, циклов и вычислительных структур и соответствующие им информационные графы.

В четвертой главе приводится описание интегрированной среды разработки. Показаны основные элементы интерфейса интегрированной среды, а также рассмотрены основные настройки конфигурации среды.

Авторы

СПИСОК СОКРАЩЕНИЙ

РВС – реконфигурируемая вычислительная система
ПЛИС – программируемая логическая интегральная схема
МВС – многопроцессорная вычислительная система
КРП – контроллер распределенной памяти

ВВЕДЕНИЕ

Для обеспечения скорости роста производительности создаются и развиваются новые архитектуры вычислительных систем. По мнению большинства специалистов в области организации высокопроизводительных вычислений, фон-неймановские архитектуры уже не в состоянии обеспечивать неуклонный рост производительности, поскольку комплексирование микропроцессоров как отдельных вычислительных устройств, так и их составных частей (многоядерные и мультитредовые процессоры) не приводит к пропорциональному увеличению производительности из-за высоких накладных расходов, необходимых для организации параллельных вычислений, представляющих собой множество взаимосвязанных последовательных процессов, поверх которых наложены процедуры межпроцессорных информационных обменов и синхронизации вычислений.

В этой связи широкое распространение начинают получать альтернативные варианты построения суперЭВМ. К числу подобных параллельных вычислительных комплексов относятся гибридные машины, содержащие в своем составе как унифицированные микропроцессоры, так и высокопроизводительные вычислительные компоненты не фон-неймановской архитектуры. К числу подобных компонентов следует отнести графические векторные процессоры и программируемые логические интегральные схемы (ПЛИС), в зарубежной архитектуре они называются FPGA (Field-Programmable Gate Array) [1,2]. Следует отметить, что гибридные вычислители хотя и повышают реальную производительность при решении вычислительно трудоемких задач, однако имеют предел потенциального роста реальной производительности, поскольку только лангируют возможности множества процессоров фон-неймановской архитектуры, связанных между собой ограниченной коммутационной системой.

Принципиально новыми свойствами обладают реконфигурируемые вычислительные системы, в которых ПЛИС используются не в качестве вспомогательных сопроцессоров, а в виде основного средства преобразования информации. Вычислительные поля, построенные на основе множества ПЛИС и сегментов распределенной памяти, позволяют многократно повысить реальную производительность при решении научно-технических задач, в том числе сильносвязанных задач (требующих больших и зачастую иррегулярных информационных обменов между компонентами системы). В ряде случаев выигрыш в скорости обработки информации в РВС составляет 2-4 десятичных порядка по сравнению с МВС кластерной архитектуры. Это преимущество достигается за счет адаптации структуры как отдельных ПЛИС, так и их совокупности, к информационной структуре решаемой задачи.

Основными сдерживающими факторами для широкого распространения реконфигурируемых вычислительных систем являются сложность и трудоемкость их программирования, а также отсутствие удобных высокоуровневых средств разработки приложений.

Для программирования РВС в настоящее время наибольшее распространение получили языки HDL-группы, позволяющие описывать структуру и функционирование вычислительной системы, а также моделировать обмен данными между блоками системы. Наиболее популярным языком этой группы является язык VHDL, который обладает высоким функциональным уровнем абстракции. Однако достоинства, позволившие языку VHDL стать стандартом де-факто в области проектирования цифровой техники, требуют от разработчика больших усилий для описания логики работы устройства в виде довольно объемной программы. Громоздкое текстовое описание устройств на VHDL обусловили популярность и широкое использование систем автоматизированного проектирования в виде специализированных графических редакторов, таких как Xilinx ISE, Altera MaxPlus или OrCad. Графические редакторы

позволяют пользователю быстро создавать проекты, используя как встроенные библиотечные блоки среды разработки, так и созданные программистом-схемотехником специализированные блоки, которые будут оттранслированы в язык VHDL на дальнейших стадиях синтеза схемотехнического решения.

Такой подход к программированию РВС требует участия в программировании системы как специалиста-схемотехника, создающего конфигурацию вычислительной системы с учетом особенностей ее архитектуры и элементной базы, так и прикладного программиста, создающего параллельную программу, описывающую потоки данных в созданной схемотехником виртуальной вычислительной структуре.

Естественное желание программировать РВС на языке высокого уровня привело к созданию ряда процедурных языков, таких как ImpulseC, Mitrion-C, Catapult C, Handel-C и др. Как видно из названия языков этой группы, все они обладают привычным для большинства программистов персональных ЭВМ синтаксисом языка С и отличаются между собой некоторыми семантическими особенностями вызова и использования тех или иных операторов. Для описания параллельных процессов в РВС в этих языках используется изначально последовательный язык, семантика которого ориентированна на взаимодействие последовательных процессов, что и не позволяет в полной мере использовать все возможности РВС. Это приводит к семантическому разрыву между исходным информационным графом алгоритма задачи, его описанием на языке высокого уровня и созданной транслятором схемотехнической реализацией, что выражается в существенном снижении эффективности приложения. Как правило, созданные с помощью указанных языков приложения имеют в 3-5 раз более низкую производительность по сравнению с приложениями, разработанными на более низких уровнях абстракции.

Данных недостатков лишен язык COLAMO, разработанный в НИИ МВС ЮФУ. Язык COLAMO является языком высокого уровня с неявным описанием параллелизма,

распараллеливание в котором достигается за счет объявления типов переменных и организации доступа к элементам массива (индексации). В настоящее время язык высокого уровня применяется для программирования реконфигурируемых вычислительных систем и позволяет создавать параллельные прикладные программы, выполняющиеся на РВС с высокой удельной производительностью. Под удельной производительностью РВС понимается отношение реальной производительности системы к числу эквивалентных вентилей системы.

Необходимость разработки учебного пособия обусловлена тем, что приведенный материал познакомит разработчиков с основными существующими языками параллельного программирования, а также поможет при создании и отладке параллельных прикладных программ, написанных на языке высокого уровня COLAMO.

Глава 1. ОБЗОР СОВРЕМЕННЫХ СРЕДСТВ РАЗРАБОТКИ ПРИКЛАДНЫХ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ ДЛЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

1.1 Языки программирования МВС

Эффективность параллельных вычислительных систем во многом зависит от программного обеспечения и, в частности, языков программирования. Развитие различных архитектур параллельных вычислительных систем способствовало созданию языков программирования, как правило, ориентированных на программирование конкретной архитектуры вычислительной системы. Для вычислительных систем с общей памятью существует большое разнообразие средств, в некоторых системах используются новые операторы (языковые расширения языков Fortran или C), в других - специальные комментарии, в третьих - обращения к специальным библиотекам (OpenMP), имеются и смешанные стратегии [3].

Для систем с распределенной памятью обычно используются механизмы взаимодействия процессов и передачи сообщений. Как правило, при программировании таких систем программа разбивается на фрагменты (подзадачи), каждая из которых выполняется на отдельном узле (процессоре), а связь между подзадачами осуществляется с помощью передачи сообщений [4].

Программирование многопроцессорных систем на традиционных языках высокого уровня (Fortran, C, Pascal) часто реализуется с помощью специальных библиотек (MPI, OpenMP или PVM), содержащих подпрограммы поддержки параллельности [5,6,7]. Такой подход к программированию не затрагивает исходный язык программирования и его компилятор, что позволяет пользователю выбирать между различными библиотеками с целью нахождения наиболее удачного средства для программирования МВС. Однако

большое число обращений к библиотечным подпрограммам в исходной параллельной программе может привести к невозможности ее оптимизации компилятором и к трудоемкому переходу от одной библиотеки к другой.

Другим методом программирования МВС является использование расширения существующих языков на основе введения специальных комментариев (директив), новых операторов и элементов описания переменных, позволяющих пользователю явно задавать параллельную структуру программы и управлять ее исполнением. Одним из таких языков является язык высокого уровня Fortran [8], созданный как машинно-независимый инструмент однопроцессорной обработки вычислительных задач. Общая тенденция к параллельной обработке данных в современных высокопроизводительных системах привела к появлению языка Fortran-90/95 [9], включающего элементы параллелизма: операции над массивами и секциями массивов, а в Fortran-95 - средства задания параллелизма циклов. Первыми расширениями языка Fortran являются язык FortranD, разработанный коллективом под руководством Ken Kennedy, и язык Fortran Vienna, созданный группой из Венского университета [10].

С целью объединения усилий и унификации языковых средств языка Fortran в январе 1992 года образована группа HPF (High Permormance Forum) [11]. Модель программирования HPF основывается на выполнении всеми процессорами одной и той же программы с отведенной для каждого процессора частью массива данных, что обеспечивается директивами HPF (спецкомментариями, которые начинаются с символов !HPF$), а параллелизм определяется инструкциями Fortran-90 и HPF.

Подход, предлагаемый в языке HPF, обеспечивает более высокий уровень программирования и ориентирован на более широкий класс параллельных вычислительных систем. Однако нерешенной остается проблема оптимального использования массивно-параллельных вычислений, а основные задачи оптимизации возлагаются теперь на компилятор, поэтому от

него требуется гораздо более сложный (интеллектуальный) анализ алгоритма.

Одним из расширений языка С является язык mpC, разработанный в конце 90-х годов прошлого века в Институте системного программирования РАН [12]. Язык mpC разработан специально для программирования параллельных вычислений на обычных сетях разнородных компьютеров. Программа на языке mpC явно определяет абстрактную неоднородную вычислительную сеть, наиболее подходящую для реализуемого параллельного алгоритма и распределяет по ней данные, вычисления и коммуникации, причем делает это динамически во время выполнения программы [13]. Параллельная программа на mpC представляет собой множество параллельных процессов, динамически распределяемых между виртуальными процессорами. Основным механизмом синхронизации параллельных процессов, взаимодействующих с помощью передачи сообщений, является барьерная синхронизация. К недостаткам данного языка можно отнести отсутствие средств управления тем, сколько процессов выполняется в программе и на каких узлах вычислительной системы, а также требование к высокому профессионализму программиста как в логическом программировании, так и в практическом использовании языка.

Среди высокоуровневых объектно-ориентированных языков можно выделить MC# - язык параллельного программирования для платформы .NET, поддерживающий создание программ, работающих во всём спектре параллельных архитектур - от многоядерных процессоров до Grid-сетей [14].

Язык MC# является расширением базовых идей языка Polyphonic C# (разработан в 2002 г. компанией Microsoft Research Laboratory (г. Кембридж, Великобритания) [15]) на случай параллельных и распределенных вычислений, в который включены высокоуровневые средства асинхронного параллельного программирования, позволяющие программировать серверные и клиент-серверные приложения.

Недостатком данного языка является возможность программирования только одной машины или нескольких

машин с зафиксированными на них асинхронными методами, взаимодействующими между собой при помощи средств удаленного вызова, представленных соответствующей библиотекой платформы .NET.

В начале 1980-х годов в рамках работ по созданию транспьютеров группой учёных из Оксфорда под руководством Дэвида Мэя был разработан процедурный язык параллельного программирования высокого уровня Occam [16]. В отличие от рассмотренных ранее языков параллельного программирования язык Occam является оригинальной разработкой и не является каким-либо расширением известного языка программирования. В основе языка Occam лежит концепция взаимодействующих последовательных процессов (Communicating Sequential Processes), разработанная Ч. Хоаром [17], позволяющая формализовать описание соответствующей вычислительной модели. Каждый процесс описывает некоторые действия, подлежащие выполнению, оперируя с идентификаторами, которые зависят от конкретной машины, применяемой для исполнения программы.

Язык Occam позволяет явно определять последовательные и параллельные процессы при помощи конструкторов SEQ и PAR соответственно. В конструкторе PAR, в отличие от конструктора SEQ, порядок выполнения действий не определен, при этом процессы считаются выполненными, если завершены все его составляющие. Для управления процессами и организации циклов используются конструкторы условного процесса IF циклического процесса WHILE и процесса выбора процессов ALT. Процессы ALT и PAR привносят в язык индетерминизм, так как считается, что при одновременном выполнении нескольких условий точно предсказать дальнейший ход событий невозможно.

Таким образом, язык Occam имеет достаточно средств для организации параллельных ветвей, синхронизации управления в параллельных ветвях и обмена информацией между ветвями. Однако Язык Occam эффективен для создания

параллельных программ для систем, содержащих небольшое число процессоров.

В 1980 году в результате выполнения проекта, предпринятого Министерством обороны США с целью разработать единый язык программирования для систем управления автоматизированными комплексами, работающими в реальном времени, был создан объектно-ориентированный язык программирования Ada, содержащий высокоуровневые средства программирования параллельных процессов [18].

Язык обладает механизмами поддержки параллельного исполнения, распределённых вычислений, а также имеет стандартные интерфейсы к другим языкам и библиотекам. Для параллельного программирования в языке определены три основных понятия: «задача», «вход задачи» и «рандеву». Понятие «задача» определяет параллельно выполняемый фрагмент программы, понятие «вход задачи» - средство синхронизации и коммуникации параллельно выполняющихся задач, механизм «рандеву» - протокол взаимодействия параллельно выполняемых задач через вход одной из них. Для организации условного межпотокового взаимодействия используется оператор Select, который осуществляет выбор параллельной задачи, с которой следует взаимодействовать.

Язык Ada обладает хорошей переносимостью по сравнению с другими языками, поскольку прошел наиболее полную и подробную стандартизацию и уверенно занимает нишу больших встроенных систем с повышенными требованиями к надежности. Однако разработка параллельных программ на языке Ada является очень трудоемкой по сравнению с другими языками, а существующие реализации Ады крайне дороги.

В 1985-1988 годах было создано средство параллельного программирования – Linda, которое не является языком программирования как таковое, а является моделью параллелизма и может быть использовано в существующих языках программирования. Взаимодействие процессов на языке Linda осуществляется по средствам глобальной *кортежной*

области (*Tuple Space*) и в отличие от языков Ada и Occam не использует никакой адресации между параллельными процессами.

Преимущество модели Linda - в чрезвычайной гибкости, она позволяет писать параллельные программы на обычных языках (C, Fortran, Lisp), дополняя их операциями межпроцессорного обмена данными, причем любой процесс можно разбить на небольшие процессы, что позволяет упростить и распараллелить процесс [19]. Однако если процесс будет разделен на большое число мелких, то может значительно (в несколько раз) возрасти время на обмен данными между процессами, а также результатами и получением новых заданий, что приведет к снижению производительности вычислительной системы. Также для организации кортежной области необходимы дополнительные затраты, т.к. она требует потенциально неограниченной глобальной памяти.

Рассмотренные языки параллельного программирования используют модель передачи сообщений и не уделяют внимания общей информационной структуре задачи. Распараллеливание осуществляется на уровне локальных участков (циклы, ветви, процедуры) изначально последовательной программы, что не позволяет достигать *максимального распараллеливания программы*.

Принципы программирования представленных языков, в основном, ориентированы на многопроцессорные вычислительные системы (МВС) с «жесткой» архитектурой, например, кластерные вычислительные системы, и не позволяют создавать параллельные программы для реконфигурируемых вычислительных систем.

1.2. Языки программирования РВС

Для программирования РВС в настоящее время наибольшее распространение получили языки HDL-группы, позволяющие описывать структуру и функционирование вычислительной системы, а также моделировать обмен данными

между блоками системы. Языки HDL-группы относятся к языкам описания и моделирования аппаратуры, среди которых можно выделить такие языки как Verilog и VHDL [20].

Язык VHDL является языком высокого уровня и позволяет выполнять как поведенческое (в виде последовательных программных предложений), так и структурное (в виде иерархии связанных компонентов) и потоковое (в виде множества параллельных регистровых операций) описания цифровых схем [21]. С точки зрения программиста, в языке VHDL можно выделить два компонента - общеалгоритмический и проблемно-ориентированный.

Общеалгоритмический компонент представляет собой язык, близкий по синтаксису и семантике к современным языкам программирования, и является строго типизированным. Однако, помимо встроенных стандартных типов данных (целый, вещественный, булевский, битовый), пользователь может вводить свои типы данных (перечислимый, диапазонный и др.).

Проблемно-ориентированный компонент позволяет описывать цифровые системы в привычных разработчику понятиях и терминах (модельное время, данные типа (Time), данные вида сигнал (Signal), средства объявления объектов и их архитектур и др.). Проблемно-ориентированный компонент можно разделить на две части: средства поведенческого описания аппаратуры (параллельные процессы и их взаимодействие) и средства потокового описания (описание на уровне межрегистровых передач). К параллельным операторам языка VHDL относятся: оператор процесса, оператор блока, параллельный оператор вызова процедур и утверждения и др.

Несмотря на все свои возможности, язык VHDL обладал недостаточной временной точностью для использования его в качестве моделирующего средства, что указало на недостатки при работе на структурном уровне абстракции. Для устранения этой проблемы были разработаны библиотека VITAL и модуль обратной корректировки информации о задержке. Основным недостатком языка является громоздкость программ, созданных на нем.

В 1984 году фирма Gateway Design Automation разработала язык описания аппаратуры Verilog [22], получивший более широкое распространение среди разработчиков, чем VHDL. Язык Verilog является контекстно-зависимым языком, синтаксически напоминающим очень популярный в среде программистов язык C++. В отличие от VHDL, обладающего более расширенными возможностями, язык Verilog имеет простые и компактные конструкции, что относительно упрощает изучение языка, при этом позволяет эффективно выполнять описание и моделирование цифровых схем.

Основной конструкцией языка является модуль (Module), аналог которого - блок entity в VHDL. Операторы в языке отличаются количеством операндов, число которых не должно превышать трех. Язык имеет фиксированные типы данных, хорошо рассчитывает задержки на уровне вентилей, однако ограничен в управлении проектом и его повторным использованием, а также не поддерживает репликацию структур в отличие от языка VHDL.

Можно выделить язык SystemC, представленный некоммерческой организацией Open SystemC Initiative, состоящей из различных компаний, университетов и сторонних разработчиков [23]. Язык SystemC представляет собой набор классов, которые свободно распространяются и позиционируются как средство моделирования цифровых схем. Помимо моделирования цифровых схем, язык может использоваться как средство описания схем устройств на уровне абстракции RTL. Здесь и далее под RTL понимается язык перемещения регистров, представляющий собой ассемблер, не зависящий от конкретного процессора. Язык SystemC обеспечивает более естественную среду для программно-аппаратного проектирования и проверки и имеет более простые средства синтеза по сравнению с традиционными методами языков VHDL и Verilog. Моделирование устройств на языке проходит в 5-10 раз быстрее, чем на языках VHDL и Verilog, но

в то же время создание RTL-описания устройства требует гораздо больше времени и сил.

Кроме высокоуровневых языков описания и моделирования аппаратуры созданы языки низкого уровня, такие как AHDL (фирма Altera) [24] и ABEL (фирма Xilinx).

Реализация исходного алгоритма на рассмотренных языках является очень трудоемкой и требует от пользователя специальных схемотехнических знаний, т.к. все конструкции языка выполняются исходно параллельно и отсутствуют средства для автоматической синхронизации между ними.

Естественное желание программировать РВС на языке высокого уровня привело к созданию ряда процедурных языков, таких как ImpulseC, Mitrion-C, Catapult C, Handel-C и др. Как видно из названия языков этой группы, все они обладают привычным для большинства программистов персональных ЭВМ синтаксисом языка С и отличаются между собой некоторыми семантическими особенностями вызова и использования тех или иных операторов.

Компания Impulse Accelerated Technologies разработала язык ImpulseC, предназначенный для программирования ПЛИС. Язык ImpulseC представляет собой набор библиотек и средств параллельного программирования, основанных на языке С [25]. Язык разрабатывался с целью упрощения процесса написания достаточно эффективных программ для ПЛИС. Модель программирования ImpulseC, как и модели многих других языков, представляет собой модифицированную версию разработанной Хоаром модели взаимодействующих последовательных процессов. Соответственно основными понятиями модели ImpulseC являются процессы и потоки.

Процессы являются независимыми, параллельно выполняющимися компонентами приложения. Обмен данными между процессами происходит посредством потоков, каждый из которых характеризуется наличием собственного буфера, благодаря которым достигается возможность организации независимых процессов. Процессы и способы обмена данными

между ними (в основном, используются потоки) задаются программистом в коде программы.

Для оптимизации созданной параллельной программы компилятор языка ImpulseC применяет следующие приемы: развертку циклов, организацию конвейера, перестановку операций. Однако при желании пользователь может отключить автоматическую оптимизацию и самостоятельно выполнить модификацию.

Благодаря совместимости языка ImpulseC с ANSI-C отладку и компиляцию параллельной программы можно осуществлять стандартными средствами программирования. Данный язык наиболее подходит для программирования приложений, для которых характерна высокая скорость обмена данными между вычислительными элементами, и обработку каждого потока данных выполнят большое количество независимых процессов, имеющих схожие вычисления. Также следует отметить, что язык применяется в случае обработки данных с фиксированной запятой, если не требуется высокая точность вычислений.

Другим языком параллельного программирования является язык Mitrion-C, разработанный компанией Mitrionics. Mitrion-C является языком высокого уровня, по синтаксису сильно похожим на C, но работает с потоком данных, а не с потоком команд.

В Mitrion-C параллелизм задается на уровне инструкций и на уровне циклов. Параллелизм на уровне инструкций является неявным, поскольку анализ зависимостей данных происходит автоматически на этапе компиляции программы. Порядок выполнения инструкций в языке не определен и зависит от степени готовности данных для каждой конструкции. Таким образом, параллелизм на этом уровне обеспечивается независимостью данных. Наличие параллелизма на уровне инструкций требует от пользователя особого внимания при разработке программы.

Отсутствие последовательности выполнения инструкций привело разработчиков языка к необходимости введения в язык

правила однократного присваивания т.к., по их мнению, результат программы с использованием многократных присваиваний не детерминирован.

Параллелизм на уровне циклов является явным и определяется ключевым словом Foreach. Пользователь сам может определять, какие циклы он хочет распараллеливать, при этом, с точки зрения ресурса ПЛИС, каждая итерация цикла будет отдельно реализоваться в аппаратуре до тех пор, пока существует доступный ресурс. Такой вид параллелизма в языке Mitrion-C является основным средством распараллеливания программы.

Для обеспечения точности вычислений в языке существует возможность вручную определять размерность скаляра, т.е. при объявлении переменной можно указать число бит, которое необходимо под неё выделить. Такая возможность позволяет повысить гибкость и эффективность программы, однако требует большей аккуратности при программировании. Если размерность переменной не указана явно, то данное значение будет вычисляться автоматически, исходя из размеров операндов и типа операции, выполняемой над данной переменной.

Для работы с массивами существует три типа данных: список, поток и вектор. Доступ к элементам списка осуществляется только последовательно, а вектор позволяет обращаться к любому элементу. Использование векторов ограничивается ресурсами ПЛИС, поэтому на практике вектора большого размера используются очень редко. Поток представляет собой последовательность элементов, обработка которых осуществляется по принципу конвейера (пока одни данные считываются из памяти, другие обрабатываются, третьи подаются на выход).

Для управления вычислениями используется условный оператор, реализация которого в архитектуре осуществляется полностью, т.е. реализуются обе ветви оператора. Такой подход приводит к затрате оборудования, поскольку одна из веток будет простаивать при выполнении, что несколько снижает

эффективность использования ПЛИС, однако такой подход позволяет повысить эффективность выполнения программы.

В целом, язык Mitrion-C обеспечивает пользователя достаточно простыми средствами распараллеливания, но предъявляет серьезные требования к разработке алгоритма, т.к. скрытый от пользователя параллелизм не позволяет контролировать количество параллельно выполняющихся процессов и данные, которые они обрабатывают, в отличие от языка COLAMO (см. параграф 1.3) [26], в котором распараллеливание определяется более строгим и формальным образом.

В 1996 году в Оксфордском университете был разработан язык Handel-C. Язык Handel-C в отличие от Mitrion-C похож на стандартный язык C не только синтаксически, но и семантически. Поскольку язык является параллельным языком программирования, то в отличие от языка C в нем присутствуют новые конструкции для организации параллелизма и взаимодействия процессов. Изначально язык Handel-C был ориентирован на программирование встроенных систем, однако в последнее время стал использоваться для программирования высокопроизводительных вычислений на основе реконфигурируемых систем. Как и во многих языках программирования для организации параллелизма, в языке Handel-C используется концепция взаимодействующих последовательных процессов, разработанная Хоаром. В отличие от языка Impulse C в языке предусмотрена возможность явного указания параллельно работающих процессов с помощью блока Par.

Переменные, объявленные в одном процессе, доступны всем «дочерним» его процессам. Такая область видимости способствует появлению коллизий, для предотвращения которых действует правило, согласно которому любой переменной можно присваивать значение только в одном параллельном процессе, а читать можно из любого процесса.

Любая переменная на языке Handel-C реализуется в виде регистра, а массивы представляют собой набор таких

переменных. Такое представление массивов позволяет получить одновременный доступ к разным его элементам. Однако такая организация массивов в языке является серьезным недостатком, поскольку требует существенных аппаратных затрат. Альтернативным способом организации массивов является использование памяти ПЛИС, при этом в каждый момент времени возможен доступ только к одному элементу массива. Среди недостатков можно отметить отсутствие возможности работы с плавающей запятой, поскольку для их реализации требуется большой объём ресурсов ПЛИС. Для работы с числами с плавающей запятой необходимо использовать внешние библиотеки.

1.3. Язык программирования COLAMO

В НИИ МВС ЮФУ программирование реконфигурируемых вычислительных систем осуществляется на языке высокого уровня COLAMO (Common Oriented Language for Architecture of Multi Objects), разработанном Левиным И.И. в 1987 году [27]. Язык COLAMO предназначен для описания реализации в архитектуре РВС специализированной вычислительной структуры на основе принципов структурно-процедурной организации вычислений [28], которая предполагает последовательную смену структурно (аппаратно) реализованных фрагментов информационного графа задачи, каждый из которых является вычислительным конвейером потока операндов.

Фундаментальным типом вычислительной конструкции в языке COLAMO является конструкция «кадр» [29]. Кадром является программно-неделимый компонент, представляющий собой совокупность операторов, которые реализуются в виде арифметико-логических команд и команд чтения/записи, выполняемых на различных функциональных устройствах, соединенных между собой в соответствии с информационной структурой алгоритма.

В работе [30] Левиным И.И. сформулировано следующее определение кадра: «Кадром является объект, определенный в виде тройки $C =< R_i, Q, W_i >$, где Q - структурно-реализованный подграф задачи, R_i и W_i - функции чтения и записи соответственно. Входные и выходные информационные массивы кадра в момент времени t_i можно определить следующим образом: $I_i = R(t_i)$ и $O_i = W(t_i)$.

В теле кадра последовательность следования операторов не является принципиальной в отличие от традиционных языков программирования, поскольку все операторы выполняются асинхронно с максимальным параллелизмом.

В языке COLAMO отсутствуют явные формы описания параллелизма, а распараллеливание достигается с помощью объявления типов доступа к переменным и индексации элементов массивов, что характерно для языков потока данных.

Массивы в языке COLAMO можно разделить по типу доступа к их элементам на три вида: массив с параллельным (векторным) доступом к элементам массива (далее векторный массив), массив с последовательным (потоковым) доступом к элементам массива (далее потоковый массив) и смешанный массив. Векторный массив задается ключевым словом Vector при описании массива, а потоковый массив - ключевым словом Stream [29]. Смешанный массив образуется комбинацией параллельных и последовательных типов доступа к элементам массива.

В зависимости от типа доступа к данным при описании массива одна и та же программа может быть реализована как параллельно, так и последовательно или конвейерно (потоково), при этом синтез соответствующей описанию вычислительной структуры осуществляет транслятор. Если вычисления связывают между собой массивы с различным типом доступа, то заданная пользователем вычислительная структура может оказаться несбалансированной, что приведет к затрате дополнительного оборудования и снизит скорость обработки потоков данных.

Изменение типа доступа позволяет достаточно просто управлять как степенью распараллеливания программы на уровне описания структур данных, так и скоростью обработки и занимаемым ресурсом, что позволяет программисту описывать различные виды параллелизма в достаточно сжатом виде.

Помимо типа доступа, для переменной в языке COLAMO определены также и тип ее хранения: мемориальный (Mem), регистровый (Reg), коммутационный (Com), и внутренняя память (InterMem) [31].

Мемориальной переменной называется величина, хранящаяся в ячейке распределенной памяти и, следовательно, сохраняющая свое значение до очередного переприсваивания. Для мемориальной переменной возможно одновременное выполнение только одного процесса. Поэтому в семантике языка COLAMO для мемориальной переменной в теле кадра действуют правило однократного присваивания и правило единственной подстановки. Правило однократного присваивания указывает на то, что мемориальная переменная в кадре изменяет свое значение только один раз. Правило единственной подстановки определяет, что переменная в кадре может использоваться только для одного процесса чтения или записи.

Для описания связей между элементами информационного графа задачи в языке COLAMO предназначена коммутационная переменная. Поскольку коммутационная переменная описывает информационные связи, то она не требует никакого аппаратного ресурса для своей реализации. Доступ к значению коммутационной переменной после выполнения кадра невозможен. Коммутационная переменная необходима транслятору для указания информационных зависимостей при построении вычислительной структуры задачи. На коммутационную переменную, как и на мемориальную переменную, действует правило однократного присваивания, но не действует правило однократной подстановки. Использование коммутационных

переменных позволяет легко разветвлять и дублировать потоки данных, но не позволяет реализовать рекурсию.

Для организации рекурсии в языке COLAMO используется регистровая переменная, которая представляет собой регистр на аппаратном уровне и используется для хранения промежуточных данных, полученных в процессе вычислений. Единственным ограничением для регистровой переменной в теле кадра является правило однократного присваивания.

Для обеспечения работы с внутренней памятью ПЛИС [32] в язык COLAMO необходимо ввести новый тип хранения переменных - InterMem **<N>**, где N - количество доступных портов при работе с внутренней памятью ПЛИС. Главной особенностью внутренней памяти является возможность осуществлять как процесс чтения, так и процесс записи в любой момент времени, но, в силу действия правила «одноструктурности», на переменную типа InterMem накладываются ограничения, соответствующие мемориальным переменным.

Для управления порядком выполнения операций и реализации ветвлений в языке COLAMO предусмотрен оператор условного перехода IF, который может быть внутренним или внешним по отношению к кадру. Условный оператор представляет собой программную конструкцию следующего вида:

If <*Условие*> Then <*СписокОператоров1*> [Else <*СписокОператоров2* >];

где If, Then, Else - ключевые слова языка,

Условие - логическое выражение,

СписокОператоров1, *СписокОператоров2* - списки операторов, выполняемых при истинности или ложности *Условия* соответственно, альтернативная ветвь Else является необязательной.

Условный оператор, расположенный в теле кадра, реализуется на структурном уровне: обе ветви условного оператора выполняются параллельно, а для выбора результата

используется мультиплексор. Реализации внешнего оператора условного перехода осуществляются на процедурном уровне путем вызова того или иного кадра в зависимости от условия.

Для организации циклов используется оператор FOR, который, по аналогии с условным оператором, может быть как внутренним, так и внешним по отношению к кадру. Внутренний оператор цикла расположен в теле кадра и реализуется структурно в виде аппаратного счетчика, а внешний оператор цикла реализуется процедурно в виде оператора цикла на языке ассемблера ARGUS [33, 34].

При реализации параллельных вычислений с помощью кадров возможна ситуация, когда функционально законченные подграфы кадров изоморфны. В этом случае целесообразно использовать подкадры (SubCadr), являющиеся структурными аналогами подпрограмм. Подкадр является независимой локальной ВС, для которой запрещена рекурсия, в том числе и скрытая, т.е. из подкадра не разрешается вызывать как собственный подкадр, так и последовательность подкадров, которая завершается вызовом текущего подкадра.

Важной особенностью подкадра является то, что при каждом его вызове в тексте программы в структуру кадра будет добавлена вычислительная структура, соответствующая этому подкадру. Использование регистровых и мемориальных переменных в подкадре при его многократных вызовах приведет к большой затрате оборудования, поэтому с целью оптимизации ресурсов в теле подкадра рекомендуется использование коммутационных переменных. Помимо рекурсии в подкадрах запрещено использование операторов передачи управления и операторов описания вычислительных структур [29].

Для объявления неизменяемой (статической) вычислительной структуры в языке COLAMO используется конструкция Let, представляющая собой функционально законченный подграф, структура которого в отличие от структуры подкадра не перегружается в процессе смены кадров, что позволяет сократить время перестройки вычислительной структуры РВС. Конструкция Let может быть отключена

оператором StopLet, в результате чего занимаемый ею аппаратный ресурс будет освобожден. По аналогии с подкадром для конструкции Let запрещена рекурсия.

Для процедурной реализации вычислений в языке COLAMO используется конструкция LocalProc, которая позволяет выполнять вычисления на указанном специализированном процессоре. Вычисления, производимые в процедуре LocalProc, транслируются в специализированные команды соответствующего процессора и выполняются последовательно.

Кроме подкадров, процедур и конструкции Let в языке присутствует конструкция SubRoutine, представляющая собой традиционную подпрограмму. Конструкция SubRoutine позволяет объединять в одну конструкцию несколько кадров и операторов управления. Использование такой конструкции позволяет сократить текст параллельной программы и сделать его более читаемым.

Язык высокого уровня COLAMO отражает все особенности структурно-процедурной организации вычислительного процесса и позволяет эффективно реализовать любой тип параллельной обработки. Использование неявного описания параллелизма и виртуальной системы коммутации упрощает программирование, что позволяет оперативно разрабатывать прикладное программное обеспечение.

1.4. Контрольные вопросы

1. Перечислите основные недостатки языков параллельного программирования многопроцессорных вычислительных систем.
2. Перечислите основные недостатки существующих языков параллельного программирования РВС.
3. Каким образом достигается неявное описание параллелизма в языке COLAMO?
4. Приведите пример языка с явным указанием параллелизма.

5. Дайте определение мемориальной переменной.
6. Дайте определение коммутационной переменной.
7. Дайте определение регистровой переменной.
8. Каким образом осуществляется переход от последовательной обработки данных к параллельной?
9. Перечислите основные вычислительные конструкции языка и их назначение.

Глава 2. МЕТОДЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ COLAMO

2.1. Организация доступа к памяти

Для доступа к ячейкам внешней памяти в языке COLAMO используется мемориальная переменная. Использование в тексте программы мемориальной переменной означает создание нового процесса, тип (чтение или запись) которого определяется в зависимости от контекста использования данной переменной. Для каждого процесса существует адресный генератор, который выполняет формирование потока адресов на адресный вход соответствующей памяти.

При организации потоков данных память позволяет осуществить только одно однократное обращение к ней. Исключение возможности некорректного доступа к памяти несколькими процессами осуществляется в языке COLAMO на синтаксическом уровне. Правило однократного присваивания [35] блокирует возможность одновременной записи нескольких процессов в одну память. Если кадр параллельной программы представить как множество операторов присваивания в виде

$$\begin{aligned}
&\text{Cadr Cadr1;}\\
&a_1 = F_1(b^1_{1,}b^1_{2,},...,b^1_{n1});\\
&a_2 = F_2(b^2_{1,}b^2_{2,},...,b^2_{n2});\\
&\quad\vdots\\
&a_k = F_k(b^k_{1,}b^k_{2,},...,b^k_{nk});\\
&\text{EndCadr;}
\end{aligned} \qquad (2.1)$$

где $A=\{a_1,a_2,...,a_k\}$ - множество мемориальных переменных, соответствующих процессу записи, $B=\{b^1_1,b^1_2,...,b^k_{nk}\}$ - множество мемориальных переменных, соответствующих процессу чтения, $F=\{f_1,f_2,...,f_k\}$ – множество функций преобразования информации, включающих условные и

циклические преобразования, а k – число операторов в теле кадра, то правило однократного присваивания можно описать следующим образом:

$$a_i \neq a_j, \text{ если } i \neq j. \quad (2.2)$$

Правило единственной подстановки [35] запрещает одновременные запись и чтение в память несколькими процессами. Математическое описание правила единственной подстановки можно представить следующим образом:

$$A \bigcap B = \varnothing. \quad (2.3)$$

Кроме того, даже одновременное чтение одной и той же памяти разными процессами является синтаксической ошибкой, поскольку для доступа к ячейкам памяти используется только один адресный вход, поэтому одновременная адресация к памяти несколькими процессами является невозможной. Однако если существует функция, позволяющая свести адресные генераторы всех процессов, выполняющих однотипный доступ (чтение или запись) к памяти, к одному общему генератору, то такое обращение к памяти является корректным.

На примере программы и эквивалентной ей граф-схемы с общим генератором G_k показан доступ двух независимых процессов к мемориальной переменной B (рис. 2.1).

```
Var a, b, c : Array Integer
[10:Stream] Mem;
Var i : Number;
Const n = 9;
Cadr summa;
    For i:=0 to n do
        a[i]:=F1(b[i]);
    For j:=0 to n do
        c[j]:= F2(b[j]);
Endcadr;
```

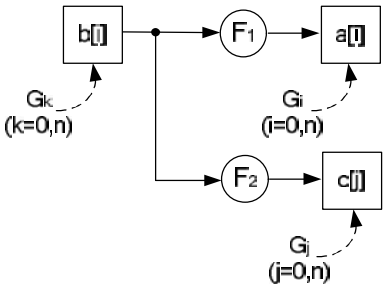


Рис. 2.1. Программа и эквивалентная ей граф-схема с общим генератором G_k

На первый взгляд, обращение к переменной B является некорректным, но поскольку адресные генераторы G_i и G_j на

рис. 2.1 соответствуют операторам цикла с индексными переменными I и J, отличающимися только именами переменных, то данные генераторы могут быть сведены к одному общему генератору G_k. Таким образом, программа является синтаксически правильной.

В большинстве случаев генераторы адресов отличаются друг от друга, поэтому сведение их к одному генератору должно компенсироваться за счет выполнения соответствующих операций на уровне информационного потока данных.

На рис. 2.2 представлены программа и эквивалентная ей граф-схема с блоком временной задержки.

```
Var a, b, c, d : Array Integer
[10:Stream] Mem;
Var i : Number;
Const n = 9;
Cadr summa;
    For i:=0 to n do
        a[i]:=F1(b[i]);
    For j:=0 to n do
        c[i]:= F2(b[j-5]);
    For k:=0 to n do
        d[k]:= F3(b[k-8]);
Endcadr;
```

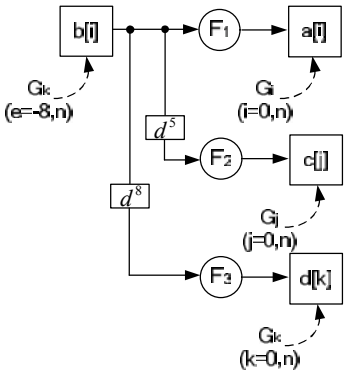


Рис. 2.2. Программа и эквивалентная ей граф-схема с блоком временной задержки

В программе адреса доступа к элементам массива B отличаются только смещением на константу. В этом случае адресные генераторы, соответствующие элементам b_i, b_{i-5} и b_{i-8}, могут быть сведены к одному генератору, а смещение элементов может быть реализовано на уровне информационных потоков данных. Поскольку элементы b_i, b_{i-5} и b_{i-8} должны быть поданы в вычислительную структуру одновременно, то смещение между ними реализуется при помощи временных

задержек. На рис. 2.2 d^5 и d^8 – блоки задержки соответственно на 5 и 8 отсчетов.

На рис. 2.3 представлены программа и эквивалентная ей граф-схема с блоком временной задержки.

```
Var a, b, c : Array Integer
[10:Stream] Mem;
Var i : Number;
Const N = 9;
Cadr summa;
    For i:=0 to N step 1 do
        a[i]:=b[i];
    For j:=0 to N step 2 do
        c[j]:=b[j];
Endcadr;
```

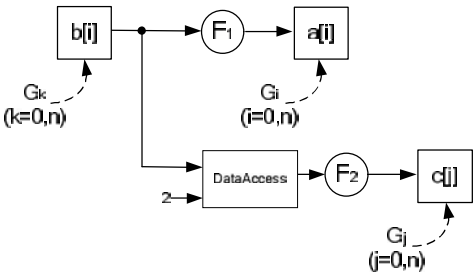


Рис. 2.3. Программа и эквивалентная ей граф-схема с блоком временной задержки

На примере программы на рис. 2.3 показан доступ двух независимых процессов к мемориальной переменной B, отличающихся только шагом доступа к элементам. Адресные генераторы, соответствующие элементам b_i и b_j, могут быть сведены к одному генератору, а выборка четных элементов из потока b_j осуществляется специализированным блоком (DataAccess).

Помимо возможности доступа к внешней памяти в языке существует возможность доступа к внутренней памяти ПЛИС. Для доступа к внутренней памяти ПЛИС в языке COLAMO используется тип хранения переменных InterMem. Отличительной особенностью внутренней памяти ПЛИС от внешней памяти является наличие одного или более портов, обеспечивающих возможность доступа к памяти нескольким независимым процессам чтения и записи. Каждый порт независимо друг от друга позволяет выполнять как процесс чтения, так и процесс записи, но только один в любой момент времени. Таким образом, внутренняя память в зависимости от количества портов позволяет получить одновременный доступ к ней нескольким процессам. В случае если количество

независимых процессов, обращающихся к внутренней памяти, больше, чем количество ее портов, то возникает нарушение правила однократного присваивания.

Для каждого порта, настроенного на чтение или запись, используется отдельный адресный генератор. Адресные генераторы внутренней памяти, соответствующие процессам чтения, могут быть сведены к одному генератору в отличие от процессов записи, поскольку в этом случае будет нарушено правило однократного присваивания.

При любом обращении к внутренней памяти будет задействован ее свободный порт независимо от того, какой процесс выполняется: чтение или запись. Так, в программе (2.4) переменная A представляет собой четырехпортовую внутреннюю память, поскольку над ней выполняются два процесса чтения по следующим адресам ($i-3$ и $i+10$) и два процесса записи по следующим адресам (i и $i-5$). С точки зрения генераторов, для организации потоков данных для внутренней памяти, рассмотренной в программе (2.6), необходимо использовать четыре независимых генератора по одному на каждый порт внутренней памяти.

```
Var a,b,c,e : Array Real [100 : Stream] Mem;
Var d : Array Real [100 : Stream] InterMem4;
Cadr ExampleInterMem4;
  For I := 0 to 30 do
    Begin
      a[i] := F1(a[i-3]);
      a[i-5] := F2(a[i+10]);
Endcadr;                                                    (2.4)
```

2.2. Сцепление переменных

В процедурных языках программирования присваивание элементов выглядит следующим образом:

$$A[i]:=B[i], \qquad (2.5)$$

где A и B – массивы, расположенные в памяти. Данный фрагмент программы четко определяет зависимость между элементами массивов A и B. Данная запись справедлива и для

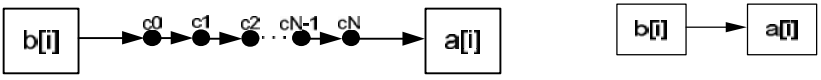
языка COLAMO. Однако в языке COLAMO существует возможность реализовать данный фрагмент программы неявным образом при помощи использования промежуточных коммутационных переменных. Использование промежуточных коммутационных переменных показано на примере программы (2.6).

```
Var a,b : Array Real [100 : Stream] Mem;
Var C0, C1, C2,..., CN : Real Com;
Var I : Number;
Cadr ExampleF;
  For i := 0 to 30 do
    Begin
      c0:=b[i];
      c1:= c0;
      c2:= c1;
          .
          .
          .
      cN:=cN-1;
      a[i] := cN;
    End;
Endcadr;                                                    (2.6)
```

где N – любое значение от 0 до бесконечности.

Программа (2.5) и программа (2.6) являются эквивалентными, поскольку коммутационные переменные C_i, при $i = \overline{0,n}$ осуществляют топологическую цепочку между мемориальными переменными (информационными вершинами) A[i] и B[i] и не приводят к затрате дополнительного оборудования. На рис. 2.4 представлены информационные графы программ (2.6) и (2.5).

Из информационных графов на рис. 2.4 видно, что данные графы эквивалентны друг другу.



а) информационный граф программы (2.6)

б) информационный граф программы (2.5)

Рис. 2.4. Информационные графы программ

Такие отношения мемориальных переменных будем называть сцеплением по аналогии с программами на языке Пролог [36, 37]. Таким образом, при программировании на языке COLAMO необходимо отслеживать информационную зависимость между мемориальными переменными не только в операторах присваивания, но и во всех информационных цепочках в пределах кадра.

Кроме того, помимо коммутационных переменных могут быть использованы коммутационные массивы. Векторные коммутационные массивы позволяют получить одновременный доступ к параллельно существующим элементам, а потоковый коммутационный массив позволяет получать элементы во времени.

В качестве примера рассмотрим программу, в которой необходимо выполнить над элементами b_i, b_{i-5} и b_{i-8} некоторое функциональное преобразование F.

```
Var a,b : Array Real [100 : Stream] Mem;
Var I : Number;
Cadr ExampleF;
  For i := 0 to 30 do
    Begin
      a[i] := F (b[i], b[i-5], b[i-8]);
Endcadr;
```
(2.7)

Поскольку массив B описан как мемориальный, то для доступа к элементам b_i, b_{i-5} и b_{i-8} будут сформированы три процесса чтения памяти, которые необходимо свести к одному адресному генератору для корректного доступа к памяти.

Однако данную программу можно реализовать при помощи коммутационного массива, что позволит задействовать только один процесс чтения мемориальной переменной B. В этом случае программа будет выглядеть следующим образом:

```
Var a,b : Array Real [100 : Stream] Mem;
Var I : Number;
Cadr ExampleF;
  For i := 0 to n do
    Begin
```

```
        c[i] := b[i];
        a[i] := F (c[i], c[i-5], c[i-8]);
Endcadr;                                                    (2.8)
```

На рис. 2.5 представлены программа и эквивалентная ей граф-схема с использованием коммутационного массива, где d^5 и d^8 – блоки задержки соответственно

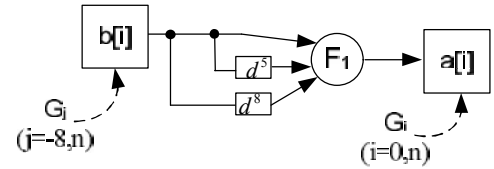


Рис. 2.5. Граф-схема программы (2.8)

Граф-схема (2.5) будет соответствовать как программе (2.7), так и программе (2.8) на 5 и 8 отсчетов. Программа (2.8) не требует от транслятора дополнительных преобразований для объединения адресных генераторов, как в случае с программой (2.7), поскольку изначально в ней используется только один адресный генератор для чтения памяти. Использование различных коммутационных массивов в программе позволяет оптимизировать синтезируемую вычислительную структуру за счет уменьшения числа регистров. На рис. 2.6 показана модифицированная программа с использованием различных потоковых коммутационных массивов и соответствующая ей эквивалентная граф-схема.

```
Var a,b : Array Real [100 :
Stream] Mem;
Var I : Number;
Cadr Example;
  For i := 0 to n do
    Begin
      c[i] := b[i];
      d[i] := c[i-5];
      e[i] := d[i-3];
      a[i] := F (c[i], d[i], e[i]);
Endcadr;
```

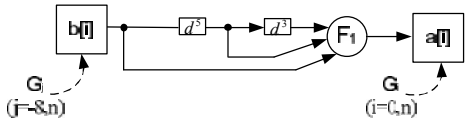


Рис. 2.6. Программа и эквивалентная ей граф-схема

Суммарное количество временных задержек в программе на рис. 2.6 сократилось за счет уменьшения числа регистров, d^5 и d^3 – блоки задержки соответственно на 5 и 3 отсчета.

Использование коммутационных переменных позволяет установить не только информационную зависимость между элементами информационных потоков данных, но и в отличие от традиционных языков программирования позволяет организовать информационные потоки по разным адресам. Использование коммутационных переменных позволяет расширить функции обработки информационных потоков данных, которые невозможно реализовать в традиционных языках программирования.

В программе на рис. 2.7 коммутационная переменная *C* связывает между собой мемориальные переменные *A* и *B*.

```
Var a : Array Real [100 : Stream; 100 : Stream] Mem;
Var b : Array Real [100 : Stream; 100 : Stream] Mem;
Var i, j, k : Number;
Cadr Example;
  For i := 0 to n do
    For j := 0 to m do
      For k := 0 to p step 2 do
          c := F1(b[i,j,k]);

  For i := 1 to r do
    For j := 0 to L step 3 do
        a[i,j] := c;
Endcadr;
```

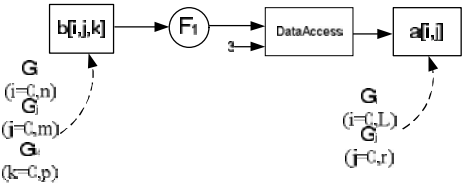


Рис. 2.7. Программа и эквивалентная ей граф-схема

Для организации информационных потоков для переменных *A* и *B* используется разное количество адресных генераторов, при этом адресные генераторы потока *A* полностью отличаются от адресных генераторов потока *B*. Однако

информационные потоки *A* и *B* сцеплены между собой коммутационной переменной *C*, соответственно каждое значение элемента *b*[i,j,k] будет записано в соответствующий элемент *a*[i,j]. Для процедурных языков данная программа является бессмысленной, поскольку во все элементы *a*[i,j] будет записано последнее значение переменной *C*. Более того, такая организация информационных потоков в процедурных языках программирования является практически невозможной. Использование коммутационных переменных позволяет выполнять повторение циклических потоков.

2.3. Параллельная и конвейерная обработка данных

На основании рассмотренных правил доступа к памяти в языке COLAMO организуется параллельный и последовательной доступ. Параллельный или последовательный доступ к элементам массива в языке COLAMO определяется ключевыми словами Vector или Stream при объявлении массива [29, 35]. Тип доступа Vector указывает на необходимость размещения элементов массива в разных каналах памяти, доступ к которым может выполняться параллельно, а тип Stream указывает на то, что элементы массива располагаются в одном канале памяти, доступ к которым выполняется последовательно. Последовательность описания векторных и потоковых составляющих массива не регламентирована.

Массив в языке COLAMO может быть определен следующим образом: $M = V \bigcup S$, где V – подмножество параметров, для которых определен параллельный тип доступа, а S – подмножество параметров, для которых определен последовательный тип доступа.

В этом случае степень максимального распараллеливания для переменной будет определяться по формуле

$$\Lambda = \prod_{i=0}^{n-1} v_i \,, \qquad (2.9)$$

где $v_i \in V$, а n – число параметров, для которых определен параллельный тип доступа.

Для скалярных переменных и массивов, имеющих только последовательный тип доступа, $\Lambda = 1$.

Рассмотрим две программы, отличающиеся между собой только способом обработки массивов.

На рис. 2.11 представлены программа и эквивалентная ей граф-схема.

```
Var a, b : Array Integer [10:Stream] Mem;
Var i : Number;
Const N = 9;
Cadr summa;
    For i:=0 to N do
        a[i]:=b[i];
Endcadr;
```

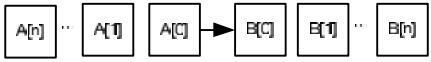


Рис. 2.8. Программа и эквивалентный ей информационный граф конвейерной обработки данных

В программе массивы *A* и *B* имеют последовательный тип доступа, что указывает на конвейерную обработку данных, следовательно, распараллеливание выражения a[i]:=b[i]; является невозможным. Соответственно элементы массива *B* будут последовательно записаны в массив *A*. Конвейерная организация вычислений может быть представлена в виде последовательности (см. рис. 2.8), где каждый элемент соответствует определенному моменту времени.

Здесь и далее информационные вершины графа (мемориальные и регистровые переменные) будем обозначать квадратами.

Программа (2.11) практически ничем не отличается от программы на рис.2.11, кроме способа обработки массивов.

```
Var a,b : Array Integer [10: Vector] Mem;
Var i : Number;
Const n = 5;
Cadr summa;
    For i:= 0 to n do
        a[i]:=b[i];
Endcadr;
```
(2.10)

Поскольку все элементы массивов a и b определены как Vector, то для данных массивов неявно указан параллельный способ обработки. В этом случае базовый подграф, представленный на рис. 2.9, будет мультиплицирован.

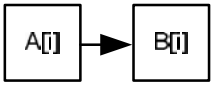


Рис. 2.9. Базовый подграф

Согласно формуле (2.10) максимально возможная степень распараллеливания по данным базового подграфа в этом случае равна 10, а реальная степень распараллеливания определяется оператором цикла и равна 5, что отражено на информационном графе данной задачи (рис. 2.10).

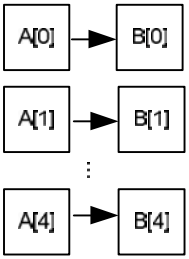


Рис. 2.10. Информационный граф задачи со степенью распараллеливания, равной 5

В синтезированном информационном графе, соответствующем программе (2.10), используются базовые подграфы (см. рис. 2.9), каждый из которых независимо друг от друга записывает элемент из массива *B* в массив *A*.

Следует отметить, что степень параллелизма распространяется не только на элементы массивов, но и на функции, выполняющие их обработку.

Программа на рис. 2.11 демонстрирует вычисление функции *F* и запись полученного результата в массив *C*. Поскольку переменные *A*, *B* и *C* имеют параллельный доступ к элементам, то распараллеливаться будут не только элементы массивов A, B и C, но и функция *F*.

На рис. 2.11 представлены программа и эквивалентный ей информационный граф распараллеливания функции F со степенью N.

```
Var a, b, c : Array Integer [10:Vector] Mem;
Var I : Number;
Const n = 9;
Cadr summa;
    For i:=0 To n Do
        c[i] := F(a[i], b[i]);
Endcadr;
```

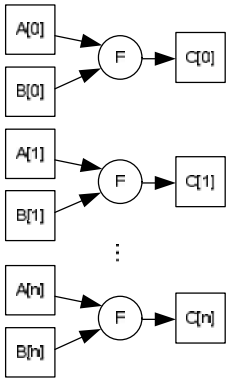


Рис. 2.11. Программа и эквивалентный ей информационный граф распараллеливания функции F со степенью N

Рассмотренные программы являются граничными примерами извлечения параллелизма, что на практике встречается очень редко, поскольку в большинстве случаев используется параллельно-последовательная обработка массивов. Использование массивов, отличающихся типами доступа в едином вычислительном фрагменте, может привести к несбалансированной вычислительной схеме, что приведет к дополнительной затрате оборудования и снижению скорости информационных потоков.

Для определения потенциальной степени распараллеливания вычислительного фрагмента введем множество $\mathrm{K} = \{\lambda_1, \lambda_2, ..., \lambda_m\}$, где λ_j - максимальная степень распараллеливания j-й переменной во фрагменте, определяемая формулой (2.9), m – количество переменных, принадлежащих вычислительному фрагменту.

Формулу для определения потенциальной степени распараллеливания вычислительного фрагмента Λ^* можно представить следующим образом:

$$\Lambda^* = \underset{j=1}{\overset{m}{Min}}(\lambda_j). \qquad (2.11)$$

Рассмотрим различные фрагменты вычислений и их синтезированные схемы в зависимости от типов переменных (скаляр или массив) и способов их обработки.

Рассмотрим программу, представленную на рис. 2.12.

```
Var a, b : Array Integer [10:Vector] Mem;
Var c : Array Integer [10:Stream] Mem;
Var I : Number;
Const k = 9;
Cadr summa;
    For i:=0 To k Do
        c[i] :=a[i];
Endcadr;
```

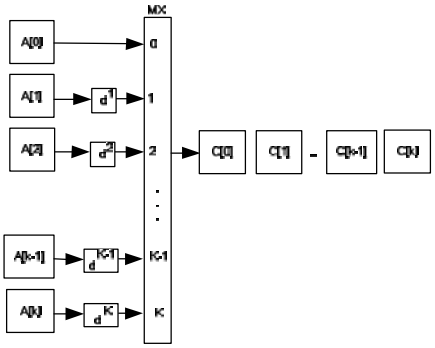


Рис. 2.12. Программа и эквивалентный ей информационный граф распараллеливания функции F со степенью N

В программе на рис. 2.12 выполняется присваивание элементов массива A, доступ к которым осуществляется параллельно массиву C, доступ к которому может быть осуществлен только последовательно, соответственно для корректной обработки данных необходимо выполнить согласование потоков данных. Для согласования потоков данных необходимо использовать блок согласования, включающий в себя мультиплексор (MX) и блоки временных задержек (d^i – блок задержки на i отсчетов).

Следует отметить, что в общем случае задержки элементов вектора могут носить случайный характер, при этом они не должны повторяться.

Если в программе выполняется смешанная обработка массивов такая, что некоторая функция F выполняет обработку векторных массивов, а полученные результаты записываются в

потоковый массив (см. программу (2.12)), то информационный граф программы может быть представлен в виде распараллеливания подграфа, реализующего функцию *F*, или в виде распараллеливания блоков согласования потоков данных.

```
Var a, b : Array Integer [10:Vector] Mem;
Var c : Array Integer [10:Stream] Mem;
Var I : Number;
Const n = 9;
Cadr summa;
    For i:=0 To n Do
        c[i] := F(a[i], b[i]);
Endcadr;
```
(2.12)

Ситуация в программе (2.12) не изменится, если в текст программы будет введена промежуточная коммутационная переменная, осуществляющая неявное присваивание результата функции *F* элементам массива C. Введение промежуточной коммутационной переменной показано в программе (2.15).

```
Var a, b : Array Integer [10:Vector] Mem;
Var e : Array Integer [10:Vector] Com;
Var c : Array Integer [10:Stream] Mem;
Var I : Number;
Const n = 9;
Cadr summa;
    For i:=0 To n Do
        Begin
            e[i] := F(a[i], b[i]);
            c[i] := e[i];
         end;
Endcadr;
```
(2.13)

Поскольку массивы E, A и B являются векторными, то согласно формуле (2.11) оператор присваивания e[i]:=F(a[i], b[i]); будет имеет степень распараллеливания, равную *N*, соответственно функция *F* будет мультиплицирована. Для реализации оператора присваивания c[i]:=e[i]; требуются мультиплексор и соответствующие блоки временных задержек. Поскольку коммутационная переменная в информационном графе является точкой и не требует аппаратного ресурса для ее

реализации, то информационный граф на рис. 2.6 будет соответствовать как программе (2.12), так и программе (2.13).

Таким образом, в независимости от явного или неявного использования коммутационных переменных распараллеливанию подвергается любой вычислительный фрагмент, оперирующий с векторными массивами (рис.2.13).

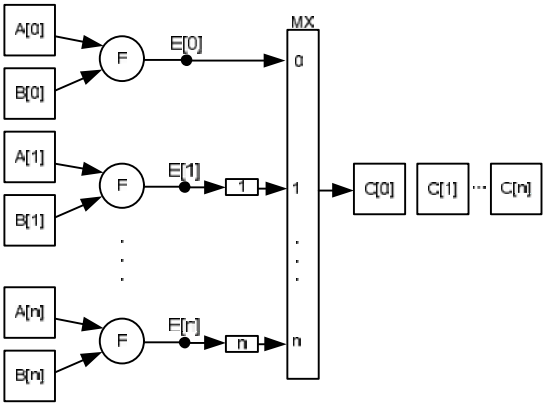


Рис. 2.13. Распараллеливание функции *F*

Если распараллеливание функции *F* на структурном уровне является невозможным (например, в силу ограниченного аппаратного ресурса РВС), то возможно распараллеливание не функции *F*, а блоков согласования потоков данных. Примером такой параллельной программы является программа на рис. 2.14.

Для согласования потоков данных между векторным массивом A и потоковым массивом E1, а также массивами B и E2 необходимо использовать блоки согласования потоков. Поскольку аргументы функции *F* (массивы e1[i] и e2[i]) и результирующий массив C являются потоковыми, то распараллеливание функции *F* не выполняется.

Программа (рис. 2.15) демонстрирует обратное преобразование потокового массива B в векторный массив A. При обратном преобразовании вместо мультиплексора необходимо использовать демультиплексор, а временные задержки на выходах демультиплексора должны быть

перевернуты, т.е. выходам 0,1,…,n должны соответствовать временные задержки n-1,n-2,…,0.

```
Var a, b : Array Integer
[10:Vector] Mem;
Var c, e1, e2 : Array Integer
[10:Stream] Mem;
Var I : Number;
Const n = 9;
Cadr summa;
    For i:=0 To n Do
        Begin
            e1[i] :=a[i];
            e2[i] :=b[i];
            c[i] := F(e1[i], e2[i]);
        end;
Endcadr;
```

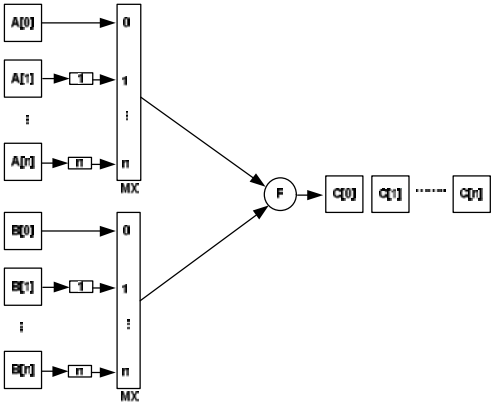


Рис. 2.14. Программа и эквивалентный ей информационный граф распараллеливания блока согласования

Если элементы массива A поступают на вход демультиплексора последовательно, а все связи являются асинхронными, то использование временных задержек является необязательным.

```
Var a,b : Array Integer [10: Stream]
Mem;
Var c : Array Integer [10: Vector]
Mem;
Var i : Number;
Const n = 9;
Cadr summa;
    For i:=0 to n do
        a[i] := b[i];
Endcadr;
```

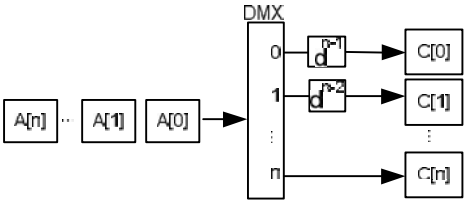


Рис. 2.15. Программа и пример информационного графа программы с демультиплексором

Следует также отметить, что при преобразовании векторного массива к потоковому массиву происходит

понижение степени распараллеливания, что приводит к падению удельной производительности вычислительной системы.

В рассмотренных программах демонстрировалось отличие способов обработки массивов только между результатом и аргументами, т.е. результирующий массив имел параллельный способ обработки, а массивы, использующиеся в качестве аргументов, имели последовательный способ обработки или наоборот.

```
Var a : Array Integer [10 : Vector] Mem;
Var b, c: Array Integer [10 : Stream] Mem;
Var i : Number;
Const N = 9;
Cadr summa;
    For i:= 0 To n Do
        c[i]:= F(a[i], b[i]);
Endcadr;
```
(2.14)

В программе (2.14) используется три массива, причем результирующий массив С и массив В, использующийся в качестве аргумента в операторе присваивания, имеют последовательный способ обработки, а массив А – параллельный. Для согласования потоков данных во фрагментах программы возможно использование векторных и потоковых коммутационных переменных. Если способ обработки массива не соответствует способу обработки фрагмента программы, в котором он используется, то для согласования потоков данных в данном фрагменте необходимо использование коммутационных массивов. В данном случае коммутационная переменная должна быть сцеплена с переменной, способ обработки которой не соответствует способу обработки данного фрагмента, при этом способы обработки данных переменных должны быть противоположными. Данное правило может быть применено к любому массиву в независимости от того, какой процесс над ним выполняется. Для согласования потоков данных при обработке функции F в программе (2.14) необходимо вместо векторного массива А подать на вход данной функции потоковый коммутационный массив D, при этом массивы А и D должны быть сцеплены между собой. Программа (рис. 2.16)

представляет собой два оператора присваивания, выполнение одного из которых является полностью последовательным, поскольку массивы B, C и D являются потоковыми, а другой требует согласования потоков, поскольку массив A является векторным, а массив D – потоковым.

Информационный граф программы, представленный на рисунке 2.16, будет включать в себя мультиплексор, блоки временных задержек и реализацию одной функции F.

```
Var a : Array Integer [10 :
Vector] Mem;
Var b, c, d: Array Integer
[10 : Stream] Mem;
Var i : Number;
Const N = 9;
Cadr summa;
    For i:= 0 To n Do
       Begin
          d[i]:= a[i];
          c[i]:= F(d[i], b[i]);
       End;
Endcadr;
```

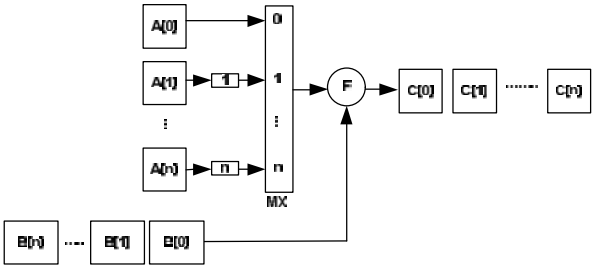


Рис. 2.16. Программа и эквивалентный ей информационный граф

В программе (рис. 2.17), согласно формуле (2.11), степень распараллеливания равна единице, соответственно обработка векторных массивов A и C должна осуществляться с использованием потоковых коммутационных переменных.

```
Var a, c : Array Integer [10 :
Vector] Mem;
Var b: Array Integer [10 :
Stream] Mem;
Var i : Number;
Const N = 9;
Cadr summa;
    For i:= 0 To n Do
       c[i]:= F(a[i], b[i]);
Endcadr;
```

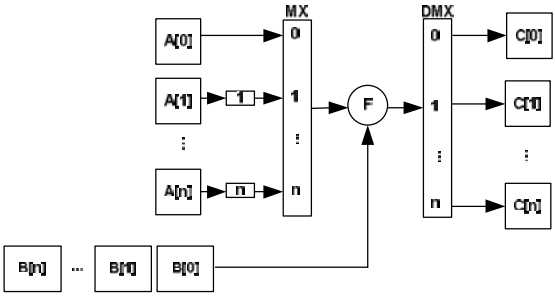
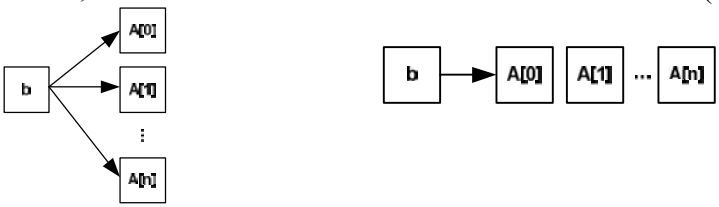


Рис. 2.17. Программа и эквивалентный ей информационный граф

Особым случаем является работа со скаляром. Следует отметить, что скалярная переменная не влияет на определение степени распараллеливания фрагмента вычислений.

Поскольку структура информационного графа зависит от способа обработки элементов массивов, то для фрагмента программы (2.15) нельзя однозначно сказать, какой информационный граф а) или б) на рис. 2.25 будет ей соответствовать. В случае если массив A в программе (2.15) является векторным, то синтезированный информационный граф будет соответствовать графу на рис. 2.18-а. Если массив A является потоковым, то синтезированный информационный граф будет соответствовать графу на рис. 2.18-б.

```
Cadr Example;
  For i := 0 to n do
    a[i] := b;
Endcadr;                                                    (2.15)
```



а) б)

Рис. 2.18. Возможные информационные графы программы

Как видно из информационных графов на рис. 2.18, присваивание векторному или потоковому массиву скалярной переменной является корректным и не требует никакого дополнительного оборудования.

Однако обратное присваивание корректно только в случае работы с потоковым массивом. В этом случае скалярная переменная будет иметь значение, соответствующее последнему присвоенному элементу потокового массива. Присваивание векторного массива скалярной переменой является синтаксически неверным (см. рис. 2.19), поскольку нарушается правило однократного присваивания языка COLAMO, т.к.

запись всех элементов векторного массива будет выполняться одновременно в одну и ту же скалярную переменную.

```
Var b: Integer Mem; - скаляр;
Var a: array Integer [5: Vector] Mem;
Var i : Number;
Const n = 4;
Cadr Example;
  For I := 0 to n do
    Begin
       b:=a[i];
    End;
Endcadr;
```

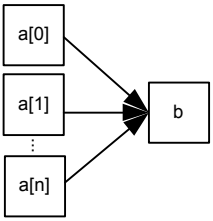


Рис. 2.19. Программа и эквивалентный ей информационный граф

Из информационного графа видно, что для корректного присваивания элементов векторного массива A скалярной переменной B необходимо выполнить последовательное присваивание всех элементов векторного массива A скаляру B. Для этого необходимо использование промежуточного потокового коммутационного массива между скалярной переменной B и векторным массивом A. Введем потоковый коммутационный массив C, элементы которого будут присваиваться скалярной переменной B, а элементы векторного массива A будут присваиваться элементам потокового массива C, как это показано на рис. 2.20.

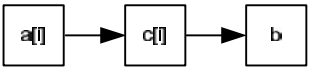


Рис 2.20. Информационный граф

Запись значений элементов потокового массива C в скалярную переменную является корректным, поскольку не нарушает принципов языка COLAMO и не требует дополнительного аппаратного ресурса. Для присваивания элементов векторного массива A элементам потокового массива C необходимо выполнить согласование потоков данных. Для этого необходимо использовать мультиплексор и блоки

задержек. В этом случае информационный граф будет иметь вид, представленный на рис.2.21.

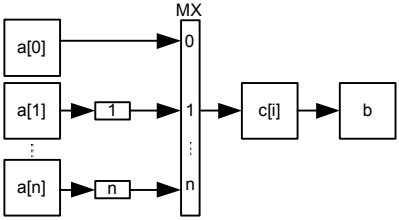


Рис. 2.21. Информационный граф с согласованными потоками данных

Таким образом, для присваивания скалярной переменной элементов векторного массива необходимо использовать мультиплексор и блоки временных задержек.

Программа (2.18) демонстрирует ситуацию неявного присваивания элементов векторного массива скалярной переменной.

```
Var b: Integer Mem;
Var a: array Integer [5: Vector] Mem;
Var d: array Integer [5: Stream] Mem;
Var i : Number;
Const n = 4;
Cadr Example;
  For I := 0 to n do
     Begin
        d[i]:=a[i];
        b:=d[i];
     End;
Endcadr;
```
(2.16)

Программа (2.16) является эквивалентной программе (рис. 2.19).

Рассмотрим три параллельные программы, демонстрирующие различные варианты взаимодействия скаляра, векторного и потокового массивов.

В программе (рис. 2.22) в качестве аргументов функции F используются скалярная переменная B и потоковый массив A, что приводит к последовательной обработке всех элементов

массива A и скаляра B функцией F. Поскольку результирующий массив C является векторным, а обработка данных функции F является последовательной, то необходимо выполнить согласование потоков данных.

```
Var b: Integer Mem;
Var a: array Integer [5: Stream]
Mem;
Var c: array Integer [5: Vector]
Mem;
Var i : Number;
Const n = 4;
Cadr Example;
  For i:=0 to n do
     c[i] := F(a[i], b);
Endcadr;
```

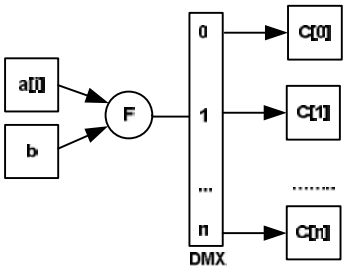


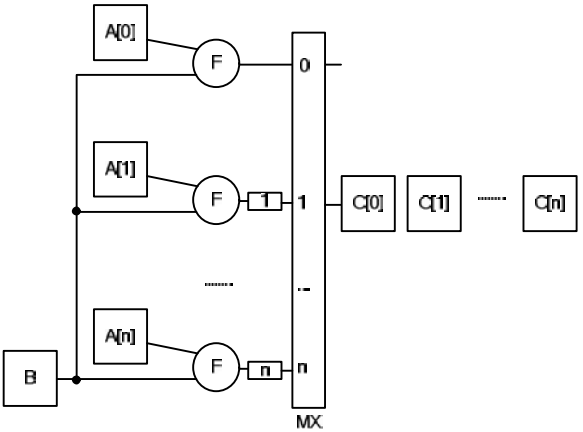
Рис. 2.22. Информационный граф программы

В программе, представленной ниже, функция обрабатывает элементы векторного массива A и скаляр B. Исходя из формулы (2.11), функция F и ее аргументы могут быть распараллелены. Однако массив C является потоковым массивом, а это означает, что степень распараллеливания оператора присваивания равна единице. В этом случае информационный граф программы (2.17) может быть реализован в двух вариантах, рассмотренных на рисунке 2.23.

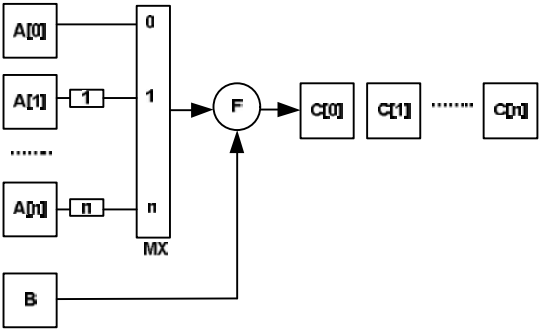
```
Var b: Integer Mem;
Var a: array Integer [5: vector] Mem;
Var c: array Integer [5: stream] Mem;
Var i : Number;
Const n = 4;
Cadr Example;
  For I := 0 to n do
       c[i] := F(a[i], b);
Endcadr;                                                    (2.17)
```

На рис. 2.23-а представлен информационный граф, построенный по принципу распараллеливания вычислительной структуры функции F и последовательной записи полученных результатов в потоковый массив C. На рис. 2.23-б представлен информационный граф, в котором вычислительная структура

функции F реализована один раз, а векторный массив A обрабатывается последовательно, таким образом, построенный информационный граф является наиболее оптимальным по сравнению с информационным графом на рис. 2.23-а.



а)



б)

Рис. 2.23. Пример информационного графа программы

В программе (2.18) выполняются обработка потокового и векторного массивов функцией F и последовательная запись полученных результатов в скалярную переменную B.

```
Var b: Integer Mem;
Var a: array Integer [5: vector] Mem;
Var c: array Integer [5: stream] Mem;
Var i: Number;
Const n=4;
```

```
Cadr Example;
  For i := 0 to n to
      b := F(a[i], c[i]);
Endcadr;
```
(2.18)

Степень распараллеливания программы (2.18) равна единице, поскольку массив C является последовательным, соответственно запись результатов функции F в скалярную переменную B корректна. Программа (2.18) является бессмысленной, поскольку значение переменной B будет равно результату функции F последней итерации цикла.

2.4. Контрольные вопросы

1. Каким образом предотвращается некорректный доступ к памяти в языке COLAMO?
2. Перечислите основные способы доступа к элементам, обрабатываемым в потоке.
3. Как вычисляется потенциальная степень распараллеливания фрагмента программы?
4. Каким образом осуществляется согласование потоков данных?

3. ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ КОНСТРУКЦИЙ ЯЗЫКА COLAMO

3.1. Особенности использования условных операторов

В языке COLAMO условный оператор синтаксически аналогичен данному оператору традиционных языков программирования. В общем виде условный оператор можно представить следующим образом:

If α Then P
Else Q,

где α - логическое выражение, P и Q – множество альтернативных операторов.

Операторы множества P выполняются в случае истины логического выражения α, в противном случае будут выполняться операторы множества Q. Альтернативная ветка Else условного оператора может отсутствовать, в этом случае условный оператор является неполным.

Условные операторы, в зависимости от места их использования, могут быть реализованы на процедурном или структурном уровне. Если условный оператор является внешним по отношению к кадру, то он реализуется процедурно, в противном случае – структурно.

Структурная реализация условного оператора в схемотехнике представляет собой мультиплексор для выбора одной из альтернативных веток условного оператора и блок управления мультиплексором, реализующий условие выбора одной из альтернативных веток. При реализации условных операторов в структурном компоненте для увеличения быстродействия аппаратно реализуются все альтернативные ветви. Однако если условный оператор содержит множество вложенных условных операторов, то будет выполняться только одна ветвь из множества возможных, причем аппаратно будут реализованы все альтернативные ветви. Таким образом, использование условных операторов, а тем более вложенных условных операторов, приводит к затрате оборудования,

которое в большинстве случаев будет простаивать. В этом случае условные операторы целесообразно реализовать не структурно, а процедурно.

Опишем условный оператор в виде оператора присваивания. Предположим, что множества P и Q представляют собой список операторов присваивания, представленных следующим образом:

$$\text{P}=\{\ p_1 = F_1(c_1^1, c_2^1, ..., c_k^1)\ ,$$

$$p_2 = F_2(c_1^2, c_2^2, ..., c_k^2), \ldots, p_n = F_n(c_1^n, c_2^n, ..., c_m^n)\ \}\ \text{и}$$

$$\text{Q}=\{\ q_1 = G_1(d_1^1, d_2^1, ..., d_k^1)\ , q_2 = G_2(d_1^2, d_2^2, ..., d_k^2)\ , \ldots,$$

$$q_n = G_n(d_1^n, d_2^n, ..., d_k^n)\ \}.$$

Если существуют $p_i \in P$ и $q_j \in Q$ такие, что $r(p_i) = r(q_j)$, где $r(x)$ - результат оператора присваивания x, то условный оператор можно представить следующим образом:

$$r(p_i) = \alpha\ \&\ F_{\blacklozenge}(c_1^i, c_2^i, ..., c_m^i) \vee \overline{\alpha}\ \&\ G_{\blacklozenge}(d_1^j, d_2^j, ..., d_k^j)\ . \quad (3.1)$$

Здесь дизъюнкция соответствует ключевому слову if, конъюнкция – ключевому слову then, а ключевое слово else – конъюнкции с инверсией логического выражения.

Для реализации условного оператора используются мультиплексор (MX) и блок управления (U). Мультиплексор реализует выбор альтернативной ветки условного оператора в зависимости от результата, получаемого на выходе блока управления. Блок управления реализует логическое выражение условного оператора. Если логическое выражение является истиной, то на выходе блока U будет установлено значение “1”, в противном случае - “0”.

Условный оператор это такая же конструкция с некоторой функциональной зависимостью, как и предыдущие конструкции, и для нее действуют те же правила распараллеливания, т.е. если все конструкции условного оператора векторные, то условный оператор распараллеливается [38].

Рассмотрим различные случаи распараллеливания программы, содержащей условный оператор.

В программе (рис.3.1) все массивы являются потоковыми, поэтому степень распараллеливания как операторов присваивания, так и условного оператора, равна единице.

В программе (рис. 3.1) блок U выполняет вычисление логического выражения B[i] = 0, и если значение элемента B[i] равно нулю, то блок U выдаст значение, равное 1, в результате чего элементу A[i] будет присвоен результат, вычисленный функцией f_1, иначе функцией f_2.

В программе (рис.3.2) все массивы, относящиеся к условному оператору, - векторные, соответственно степень распараллеливания условного оператора равна N.

```
Var a, b: array Integer [100:
Stream] Mem;
Var i: Number;
Const n=99;
Cadr ExampleIfStream;
      For i:=0 to n do
        If b[i] = 0 then
          a[i] := f1(c[i]);
        Else
          a[i] := f2(d[i]);
Endcadr;
```

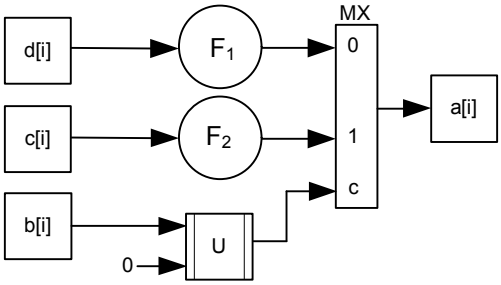


Рис. 3.1. Программа и эквивалентный ей информационный граф условного оператора со степенью распараллеливания, равной 1

Таким образом, информационный граф на рис. 3.2 будет представлять собой мультиплицирование информационного графа, рассмотренного на рис. 3.1, N раз.

В случае использования в условном операторе массивов с различным типом доступа степень распараллеливания логического выражения (Λ^1) и альтернативных веток оператора (Λ^2) вычисляется независимо друг от друга.

```
Var a, b, c, d: array Integer [10:
Vector] Mem;
Var i: Number;
Const n=9;
Cadr ExampleIfVector;
      For i:=0 to n do
        If b[i] = 0 then
           a[i] := f1(c[i]);
        Else
           a[i] := f2(d[i]);
Endcadr;
```

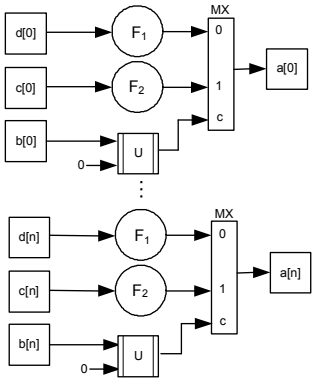


Рис. 3.2. Программа и эквивалентный ей информационный граф условного оператора со степенью распараллеливания, равной N

Если $\Lambda^1 < \Lambda^2$, то будет выполнено распараллеливание альтернативных веток условного оператора. В этом случае для согласования потоков данных между логическим выражением условного оператора и его альтернативными ветками необходимо на входы мультиплексоров, реализующих выбор одной из альтернативных веток, установить блоки временных задержек, значение которых определяется номером элемента массива, поступающего на конкретный вход мультиплексора.

Подобный случай представлен в программе (рис. 3.3), в которой все массивы в ветках условного оператора выполняются параллельно, а так как массивы A, C и D являются векторными, то степень распараллеливания альтернативных веток условного оператора $\Lambda^2 = 5$.

Степень распараллеливания логического выражения $\Lambda^1 = 1$, поскольку обработка логического выражения условного оператора выполняется последовательно, т.к. массив B является потоковым. Согласно программе (рис. 3.1), каждой паре элементов из векторных массивов C и D соответствует элемент из потокового массива B. Поскольку элементы векторных массивов C и D обрабатываются одновременно, а элементы массива B последовательно, то элементы массивов C и D должны быть задержаны на соответствующее количество

отсчетов. Так, элементы C[1] и D[1], соответствующие элементу B[1], должны быть задержаны на один отсчет, а элементы C[n] и D[n] - соответственно на n отсчетов.

```
Var a, c, d: array Integer [5: Vector] Mem;
Var b: array Integer [5: Stream] Mem;
Var i: Number;
Const n=4;
Cadr ExampleIfVectorStream;
      For i:=0 to n do
        If b[i] = 0 then
           a[i] := c[i];
        Else
           a[i] := d[i];
Endcadr;
```

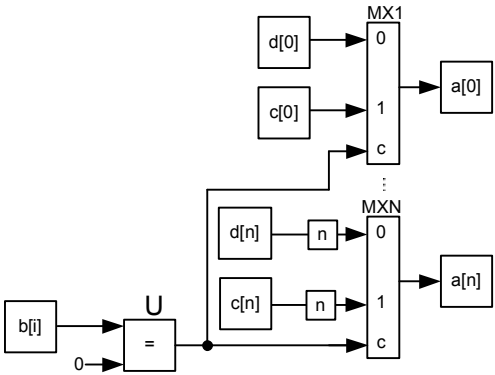


Рис. 3.3. Программа и эквивалентный ей информационный граф

Если $\Lambda^1 > \Lambda^2$, то будет выполнено распараллеливание логического выражения условного оператора. В этом случае для согласования потоков данных между логическим выражением и альтернативными ветками условного оператора необходимо использовать блоки согласования потоков данных для каждой векторной переменной в логическом выражении.

Подобная ситуация показана в программе (рис.3.4), поскольку в логическом выражении условного оператора используется векторный массив, а в ветках условного оператора - только массивы, имеющие последовательный тип доступа.

Поскольку степень распараллеливания альтернативных веток условного оператора равна единице, то для реализации выбора одной из веток условного оператора потребуются один мультиплексор и один блок управления. Поскольку массив B является векторным, а данные на вход блока управления должны подаваться последовательно, то необходимо использовать дополнительный мультиплексор и блоки временных задержек [39].

```
Var a, c, d : array Integer [5 :
Stream] Mem;
Var b : array Integer [5 :
Vector] Mem;
Var i, n : Number;
Define n=4;
Cadr Example;
    For i := 0 To n Do
       If b[i] = 0 then
          a[i] := c[i]
       Else
          a[i] := d[i];
Endcadr;
```

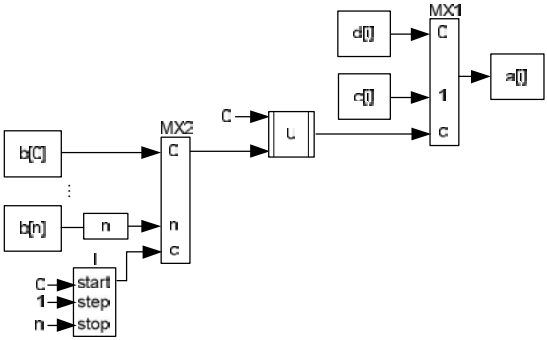


Рис. 3.4. Программа и эквивалентный ей информационный граф

Блок I в информационном графе на рис. 3.2 представляет собой аппаратную реализацию оператора цикла.

Если условный оператор является неполным, т.е. отсутствует альтернативная ветка Else, то формула (3.2) будет иметь следующий вид:

$$r(p_i) = \alpha \ \& \ F_{\blacklozenge}(c_1^i, c_2^i, \ldots, c_k^i) \vee \overline{\alpha} \ \& \ r(p_i), \qquad (3.2)$$

Таким образом, неполный условный оператор представляет собой скрытую рекурсию. Распараллеливание неполного условного оператора аналогично условному оператору с альтернативной веткой.

Структура информационного графа при работе с условными операторами зависит не только от способа обработки принадлежащих ему переменных, но и от способа хранения переменной, для которой выполняется условный оператор.

Для простоты рассмотрения особенностей взаимодействия условного оператора и переменных с различным способом хранения предположим, что условный оператор является полным, а множества A и B состоят из одного оператора присваивания. Таким образом, условный оператор будет иметь следующий вид:

$$\text{If } \alpha \text{ Then } a_i = F_1(c_1, c_2, \ldots, c_n) \text{ Else } a_j = F_2(d_1, d_2, \ldots, d_m). \qquad (3.3)$$

Рассмотрим случай, когда переменная *a* является **мемориальным** массивом. При работе с мемориальным

массивом обработка может выполняться не только на уровне данных, но и на уровне адресов. Поскольку все операторы в теле кадра выполняются параллельно, то результаты функций F_1 и F_2 могут быть получены одновременно, соответственно и адреса записи для массива A (I и J) должны быть получены также одновременно. Правило однократного присваивания в данном случае не нарушается, несмотря на то что над мемориальным массивом A выполняется два независимых процесса записи, поскольку за счет условного оператора в один момент времени будет выполняться только один процесс записи.

Реализация адресации к элементам мемориального массива может быть осуществлена как на структурном уровне (внешняя адресация), так и на процедурном уровне (операторами языка ARGUS). Таким образом, в зависимости от типа адресации к элементам мемориального массива информационный граф будет отличаться. Если адреса записи элементов в массив в обеих ветках условного оператора равны (т.е. $i = j$), то запись полученных результатов функциями F_1 и F_2 в независимости от логического выражения α всегда будет выполняться по одному и тому же адресу, таким образом, необходимость в реализации внешней адресации отсутствует. Соответственно информационный граф будет включать в себя один мультиплексор для выбора одной из альтернативных веток условного оператора и один блок управления, реализующий логическое выражение α.

Если адреса записи элементов в массив в обеих ветках условного оператора не равны (т.е. $i \neq j$) или используется косвенная адресация к массиву, то для адресации к массиву a необходимо использовать внешнюю адресацию, для организации которой используются блоки генерации адресов. В случае если $i \neq j$, то условный оператор используется не только для выбора одного из результатов, полученных в альтернативных ветках, но и для выбора одного из адресов записи для выбранного результата. Таким образом, информационный граф будет содержать два мультиплексора,

один из которых выполняет выбор данных, а другой - выбор адресов. Управление данными мультиплексорами осуществляется одним и тем же блоком управления.

Программа на рис. 3.5 демонстрирует использование условного оператора и мемориальной переменной, для которой в обеих ветках условного оператора совпадают адреса записи.

```
Var a, c, d : array Integer [5 : Stream]
Mem;
Var b : array Integer [5 : Stream] Mem;
Var i, n : Number;
Define n=4;
Cadr Example;
    For i := 0 To n Do
       If b[i] = 0 then
         a[i] := c[i]
       Else
         a[i] := d[i];
Endcadr;
```

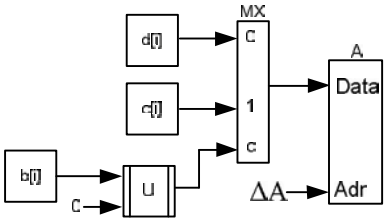


Рис. 3.5. Информационный граф программы

В информационном графе на рис. 3.5 на вход адреса КРП, соответствующего массиву A, подается значение $\Delta A = -1$, соответствующее «мусорной» ячейке в КРП.

Поскольку в программе (рис. 3.6) выполняется запись в массив A по разным адресам, то условный оператор действует не только на переменную A, но и на выбор ее адресов. Таким образом, информационный граф программы будет содержать мультиплексор (MX1) для выбора одного из результатов функций F и G и мультиплексор (MX2) для выбора адреса записи в массив A. В информационном графе мультиплексор MX2 подключен на адресный вход КРП A и осуществляет выбор поступающих на него адресов от блоков G_1 и G_2. Блок G_1 генерирует адреса записи в диапазоне от 0 до n-2, а блок G_2 - в диапазоне от 2 до n.

Условные операторы, рассмотренные в предыдущих программах, имели обе альтернативные ветви Then и Else, что всегда указывало на необходимость управления потоками

данных, а в программе (2.28) указывало на необходимость управления адресами записи.

При отсутствии одной из альтернативных ветвей достаточно выполнять управление не потоками данных, а адресами, по которым необходимо осуществить запись полученного результата.

В программе (рис.3.7) используется условный оператор без альтернативной ветки Else.

```
Var a,b,d : Array Integer [100 : Stream] Mem;
Var i, n : Number ;
Define n=99 ;
Cadr IfMemAdr;
  For i := 2 to n do
    Begin
      If b[i] = 0 then
        a[i] := F1(c[i])
      Else
        a[i-2] := F2(d[i]);
    End;
Endcadr;
```

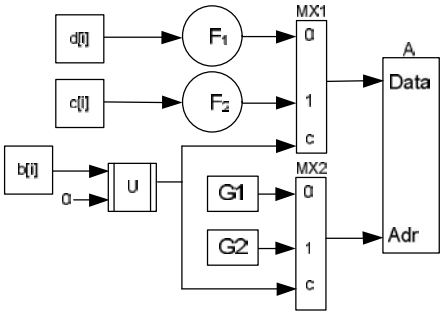


Рис. 3.6. Информационный граф программы

```
Var a,b,d : Array Integer [100 : Stream] Mem;
Var i, n : Number ;
Define n=99 ;
Cadr IfMemAdr;
  For i := 2 to n do
    Begin
      If b[i] = 0 then
        a[i] := F1(c[i])
    End;
Endcadr;
```

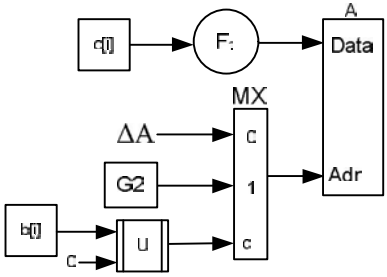


Рис. 3.7. Информационный граф программы без альтернативной ветви в условном операторе

В данном случае вычисленный результат функции F_1 приходит непосредственно на вход данных контроллера распределенной памяти (КРП), соответствующего массиву A, а

для управления адресами записи используется мультиплексор MX. При отсутствии альтернативной ветви на соответствующий вход мультиплексора MX подается адрес ΔA, соответствующий «мусорной» ячейке памяти, а блок G_2 выполняет генерацию адреса распределенной памяти на основании начального адреса записи, соответствующего элементу массива A[I].

Рассмотрим случай, когда переменная *a* является **регистровой**. В этом случае в отличие от мемориальной переменной необходимость в вычислении адресов отсутствует. Регистровая переменная представляет собой блок, имеющий два входа: вход данных (Data) и вход «разрешения записи»(CE). Вход CE позволяет управлять записью данных в регистр. Если на вход CE приходит «1», то значение на входе Data будет записано в регистр, в противном случае нет.

При использовании полного условного оператора все его альтернативные ветки будут реализованы, а мультиплексор, выполняющий выбор одной из альтернативных веток, подключен на вход данных (Data) регистра. Поскольку запись в регистр будет выполняться постоянно в независимости от условия условного оператора, то на входе CE регистра будет установлена константа, равная «1». В программе (рис. 3.8) используются условный оператор и регистровая переменная.

```
Var b,c,d : Array Integer
[10 : Stream] Mem;
Var r : Integer Reg;
Var i, n : Number;
Define n=9;
Cadr IfReg;
  For i := 0 to n do
    Begin
      If b[i] = 0 then
        r := F1(c[i])
      Else
        r := F2(d[i]);
    End;
Endcadr;
```

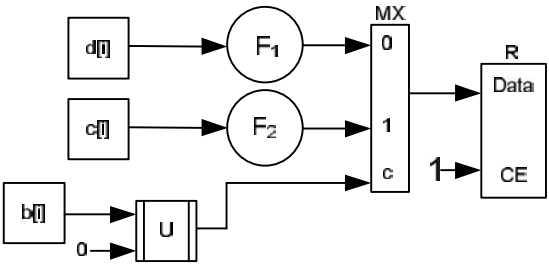


Рис. 3.8. Информационный граф взаимодействия регистровой переменной и условного оператора

В случае использования неполного условного оператора (отсутствует альтернативная ветка Else) на нулевой вход мультиплексора, соответствующего альтернативной ветке Else, должно приходить текущее значение регистра R, так как в случае невыполнения условия сравнения значение регистра R должно остаться прежним. В этом случае в информационном графе появляется скрытая рекурсия за счет обратной связи с регистра на вход мультиплексор (см. рис. 3.9).

```
Var b,c : Array
Integer [10 : Stream]
Mem;
Var r : Integer Reg;
Var i, n : Number;
Define n=9;
Cadr IfReg;
  For i := 0 to n do
    Begin
      If b[i] = 0 then
        r := F1(c[i]);
    End;
Endcadr;
```

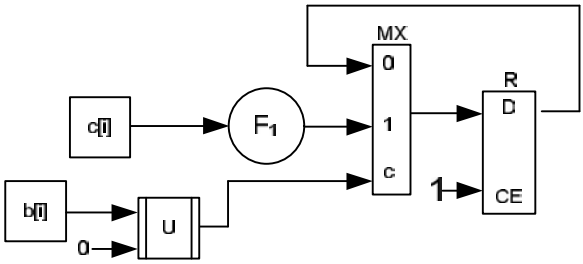


Рис. 3.9. Информационный граф взаимодействия регистровой переменной и условного оператора без альтернативной ветки

Для оптимизации информационного графа и удаления обратной связи необходимо использование входа CE регистра. При использовании неполного условного оператора мультиплексор и блок управления позволяют записать в регистр новое вычисленное значение или переписать значение регистра своим же значением. Подобное действие выполняет и вход CE регистра, но он только разрешает или запрещает запись в регистр. Таким образом, на вход CE может быть подключен блок управления U. Если результатом работы блока управления является истина, то на вход CE поступит «1», соответственно

значение, вычисленное функцией F_1 (см. рис. 3.9), будет записано в регистр, в противном случае значение регистра не изменится.

В программе (см. рис. 3.9) используются неполный условный оператор и регистровая переменная. Отсутствие альтернативной ветки условного оператора приводит к необходимости перезаписи значения регистровой переменной. Оптимизированный граф информационного графа программы (рис. 3.9) представлен на рис. 3.10.

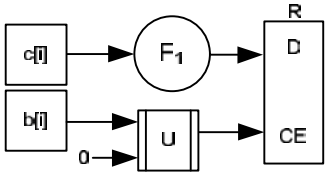


Рис. 3.10. Оптимизированный информационный граф

Как видно из рис. 3.10, в информационном графе отсутствуют мультиплексор и обратная связь для перезаписи значения регистра R.

В случае если полный условный оператор выполняется для коммутационной переменной, то для нее, так же как и для регистровой переменной, необходимо реализовать только альтернативные ветви условного оператора. Выбранное мультиплексором значение передается непосредственно на блок, выполняющий ее дальнейшую обработку, при этом данное значение не записывается в коммутационную переменную, поскольку коммутационная переменная является дугой графа. Таким образом, вычисленное значение, соответствующее переменной T, является недоступным для пользователя.

В программе (рис. 3.11) используется коммутационная переменная, значение которой определяется условным оператором.

```
Var a,b,c,d : Array Integer [10 :
Stream] Mem;
Var t : Integer Com;
Var i, n : Number;
Define n=9;
Cadr IfCom;
  For i := 0 to n do
    Begin
      If b[i] = 0 then
        t := F1(c[i])
      Else
        t := F2(d[i]);
      a[i] := t + c[i];
    End;
Endcadr;
```

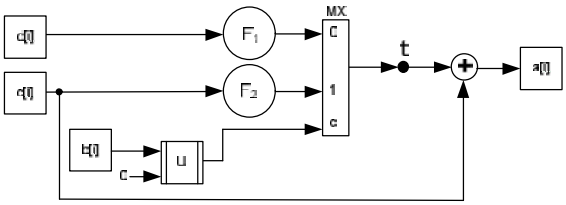


Рис. 3.11. Информационный граф взаимодействия коммутационной переменной и условного оператора

Если для выбора результата, записываемого в коммутационную переменную, используется неполный условный оператор, то так же как и для регистровой переменой, используются мультиплексор и блок управления, но на нулевой вход мультиплексора будет подключена константа, равная «0» (рис. 3.12), так как в случае невыполнения условия сравнения значение коммутационной переменной T должно быть равно “0”.

```
Var a,b,c,d : Array Integer [10 :
Stream] Mem;
Var t : Integer Com;
Var i, n : Number;
Define n=9;
Cadr IfCom;
  For i := 0 to n do
    Begin
      If b[i] = 0 then
        t := F1(c[i]);
      a[i] := t + c[i];
    End;
Endcadr;
```

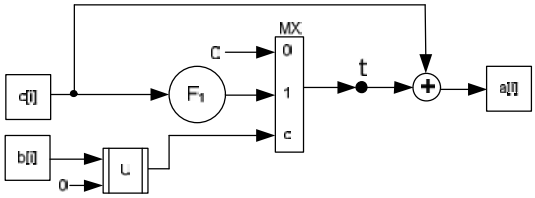


Рис. 3.12. Информационный граф взаимодействия регистровой переменной и условного оператора без альтернативной ветки

Для оптимизации информационного графа необходимо заменить мультиплексор на блок логического суммирования, на входы которого будут подаваться результат, полученный функцией F_1, и результат работы блока управления.

Программа (рис. 3.12) демонстрирует работу коммутационной переменной и условного оператора без альтернативной ветки (рис. 3.13).

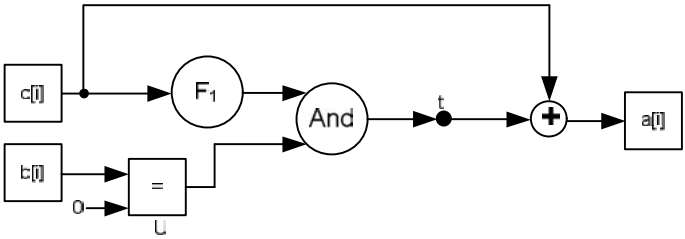


Рис. 3.13. Оптимизированный информационный граф взаимодействия коммутационной переменной и условного оператора без альтернативной ветки

Рассмотрим случай, когда переменная *a* является **внутренней памятью**. Для простоты рассмотрения внутренней памяти здесь и далее будем использовать однопортовую и двухпортовую внутреннюю память.

Однопортовая внутренняя память представляет собой блок, который имеет информационный вход/выход (Data), адресный вход (Adr) и вход «разрешения записи» (Type). Вход Type недоступен пользователю, к нему сигналы подключается транслятором. Если порт выполняет процесс чтения, то на вход Type будет подключен потенциал «0», в противном случае подключен сигнал стробирования данных. Однопортовая внутренняя память позволяет получить доступ к ней только одному процессу, тип которого определяется значением на входе Type.

Использование адресного входа (Adr) является обязательным, поэтому любая адресация к переменной InterMem расценивается как косвенная и реализуется всегда только на структурном уровне. Следовательно, при использовании

условного оператора с переменной InterMem необходимо выполнять не только управление потоками данных, но и адресами записи.

При рассмотрении особенностей взаимодействия условного оператора и внутренней памяти предположим, что условный оператор является полным, а каждая его ветка состоит из одного оператора присваивания. Таким образом, условный оператор будет иметь следующий вид:

If α Then $a_i = F_1(c_1, c_2, ..., c_n)$ Else $a_j = F_2(d_1, d_2, ..., d_m)$.(3.4)

где а – переменная типа InterMem.

Если в программе (3.4) адреса записи элементов a_i и a_j равны (т.е. $i = j$), то управление данными адресами не осуществляется, а сам адрес напрямую подключается к соответствующему входу адреса (Adr) внутренней памяти.

Если адреса записи элементов a_i и a_j не равны (т.е. $i \neq j$), то управление осуществляется не только на уровне данных, но и на уровне адресов записи.

В обоих случаях использование условного оператора приводит к тому, что над переменной А будет осуществляться только один процесс чтения в независимости от вложенности условных операторов, соответственно для реализации программы (3.4) достаточно описать переменную А как однопортовую внутреннюю память.

Примеры использования однопортовой внутренней памяти и условного оператора рассмотрены в программе (рис. 3.14).

Поскольку адреса записи в программе (рис. 3.14) совпадают, то управление потоками данных осуществляется мультиплексором (MX), а на вход Adr подаются адреса с блока, аппаратно реализующего оператор цикла с индексной переменной I.

Отсутствие альтернативной ветки в условном операторе требует осуществления управления только на уровне адресов, а поток данных должен подаваться сразу на вход Data внутренней памяти.

```
Var b,c,d : Array Real [10 : Stream] Mem;
Var a : Array Real [10 : Stream] InterMem;
Var i, n : Number;
Define n=9;
Cadr ExampleInterMem;
  For I := 2 to 9 do
   Begin
    If  b[i] = 0 then
       a[i] := c[i]
    Else
       a[i] := d[i];
     End;
Endcadr;
```

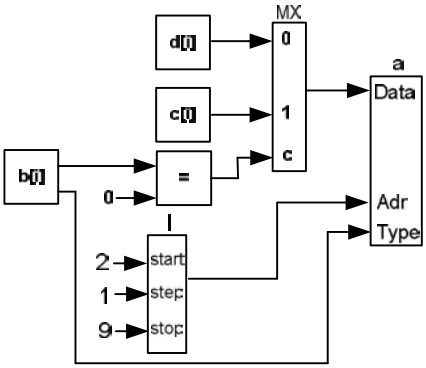


Рис. 3.14. Информационный граф взаимодействия однопортовой внутренней памяти и условного оператора

В этом случае на вход мультиплексора, которому соответствует отсутствующая ветка условного оператора, необходимо установить адрес $\Delta A = -1$, который указывает на мусорную ячейку внутренней памяти. Программа (рис. 3.15) демонстрирует отсутствие альтернативной ветки Else условного оператора.

```
Var b,c,d : Array Real [10 : Stream] Mem;
Var a : Array Real [10 : Stream] InterMem;
Var i, n : Number;
Define n=9;
Cadr ExampleInterMem;
  For I := 2 to 9 do
   Begin
    If  b[i] = 0 then
       a[i] := F1(c[i]);
     End;
Endcadr;
```

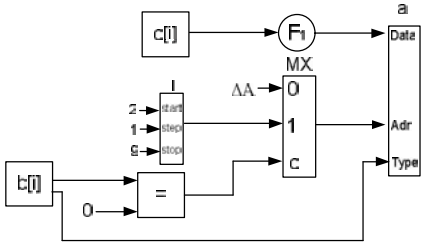


Рис. 3.15. Информационный граф взаимодействия переменной InterMem и условного оператора без альтернативной ветки

В информационном графе (рис. 3.15) на вход данных (Data) блока внутренней памяти результат вычисления функции F_1 приходит напрямую, а над адресами записи необходимо выполнять управление.

На нулевой вход мультиплексора подключена константа $\Delta A = -1$, поскольку нулевой вход мультиплексора

соответствует отсутствующей альтернативной ветке Else условного оператора.

Правила синтезирования информационного графа при использовании условного оператора и переменной InterMem будут отличаться в зависимости от того, какой процесс - чтение или запись выполняется над переменной.

В случае если над переменной InterMem выполняется процесс чтения, то в отличие от процесса записи управление на уровне данных никогда не будет осуществляться в независимости от того, какой условный оператор выполняется (полный или неполный).

При использовании многопортовой памяти используются те же самые принципы синтезирования информационного графа, что и для однопортовой внутренней памяти.

Предположим, что условный оператор будет иметь следующий вид:

$$\text{If } \alpha \text{ Then } a_i = F_1(a_{i-3}) \text{ Else } a_{i+3} = F_2(a_{i-5}). \qquad (3.5)$$

где a – переменная, объявленная как двухпортовая внутренняя память.

В этом случае над переменной A выполняется четыре независимых процесса. Согласно вышеизложенному для реализации программы (3.5) необходимо использовать четырехпортовую внутреннюю память.

Однако использование условного оператора позволяет сократить число процессов, выполняющих одновременный доступ к переменной A, с 4 до 2, поскольку в один момент времени будет выполняться только одна из двух возможных веток условного оператора, в каждой из которых осуществляется по два независимых процесса.

Таким образом, для реализации программы (3.5) достаточно использование двухпортовой внутренней памяти. Информационный граф программы (3.5) представлен на рис. 3.16.

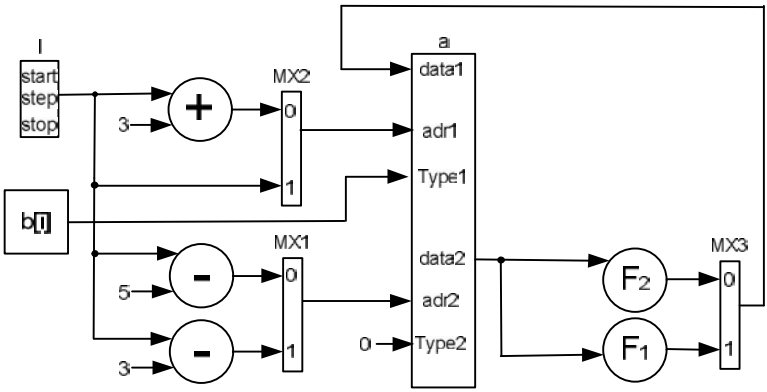


Рис. 3.16. Информационный граф программы (3.5)

Для простоты рассмотрения информационного графа программы (рис 3.16) блок управления мультиплексорами MX1, MX2 и MX3 не приведен. Мультиплексор MX1 выполняет переключение между адресами записи, мультиплексор MX2 - между адресами чтения, а мультиплексор MX3 - между результатами функций F_1 и F_2, при этом все мультиплексоры управляются одним и тем же блоком управления.

Если количество независимых процессов в любой из веток условного оператора будет больше, чем «2», то использование двухпортовой внутренней памяти является синтаксической ошибкой, поскольку приведет к нарушению правила однократного присваивания.

В случае отсутствия альтернативной ветки Else в программе (3.5) управление будет осуществляться только на уровне адресов как чтения, так и записи, а на вход соответствующий альтернативной ветки Else мультиплексоров управления будет подана константа $\Delta A = -1$.

3.2. Особенности использования операторов цикла

Оператор цикла предназначен для перебора диапазона значений, определяемых начальным и конечным значениями с заданным шагом. Оператор цикла в языке COLAMO в

зависимости от контекста использования его индексной переменной может быть реализован как структурно, так и процедурно. Если индексная переменная цикла используется в вычислениях в качестве аргумента, то оператор цикла реализуется в структурном компоненте параллельной программы в виде аппаратно реализованного блока. Если индексная переменная цикла используется в качестве адреса массива, то оператор цикла реализуется процедурно.

Как правило, использование операторов цикла приводит к необходимости его реализации как на структурном, так и на процедурном уровне.

В данном параграфе будут рассмотрены особенности обработки оператора циклов при формировании процедурного компонента параллельной программы.

С точки зрения процедурного компонента параллельной программы оператор цикла можно представить в виде генератора (G), основной задачей которого является формирование потоков данных на чтение или на запись. Сам генератор является виртуальным объектом и в генерации структурного компонента не участвует.

Количество генераторов, необходимых для генерации потоков данных, и количество элементов в каждом потоке определяются операторами циклов и контекстом их использования в параллельной программе. Из программы (рис. 3.17) видно, что индексная переменная I оператора цикла используется для адресации к элементам массивов, однако, какой тип доступа - параллельный или последовательный - определить невозможно без описания массивов A и B.

```
Cadr ExampleFor;
  For i := 0 to n do
      a[i] := F1(b[i]);
EndCadr;
```

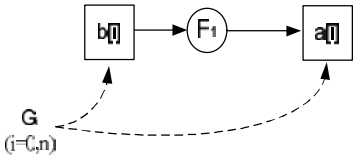


Рис. 3.17. Программа и эквивалентная ей граф-схема при использовании потоковых массивов

Если массивы A и B являются потоковыми, то для генерации потоков данных необходимо использовать два генератора, один из которых формирует поток данных размерностью $n+1$ на чтение, а другой формирует поток данных той же размерности, но на запись. Если размерность генерируемых потоков разными генераторами совпадает, то для простоты будем использовать один генератор.

Оператор цикла на рис. 3.17 представлен в виде генератора G_1, который генерирует поток данных размерностью $n+1$ в режиме чтения для контроллера распределенной памяти, соответствующего массиву B, и в режиме записи для контроллера распределенной памяти, соответствующего массиву A.

Если массивы A и B являются векторными, то данные массивы будут распараллелены, соответственно и блоки генерации потоков данных необходимо также распараллелить. Тогда граф-схема программы (рис. 3.17) будет состоять из $n+1$ генераторов, каждый из которых будет формировать поток данных, состоящий из одного элемента. Таким образом, параллельно-конвейерная программа будет состоять из $n+1$ операторов чтения и $n+1$ операторов записи, каждый из которых будет обрабатывать только один элемент потока. Граф-схема программы при использовании потоковых массивов A и B показана на рис. 3.18.

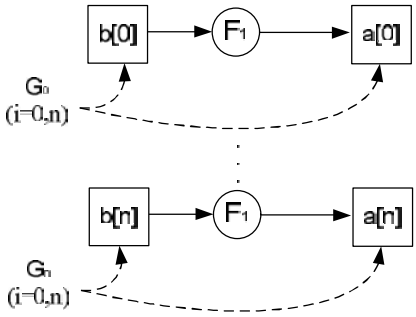


Рис. 3.18. Граф-схема программы при использовании векторных массивов

При использовании вложенных циклов количество генераторов, необходимых для генерации одного потока данных, будет равно количеству вложенных переменных. Однако в отличие от традиционных языков программирования использование оператора цикла не гарантирует повторного выполнения операторов, расположенных в теле цикла. Если отсутствует информационная зависимость между индексной переменной цикла и переменными в теле цикла, то данный цикл будет исключен из рассмотрения при формировании процедурного компонента программы. Это же правило распространяется и на вложенные циклы.

Так в программе (3.6) используются два вложенных цикла с индексными переменными I и J.

```
Var a,b : Array Integer [100 : Stream] Mem;
Var i,j : Number ;
Const N = 10;
Const M = 100;
Cadr ExampleFor;
  For i := 0 to N - 1 do
     For j := 0 to M - 1 do
         a[i] := F1(b[i]);
EndCadr;
```
(3.6)

Поскольку индексная переменная I не участвует в формировании адресов массивов A и B и в вычислениях, то оператор цикла, соответствующий индексной переменной I, будет удален из рассмотрения. Если вложенные циклы имеют информационную зависимость, например, используются в адресации к одной и той же переменной, то для формирования потока данных, соответствующего данной переменной, будут использоваться два генератора, соответствующих данным операторам цикла. Как правило, оператор, соответствующий последнему вложенному циклу, используется для генерации основного потока данных, а все остальные операторы - для реализации смещения адресов и зацикливания.

В программе (рис. 3.19) используется два вложенных цикла, связанных между собой, поскольку их индексные переменные необходимы для формирования адресов при доступе к массивам A и B.

```
Var a,b : Array Integer [10 :
Stream, 100 : Stream] Mem;
Var i,j : Number ;
Const N = 9 ;
Const M = 49 ;
Cadr ExampleFor;
  For i := 0 to N do
    For j := 0 to M do
     Begin
         a[i,j] := F1(b[i,j]);
     End;
EndCadr;
```

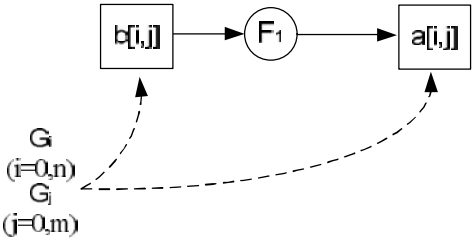


Рис. 3.19. Программа и эквивалентная ей граф-схема с использованием взаимосвязанных вложенных операторов цикла

Параллельно-конвейерная программа на языке ARGUS будет состоять из операторов чтения и записи и из операторов смещения адреса и операторов цикла.

```
Var A, B, Inc : ParamAddress;
Var StepA, StepB, RepeatA, RepeatB : ParamAddress;
Label M0, M1;
Define A = 0; Define B = 0;
Define StepA = 1; Define StepA = 1;
Define RepeatA = 50; Define RepeatB = 50;
Define Inc = 100;
Define Count = 9;
Cadr ExampleFor
   DMC[0.0]#
     M0 : Read A Step StepA Repeat RepeatA;
     AddPP A, Inc;
     Loop Count Goto M0;
   DMC[0.1]#
     M1 : Read B Step StepB Repeat RepeatB;
     AddPP B, Inc;
     Loop Count Goto M1;
EndCadr;
```

(3.7)

В случае если вложенные операторы цикла используются при индексации к разным массивам, но между этими массивами существует информационная зависимость, то данные операторы цикла также являются информационно зависимыми.

В программе (рис. 3.20) для обращения к элементам массива B используется только индексная переменная I, а для массива A оба цикла по I и по J. Однако массивы A и B информационно связаны между собой коммутационной переменной T, а это значит, что оба оператора цикла будут использоваться для формирования потоков данных для массивов A и B.

```
Var a : Array Integer [10 : Stream,
100 : Stream] Mem;
Var b : Array Integer [100 : Stream]
Mem;
Var t : Integer Com;
Var i,j : Number ;
Const N = 9 ;
Const M = 49 ;
Cadr ExampleFor;
  For i := 0 to N do
    For j := 0 to M do
     Begin
         t := F1(b[i]);
         a[i,j] := t;
     End;
EndCadr;
```

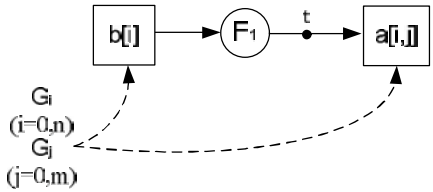


Рис. 3.20. Программа и эквивалентная ей граф-схема с двумя генераторами

Параллельно-конвейерная программа для данного примера будет выглядеть следующим образом:

```
Var A, B, Inc : ParamAddress;
Var StepA, StepB, RepeatA, RepeatB : ParamAddress;
Label M0, M1;
Define A = 0; Define B = 0;
Define StepA = 1; Define StepA = 1; Define RepeatA = 50;
Define RepeatB = 50;
Define Inc = 100;
Define Count = 9;
```

```
Cadr ExampleFor
   DMC[0.0]#
     M0 : Read A Step StepA Repeat RepeatA;
          AddPP A, Inc;
     Loop Count Goto M0;
   DMC[0.1]#
     M1 : Read B Step StepB Repeat RepeatB;
     Loop Count Goto M1;
EndCadr;
```
(3.8)

Поскольку на массив B оператор цикла с индексной переменной I в явном виде не оказывает влияние, то смещение адресов, как в случае с массивом A, осуществляться не будет, а сам массив B будет прочитан $Count+1$-раз, начиная с одного и того же начального адреса B.

Если для индексации элементов информационно связанных массивов используются индексные переменные разных операторов цикла, то в общем случае для генерации потоков данных будут использоваться все генераторы, соответствующие данным операторам цикла.

Однако если среди этих операторов существуют операторы цикла с одинаковыми параметрами, то данные операторы могут быть объединены под один генератор. Если параметры операторов цикла отличаются, то в этом случае при генерации потоков данных возможна их рассинхронизация.

В программе (3.9) массивы A и B связаны между собой коммутационной переменной T, а для доступа к элементам используются разные операторы циклов.

Поскольку операторы циклов с индексной переменной I имеют одинаковые параметры, то данные циклы могут быть объединены под один генератор. Циклы с индексной переменной J имеют разные параметры, соответственно данные операторы будут представлять собой два независимых генератора.

```
Var a, b : Array Integer [10 : Stream, 100 : Stream] Mem;
Var t : Integer Com;
Var i,j : Number ;
Const N = 9 ;
```

```
Const M = 49 ;
Cadr ExampleFor;
  For i := 0 to N do
    For j := 0 to M do
        t := F1(b[i,j]);

  For i := 0 to N do
    For j := 5 to M do
        a[i,j] := t;
EndCadr;
```
(3.9)

Наиболее сложным случаем является ситуация, когда доступ к массиву осуществляется несколькими независимыми процессами, при этом операторы циклов, использующиеся для доступа к элементам массива, для каждого процесса отличаются друг от друга.

В этом случае генераторы, соответствующие данным операторам цикла, должны быть объединены в генератор, который бы отвечал всем требованиям объединяемых генераторов.

Такой генератор будет иметь в качестве начального значения самое минимальное из всех начальных значений, соответствующих объединяемым генераторам в качестве конечного значения, - максимальное конечное значение соответствующих генераторов, а для значения шага необходимо выбрать значение, позволяющее не только идти по массиву с максимально возможным шагом, но и обеспечивающее покрытие всех необходимых адресов чтения для соответствующих генераторов. Однако результат работы такой программы является труднопрогнозируемым.

Использование во вложенных циклах параметров, значения которых могут меняться на каждой итерации его выполнения, приведет к снижению производительности вычислительной системы и аппаратным затратам, поскольку требуются дополнительные операторы для вычисления параметров циклов в процедурном компоненте или аппаратная реализация вычисления параметров в структурном компоненте.

Если в качестве параметра оператора цикла используются индексные переменные или константы, то вычисление таких параметров может быть выполнено на процедурном уровне с целью уменьшения аппаратных затрат в структурном компоненте.

Если в качестве параметра оператора цикла используется мемориальная переменная или некоторая вычислительная функция, то реализация вычисления параметра цикла выполняется как на структурном уровне, так и на процедурном.

В случае использования в качестве параметра цикла мемориальной переменой в независимости от того, на структурном или процедурном уровне выполняется реализация, необходимо выполнить пересылку значения, соответствующего используемой мемориальной переменной (при реализации на процедурном уровне), или вычисленного значения параметра (при реализации на структурном уровне) во все контроллеры распределенной памяти, соответствующие переменным, расположенным в теле данного оператора цикла.

Пересылка данных между контроллерами распределенной памяти на процедурном уровне невозможна, следовательно, необходимо реализовать в информационном графе связи между контроллерами распределенной памяти соответствующих переменных.

При структурной реализации вычисленное значение параметра пересылается в дополнительную регистровую переменную и во все контроллеры распределенной памяти кроме КРП, соответствующего переменной, используемой в параметре цикла, поскольку данное КРП настроено на процесс чтения. Для записи вычисленного значения в данное КРП используется регистровая переменная, в которую ранее было помещено вычисленное значение параметра. Для пересылки вычисленного значения в процедурном компоненте параллельной программы будут реализованы дополнительные кадры, выполняющие пересылку и запись вычисленных значений в соответствующие контроллеры распределенной памяти.

При процедурной реализации вычисления параметра цикла в структурном компоненте отсутствует аппаратная реализация соответствующего параметра цикла. Однако присутствуют проложенные связи между контроллерами распределенной памяти для пересылки соответствующей информации. При такой реализации значение мемориальной переменной, используемой в параметре цикла, будет разослано во все контроллеры распределенной памяти, после чего средствами языка ARGUS будет выполнен расчет данного параметра цикла в каждом КРП. Данная реализация также требует использования дополнительных кадров для пересылки данных между КРП.

В программе (рис. 3.21) в выражении начального параметра вложенного оператора цикла используется мемориальная переменная B. В этом случае реализация вычисления данного параметра может быть реализована как структурно, так и процедурно.

```
Var a : Array Integer [10 : Stream, 100 : Stream] Mem;
Var b : Array Integer [10 : Stream] Mem;
Var i,j : Number ;
Cadr ExampleFor;
  For i := 0 to 5 do
    For j := b[i] + 3 to 89 do
     Begin
          a[i,j] := F1(i,j);
     End;
EndCadr;
```

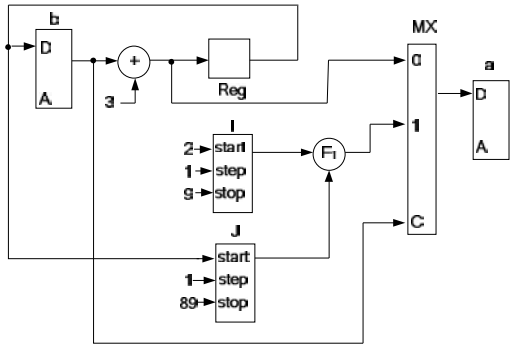


Рис. 3.21. Программа и эквивалентная ей структурная реализации параметра цикла программы

В контроллер распределенной памяти, соответствующий переменной A, будет выполняться запись значений из разных источников, одним из которых является блок вычисления параметра цикла, а другим - функция F_1. Для выбора

соответствующего значения на запись в переменную A используется мультиплексор.

При процедурной реализации в информационном графе отсутствуют блоки, выполняющие вычисление соответствующего параметра цикла, а присутствуют только связи для пересылки значения элемента B[i] в контроллер распределенной памяти, соответствующего переменной A. Параллельно-конвейерная программа при процедурной реализации вычисления параметра цикла будет отличаться от структурной реализации только набором операторов, выполняемых в кадре Sub2.

```
Cadr Sub2
   DMC[0.1]#
     MovePP ParamStart, B;
     AddPP ParamStart, Inc
     Read ParamStart Step OneStep Repeat OneRepeat;
    DMC[0.0]#
      Write ParamStart Step OneStep Repeat OneRepeat;
EndCadr;                                                    (3.10)
```

В кадре Sub2 выполняются суммирование элемента B[i] с константой «3» и запись полученного значения по адресу ParamStart контроллера распределенной памяти, соответствующего переменной A.

3.3. Особенности использования вычислительных структур

Основными конструкциями языка COLAMO являются конструкции кадр (Cadr), подкадр (SubCad), конструкция Let, конструкция (Implicit) и конструкция SubRoutine, рассмотренные в параграфе 1.3. В данном параграфе рассматривается трансляция конструкций подкадр и Let.

Любой вызов конструкции SubCadr в тексте программы требует добавления в точку вызова подкадра в информационном графе программы, соответствующего подкадру информационного подграфа. Распараллеливание подкадра подчиняется тем же самым правилам, что и для ранее

рассмотренных конструкций. Распараллеливание подкадра соответствует многократному вызову данного подкадра, что приводит к мультиплицированию аппаратной реализации данного подкадра.

На примере программы (рис.3.22) показан двойной вызов одного и того же подкадра Calculate, соответственно данная конструкция будет реализована в информационном графе программы дважды.

```
Var a, b, e : Array Integer [100 :
Stream] Mem;
Var c, d : Array Integer [100 :
Stream] Mem;
Var i : Number;
Const n = 100;
SubCadr Calculate (In : In1, In2;
Out : Out1);
Var In1, In2, Out1 : Integer Com;
      Out1 := F1(In1, In2);
EndSubCadr;
Cadr One;
   For i := 0 to n - 1 do
      Calculate(a[i], b[i], c[i]);
      d[i] := F2(c[i]);
      Calculate(d[i], b[i], e[i]);
EndCadr;
```

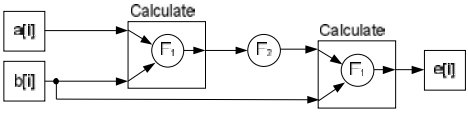


Рис. 3.22. Программа и эквивалентный ей информационный граф

Вызов конструкций SubCadr в кадрах многокадровой параллельной программы приводит к нарушению принципа «одноструктурности» параллельной программы, поскольку вычислительные структуры каждого кадра будут неизоморфны. На первый взгляд, использование в разных кадрах одной и той же конструкции SubCadr не нарушает принципа «одноструктурности» параллельной программы, поскольку вычислительные структуры каждого кадра будут эквивалентны. Однако данное предположение является неверным, поскольку при каждом вызове конструкции SubCadr в информационный граф программы будет добавлен информационный подграф,

соответствующий данной конструкции. Таким образом, полученный информационный граф параллельной программы не будет соответствовать информационному подграфу каждого из кадров.

Для реализации многокадровых параллельных программ необходимо использовать статическую конструкцию Let. Конструкция Let реализуется в информационном графе только один раз и в отличие от подкадра не мультиплицируется при ее вызове. Распараллеливание конструкции Let, в отличие от конструкции SubCadr, запрещено. В многокадровых параллельных программах информационные потоки, соответствующие конструкции Let, в разных кадрах могут иметь разные источники и приемники.

Поскольку конструкция Let является одной для всех кадров, то необходимо выполнять переключение информационных потоков при смене кадров. Переключение информационных потоков при вызове конструкции Let может быть осуществлено при помощи сдвоенных контроллеров распределенной памяти или с помощью дополнительных коммутаторов.

Под сдвоенным КРП понимаются два канала памяти с возможностью одновременного чтения и записи по разным каналам. Автоматическое использование сдвоенного КРП транслятором языка позволяет упростить схемотехническую организацию потоков данных в вычислительной системе.

Для простоты рассуждения допустим, что параллельная программа состоит из двух кадров: $C_1 =< R_1, Q_1, W_1 >$ и $C_2 =< R_2, Q_1, W_2 >$. Пусть функции чтения и записи (R и W) выполняются над мемориальными переменными, причем подмножества информационных потоков, соответствующих значениям функций R и W в пределах одного кадра, являются непересекающимися, а в пределах множества кадров их пересечение возможно.

Каналы памяти α и β могут быть объединены в сдвоенный канал памяти, если для переменных a и b,

расположенных в этих каналах памяти, соответственно будут выполнены следующие условия: $a \in (X_1 \cap Y_2)$ и $a \notin X^2$; $b \in (X_2 \cap Y_1)$ и $b \notin Y_2$, где $X_i = R_i(t_i)$, $Y_i = W_i(t_i)$, $i = \overline{1,2}$. Данное утверждение можно расширить на множество кадров.

Если формирование сдвоенного канала памяти при вызове LET невозможно, то переключение источников данных при смене кадров выполняется с помощью коммутаторов, аппаратно-реализованных операторов условного перехода, управление которым осуществляется программой контроллера распределенной памяти. Переключение информационных потоков с помощью коммутаторов требует как дополнительного оборудования для их реализации, так и предварительной настройки коммутаторов перед выполнением каждого кадра. Поэтому использование сдвоенных КРП является предпочтительным.

На примере программы (рис.3.23) показана двухкадровая параллельная программа с конструкцией Let. Поскольку над мемориальными переменными A и C в разных кадрах выполняются разные процессы (т.е. переменной A в кадре One соответствует процесс чтения, а в кадре Two – процесс записи, соответственно переменной C в кадре One соответствует процесс записи, а в кадре Two – процесс чтения), то данные переменные могут быть объединены в сдвоенный контроллер распределенной памяти.

Однако следует отметить, что если любая из этих переменных, используемая на чтение, будет подана еще хотя бы на один вход конструкции Let в одном из кадров, то организация сдвоенного КРП является невозможной.

Если на один и тот же вход конструкции Let поступают информационные потоки, соответствующие разным мемориальным переменным, то на соответствующий вход конструкции Let будет подключен коммутатор (мультиплексор), а на входы данного коммутатора будут поступать соответствующие информационные потоки. Настройка

коммутатора выполняется на процедурном уровне после смены каждого кадра.

```
Var a, b, c : Array Integer [100 : Stream] Mem;
Var i : Number;
Const n = 100;
Let Calculate (In : In1, In2; Out : Out1);
Var In1, In2, Out1 : Integer Com;
      Out1 := F(In1, In2);
EndLet;
Cadr One;
   For i := 0 to n - 1 do
      Calculate(a[i], b[i], c[i]);
EndCadr;
Cadr Two;
   For i := 0 to n - 1 do
      Calculate(c[i], b[i], a[i]);
EndCadr;
```

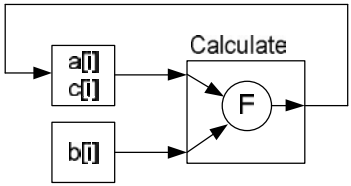


Рис. 3.23. Программа и эквивалентная ей реализация двухкадровой параллельной программы с использованием сдвоенного КРП

На примере программы (рис. 3.24) показана двухкадровая параллельная программа для организации потоков данных, в которой используются дополнительные коммутаторы.

```
Var a, b, c, d : Array Integer [100 :
Stream] Mem;
Var i : Number;
Const n = 100;
Let Calculate (In : In1, In2; Out : Out1);
Var In1, In2, Out1 : Integer Com;
      Out1 := F(In1, In2);
EndLet;
Cadr One;
   For i := 0 to n - 1 do
      Calculate(a[i], b[i], d[i]);
EndCadr;
Cadr Two;
   For i := 0 to n - 1 do
      Calculate(c[i], b[i], d[i]);
EndCadr;
```

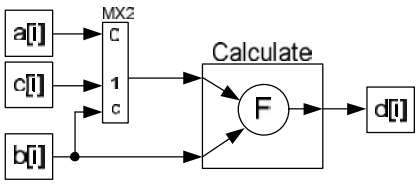


Рис. 3.24. Программа и эквивалентная ей реализация двухкадровой параллельной программы с использованием дополнительных коммутаторов

Поскольку организация сдвоенного КРП между переменными A и C невозможна, то для организации потоков данных должны использоваться коммутаторы.

В общем случае количество входов или выходов коммутатора зависит от количества кадров параллельной программы.

Однако источники или приемники информационных потоков при смене кадров могут не изменяться, поэтому в этом случае количество входов или выходов на коммутаторе может быть сокращено, что приведет к уменьшению аппаратных затрат, но потребует более сложного управления коммутаторами на процедурном уровне.

3.4. Контрольные вопросы

1. Опишите правила распараллеливания условного оператора.
2. Опишите особенности реализации условных операторов в зависимости от способа хранения переменных.
3. Каким образом определяется зависимость между операторами циклов?
4. В каком случае операторы цикла приведут к снижению производительности РВС?
5. В каком случае будет использоваться сдвоенный контроллер распределенной памяти?

4. СРЕДА РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРИКЛАДНЫХ ПРОГРАММ

Основной задачей интегрированной среды разработки является ускорение процесса разработки параллельных программ различной конфигурации за счет интеграции всех необходимых для этого средств в одну среду, обеспечивающую интерактивную подготовку исходных текстов и загрузочных модулей параллельных решений.

Интегрированная среда разработки обеспечивает пользователя всеми современными возможностями комплексных средств разработки, таких как Borland Developer Studio или Microsoft Visual Studio:

– интерактивной разработкой параллельных программ;

– обеспечением объектного представления программ и отдельных ее компонентов;

– обеспечением эффективного механизма отладки параллельных программ, в том числе и со связью между исполняемым и исходным кодами;

– выделением ключевых слов синтаксиса языка;

– быстрой навигацией по ошибкам текста программы;

– навигацией по тексту программы с помощью ссылок;

– интерактивным доступом к контекстному меню.

Интегрированная среда позволяет пользователю выполнять трансляцию параллельных программ в двух режимах:

- автоматическом режиме трансляции;

- полуавтоматическом режиме трансляции.

Полуавтоматический режим трансляции позволяет пользователю контролировать ход выполнения трансляции и по необходимости вносить изменения на этапе лексического и синтаксического анализа.

4.1. Интерфейс интегрированной среды разработки

Интегрированная среда поддерживает многооконную организацию пользовательского интерфейса, где используется

два типа окон: к первому относятся окна специализированного текстового редактора, предназначенного для ввода исходных текстов параллельных программ, ко второму – множество вспомогательных окон, через которые осуществляется доступ к элементам управления процессом разработки.

Главная форма интегрированной среды показана на рис. 4.1. Интегрированная среда разработки предназначена для создания, документирования, запуска и отладки параллельных программ для реконфигурируемой вычислительной системы на языке высокого уровня COLAMO.

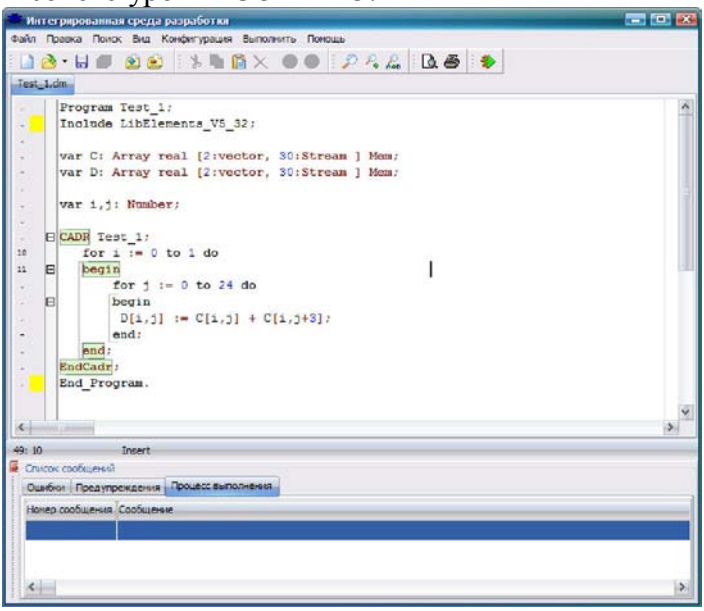


Рис. 4.1. Главная форма интегрированной среды разработки

Пользовательский интерфейс включает:

1. Меню и панели инструментов. Через основное меню программы и панели инструментов осуществляется доступ к основным функциям среды разработки (рис. 4.2).

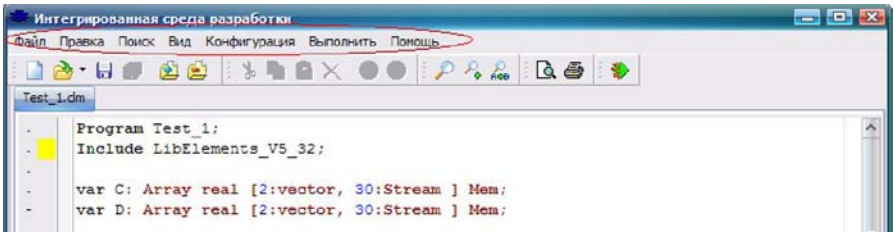


Рис. 4.2. Меню программы

2. Специализированный текстовый редактор. Разработка программ на языке высокого уровня COLAMO осуществляется в специализированном текстовом редакторе (рис. 4.3).

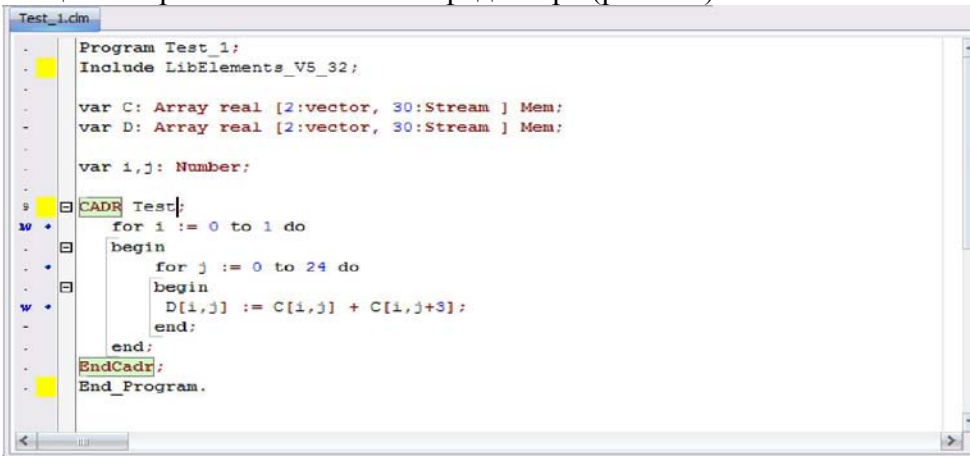


Рис. 4.3. Специализированный текстовый редактор

Отличительными особенностями этого редактора являются:

- соответствующие синтаксису языка Colamo атрибуты вывода текста;
- интерактивное взаимодействие управляющими элементами, позволяющее осуществлять быструю контекстную навигацию по тексту;
- вывод на печать с сохранением атрибутов вывода;
- поиск и замена строк.

3. Поля редактора. Служат для индикации различных атрибутов строк параллельной программы, таких как точка останова, предупреждение или ошибка, точка исполнения.

4. Информационная панель редактора текста программ. На панель выводится информация о положении курсора, индикатор режима вставки, индикатор модификации текста (рис. 4.4).

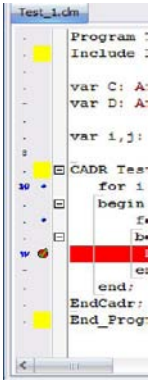


Рис. 4.4. Информационная панель редактора текста программ

5. Список сообщений пользователю. Список строк, формируемый во время трансляции, в котором выводится информация об ошибках и предупреждениях. Возможна навигация по тексту программы через список сообщений (рис.4.5).

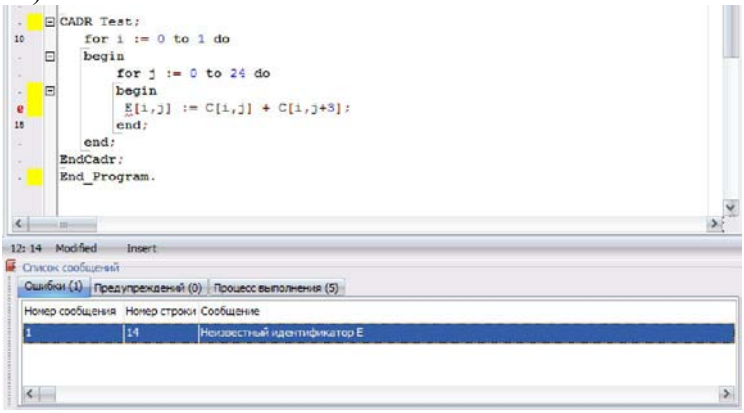


Рис. 4.5. Панель «Список ошибок»

Окно сообщений позволяет переходить к соответствующим позициям в тексте программы. Поля редактора отображают текущее состояние каждой строки. В зависимости от содержания строки могут содержать ошибки или предупреждения.

Интегрированная среда разработки прикладных параллельных программ позволяет одновременно работать с несколькими исходными файлами посредством элемента управления с многостраничной организацией.

Управление процессом разработки осуществляется как через основное меню среды, так и через панель инструментов. Меню «Файл», «Правка», «Поиск», «Вид», «Помощь» содержат общепринятые команды для таких сред.

Интегрированная среда разработки параллельных прикладных программ для реконфигурируемых вычислительных систем обеспечивает широкие сервисные возможности и существенно облегчает работу программиста.

4.2. Параметры интегрированной среды разработки

Интегрированная среда разработки позволяет создавать программы на двух языках: на языке высокого уровня COLAMO и на языке ассемблера ARGUS. Для корректной трансляции создаваемых проектов, в независимости от типа языка, необходимо указать соответствующие параметры в настройках интегрированной среды. Для настройки интегрированной среды разработки необходимо выбрать меню “Конфигурация”. Все параметры интегрированной среды разбиты на пять групп:

- библиотеки;
- компилятор;
- базовые модули
- синтезатор;
- настройки оболочки.

Пункт «Библиотеки» содержит перечень библиотек, подключенных в среде, и набор полей для указания директорий хранения промежуточных и результирующих данных, полученных в результате компиляции (рис. 4.6). Поле «Директория предварительной генерации» указывает директорию для хранения промежуточных данных, генерируемых в результате трансляции проектов на языке COLAMO. В поле «Директория предварительной генерации» указывается путь для хранения результатов, полученных при трансляции проектов на языке ARGUS. Поле «Директория специальных макроопераций» используется при программировании на языке ARGUS и предназначено для хранения специализированных загрузочных файлов, подключаемых в проекте.

В списке «Используемые библиотеки» указывается перечень библиотек, которые могут быть подключены в проекте при программировании на языке COLAMO. Использование библиотек, отсутствующих в данном списке, приведет к синтаксической ошибке на этапе трансляции.

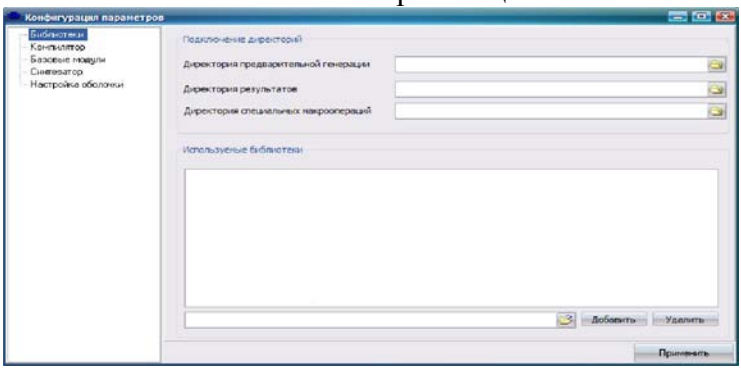


Рис. 4.6 Окно настройки параметров. Пункт «Библиотеки»

Пункт «Компилятор» позволяет установить настройки для трансляторов языка COLAMO и ARGUS. Окно настройки параметров при выборе пункта «Компилятор» показано на рис. 4.7.

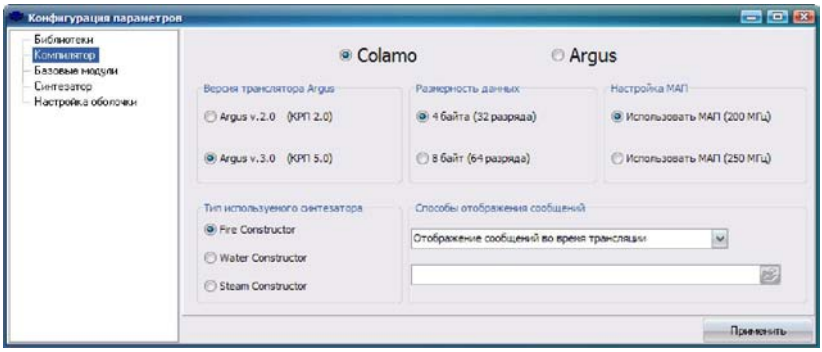


Рис. 4.7 Окно настройки параметров. Пункт «Компилятор»

В данном окне пользователь может выбрать один из языков программирования: COLAMO или ARGUS. Поскольку существует две модификации языка ассемблера ARGUS, то в блоке «Версия транслятора Argus» можно выбрать одну из модификаций в зависимости от типа платы, на которой будет выполняться создаваемая параллельная программа. Блоки «Размерность чисел» и «Настройка МАП» используются при программировании на языке ARGUS и позволяют указать разрядность обрабатываемых данных и тип используемого макропроцессора.

Блок «Тип используемого синтезатора» используется для указания синтезатора, применяемого при трансляции. При трансляции параллельной программы на языке COLAMO используется синтезатор Fire Constructor, а при программировании на языке ARGUS - синтезатор Steam Constructor (используется при программировании на уровне макрообъектов) или Water Constructor (используется при программировании на уровне макропроцессоров).

Блок «Способы отображения сообщений» позволяет выбрать режим вывода сообщений пользователю. Режим «Отображение сообщений во время трансляции» позволяет в реальном времени получать все сообщения, выдаваемые транслятором во время работы. При работе в режиме «Отображение сообщений после трансляции» все сообщения,

полученные в результате работы транслятора, будут отображены в среде только после завершения работы транслятора. Выбор режима «Отображение сообщений в файл» позволяет автоматически сохранять все полученные сообщения в результате трансляции в специализированный файл, расположенный в той же директории, где находится и транслируемый проект.

Пункт «Базовые модули» позволяет описывать параметры базовых модулей и их количество. Окно настройки параметров при выборе пункта «Базовые модули» показано на рис. 4.8.

В полях «Тип базовых модулей» и «Частота работы базовых модулей» указываются тип платы и частота, на которой будет выполняться создаваемая программа. Данные поля используются при трансляции параллельных прикладных программ на языке COLAMO.

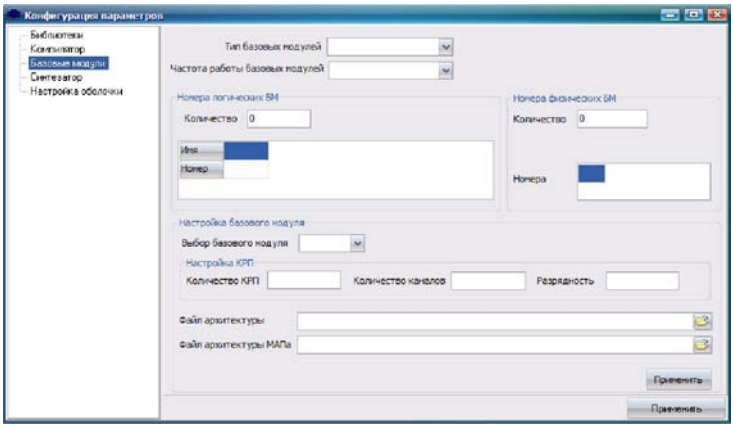


Рис. 4.8 Окно настройки параметров. Пункт «Базовые модули»

Блоки «Номера логических ВМ» и «Номера физических ВМ» используются для указания количества базовых модулей, доступных для реализации создаваемой параллельной программы. Блок «Номера логических ВМ» позволяет для каждого физического номера базового модуля указать логическое обозначение и использовать его при разработке параллельных программ. Такая возможность позволяет

создавать масштабируемые параллельные программы. Параметры в блоке «Настройка базового модуля» позволяют указать настройки для каждого базового модуля в отдельности. Поле «Выбор базового модуля» позволяет выбрать необходимый базовый модуль, а поля «Файл архитектуры», «Файл архитектуры МАПа» и блок «Настройка КРП» определяют основные параметры для выбранного базового модуля.

Пункт “Синтезатор” позволяет настроить работу выбранного синтезатора в пункте «Компилятор». Окно настройки параметров при выборе пункта «Синтезатор» показано на рис. 4.9.

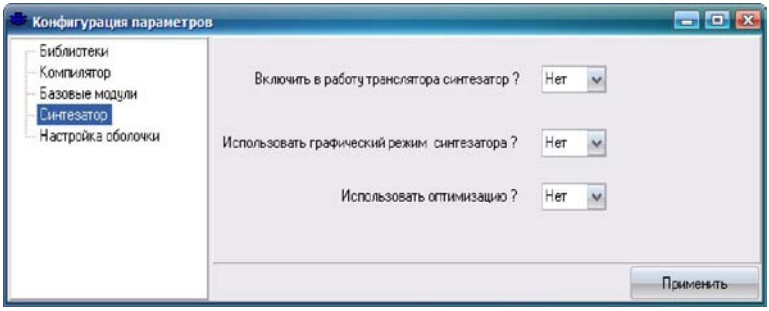


Рис. 4.9 Окно настройки параметров. Пункт «Синтезатор»

Настройки в данном окне позволяют задействовать в процессе трансляции выбранный синтезатор, а также указать необходимость выполнения оптимизации и использования графического интерфейса синтезатора.

Пункт “Настройка оболочки” позволяет пользователю настроить внешний вид текстового редактора, а также выбрать или настроить цветовую схему подсветки операторов используемого языка программирования. Окно настройки параметров при выборе пункта «Синтезатор» показано на рис. 4.10.

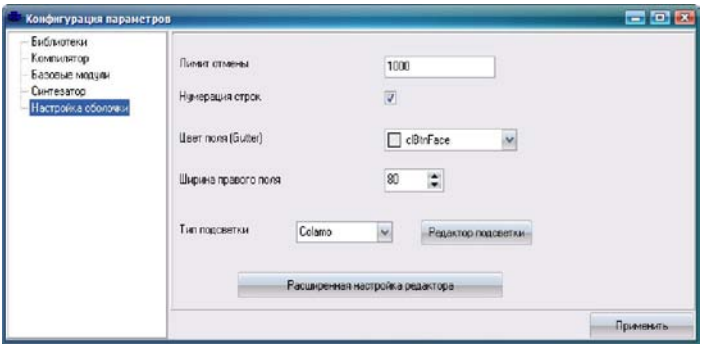


Рис. 4.10 Окно настройки параметров. Пункт «Настройка оболочки»

4.3. Контрольные вопросы

1. Каким образом выбирается язык программирования в интегрированной среде?
2. Каким образом выбирается синтезатор для трансляции параллельной прикладной программы?

ЗАКЛЮЧЕНИЕ

Широкое распространение РВС сдерживается сложностью их программирования и отсутствием удобных средств разработки параллельных прикладных программ. В настоящее время создано более 100 языков параллельного программирования, отличающихся организацией распараллеливания и, как правило, предназначенных для программирования определенной архитектуры вычислительной системы.

Традиционные языки параллельного программирования МВС неэффективны для программирования реконфигурируемых вычислительных систем, поскольку ориентированы на построение управляющих графов программ, в то время как для ПЛИС целесообразна явная реализация информационных графов задачи.

В то же время существующие системы программирования РВС на низком уровне требуют больших временных затрат на создание и отладку прикладной программы, а на высоком уровне не обеспечивают создание прикладных программ, выполняющихся на РВС с высокой удельной производительностью.

Разработанный в НИИ МВС язык высокого уровня COLAMO является удобным и эффективным средством разработки параллельных прикладных программ для РВС, поскольку отражает все особенности структурно-процедурной организации вычислительного процесса и позволяет эффективно реализовать любой тип параллельной обработки. Использование неявного описания параллелизма и виртуальной системы коммутации упрощает программирование, что позволяет оперативно разрабатывать прикладное программное обеспечение.

Рассмотренные в учебном пособии методы организации параллельной и конвейерной обработки данных, а также особенности использования операторов и вычислительных конструкций языка позволят разработчикам создавать

параллельные прикладные программы, выполняющиеся на РВС с высокой удельной производительностью.

ЗАДАЧИ ДЛЯ САМОКОНТРОЛЯ

1) Напишите программу, реализующую суммирование элементов двух массивов со степенью распараллеливания, равной 5.
2) Нарисуйте информационный граф следующей программы:

```
Var a : Array Integer [4 : vector, 100 : Stream] Mem;
Var b, d, f : Array Integer [4 : Stream, 100 : Stream] Mem;
Var e : Integer Mem
Var i, j : Number;
Cadr Example;
For i := 0 to 1 do
  For j := 10 to 50 do
    Begin
      a[i,j] := b[i,j] – e;
      d[i,j] := b[i,j -5] + f[i,j];
    End;
EndCadr;
```

3) Нарисуйте информационный граф следующей программы:

```
Var a,b, c, d, e, f : Array Integer [4 : Stream, 100 : Stream]
Mem;
Var i, j : Number;
Cadr Example;
For i := 0 to 1 do
  For j := 10 to 50 do
    Begin
      a[i,j] := b[i,j] – c[i,j];
      If b[i,j - 5] >= 10 then
        d[i,j] := b[i,j -3] + e[i,j];
      f[i,j] := d[i,j] + c[i,j];
    End;
EndCadr;
```

4) Нарисуйте информационный граф следующей программы:

```
Var a,b, c, d, e, f, t : Array Integer [100 : Stream] Mem;
Var i : Number;
Cadr Example;
  For i := 10 to 80 do
    Begin
      c[i] := a[i] – b[i];
      d[i] := c[i-5] + a[i];
      e[i] := c[i-7] + b[i];
      f[i] := c[i-2] + t[i];
    End;
EndCadr;
```

5) Перепишите предыдущую программу так, чтобы количество временных задержек в информационном графе было минимальным.
6) Нарисуйте информационный граф и определите, является ли данная программа корректной:

```
Var a, b, c, d, e : Array Integer [100 : Stream] Mem;
Var i : Number;
Cadr Example;
  For i := 10 to 80 do
    Begin
      c[i] := a[i] – b[i];
      c[i-5] := a[i-5] + b[i];
      If b[i - 5] >= 20  then
        d[i] := b[i] + e[i];
      Else
        d[i] := b[i -3] + e[i];
    End;
EndCadr;
```

7) Нарисуйте информационный граф следующей программы:

```
Var b, c, e, f, t : Array Integer [100 : Stream] Mem;
Var a, d : Array Integer [100 : Stream] Com;
Var i, j : Number;
Cadr Example;
  For i := 10 to 80 do
```

```
    Begin
      a[i] := b[i] – c[i];
      d[i] := a[i-5] + e[i];
    End;
  For j := 15 to 80 do
      f[j] := d[j-7] + t[j];
EndCadr;
```

8) Определите, является ли данная программа корректной. Если нет, то объясните, почему.

```
Var a,b, c, d, e : Array Integer [100 : Stream] Mem;
Var i : Number;
Let Func (in : a1, b1; c1 out : d1);
Var a1, b1, c1, d1 : Integer Com;
    d1 := (a1 + b1) – c1;
EndLet;
Cadr Example1;
 Func(a[i], b[i], c[i], d[i]);
EndCadr;
Cadr Example2;
 Func(d[i], e[i], c[i], a[i]);
EndCadr;
```

9) Нарисуйте информационный граф следующей программы:

```
Var a,b, c, d, e : Array Integer [100 : Stream] Mem;
Var i : Number;
Let Func (in : a1, b1; c1 out : d1);
Var a1, b1, c1, d1 : Integer Com;
    d1 := (a1 + b1) – c1;
EndLet;
Cadr Example1;
 Func(a[i], b[i], c[i], d[i]);
EndCadr;
Cadr Example2;
 Func(d[i], e[i], c[i], a[i]);
EndCadr;
```

10) Напишите программу, реализующую пирамидальное суммирование элементов семи массивов.
11) Напишите программу, реализующую поиск максимального элемента в массиве.
12) Напишите программу, реализующую умножение матрицы на вектор.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Spartan-3 FPGA Family: Complete Data Sheet [Электронный ресурс] / Xilinx Inc., 2005.

2. Зотов, В.Ю. Проектирование встраиваемых микропроцессорных систем на основе ПЛИС фирмы Xilinx в САПР WebPACK ISE [Текст]: монография / В.Ю. Зотов.- М.: Горячая линия-Телеком, 2006. - 520 с.

3. Левин, М.П. Параллельное программирование с использованием OpenMP [Текст]: монография / М.П. Левин. – Изд-во Бином. Лаборатория знаний, 2008.

4. Эндрюс, Грегори Р. Основы многопоточного, параллельного и распределённого программирования [Текст] = Foundations of Multithreaded, Parallel, and Distributed Programming : [учеб. пособие] / Г.Р. Эндрюс ; пер. с англ. А.С. Подосельника и [др.]; под ред. А.Б. Ставровского. - М. ; СПб. ; Киев : Вильямс, 2003. - 505 с.

5. Воеводин, В.В. Параллельное программирование [Текст]: монография / В.В. Воеводин, Вл.В. Воеводин. – С-Пб. Изд-во «БХВ-Петербург», 2002.

6. Barbara Chapman. Using OpenMP: Portable Shared Memory Parallel Programming [Текст]: Изд-во МИТ Пресс - MIT Press, Cambridge, Massachusetts., 2008.

7. Хьюз, К. Параллельное и распределенное программирование с использованием C++ [Текст]: монография / К. Хьюз, Т. Хьюз.. – Изд-во Вильямс, 2004.

8. Hempel, R. The Argonne/GMD Macros in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model, Feb. 1991.

9. Меткалф, М., Рид Дж. Описание языка программирования Фортран 90 : монография / М. Меткалф, Дж. Рид. - М.: Мир, 1995.

10. Денисов Е.А. Параллельное программирование - новый образ мысли. http://www.rapas.ru/computerra.html.

11. High Performance Fortran Language Specification. High Performance Fortran Forum, May 3, 1993, Version 1.0.

12. Lastovetsky A.: Parallel Computing on Heterogeneous Networks, John Wiley & Sons, New Jersey, 2003, 159-254.

13. http://www.parallel.ru/tech/mpc/mpC-rus.html

14. http://u-pereslavl.botik.ru/~vadim/MCSharp/index.ru.php

15. Сердюк Ю.П., Петров А.В. Параллельное программирование для многоядерных процессоров. Электронная книга. http://www.intuit.ru/department/supercomputing/ppmcp/

16. Джоунз, Г. Программирование на языке Оккам [Текст]: монография / Г. Джоунз; пер. с англ. – М. : Мир, 1989 . – 208 с. – ISBN 5-03-001155-2.

17. Хоар Ч. Взаимодействующие последовательные процессы. - М.: Мир, 1989.-265с.

18. Де Сантис, П Язык Ада [Текст]: монография / Пабло де Сантис. – Изд-во АСТ, 2005.

19. Бен-Ари, М. Языки программирования. Практический сравнительный анализ [Текст]: монография / М. Бен-Ари. – М.: Мир, 2000.

20. Botros, Nazeih HDL. Основы программирования VHDL И Verilog" [Текст]: монография / Nazeih Botros. - Изд-во "Da Vinci инженерно Пресс, 2006. - 506 с.

21. ГОСТ Р 50754-95. Язык описания аппаратуры цифровых систем VHDL. Описание языка [Текст] / Российский НИИ информационных систем (РосНИИ ИС); Всероссийская ассоциация организаций, заинтересованных в применении языка VHDL (ВАЯПС), 01.01.1996.

22. Поляков, А.К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры [Текст]: монография / А.К. Поляков. - Издательство: Солон-Пресс ISBN: 5-98003-016-6, 2003. - 320 с.

23. Максфилд, К. Проектирование на ПЛИС. Курс молодого бойца [Текст]: монография / К. Максфилд. – М.: Издательский дом «Додэка-XXI» 2007.

24. Стешенко, В.Б. ПЛИС фирмы ALTERA: элементная база, система проектирования и языки описания аппаратуры

[Текст] / В.Б. Стешенко // Издание 3-е, стереотипное. - М.: Издательский дом «Додэка_XXI», 2007.

25. www.ImpulseC.com

26. Левин, И.И. Язык параллельного программирования высокого уровня для структурно-процедурной организации вычислений [Текст] / И.И. Левин // Труды Всероссийской научной конференции "Высокопроизводительные вычисления и их приложения", Черноголовка. – М.: Изд-во МГУ, 2000. - С. 108-112.

27. Левин, И.И. Язык программирования вы-сокого уровня для многопроцессорной системы с программируемой архитектурой [Текст]: сборник трудов / И.И. Левин, В.Ф. Гузик, О.О. Сафронов // Распределенная обработка информации. Новосибирск, 1991.

28. Левин, И.И. Ресурсонезависимое параллельное программирование [Текст] / И.И. Левин // Искусственный интеллект. - Донецк: Наука і освіта, 2002. - №3. - С. 277-285.

29. Каляев, И.А. Реконфигурируемые мультиконвейерные вычислительные структуры [Текст]: монография / И.А. Каляев, И.И. Левин, Е.А. Семерников, В.И. Шмойлов; под общ. ред. И.А. Каляева. - 2-е изд., перераб. и доп. - Ростов-на-Дону: Изд-во ЮНЦ РАН, 2009. - 344 с.

30. Левин, И.И. Методы и программно-аппаратные средства параллельных структурно-процедурных вычислений [Текст]: диссертация на соискание ученой степени доктора технических наук: 05.13.11, 05.13.15: защищена 8.10.2004: утверждена 8.04.2005 / Левин Илья Израилевич. – Таганрог, 2004. - 363 с. – Библиогр.: с. 285-300.

31. Левин, И.И. Семантические особенности описания переменных на языке программирования COLAMO [Текст] / И.И. Левин, А.И. Дордопуло, В.А. Гудков // Труды Третьей международной научной конференции «СУПЕРКОМПЬЮТЕРНЫЕ СИСТЕМЫ И ИХ ПРИМЕНЕНИЕ. SSA’2010». Республика Беларусь, Минск. – С. 176-179.

32.http://www.oracul.kiev.ua/index.php?page=news&news_id=222

33. Левин, И.И. О языке макроассемблера комплекта БИС с програм-мно-перестраиваемой структурой [Текст] / И.И. Левин, О.О. Сафронов // Архитектура ЭВМ и машинное моделирование. – Таганрог: Изд-во ТРТИ, 1989.

34. Гудков, В.А. Расширения структурно-процедурного языка программирования ARGUS для многопроцессорных вычислительных систем с макрообъектной архитектурой [Текст] / В.А. Гудков // Материалы Третьей ежегодной научной конференции студентов и аспирантов базовых кафедр ЮНЦ РАН. – Ростов/Д: Изд-во ЮНЦ РАН, 2007. – С. 135-136.

35. Каляев А.В. Модульно-наращиваемые многопроцессорные системы со структурно-процедурной организацией вычислений [Текст]: монография / А.В. Каляев, И.И. Левин; под ред. А.В. Каляева. - М.: Янус-К, 2003. - 380 с.

36. Клоксин, У. Программирование на языке Пролог [Текст]: монография / У. Клоксин, К. Меллиш. – М.: Мир, 1987.

37. Стерлинг, Л. Искусство программирования на языке Пролог [Текст]: монография / Л. Стерлинг, Э. Шапиро. - М.: Мир, 1990. - 235 с.

38. Гудков, В.А. Методы трансляции операторов условного перехода на языке COLAMO в структурную составляющую [Текст] / В.А. Гудков // Материалы Международной научно-технической конференции «Суперкомпьютерные технологии: разработка, программирование, применение (СКТ-2010)». Т.1. – Таганрог: Изд-во ТТИ ЮФУ, 2010. – С.202-206.

39. Гудков, В.А. Реализация операторов условного перехода в структурной составляющей параллельной программы на языке COLAMO [Текст] / В.А. Гудков // Материалы Седьмой Международной научной молодежной школы «Высокопроизводительные вычислительные системы». – Таганрог: Изд-во ТТИ ЮФУ, 2010. – С.196-201.

Илья Израилевич Левин
Алексей Игоревич Дордопуло
Вячеслав Александрович Гудков

ПРОГРАММИРОВАНИЕ РЕКОНФИГУРИРУЕМЫХ ВЫЧИСЛИТЕЛЬНЫХ УЗЛОВ НА ЯЗЫКЕ COLAMO

Ответственный за выпуск
Редактор Селезнева Н.И.
Корректоры: Чиканенко Л.В., Надточий З.И.

ЛР№020565 от 23 июня 1997 г.

Формат 60x841/16.	Подписано к печати -28.04.2011г.
Печать офсетная	Бумага офсетная.
Усл. п.л. - 5,6.	Уч.- изд-л. - 5,5.
Заказ №	Тираж 50 экз.

"С"

Издательство Технологического института
Южного федерального университета
ГСП 17А, Таганрог, 28, Некрасовский, 44
Типография Технологического института
Южного федерального университета
ГСП, 17А, Таганрог, 28, Энгельса, 1