Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Пермский национальный исследовательский
политехнический университет»

.

О.В. Гончаровский

ПРОЕКТИРОВАНИЕ ВСТРОЕННЫХ УПРАВЛЯЮЩИХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Издательство
Пермского национального исследовательского
политехнического университета
2013

УДК 004.415.2
ББК 32.973.202-018.2

Рецензенты:

Начальник отдела ТО-5 ОАО «Стар»,
кандидат технических наук *С.В. Березняков;*

кандидат технических наук, доцент *Т.С. Леготкина,*
Пермский национальный исследовательский политехнический университет

Гончаровский О.В.
Проектирование встроенных управляющих систем реального времени: учеб. пособие / О.В. Гончаровский, Н.Н.Матушкин, А.А. Южаков – Пермь: Изд-во Перм. нац. исслед. политехн. ун-та, 2013. – 1хх с

ISBN ххх-х-ххх-хххх-х

В учебном пособии рассмотрены организация аппаратной части встроенных микропроцессорных систем. Рассмотрена методология проектирования компонент встроенных систем на программируемой логике по модели программно-управляемого автомата. Рассмотрена методология проектирования прикладного программного обеспечения встроенных систем ориентированная на модель, позволяющая освоить разработку функциональных спецификаций для систем управления двигателями летательных аппаратов.

Предназначено для студентов, обучающихся по профилю подготовки магистров 22040056.68 «Информационные технологии в проектировании управляющих систем реального времени».

Учебное пособие может быть полезно студентам смежных направлений подготовки.

УДК 004.415.2
ББК 32.973.202-018.2

ISBN ххх-х-ххх-хххх-х ПНИПУ, 2013

Оглавление

ПРЕДИСЛОВИЕ

Дисциплина «Проектирование встроенных управляющих систем реального времени» является частью сетевой магистерской программы 22040056.68 «Информационные технологии в проектировании управляющих систем реального времени» по направлению 220400 «Управление в технических системах».

Сетевой характер программы обусловлен стремлением к повышению качества подготовка за счет объединения усилий (материальной базы и квалификации специалистов) нескольких учебных заведений. Современный подход к проектированию встроенных управляющих систем реального времени опирается на парадигму модельно-ориентированного проектирования их прикладного программного обеспечения. В этой области существуют различные инструменты, позволяющие автоматически генерировать код приложения на том или ином языке программирования (обычно Си). Участвующие в сетевой программе университеты ориентируются на различные объекты управления и используют различные средства разработки программного обеспечения. Теоретические основы курса являются общей базой для освоения различных средств разработки. На лабораторных занятиях в ПНИПУ применяется интегрированная среда разработки SCADE Suite, используемая ведущими мировыми авиастроительными компаниями для разработки безопасных приложений встроенных систем управления и оригинальная аппаратная мобильная платформа, делающая максимально наглядным результат проделанной работы. В академических кругах SCADE Suite пока еще широко не используется, и немногие российские университеты обладают опытом его применения.

Объектом изучения дисциплины являются встроенные микропроцессорные системы. Большинство учебников по встроенным системам уходят своими корнями в небольшие микропроцессорные системы на базе микроконтроллеров, в которых **встроенная система определяются как система обработки информации, включенная в состав изделия**. Она рассматривается как часть законченного изделия, выполняющая специальные функции (характерны только для этого изделия). Эти функции реализуются методами и средствами информационных технологий.

Современное определение встроенных систем ориентируется на системы с широкими возможностями сравнимыми с компьютерами общего назначения. Согласно [1] **встроенная система – это компьютер с перечнем ограничений большим, чем компьютер общего назначения**. Характерные ограничения касаются предназначения приложения, форм-фактора, потребляемой мощностью, системных ресурсов и функций, а также предположений о поведении пользователей.

Встроенные системы обычно разрабатываются для оной прикладной задачи или класса таких задач и имеет предопределенное множество ресурсов и функций .

Встроенные системы от остальных систем обработки информации отличаются такими характеристиками как:

• Повышенная надежность.

• Повышенная безопасность.

• Критичность к реальному времени.

• Структура стоимости: периодические издержки на разработку аппаратной части (решающее значение для потребительского рынка); разовые затраты на разработку программного обеспечения (возможны ограничения на время вывода нового изделия на рынок).

• Ограничения, которых обычно нет для настольных компьютеров: объем памяти, среда разработки, потребляемая мощность, интерфейс оператора.

• Быстродействие процессора не является гарантией времени отклика.

• Интерфейс с окружающим миром через сенсоры и исполнительные механизмы.

• Смешанное поведение – взаимодействие с непрерывной окружающей средой с помощью логических функций и конечных автоматов.

Результаты исследования структуры рынка встроенных систем говорят о том, что три четверти встроенных систем это системы реального времени, т.е. системы с ограничением на время решения задачи и больше половины являются распределенными системами.

Выделяют 4 основных типа встроенных систем, определяемых областью применения (или спецификой решаемых задач):

• Управление: автоматизация технологических процессов, аэрокосмическая техника, автомобильная техника, бытовая техника.

• Системы связи: мобильные телефоны, сетевое оборудование и оборудование систем передачи.

• Обработка аудио и видеосигналов.

• Карманные компьютеры и портативные игры.

В этих системах микропроцессорные средства чаще всего выполняют функции, которые можно сгруппировать следующим образом.

• Реализация классических законов управления, нечёткого управления (Fuzzy logic) и других.

• Реализация последовательностной логики: конечные автоматы, модули переключения между различными законами управления.

• Обработка сигналов: сжатие данных мультимедиа, цифровая фильтрация, гармонический анализ.

• Специализированное сопряжение с кнопками, звуковыми и световыми индикаторами, высокоскоростным вводом-выводом.

• Диагностика: обнаружение неисправностей и реконфигурация.

Характеристики встроенных систем, выполняемые ими функции, сделали их применение тотальным, охватывающим практически все области человеческой деятельности.

В учебном пособии рассматривается первый тип встроенных систем, а именно управляющие системы реального времени. На рис. 1 приведена структура аппаратных средств управляющей систем реального времени.

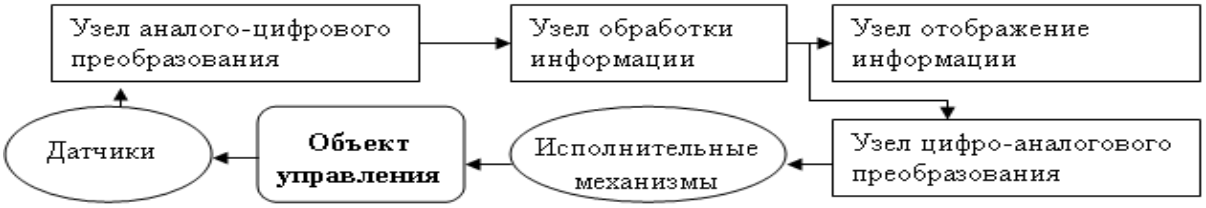


Рис.1. Структура аппаратных средств управляющей систем реального времени

Сигналы о состоянии объекта управления поступает во встроенную систему в общем случае через аналоговые датчики. Преобразование этих сигналов в цифровую форму выполняет узел *аналого-цифрового преобразования* (АЦП или ADC), в состав которого входит устройство выборки-хранения (УВХ). Узел обработки информации в соответствии с целевой функцией формирует сигналы для устройства отображения информации и для воздействия на объект управления через узел *цифро-аналогового преобразования* (ЦАП или DAC) и исполнительные механизмы (в общем случае аналоговые устройства).

Узел обработки информации является основным элементом встроенных систем. Наиболее важное ограничение для него – это мощность, уменьшение потребления которой приводит к уменьшению размеров источника питания, регулятора напряжения, системы охлаждения и уменьшению потребления энергии. Уменьшение потребления энергии важно для мобильных устройств, т.к. технологии производства батарей совершенствуются очень медленно и, следовательно, стоимость энергии остается высокой. Также уменьшение потребления энергии уменьшает требования к охлаждению и увеличивает надежность (время жизни электронных устройств уменьшается при повышении температуры).

Узлы обработки информации могут быть реализованы с помощью специализированных схем, микропроцессорных систем и программируемой логики.

Для реализации задач, требующих высокой производительности и большого рынка сбыта могут быть разработаны специализированные схемы (полностью заказные СБИС, Application-Specific Circuits – ASICs). Однако стоимость разработки и изготовления таких кристаллов может быть очень большой. Например, стоимость комплекта масок для формирования топологии на кристалле может достигать 100000 – 1000000 долларов. Остается фактом экспоненциальный рост в последние годы стоимости комплекта масок. Кроме того этот подход имеет длительный период проектирования при отсутствии гибкости: корректировка ошибок проекта обычно требует нового комплекта масок и нового запуска производства.

Следовательно, специализированные схемы подходят, только если требуется максимальная энергетическая эффективность и большое количество таких устройств. По этой причине они не рассматриваются в учебном пособии.

В случае микропроцессорной реализации управляющих систем реального времени важной является предсказании производительности системы. Основная проблема состоит в том, что необходимо иметь наихуд-

шие сведения о системе раньше, чем средние значения о ней. Такая информация необходима для гарантии реализации ограничений реального времени. Для получения информации о производительности уже на ранних стадиях проектирования было предложено два различных класса техник – это предполагаемые и точные значения производительности. Техника точного значения использует реальный программный код (в бинарной форме) на близкой к реальной аппаратной платформе. Это возможно только если существуют интерфейсы с инструментами разработки аппаратного и программного (компиляторам) обеспечения. Препятствиями для точного прогнозирования на этапе проектирования являются эффекты использования в архитектурах современных процессоров конвейеров, кэш памяти, механизмов прерываний виртуальной памяти. Вот почему этим вопросам уделяется внимание в первом разделе учебного пособия.

Предметом изучения дисциплины является разработка архитектуры платформы встроенной управляющей системы реального времени и создание прикладного программного обеспечения, основанное на модели.

Проектирование встроенных систем является сложной задачей, которая разбивается на ряд подзадач приемлемой сложности [2]. Эти подзадачи решаются последовательно друг за другом, но часто решения некоторых из них выполняются итеративно. Проектирование начинается с идей. Идеи должны включать знания прикладной области, а затем должны быть трансформированы в спецификацию. Также необходимы сведения о стандартных компонентах аппаратного и системного программного обеспечения. Эти компоненты по возможности многократно используются в проекте. Вся информация сохраняется в репозитории проекта.

С использованием репозитория проектные решения могут приниматься в итерационной манере. Во время проектных итераций прикладные задачи отображаются на исполнительную платформу, и генерируется новая проектная информация. Генерация включает в себя отображение операций на одновременные задачи, отображение операций или на аппаратуру или на программное обеспечение, компиляцию и планирование задач. В общем случае отображение выполняется на мультипроцессорную систему, принадлежащую к одному из двух классов.

Однородные (гомогенные) мультипроцессорные системы: все процессоры системы обеспечивают одинаковую функциональность. Это вариант многоядерной архитектуры персональных компьютеров. Совместимость программ различных процессоров – ключевое преимущество, которое используется планировщиком задач во время работы программ, а также при проектировании с целью достижения отказоустойчивости. Также упрощается разработка платформы и инструментов проектирования из-за однотипности процессоров.

Неоднородные (гетерогенные) мультипроцессорные системы: процессоры принадлежат к различным типам, что обеспечивает большую эффективность системы.

Задача проектирования на этапе отображения формулируется следующим образом.

Дано:

1. Множество прикладных задач.
2. Примеры, описывающие, как эти задачи будут использоваться.
3. Множество архитектур-кандидатов:

– процессоры (возможно неоднородные);
– архитектуры коммуникаций (возможно неоднородные);
– возможные политики планирования выполнения задач.

4. Технические требования:

– сохранение времени выполнения задачи и/или максимизировать производительность.
– минимизация цены, энергопотребления, …

Найти:

1. Отображение задач на процессоры.
2. Варианты планирования выполнения задач.
3. Архитектуру.

Проектные решения должны быть *оценены*(evaluation) по отношению к различным техническим требованиям, таким как производительность, энергосбережение, технологичность и т.д. Оценка проекта это процесс вычисления ключевых количественных характеристик системы.

При современном состоянии искусства проектирования ни один этап не гарантирует корректный результат. Следовательно, необходима *валидация* (validation) проекта, т.е. действия по проверке соответствует или нет проект его целям, удовлетворяет или нет всем ограничениям, и будет ли выполняться, как задумано. Валидацию, выполняемую с математической строгостью называют *верификацией* (формальной). Из-за важности такого качества встроенных систем как эффективность, значимой становится оптимизация проекта.

Трудоемкость дисциплины составляет 180 часов, из них лекции 6 часов, практических занятий 18 часов, лабораторный практикум 16 часа и самостоятельная работа 100 часов.

Цель учебной дисциплины состоит в освоении разработки аппаратной платформы и современных методов проектирования программного обеспечения встроенных управляющих систем реального времени.

Учебное пособие призвано обеспечить формированию следующих дисциплинарных профессиональных специальных компетенций относящихся к этой области деятельности магистров:

– Способность разрабатывать структурную схему аппаратной платформы встроенной управляющей системы реального времени (ПСК1-1).

– Способность использовать интегрированную среду разработки для модельно-ориентированного проектирования прикладного программного обеспечения встроенных управляющих систем реального времени (ПСК2-1).

Учебное пособие разбито на два раздела. Первый раздел посвящен аппаратным средствам встроенных систем, а второй – программному обеспечению встроенных систем. Разделы содержат примеры. После каждого

раздела приведены вопросы для самоконтроля. Завершает учебное пособие набор заданий для поддержки формирования элементов компетенций «умеет».

Задача первого раздела учебного пособия состоит в формировании таких компонент компетенции ПСК-1-1 как:

– Знать архитектуру процессорных узлов встроенных систем.

– Знать типы и формы параллелизма процессорных узлов.

– Знать иерархию и технологии памяти встроенных систем.

– Знать структуру контроллеров прерываний, устройств и интерфейсов ввода-вывода.

– Уметь разрабатывать структурную схему аппаратной платформы встроенной управляющей системы реального времени.

Задача второго раздела учебного пособия состоит в формировании таких компонент компетенции ПСК-2-2 как:

– Знать содержание и взаимосвязи этапов проектировании встроенных систем управления реального времени.

– Знать автоматные и потоковые модели вычислений встроенной системы, язык программирования Scade и интегрированную среду разработки прикладного программного обеспечения SCADE.

– Уметь создавать автоматные модели для встроенных систем.

– Уметь создавать потоковые модели и их комбинации для встроенных систем.

Учебное пособие состоит из двух глав. Первая глава посвящена элементам необходимым для разработки структурной схемы аппаратной платформы встроенной управляющей систем реального времени. Вторая глава посвящена разработке прикладного программного обеспечения управляющей системы реального времени основанной на модели.

1. АППАРАТНАЯ ПЛАТФОРМА ВСТРОЕННОЙ УПРАВЛЯЮЩЕЙ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Программисты встроенных систем должны уметь читать схемы аппаратных платформ, на которых работают его приложения. Нет необходимости понимания деталей тактирования или управления питанием проекта, но как минимум необходимо понимание платформы на уровне блоков (структурная схема): какие используются интерфейсы для подключения устройств, какой вывод порта общего назначения используются для связи со светодиодным индикатором и так далее. Всегда предпочтительнее обращаться к схеме, чем к документации, описывающей встроенную систему на высоком уровне (она может содержать ошибки). Чтение схемы подобно чтению исходного кода – маловероятно, что то, что вы видите в схеме, не представляет реальность.

Разработчик «железа» встроенных систем делает выбор среди большого разнообразия аппаратных платформ. Вопрос, который часто встает – какая платформа наилучшим образом удовлетворяет заданному приложению? Решение, обычно сделанное экспертом, находится среди близких друг другу платформ. Поэтом большой интерес к тому, как это делается. Какие знания и умения для этого необходимы? Первый раздел поможет дать ответ на это, рассматривая вопросы организации и архитектуры аппаратных платформ встроенных систем.

Под аппаратной платформой понимают группу совместимых микропроцессорных систем (компьютеров), которые могут выполнять одинаковые программы и служат как основа или база для создания близких по назначению систем. Взаимозаменяемым понятием часто выступает **аппаратная среда** (environment) – определенная конфигурация аппаратных средств.

“a platform is a set of rules and guidelines of the hardware and software architecture, together with a suite of building blocks that fit into that architecture.”

Платформа – это набор правил и руководств по архитектуре аппаратного и программного обеспечения, вместе с набором блоков, которые входят в архитектуру.

Ключевой характеристикой всех встроенных систем является то, что они спроектированы для выполнения специфической задачи или функции. Программное обеспечение, спроектированное для платформы, является специфичным для всей этой функции устройства. Во многих случаях свойства компьютера системы не видимы для пользователя, т.е. только функция, обеспечиваемая системой, является его содержанием. Конечный пользователь встроенных систем обычно сам не устанавливает в них приложения (хотя возможности по замене приложения на новую версию могут иметь место).

Платформы встроенных систем покрывают огромное число устройств от домашних беспроводных роутеров до сложных навигационных и мультимедиа систем легковых автомобилей и промышленных управляющих си-

стем роботизированной сборки таких автомобилей. Характеристики платформ образуют множество различных требований к их интерфейсам и производительности. При разработке встроенной системы наиболее вероятным является выбор между различными SoC-устройствами (системы на кристалле), включающими только те периферийные компоненты, которые необходимые конкретному приложению. На рис. 1.1 приведена характерная платформа на основе SoC [1].

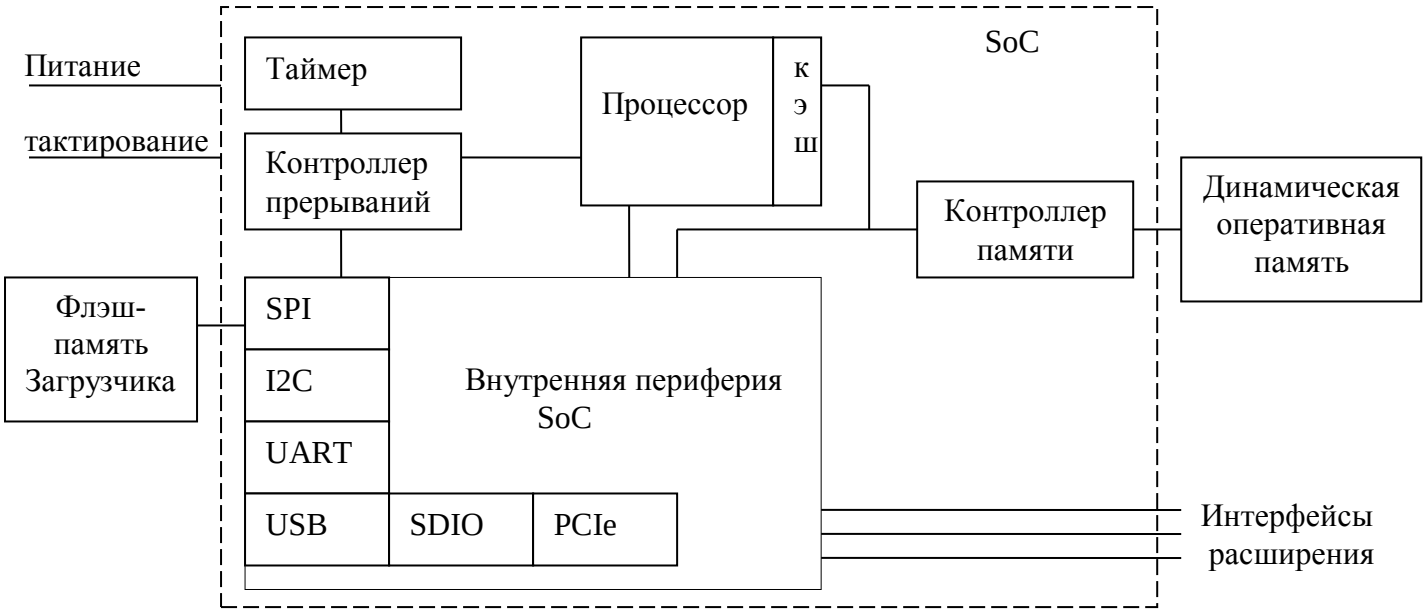


Рис. 1.1. Пример платформы на основе SoC

Встроенные системы имеют хотя бы один процессор (CPU - центральный процессорный узел) – центр всей платформы. CPU является первичным исполнительным окружением прикладной программы (приложения). CPU обычно исполняет код операционной системы, которая запускает приложения необходимые для работы платформы.

. В современных системах встроенные SoC иногда содержат дополнительные специализированные процессорные узлы (DPU), разработанные для специфических приложений. Например, в современных смартфонах CPU выполняет приложения видимые пользователю, в то время как другой DPU (baseband processor – процессор передачи в основной полосе частот) исполняет программы стека протоколов беспроводной связи. В других платформах DPU занимаются обработкой звука и изображения с камеры. Программы, исполняемые DPU, называют «зашитыми» (firmware). Они не требуют операционной системы

Процессоры современных систем 32-битные. Это означает, что все регистры внутри процессора содержат максимально 32 разряда. Встроенные системы с низкой производительностью часто используют 16-битные или даже 8-битные микроконтроллеры, но увеличение объема работы и требования по сетевому взаимодействию требуют для управления в таких платформах 32-битных процессоров. Аналогичным образом, когда требуется высокая производительность или большой объем памяти, появляются 64-битные системы.

На рис.1.2 приведена структурная схема простой микропроцессорной системы (MPS). Эта схема является частью понятия *организации MPS*, которое включает состав программно-аппаратных средств, связи между ними и их функциональные характеристики. Кроме CPU простая MPS содержит основную память (Main Memory – MM) для хранения программ и данных, *устройства ввода-вывода* (I/O), связывающие MPS с внешним миром и *магистраль* (System Bus – SB), объединяющую эти три компонента в единое целое. Основная память в общем случае включает в себя два типа устройств: *оперативные запоминающие устройства* (RAM – ОЗУ) и *постоянные запоминающие устройства* (ROM – ПЗУ).

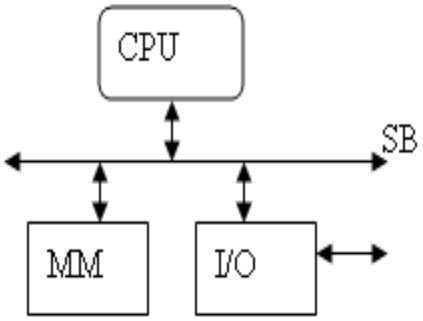


Рис.1.2. Структурная схема простой микропроцессорной системы

CPU считывает из MM команды, образующие программу, декодирует их. Команда – это элементарное действие, которое определяется типом используемых данных, источником их получения, операцией над ними, приемником результата, а также источником получения следующей команды. По результату декодирования ПУ выполняет выборку данных из MM или I/O, выполняет их обработку и пересылает результат обратно в MM или I/O – программно-управляемый обмен. Передача данных между MM и I/O, минуя CPU, выполняется по каналу прямого доступа в память.

С точки зрения CPU MM и I/O представляют собой линейно упорядоченный набор байт со своими номерами (адресами). Диапазон значений адресов памяти, которые может сформировать CPU для передачи по магистрали, называют логическим (линейным или исполнительным) адресным пространством. Диапазон адресов, реализованных в MM и I/O, называют физическим адресным пространством.

Для хранения программ и данных может использоваться одно пространство памяти. Такая организация получила название *архитектуры Дж. фон Неймана* – кодирование программ в формате, соответствующем формату данных. Программы и данные хранятся в едином пространстве, и нет никаких признаков, указывающих на тип информации в ячейке памяти.

Для хранения программ и данных может использоваться одно пространство для хранения программ, а другое для хранения данных. Такая организация получила название *архитектуры Гарвардской лаборатории*. Память программ и память данных разделены и имеют свои собственные адресные пространства и способы доступа к ним.

На рис.1.3 приведена обобщенная структурная схема CPU. Узел интерфейса (УИ) обеспечивает доступ к магистрали MPS.

Кэш память предназначена для улучшения взаимодействия основной памяти и процессорного ядра.

Узел управления памятью (УУП или MMU) в общем случае предназначен для организации виртуального адресного пространства.

Узел загрузки/сохранения (УЗС) выполняет команды загрузки и сохранения данных, обеспечивая пересылки данных между регистрами и памятью.

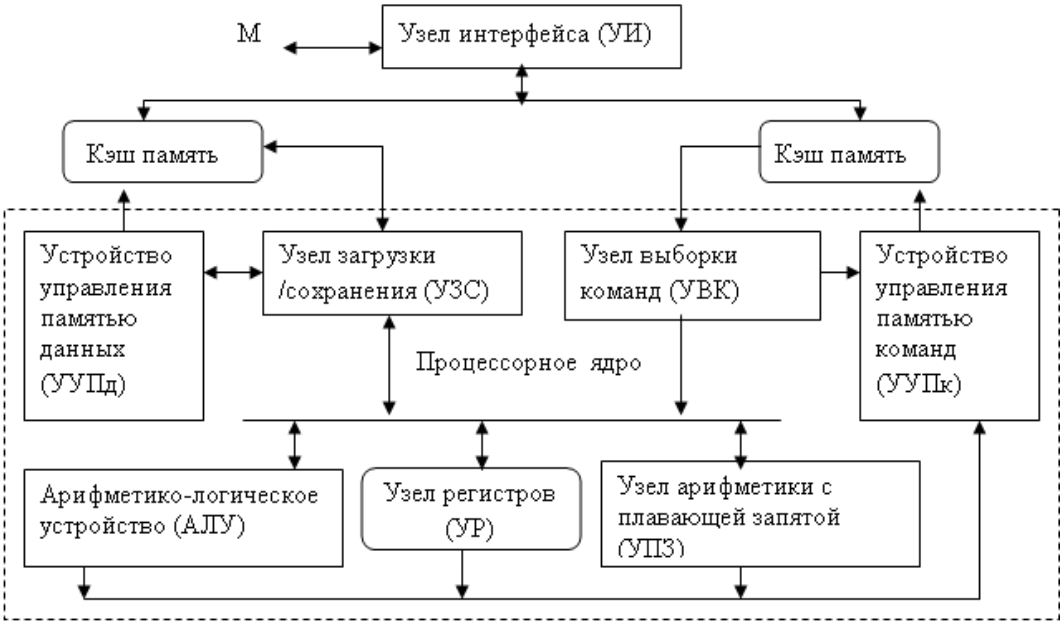


Рис.1.3. Структурная схема процессора

Арифметико-логическое устройство (АЛУ или ALU) выполняет операции целочисленной арифметики и логические операции над данными из внешней памяти или узла регистров (УР), а узел арифметики с плавающей запятой (УПЗ) выполняет операции над данными, представленными в одноименном формате.

Узел регистров предназначен для хранения данных и адресов внешней памяти процессора.

Среди регистров CPU выделяется *счетчик команд* (PC – Program Counter), находящийся в УВК и указывающий на адрес команды, которую необходимо выполнить. В УВК находится также *регистр команд* (IR – Instruction Register), содержащий команду, выполняемую в данный момент.

CPU выполняет каждую команду за несколько шагов:

1. Вызывает следующую команду из памяти и переносит ее в регистр команд.

2. Меняет содержимое счетчика команд, который теперь указывает на следующую команду.

3. Определяет тип выбранной команды.

4. Если команда использует слово из памяти, определяет, где находится это слово.

5. Передает слово, если это необходимо, в регистр CPU.

6. Выполняет команду.

7. Переходит к шагу 1 для выполнения следующей команды.

Такую последовательность шагов называют выборка-декодирование-исполнение.

1.1. Архитектура процессорных узлов встроенных систем

Под *архитектурой* CPU понимают функциональные возможности CPU, используемые для представления программ и данных, а также для управления процессом вычислений.

Согласно [3] архитектура процессора определяется следующими элементами:

1. *Множество команд*. Множество команд в общем случае содержит группу команд передачи данных, группу команд целочисленной арифметики, группу команд арифметики с плавающей точкой, группу логических команд, группу команд управления последовательностью выборки, группу специальных команд. Множество команд определяет также форматы кодирования команд, способы адресации данных.

2. *Программная модель*. Программная модель определяет множество регистров и соглашение о памяти, включающее детали нумерации бит и байт и как данные (целые или в форме с плавающей запятой) хранятся в памяти.

3. *Модель памяти*. Модель памяти определяет размер адресного пространства и разбиение адресного пространства на блоки (сегменты) и страницы, порядок байт, когерентность памяти и различные виды ее защиты.

4. *Модель прерываний*. Модель прерываний определяет полное множество прерываний и условий, которые генерируют прерывания. Модель прерываний специфицирует характеристики прерываний, такие как определенные или неопределенные, синхронные или асинхронные, маскируемые или не маскируемые. Модель прерываний определяет вектора прерываний и множество регистров, которые задействуются при прерываниях. Модель прерываний также обеспечивает адресное пространство для реализации специфических прерываний.

5. *Модель управления памятью*. Модель управления памятью определяет ее разбиение, конфигурирование и защиту. Модель управления памятью также специфицирует выполнение трансляции адресов (преобразование логического адреса в физический или действительный), логическое, виртуальное и физическое адресное пространство, специфические управляющие команды и другие характеристики.

6. *Модель хранения времени*. Модель хранения времени определяет средства, позволяющие устанавливать время суток, а также ресурсы и механизмы, необходимые для поддержки исключений, зависящих от времени.

1.1.1. Множество команд

Команда состоит из кода операции и некоторой дополнительной информации, например, откуда поступают операнды и куда должны отправляться результаты - это формат команды.

Через код операции передается информация о выполняемых командой действиях.

Механизм определения, где находятся операнды (их адреса), называют *адресацией*. В команде может присутствовать ноль, один, два или три адреса.

В одних процессорах все команды имеют одинаковую длину; в других команды могут быть разной длины. Если все команды одной длины, то это упрощает декодирование, но часто требует большего пространства, поскольку все команды должны бать такой же длины, как самая длинная.

Множество команд процессора часто называют *системой команд*. Это множество можно разбить на несколько подмножеств, содержащих однотипные команды: команды пересылки данных, арифметические и логические команды, команды ветвления и команды управления процессором.

Разработка кодов операций является важной частью архитектуры. Однако значительное число битов программы используется для того, чтобы определить, откуда нужно брать операнды, а не для того, чтобы узнать, какие операции нужно выполнить.

Самый простой способ определения операнда – включить в адресную часть команды сам операнд, а не адрес операнда или другую информацию, описывающую, где находится операнд. Такой операнд называют непосредственным операндом, а адресацию - *непосредственной адресацией*. Этим способом можно работать только с константами, зато не требуется обращения к памяти для чтения операнда.

Следующий способ определения операнда – просто дать его полный адрес в команде. Такой способ называют *прямой адресацией*. Этот способ можно использовать только для доступа к переменным, адреса которых известны заранее.

Регистровая адресация сходна с прямой адресацией, только вместо адреса ячейки ММ указывается адрес регистра. Этот способ адресации является самым распространенным.

При *косвенной регистровой адресации* определяемый операнд берется из памяти или отправляется в память, но адрес не зафиксирован жестко в команде, как при прямой адресации. Вместо этого адрес содержится в регистре и его называют указателем. Преимущество косвенной адресации состоит в том, что можно обращаться к памяти, не имея в команде полного адреса.

Индексная адресация предполагает, что адрес памяти является суммой указателя и небольшого смещения в самой команде.

Относительная индексная адресация предполагает, что адрес памяти вычисляется путем суммирования значения двух регистров и смещения.

1.1.2. Программная модель

1.1.2.1. Регистры

Все CPU содержат набор регистров, обеспечивающий быстрый доступ к часто используемым данным, избегая обращения к ММ. Другими словами регистры – это небольшая сверхоперативная память с минимальным временем доступа.

Регистры можно разделить на *адресные регистры* (АР или AR - Address Register), регистры данных (РД или DR – Data Register), *регистры общего назначения* (РОН или GPR – General Purpose Register) и специальные регистры (СР или DR – Dedicated Register).

Адресные регистры содержат адреса ММ. Регистры данных содержат промежуточные результаты вычислений и могут специализироваться на хранении различных типов данных (целых чисел или в формате с плавающей запятой).

Регистры общего назначения содержат как адресную информацию, так и данные. В одних CPU РОН полностью симметричны и взаимозаменяемы, а в других - могут быть специализированы.

Специальные регистры включают счетчик команд и другие регистры с особой функцией (регистры специальных функций). Некоторые СР доступны только в особом режиме работы CPU – привилегированном или супервизорном. Эти регистры используются только *операционной системой* (ОС) – специальной программой, управляющей выполнением пользовательских программ. Есть один специальный регистр, который представляет собой привилегированно-пользовательский гибрид. Это флажковый регистр или PSW (Program State Word – *слово состояния программы*). PSW содержит различные биты, которые нужны для работы CPU. Самые важные биты - это коды условий. Они устанавливаются в каждом цикле с участием АЛУ и отражают состояние результата выполненной операции. Биты кода условий включают:

N – результат отрицательный (Negative);

Z – результат равен 0 (Zero);

C – перенос из самого левого бита (Cary out);

V – результат вызвал переполнение (oVerflow);

AC – перенос из третьего бита (Auxiliary cary – служебный перенос);

P – результат четный (Parity).

Коды условий используются в командах условного перехода. Другие поля PSW указывают режим CPU (пользовательский или привилегированный), трассовый бит (используется для отладки), уровень приоритета процессора, а также статус разрешения прерываний.

Указатель стека (SP – Stack Pointer) содержит адрес вершины стека. *Стек* - это память магазинного типа с дисциплиной обслуживания LIFO, организованная в ММ и содержащая данные и/или адреса возврата. Стек может содержать некоторую структуру данных, местоположение которой определяется двумя указателями SP и LV, где LV – нижняя граница распо-

ложения данных в стеке. Такую структуру называют фреймом (кадром) стека.

В некоторых процессорах только часть РОН «видны» программе в любой момент времени. Эта особенность, называемая регистровыми окнами. Она предназначена для повышения эффективности вызова подпрограмм. Регистровые окна имитируют стек. То есть существует несколько наборов регистров, точно также как и несколько фреймов в стеке. Специальный регистр CWP (Current Window Pointer – указатель текущего окна) содержит номер регистрового окна, доступного программе. Команда вызова подпрограммы такого процессора скрывает старый набор регистров и путем изменения CWP предоставляет новый набор, который может использовать вызванная подпрограмма. Однако некоторые регистры могут перекрываться, что обеспечивает эффективный способ передачи параметров между подпрограммами.

1.1.2.2. Типы данных

Процессоры поддерживают различные типы данных на уровне системы команд. Типы данных можно разделить на две категории: числовые и нечисловые.

Среди числовых типов данных главными являются целые числа различной длины (8,16,32 или 64 бита). Некоторые процессоры поддерживают целые числа и со знаком и без знака. Знаковый бит является старшим битом слова, а отрицательные числа представляются в дополнительном коде.

Нечисловые типы данных используются для представления символьной информации. Наиболее распространенными символьными кодами являются ASCII (7-битовый символ) и UNICODE (16-битоаые символы). На уровне команд часто имеются особые команды, предназначенные для операций с цепочками символов. Эти цепочки иногда разграничиваются специальными символами в конце. Вместо этого для определения конца цепочки может использоваться поле длины цепочки. Команды могут выполнять копирование, поиск, редактирование цепочек и другие действия.

1.1.3. Модели памяти

1.1.3.1. Адресное пространство

Процессор взаимодействует с памятью, разделенной на ячейки с последовательными адресами. Наиболее распространенный размер ячейки 8 бит или байт.

Байты обычно группируются в 4-байтовые или 8-байтовые слова. Некоторые процессоры требуют, чтобы слова были выровнены в своих естественных границах. Так, например, 4-байтовое слово может начинаться с адреса 0,4,8 и т.д., но не с адреса 1 или 2. Точно так же слово из 8 байтов может начинаться с адреса 0,8 или 16, но не с адреса 4 или 6. Выравнивание адресов требуется довольно часто, поскольку память работает более

эффективно. Возможность считывать слова с произвольными адресами требует усложнения процессора и увеличивает времени доступа к памяти.

1.1.3.2. Порядок байт

Порядок байт важен для размещения в памяти числовых типов данных. Порядок, называемый *обратным* (big-endian), подразумевает, что старший значащий байт числа размещается по младшему адресу памяти. Порядок, называемый *прямым* (little-endian), подразумевает, что старший значащий байт числа размещается по старшему адресу памяти. В качестве примера рассмотрим отображение структуры данных S, представленной на Си для 32-разрядного процессора:

```
struct {
int a;    /* 0x1112_1314 слово */
double b; /* 0x2122_2324_2526_2728 двойное слово */
char * c;  /* 0x3132_3334 слово */
char d[7]; /* 'L','M','N','O','P','Q','R' массив байт */
short e;   /* 0x5152 полслова */
int f;     /* 0x6162_6364 слово */
} S;
```

В случае орядка big-endian в памяти будут следующие значения:

```
содержимое 11 12 13 14 (x) (x) (x) (x)
адрес      00 01 02 03  04 05  06  07
содержимое 21 22 23 24  25  26  27 28
адрес      08 09 0A 0B 0C 0D 0E 0F
содержимое 31 32 33 34 ‘L’ ‘M’ ‘N’ ‘O’
адрес      10 11 12 13  14  15   16   17
содержимое ‘P’ ‘Q’ ‘R’ (x) 51  52  (x) (x)
адрес      18  19  1A  1B 1C 1D 1E 1F
содержимое 61 62 63 64 (x) (x) (x) (x)
адрес      20 21 22 23 24  25  26  27
```

В случае порядка little-endian в памяти будут следующие значения:

```
содержимое 14 13 12 11 (x) (x) (x) (x)
адрес      00 01 02 03  04  05 06  07
содержимое 28 27 26 25  24  23  22 21
адрес      08 09 0A 0B 0C 0D 0E 0F
содержимое 34 33 32 31 ‘L’ ‘M’ ‘N’ ‘O’
адрес      10 11 12 13  14   15   16  17
содержимое ‘P’ ‘Q’ ‘R’ (x) 52 51  (x) (x)
адрес      18   19  1A 1B 1C 1D 1E 1F
содержимое 64 63 62 61 (x) (x) (x) (x)
адрес      20 21 22 23 24  25  26  27
```

1.1.3.3. Когерентность памяти

Еще один аспект модели памяти – семантика памяти. Естественно ожидать, что команда чтения из памяти (Load), которая встречается после команды записи в память (Store) и которая обращается к тому же адресу, возвратит только что сохранное значение, т.е. действия с памятью согласованы (когерентны). Однако в процессорах с параллельным выполнением команд (конвейерные процессоры) преобразования переупорядочиваются. Таким образом, существует реальная опасность, что память не будет действовать так, как ожидалось. Ситуация осложняется в случае с мультипроцессором, когда каждый процессор посылает разделяемой памяти поток запросов на чтение и запись, которые тоже могут быть пере упорядочены.

Первый подход к обеспечению когерентности памяти состоит в том, что процессор все запросы к памяти упорядочивает таким образом, чтобы каждый из них завершался до того, как начнется следующий. Это вредит производительности, но дает простейшую семантику памяти (все операции выполняются в строгом программном порядке).

При втором подходе упорядоченное обращение к памяти может быть достигнуто с помощью команды Sync, которая блокирует запуск всех новых операций памяти до тех пор, пока предыдущие операции не будут завершены. Это сильно затрудняет работу разработчикам компиляторов.

Концепция выполнение инструкций по порядку.

1. Считывание инструкции.
2. Если все операнды инструкции доступны, то она передаётся на выполнение соответствующему исполнительному модулю, иначе процессор останавливается, ожидая готовности операндов.
3. Инструкция выполняется в соответствующем модуле.
4. Модуль записывает результат обратно в регистровый файл.

Концепция выполнение не по порядку.

1. Считывание инструкции.
2. Помещение инструкции в очередь.
3. Инструкция находится в очереди до тех пор, пока её операнды не станут доступны. Таким образом, инструкция может покинуть очередь прежде, чем инструкция попавшая туда раньше.
4. Выбранная из очереди инструкция выполняется в соответствующем модуле.
5. Результат помещается в очередь.
6. Только после того, как все инструкции, которые были в очереди впереди данной, выполнятся, её результат помещается в регистровый файл.

Данная парадигма применяется при разработке суперскалярных процессоров, позволяя повысить процент загрузки исполнительных модулей, а, следовательно, и их производительность.

1.1.3.4. Защита памяти

Защита памяти предусматривает механизмы ограничения доступа к памяти по записи или чтению со стороны программ. Защита связана с разбиением памяти на блоки, страницы или сегменты. Для этих объектов может быть разрешено только чтение, только запись, запись и чтение со стороны программы пользователя или супервизора. Вариант защиты задается в дескрипторных таблицах сопоставленных этим разбиениям. Нарушения требований защиты вызывают особое поведение процессора для их преодоления.

1.1.4. Модель прерываний

Прерывания – это изменения в потоке управления, вызванные какими-либо событиями. Эти события могут быть внешними по отношению к процессору (инициируются устройствами ввода-вывода) - асинхронные прерывания или вызваны результатами выполнения команд – синхронные прерывания (системные прерывания, исключения или ловушки). Прерывание останавливает работу программы и передает управление не содержащейся в явном виде в программе *подпрограмме обработки прерывания* (ISR – Interrupt Serves Routine или interrupt handler – обработчик прерывания) для выполнения особых действий. Адрес первой команды ISR определяется вектором прерывания, сопоставленным тому или иному прерыванию. После завершения ISR управление передается прерванной программе. В случае асинхронного прерывания программа должна продолжить процесс в том же самом состоянии, в котором находилась, когда произошло прерывание. Соответствующую ISR называют прозрачной.

Более общим является понятие *исключение* (exception) - необычная, непредусмотренная или ошибочная ситуация, которая может возникнуть при выполнении программы и изменить её нормальное функционирование.

Наиболее распространенные условия, которые могут вызвать исключения, это переполнение и исчезновение значащих разрядов при операциях с плавающей запятой, переполнение при операциях с целыми числами, нарушения защиты памяти, неопределяемый код операций, переполнение стека, обращение к несуществующим или нечетным адресам ММ, деление на 0. К исключениям относят также и прерывания.

Физический интерфейс простой системы прерываний может быть представлен единственной линией IRQ (Interrupt Request – запрос прерывания). Высокий уровень напряжения на линии IRQ, например, воспринимается как запрос на прерывание. На линию IRQ могут быть мультиплексированы запросы от нескольких источников. Однако в этом случае после принятия общего запроса к обслуживанию возникает задача идентификации источника, выставившего запрос, и передачи управления на соответствующую подпрограмму ISR. Эта задача решается только программным методом с помощью специальной процедуры, называемой *полингом* (polling).

Функция полинга состоит в последовательном опросе состояния всех

устройств (чтение регистров состояния) для выявления готовности к обслуживанию.

Для увеличения числа одновременно обслуживаемых источников прерываний в систему вводится несколько линий с фиксированными векторами прерываний. Такую систему называют радиальной. Часть радиальных линий могут быть внутренними для приема исключений процессора.

В зависимости от числа запросов, одновременно находящихся на обслуживании, различают одно- и многоуровневые системы прерываний. В одноуровневой системе в каждый момент времени допускается лишь один запрос. Обработка всех других запросов откладывается до окончания текущего обслуживания блокировкой всех остальных.

Если несколько устройств одновременно запросили обслуживание, система прерываний выбирает одно из них на основании приоритета, отражающего важность и срочность его обслуживания. Как наиболее естественная, выделяется линейно упорядоченная фиксированная система приоритетов.

Повышение гибкости системы приоритетов связано с их динамическим изменением по заданному алгоритму. Однако в каждый момент времени все приоритеты строго упорядочены, что обеспечивает однозначный выбор одного из них.

Одной из систем с динамически изменяемыми приоритетами является циклическая система. В ней после каждого очередного обслуживания запроса происходит циклический сдвиг приоритетов с присвоением нижнего только что обработанному запросу. Это приводит к равномерному распределению приоритетов.

Многоуровневая система прерываний разрешает многократное (по числу уровней) прерывание одних ISR другими. Для этого каждому уровню ставится в соответствие некоторое подмножество запросов из их общего числа и строго упорядоченный приоритет. Подпрограммы ISR некоторого уровня могут быть прерваны лишь запросами более высокого уровня. Приоритеты запросов сравниваются с приоритетом процессора и, если он ниже, прерывают его работу. Для разрешения конфликтов внутри группы запросов одного уровня существует вторичная система приоритетов.

1.1.5. Модель управления памятью

Виртуальная память - это способ выполнения программ, размер которых превышает размер доступной физической памяти. Части программ (оверлеи) хранятся во вторичной памяти и пересылаются в MM по мере надобности. Пересылка осуществляется автоматически операционной системой и невидима для программы.

Виртуальная память основана на разделении понятий *адресного пространства* и *адресов памяти* (физические адреса). *Виртуальное адресное пространство* – это множество адресов, к которым может обращаться программа данного CPU, а реальные адреса MM MPS – физическим адресным пространством. Виртуальное адресное пространство CPU намного

больше его адресного пространства, определяемого разрядностью его адресного регистра.

1.1.5.1. Страничная организация памяти

Пусть адресное пространство CPU равно 65536 байт, а физическое адресное пространство MPS содержит ячейки с 0 по 4095. Можно было бы настроить CPU MPS так, чтобы при обращении к адресу 4096 должно использоваться слово из памяти 0, а при обращении к адресу 4097 – слово из памяти с адресом 1, при обращении к адресу 8191 – слово из памяти с адресом 4095. Другими словами, определено отображение из адресного пространства в действительные адреса памяти – механизм *трансляции адресов*. Что произойдет, если CPU обратится к адресу 8192. В CPU без виртуальной памяти произойдет исключение по несуществующему адресу физической памяти. В CPU с виртуальной памятью будет иметь место следующая последовательность шагов:

1. Байты из MM будут отправлены во вторичную память.
2. Байты с 8192 по 12287 будут загружены из вторичной памяти в MM.
3. Отображение адресов изменится: теперь адреса с 8192 по 12287 соответствуют ячейкам физической памяти с 0 по 4095.
4. Выполнение программы продолжится, как ни в чем не бывало.

Такая технология автоматического отображения называется *страничной организацией памяти*, а части программы, которые считываются из вторичной памяти, страницами. *Страница* – набор соседних байтов фиксированной длины, не имеющих непосредственной связи с логической структурой программы.

Виртуальное адресное пространство разбивается на ряд страниц равного размера, обычно от 512 до 64 Кбайт, хотя иногда встречается 4 Мбайт. Размер страницы всегда должен быть степенью двойки. Физическое адресное пространство тоже разбивается на части равного размера таким образом, чтобы каждая такая часть MM вмещала ровно одну страницу. Эти части MM называют *страничными кадрами*.

Рассмотрим для примера, как можно 32-разрядный логический адрес (пусть виртуальная память тоже 32 разряда, а размер страницы 4 Кбайт) отобразить на физический адрес MM объемом 32 Кбайт. В CPU это отображение выполняет MMU. Преобразование 32-битного логического адреса в 15-битный адрес MM (8 страничных кадров) выполняется следующим образом. Узел MMU разделяет логический адрес на 20-битный номер виртуальной страницы и 12-битовое смещение внутри страницы. Номер виртуальной страницы используется в качестве индекса в *таблице страниц* для нахождения нужной страницы. На рис.1.4 номер виртуальной страницы равен 5, поэтому из таблицы выбирается элемент 5 с номером страничного кадра 6 .Сначала MMU проверяет, находится ли нужная страница в текущий момент в MM, читая бит присутствия в данном элементе *таблицы страниц*. В примере этот бит равен 1, т.е. страница в памяти.

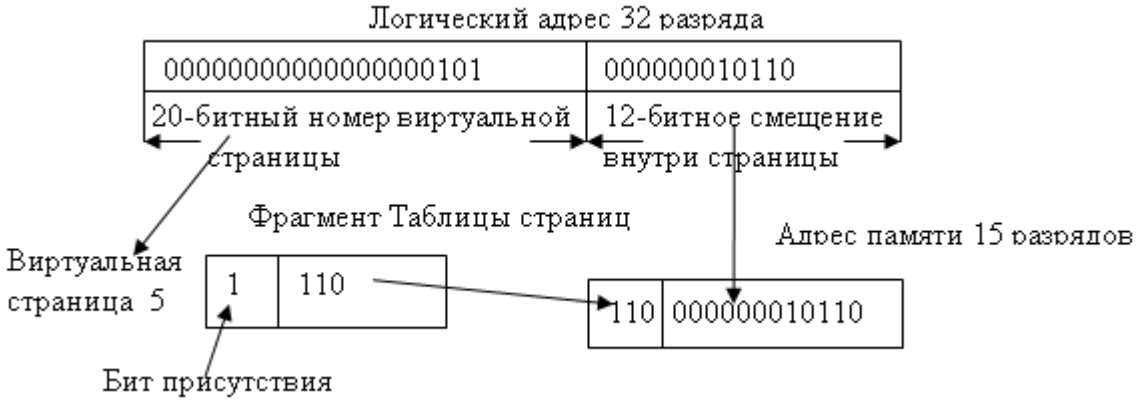


Рис.1.4. Формирования адреса ММ

При обращении к адресу страницы, которой нет в памяти, происходит исключение из-за отсутствия страницы. В случае такой ошибки операционная система должна считать нужную страницу из вторичной памяти, ввести новый адрес физической памяти в таблицу страниц, а затем повторить команду, которая вызвала исключение. Такой метод работы с виртуальной памятью называется *вызовом страницы по требованию*. Для каждой программы распределение памяти уникально и при переключении с одной программы на другую меняется, поэтому в системах с разделением времени такой подход не годится.

Другой подход основан на наблюдении, что большинство команд обращаются к адресному пространству не равномерно. Обычно большинство обращений относится к небольшому числу страниц. При обращении к памяти можно вызвать команду, вызвать данные или сохранить данные. В каждый момент существует набор страниц, которые использовались при последних m обращениях. Этот набор называют *рабочим множеством*. Поскольку рабочее множество меняется очень медленно, можно опираясь на последнее перед остановкой программы рабочее множество, предсказать, какие страницы понадобятся при новом запуске программы. Эти страницы можно загрузить заранее перед очередным запуском программы.

При обращении программы к странице, которая отсутствует в памяти ее нужно вызвать из вторичной памяти. Однако, чтобы освободить для не место, нужно во вторичную память отправить какую-нибудь страницу. По одному из алгоритмов удаляется та страница, которая использовалась наиболее давно, поскольку вероятность того, что она будет в текущем рабочем множестве, очень мала. Этот алгоритм называется LRU (Last Recently Used – наиболее давно использовавшиеся элементы). Иногда LRU приводит к патологическим ситуациям (например программа, цикл которой простирается на несколько страниц).

Другой алгоритм – FIFO (First-in First-out – первым поступил, первым выводится) удаляет ту страницу, которая раньше всех загружалась, независимо от того, когда в последний раз производилось обращение к этой странице. С каждым страничным кадром связан отдельный счетчик. Изначально все счетчики установлены на 0. После ошибки из-за отсутствия страниц счетчик каждой страницы, находящейся в памяти, увеличивается

на 1, а счетчик только что вызванной страницы принимает значение 0. Когда нужно выбрать страницу для удаления, выбирается страница с самым большим значением счетчика.

Если размер рабочего множества больше, чем число допустимых страничных кадров, ни один алгоритм не дает хороших результатов, и ошибки из-за отсутствия страниц будут происходить часто. Если программа постоянно вызывает подобные ошибки, то говорят, что наблюдается пробуксовка (thrashing).

Если страница, которую нужно удалить не менялась, с тех пор как ее считали, то необязательно ее записывать обратно во вторичную память, поскольку точная копия уже существует. В MMU содержится бит для каждой страницы, который равен 0 при загрузке страницы и принимает значение 1, когда изменяются данные в этой странице. По этому биту ОС определяет необходимость перезаписи страницы.

1.1.5.2. Сегментация памяти

Страничная организация памяти представляет одномерную виртуальную память, в которой виртуальные адреса идут один за другим от 0 до максимального адреса. Удобнее использовать два или несколько виртуальных адресных пространств. Например, компилятор может иметь несколько таблиц, которые создаются в процессе компиляции:

1. Таблица символов, которая содержит имена и атрибуты переменных.
2. Исходный текст, сохраняемый для листинга.
3. Таблица, содержащая все использующиеся целочисленные константы и константы с плавающей запятой.
4. Дерево, содержащее синтаксический анализ программы.
5. Стек, используемый для вызова процедур в компиляторе.

Каждая из первых четырех таблиц постоянно растет в процессе компиляции. Последняя таблица растет и уменьшается непредсказуемо. В одномерной памяти эти таблицы пришлось бы разместить в виртуальном адресном пространстве в виде смежных областей.

Что произойдет, если программа содержит очень большое число переменных? Адресное пространство, предназначенное для таблицы символов, может переполниться, даже если в других таблицах много свободного места.

Компилятор может забирать свободное пространство у одних таблиц и передавать его другим таблицам, но это похоже на управление оверлеями вручную. Нужно освободить программиста от расширения и сокращения таблиц, подобно тому, как виртуальная память исключает необходимость следить за разбиением программы на оверлеи. Для этого нужно создать много абсолютно независимых адресных пространств, которые называют *сегментами*. Каждый сегмент состоит из линейной последовательности адресов от 0 до какого-либо максимума. Разные сегменты могут иметь разную длину. Длина сегмента может меняться во время выполнения программы.

Так как каждый сегмент основывает отдельное адресное пространство, разные сегменты могут изменяться независимо друг от друга.

Если процедура занимает отдельный сегмент n и впоследствии изменяется и перекомпилируется, то другие процедуры менять не нужно, т.к. начальный адрес процедуры не изменился.

Сегментация облегчает разделение общих процедур или данных между несколькими программами. Эти процедуры или данные размещают в одном сегменте, т.е. нет необходимости иметь копии этих объектов в каждой программе.

Разные сегменты могут иметь разные виды защиты. Например, сегмент с процедурой можно определить “только для выполнения”, запретив тем самым считывание и запись в него. Для массива с плавающей запятой разрешается только чтение и запись, но не выполнение и т.д. Такая защита часто помогает обнаружить ошибки в программе.

Защита имеет смысл только в сегментированной памяти и не имеет смысла в одномерной памяти. Поскольку каждый сегмент включает в себя объект только одного типа, он может использовать защиту, подходящую для этого типа.

Сегментация реализуется разделением сегмента на страницы и вызова их по требованию. В этом случае одни страницы сегмента могут находиться в ММ, а другие – во вторичной памяти. Чтобы разбить сегмент на страницы, для каждого сегмента создается своя таблица страниц.

1.1.6. Типы процессоров

Микропроцессор (μP) представляет собой процессорный узел, реализованный в виде одной микросхемы. μP чаще всего используются для вычислений общего назначения и применяются в основном в персональных компьютерах, серверах, суперкомпьютерах и высокопроизводительных встроенных системах реального времени.

Микроконтроллер (μC) представляет собой объединение на одном кристалле процессорного узла, постоянной памяти для хранения программ (десятки килобайт), оперативной памяти для хранения данных (несколько килобайт) и набора различных устройств ввода-вывода. μC широко применяются в недорогих встроенных системах реального времени.

Система на кристалле (SoC) как и μC размещается на одном кристалле и также выполняет функции целого устройства (например, компьютера). Отличие состоит в большей сложности, ориентированности на специализированную задачу и подходом к проектированию - аппаратная часть собирается из стандартных отлаженных блоков (IP-ядер), а для сборки программной части используются готовые драйверы.

Выделяют процессоры, основанные на SoC. Они содержат процессорное ядро и большое количество периферийных интерфейсов [4].

Системы на кристалле могут содержать несколько процессорных узлов в общем случае разнородных. Такие устройства называют мультипроцессорными системами на кристалле (MPSoC), например [5].

Цифровой сигнальный процессор (Digital signal processor - DSP) [6] — это специализированный микропроцессор, предназначенный для цифровой обработки сигналов в реальном масштабе времени. Задачи цифровой обработки сигналов имеют несколько общих моментов.

Во-первых, большое число обрабатываемых данных, которые представляют отсчеты физических величин, поступающих с заданным периодом дискретизации (отсчеты радиосигналов, изображения, речи).

Во-вторых, обычно выполняются сложные математические операции, включающие фильтрацию, идентификацию, спектральный анализ, машинное обучение и другие. Это интенсивные операции, для поддержки которых и разработаны DSP, например, операция «умножение с накоплением» (MAC) $A = A + X \times B$ обычно исполняется за один такт, где A – аккумулятор, X – входной отсчет, а B – некоторый постоянный коэффициент.

Коммуникационные микропроцессоры, например [7], разработаны для интеграции с сетевым и другим коммуникационным оборудованием и состоит из высокопроизводительного процессорного ядра, гибкого контроллера памяти и коммуникационного процессорного модуля (CPM). CPM содержит независимый специализированный RISC процессор (CP). Этот процессор разгружает центральный процессорный узел от задач взаимодействия с периферийным оборудованием.

CPM содержит:

– последовательные коммуникационные контроллеры (SCC), поддерживающие протоколы Ethernet, ATM, HDLC и другие;

– последовательные контроллеры администрирования (SMC);

– интерфейсы с временным разделением каналов;

– интерфейсы характерные для обычных микропроцессорных систем.

Графические процессоры (GPU) [8] – специализированные процессоры, разработанные для выполнения вычислений требуемых для визуализации графики. Они поддерживают 3D графику, построение теней и цифровое видео. Доминируют в этой области Intel, NVIDIA иAMD.

Некоторые встроенные приложения, в частности игры, хорошо подходят для GPU. Графические процессоры эволюционируют в сторону общей модели программирования и поэтому начинают примеряться в других приложениях, требующих интенсивных вычислений, таких как измерительная техника.

Обычно GPU потребляют много энергии и поэтому не находят применения для встроенных приложений с жесткими требованиями к энергопотреблению.

1.2. Типы и формы параллелизма процессоров

Большинство современных процессоров обеспечивают различные формы параллельной работы. Эти механизмы существенно влияют на время выполнения программ микропроцессорной системой, поэтому разработчики встроенных систем должны понимать их.

Понятие конкурентная (одновременная) работа является центральным для встроенных систем. Говорят, что компьютерная программа конкурентная, если ее различные части концептуально могут выполняться совместно. Говорят, что компьютерная программа параллельна, если ее различные части физически выполняются совместно на различном оборудовании (например, многоядерный процессор или группа различных процессоров)

Не конкурентные программы задают строгую последовательность выполнения команд. Языки программирования, выражающие такие программы, называют императивными (например, язык Си). Использование Си для написания конкурентных программ требует дополнительных шагов вне языка. Обычно это использование библиотеки потоков (thread), которая обеспечивается операционной системой. В Java, являющимся императивным языком включены конструкции, напрямую поддерживающие потоки.

Рассмотрим следующие операторы Си кода:

```
double pi, piSquared,
piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = pi * pi * pi;
```

Последние два оператора являются независимыми и, следовательно, могут выполняться параллельно или в обратном порядке. Это не повлияет на результат.

Перепишем эту последовательность следующим образом, и она перестанет быть независимой:

```
double pi, piSquared,
piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = piSquared * pi;
```

В этом случае последний оператор зависит от предыдущего.

Компилятор может анализировать зависимости между операторами и формировать параллельный код, если целевая микропроцессорная система поддерживает параллельную работу. Такой анализ называют анализом потока данных. Сегодня многие микропроцессоры поддерживают параллельное выполнение, используя архитектуру с множественной выдачей потоков команд или VLIW (Very Large Instruction Word, очень длинный формат команды).

Процессоры с множественной выдачей потоков команд могут выполнять одновременно независимые команды. Аппаратные средства анализируют команды на лету, и когда зависимость не обнаруживается, выполняют одновременно более чем оду команду. VLIW процессоры имеют ко-

манды ассемблерного уровня, которые определяют конкурентные операции. В этом случае от компилятора обычно требуют вставить подходящие команды в программу.

В обоих случая (множественна выдача и VLIW) императивная программа анализируется на предмет одновременности для разрешения ее параллельного выполнения. Общее требование состоит в увеличении скорости выполнения программы. Цель - увеличение производительности, когда предположение о завершении задачи раньше всегда лучше завершения позже. Однако в контексте встроенных систем одновременность играет более существенную роль, чем просто улучшение производительности. Программе встроенной системы часто необходимо отслеживать и реагировать на множество одновременных источников и одновременно управлять множеством выходных устройств. Программа встроенной системы почти всегда одновременная программа и одновременность является присущей частью логики программы. Это не только путь улучшения производительности. В действительности завершение задачи раньше не является с необходимостью лучше, чем завершение позже. Суть в своевременности, т.е. действия в физическом мире часто необходимо делать в правильное время - не раньше и не позже. Например, регулятор бензинового двигателя: раннее зажигание не лучше позднего. Оно должно произойти в правильное время.

Также как императивные программы могут выполняться последовательно или параллельно, конкурентные программы могут выполняться последовательно или параллельно. Последовательное выполнение конкурентной программы сегодня обычно выполняется многозадачной операционной системой, которая чередует выполнения множества задач в простом последовательном потоке команд. Аппаратура может распараллелить это выполнение, если процессор поддерживает множественную выдачу или VLIW. Таким образом, конкурентная программа преобразуется операционной системой в последовательный поток и обратно в конкурентную программу аппаратным обеспечением для улучшения производительности. Это множественное преобразование сильно затрудняет получение результата в правильное время.

Параллелизм в аппаратном обеспечении призван повысить производительность приложений, требующих интенсивных вычислений. С точки зрения программиста одновременность вырастает как следствие разработки аппаратура для увеличения производительности. Другими словами приложение не требует, чтобы множество действий происходило одновременно, требуется лишь, чтобы все происходило очень быстро. Конечно, много интересных приложений объединят обе формы одновременности, возникающие из параллелизма и требований приложения.

В этом параграфе рассматриваются такие параллелизмы как конвейерная обработка, параллелизм уровня команд и многоядерные архитектуры.

1.2.1. Конвейеризация

Большинство современных процессоров являются конвейерными. На рис. 1.5 [9] приведен простой пятиступенчатый конвейер для 32-разрядного процессора, где:
fetch – ступень выборки команды;
decode – ступень декодирования команды;
execute – ступень выполнения команды;
memory – ступень чтения или записи в память;
writeback – ступень обратной записи;
PC – счетчик команд;
Add –устройство сложения;
Mux – мультиплексор;
Instruction memory – память команды;
Decode – устройство декодирования;
Zero? – устройство определения равенства 0 операнда;
Register bank – банк регистров;
ALU – арифметико-логическое устройство;
Data memory – память данных;
data hazard (memory read or ALU result) - риск сбоя данных при чтении памяти или результата ALU;
data hazard (computed branch) - риск сбоя данных при безусловном переходе;
control hazard (conditional branch) - риск сбоя управления;
branch taken –переход по команде ветвления.

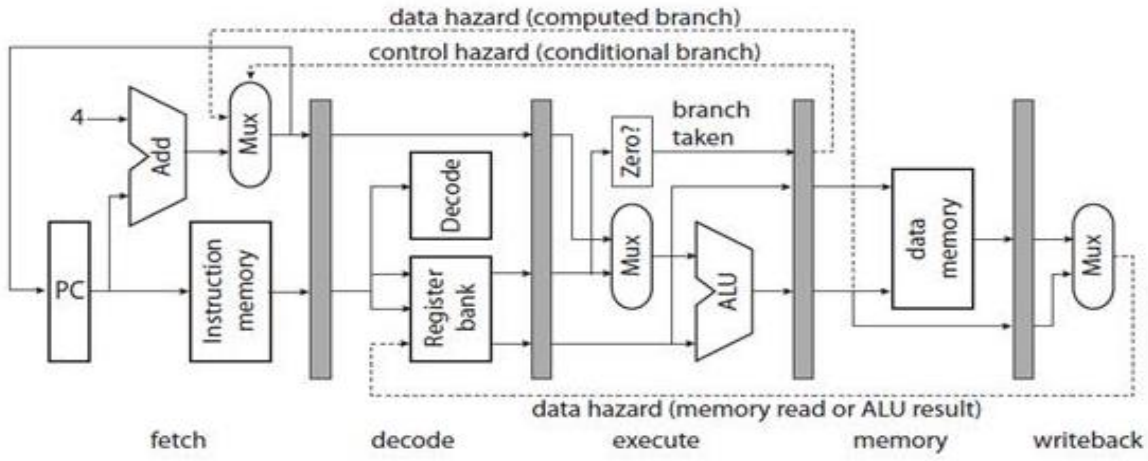


Рис. 1.5. Простой конвейер

Затененные прямоугольники на рисунке – регистры-защелки, синхронизированные тактовой частотой процессора. По фронту тактового сигнала данные на их входах запоминаются, а выходы остаются неизменными до прихода следующего фронта, что дает возможность схемам, находящимся между защелками, установить свое значение.

На ступени выборки команды PC задает адрес памяти команд, которая содержит кодированные 32-разрядные слова команд. На этой ступени PC увеличивается на 4 (байта), указывая на адрес следующей команды, за

исключением команд условного перехода, которые сами обеспечивают установку нового адреса в PC.

На ступени декодирования из команды извлекаются адреса регистров, по которым из банка регистров выбираются данные.

На ступени выполнения, выбранные из банка регистров или из PC (для команд безусловного перехода) данные преобразуются с помощью ALU.

На ступени памяти выполняются операции записи или чтения по адресу, указанному в регистре банка.

На ступени обратной записи выполняется запоминание результата в регистровом файле.

В случае DSP добавляется одна или две ступени для выполнения умножения и вычисления адресов двух портовой памяти с помощью отдельных ALU. Двух портовая память допускает одновременный доступ к двум операндам.

Оборудование конвейера между регистрами-защелками функционирует параллельно. Следовательно, мы видим, что одновременно выполняется пять команд каждая на своей ступени. Это легко визуализировать с помощь таблицы распределения на рис. 1.6.

Слева на рисунке показаны одновременно используемые аппаратные ресурсы. Регистровый банк появляется три раза, т.к. в цикле команды могут быть два чтения и одна запись в банк регистров.

Таблица распределения показывает последовательность команд A, B, C, D, E программы. В цикле 5 выбирается E в то время как D читает из банка регистров, C использует ALU, B читает или записывает в память данных, а A записывает результат в банк регистров. Запись A происходит в цикле 5, а чтение B в цикле 3. Таким образом значение прочитанное B не может быть значением, записываемым A. Это вызывает риск сбоя данных при чтении (на рис.6. отмечено пунктирной линией). Обычно программист предполагает A выполняется перед B и результат A доступен B, что в действительности не так.

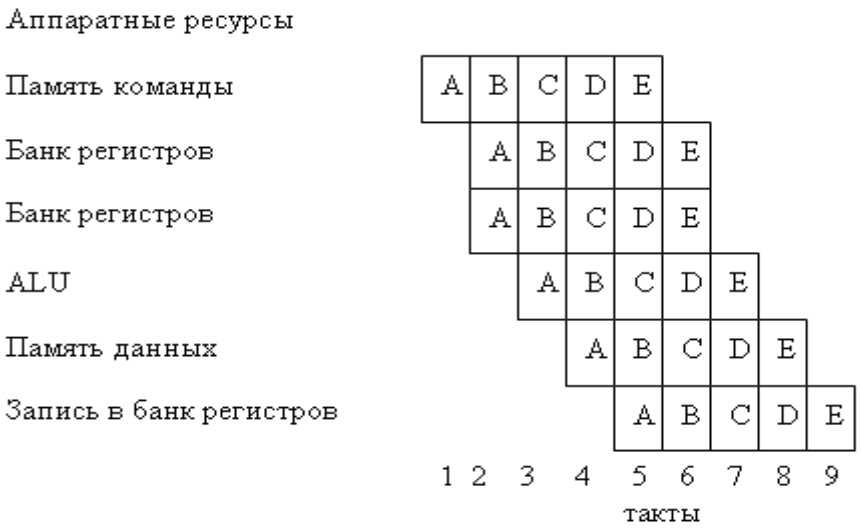


Рис.1.6. Таблица распределения для конвейера на рис. 1.5

В компьютерных архитектурах проблема рисков сбоя решается различными путями. Простейшее решение известно как явный конвейер, когда риски просто документируются и программист (или компилятор) должен учитывать их. Например, где B читает регистры, записанные A, компилятор должен вставить три пустые команды (ничего не делают) между A и B для обеспечения записи перед чтением. Эти пустые команда образуют «пузыри», распространяющиеся по конвейеру.

Более сложное решение основано на взаимоблокировке, когда аппаратура декодирует B и обнаруживает, что B читает регистр записываемый A (риск сбоя), то выполнение B задерживается на три такта пока A не завершит ступень обратной записи. Это иллюстрирует рис. 1.7.

Задержку можно уменьшить до двух тактов, если реализовать чуть более сложную логику продвижения команд в конвейере, которая обнаруживает запись A в туже ячейку из которой читает B, а затем напрямую обеспечивает данными B перед их обратной записью. Это автоматизирует введение «пузырей».

Еще более сложной техникой является выполнение с изменением последовательности, когда аппаратура обнаруживает риск сбоя и вместо задержки выполнения B продолжает выборку C и если C не читает регистры записываемые A или B и не записывает регистры читаемые B, тогда продолжается выполнение C перед B. Это уменьшает число «пузырей».

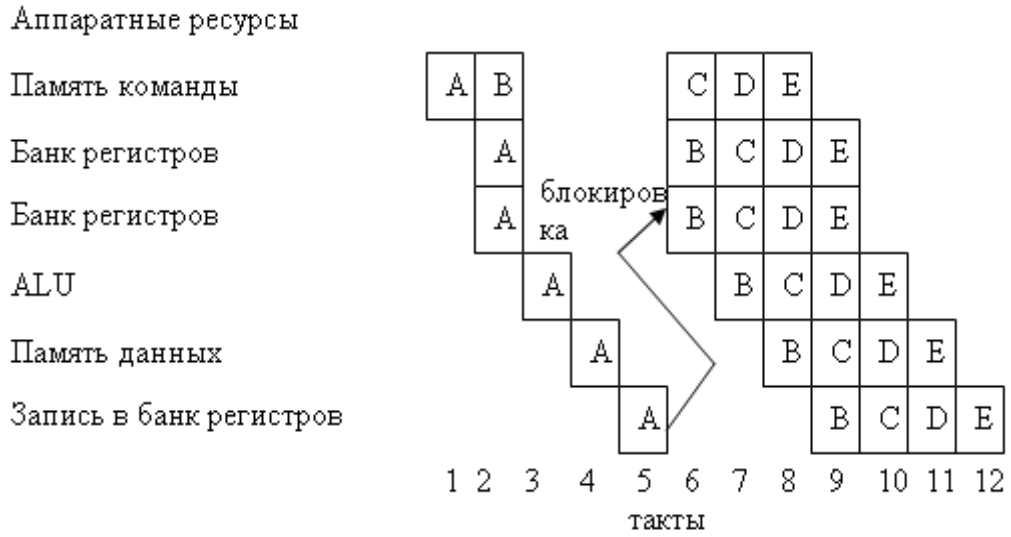


Рис.1.7. Таблица распределения для конвейера на рис. 1.5 с взаимоблокировкой, когда B читает регистр, записываемый A.

Другая разновидность рисков сбоя конвейера на рис. 6 – риск сбоя управления выборкой команд. Команда условного перехода изменяет PC, если определенный регистр равен 0. В этом случае, если A является командой условного перехода, она достигает ступени чтения или записи в память перед тем как изменяет PC. Следующие за A команды будут выбраны, декодированы, но в какой то момент времени выясняется что их не нужно было исполнять.

Существует несколько способов преодоления рисков сбоя управления выборкой команд. При задержанном переходе просто документируется, что команда перехода занимает столько-то тактов. Программист (или компилятор) должен обеспечить, чтобы команды, следующие за командами перехода, были пустыми или делали полезную работу, не зависящую от ветвления. Взаимоблокировка обеспечивает автоматическое введение «пузырей» как рисках сбоя данных.

Наиболее сложный способ преодоления рисков сбоя управления выборкой команд это спекулятивное выполнение команд. Аппаратура предполагает что переход, вероятно, будет иметь место и начинает выполнять предполагаемые команды. Если предположение не оправдалось, приходится удалять побочные результаты (такие как запись в регистры), вызванные спекулятивным выполнением команд.

За исключение явной конвейеризации и задержанных переходов все остальные способы вносят вариативность во время выполнения команд. Анализ времени выполнения программы может стать очень трудным в случае длинного конвейера с замысловатым продвижением и спекулятивным выполнением.

Явные конвейеры наиболее характерны для DSP, которые часто используются в задачах, где важно точное время. Изменение порядка выполнения и спекулятивное выполнение характерны в процессорах общего назначения, где время имеет значение в общем смысле. Разработчику встроенных систем необходимо понимать требования приложения и не останавливаться на процессорах, для которых неочевидна точность расчета времени выполнения программ.

1.2.2. Параллелизм уровня команд

Процессоры, поддерживающие параллелизм уровня команд (ILP) способны выполнять несколько независимых операций в каждом цикле команды. Рассмотрим четыре основных типа ILP: процессоры CISC, параллелизм части слова, суперскалярные процессоры и процессоры VLIW.

Процессоры со сложными командами (точнее со специализированными) называют CISC (complex instruction set computer) процессорами в противоположность RISC (reduced instruction set computers) процессорам с сокращенным множеством команд .

Процессор DSP типичный CISC процессор, включающий специальные команды поддержки программной реализации фильтров с конечной импульсной характеристикой (FIR фильтров). Такой процессор реализует FIR с производительностью одна команда на ветвление в фильтре. Недостаток такого процессора состоит в экстремальном напряжении сил компилятором для включения таких команд в программу. Поэтому DSP используют библиотеки, написанные и оптимизированные на ассемблере.

Параллелизм части слова **[**10**].** Многие встроенные приложения оперируют с данными меньшей разрядности, чем слово процессора. Для поддержки таких типов данных некоторые процессоры используют паралле-

лизм на уровне части слова, когда многоразрядное ALU разбивается на минимальные части, допускающие одновременное выполнение арифметических или логических операций над маленькими словами.

Векторный процессор один из тех, у кого множество команд включает одновременные операции над элементами множественных данных. Параллелизм уровня команд является формой векторной обработки.

Суперскалярные процессоры **[**11**]** используют обычное множество команд, но аппаратура может одновременно распределять несколько команд по индивидуальным узлам обработки, когда обнаруживается, что такое распределение не изменяет поведения программы. Такие процессоры поддерживаю изменение последовательности выполнения, когда поздняя команда потока выполняется перед ранней.

Недостаток суперскалярной архитектуры с точки зрения встроенных систем состоит в трудности предсказания времени выполнения программы. В контексте многозадачности (прерывания и потоки) это время даже может и не иметь высокой повторяемости. Время выполнения может быть очень чувствительно к точному времени прерываний, так незначительные вариации этого времени могут существенно повлиять на время выполнения программы.

Процессоры с архитектурой VLIW **[**2**]** часто используются во встроенных системах вместо суперскалярных для получения высокой повторяемости и предсказуемости времени выполнения программ. Так же как суперскалярные процессоры, VLIW процессоры включают несколько функциональных узлов, но вместо динамического определения какие команды могут выполняться одновременно каждая команда определяет то что каждый функциональный узел должен делать в отдельном цикле. Таким образом, VLIW объединяет несколько независимых операций в отдельную команду. Как и в суперскалярных процессорах множество операций выполняется одновременно на различном оборудовании. Однако порядок и одновременность выполнения фиксирована в программе в противоположность принятия решений на лету у суперскалярных процессоров. Это требует от программиста или компилятора обеспечить, чтобы одновременные операции были действительно независимыми. В обмен на эти дополнительные сложности в программировании время выполнения становится предсказуемым. Для приложений требующих еще увеличения производительности VLIW процессоры могут быть усложнены.

Многоядерные процессоры являются комбинацией нескольких CPU на одном кристалле. Разнородные (гетерогенные) многоядерные процессоры объединяют разнотипные CPU в противоположность однородным (гомогенным) процессорам, объединяющим одинаковые CPU. Для встроенных систем многоядерная архитектура несет значительные преимущества, чем одноядерная из-за задач реального времени и задач, критичных по безопасности. По этой причине разнородные многоядерные процессоры используются в мобильных телефонах, т.к. функции по обработке речи и радиосигналов являются функциями жесткого реального времени со значительной

вычислительной нагрузкой. В такой архитектуре пользовательские приложения не взаимодействуют с функциями реального времени.

1.3. Технологии и иерархия памяти

1.3.1. Технологии памяти

Выбор технологии памяти имеет важное последствие для разработчика встроенной системы. Например, программиста беспокоит изменяться или нет данные после выключения питания или вхождения в режим энергосбережения. Память, содержимое которой теряется после пропадания питания, называют *энергозависимой памятью* (volatile memory). Обычно с ней ассоциируют оперативное запоминающее устройство (ОЗУ) или RAM (Random Access Memory) – память, в которой единицы данных (байты или слова) могут быть записаны и считаны относительно быстро. Для ее организации используют технологии *статической память* SRAM (Static RAM) и *динамической памяти* DRAM (Dynamic RAM). .

Встроенные системы без исключения нуждаются в сохранении данных даже при выключенном питании. Для этого существует несколько опций. Резервное батарейное питание является одной из них, при котором энергия не исчезает. Батареи, однако, срабатываются, следовательно, требуется более совершенная опция в собирательном смысле известная как *энергонезависимая память* (non-volatile memory). Обычно с ней ассоциируют постоянную память (ROM).

1.3.1.1. Статическая память и ее контроллер

Используемая для создания SRAM технология та же что и для логических элементов, процессоров, PLD и SoC. Информация сохраняется триггерами на протяжении всего времени, пока на SRAM подано питание. Одинаковая технология позволяет добавлять блоки SRAM в процессоры, SoC и PLD (противоположно DRAM, которая использует полностью отличную технологию и поэтому не находит прямого применения на кристаллах процессоров, SoC и PLD). Быстродействие SRAM обычно много больше DRAM. Часто ответ SRAM составляет пару тактов частоты CPU.

При размещении блока SRAM на кристалле CPU или SoC его позиционируют в определенное место адресного пространства микропроцессорной системы. Системное программное обеспечение и драйверы устройств могут выделять части SRAM для своих нужд. Редко операционная система динамически распределяет эту память. Она остается за BSP (Board Support Package – пакет поддержки платы), поддерживающей различные особенности микропроцессорной системы. SRAM обычно выделяется для специальной структуры данных, к которой очень часто обращается процессор или элементы устройств ввода-вывода для временного хранения потоковых данных. В общем блоки SRAM не являются кэш-когерентными

с системой основной памяти и поэтому отображаются на некэшируемое адресное пространство. В некоторых платформах часть инфраструктуры кэш может быть перенастроена как SRAM и ячейки SRAM в кэш блоке воспринимаются просто как область памяти. Это известно под именем «кэш как RAM» с очень малым латентным временем, что делает ее сильно связанной с CPU. Ячейки в кэш CPU часто делают из ячеек высокоскоростных SRAM. Это естественный компромисс. Плотность SRAM намного меньше DRAM, поэтому, память на кристалле измеряется мегабайтами, тогда как в случае с DRAM – гигабайтами.

Доступ по записи/чтению в SRAM намного быстрее, чем в DRAM. Поэтому нет необходимости применять сложные механизмы конвейеризации посредством контроллера SRAM. Контроллер SRAM обычно или управляет одной транзакцией в каждый момент времени или конвейеризирует небольшое число ожидающих выполнения транзакций при помощи шины с простым дроблением транзакции. Это не приводит к такому повышению производительности, как в случае DRAM.

Транзакции по запись только части слова уменьшает производительность системы, так как контроллер SRAM сначала должен прочитать все слово, вставить в него новый байт, в затем записать слово обратно в память. Транзакции по чтению невыровненных слов также уменьшают производительность системы, так как контроллер SRAM сначала должен прочитать одно слово и выделить из него требуемые байты, затем следующее и выделить из него требуемые байты, затем объединить выделенные данные и отправить их в CPU.

Асинхронный интерфейс SRAM содержит следующие сигналы:

- A0…An – адрес ячейки;
- D0…Dm – вход/выходные данные;
- /WE – (Write Enable) – инициализация записи данных;
- /OE (Output Enable) – инициализация чтения данных;
- /CS – выборка кристалла.

На рис. 1.8 приведена временная диаграмма цикла записи SRAM. Время установки (setup time) определяет минимальное время для подачи сигнала /WE после установки линий адреса (время необходимое для дешифрации адреса ячейки). Время сохранения (hold time) определяет минимальное время до очередного изменения адресных линий после снятия сигнала /WE (время необходимое для предотвращения перезаписи данных в другую ячейку). На рис. 1.9 приведена временная диаграмма цикла чтения SRAM.

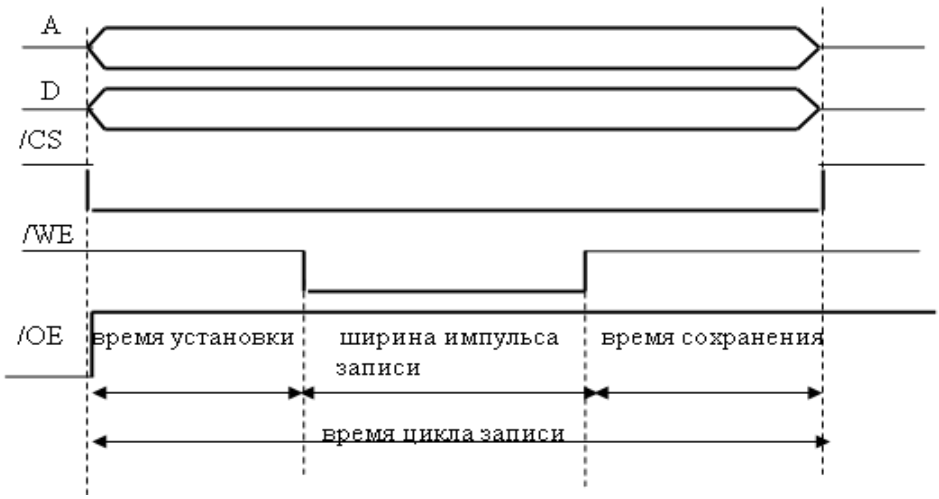


Рис. 1.8 . Временная диаграмма цикла записи статического ОЗУ

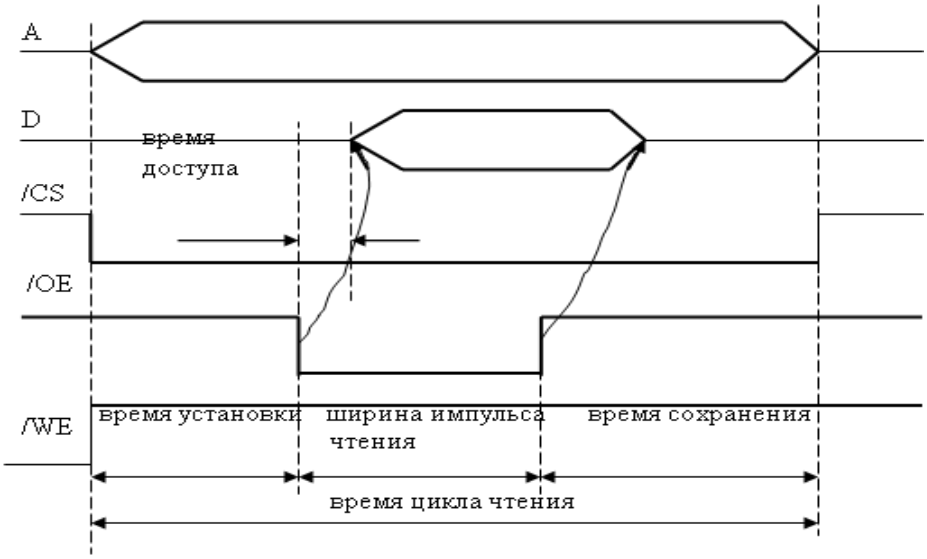


Рис. 1.9. Временная диаграмма цикла чтения статического ОЗУ

1.3.1.2. Динамическая память и ее контроллер

DRAM является формой энергозависимой памяти. Современные DRAM являются синхронными (SDRAM), т.е. все действия происходят по фронту или срезу тактового сигнала (Clock). Значение бита в ячейке DRAM запоминается на очень маленькой емкости. Эта емкость много меньше логического вентиля и, следовательно, плотность ячеек выше, чем в SRAM. Эта разница значительна. Чем выше плотность, тем ниже стоимость одного бита памяти. Однако существует системный компромисс – время операции чтения в DRAM намного больше, чем в памяти на вентилях CPU и SRAM. Задержка чтения является ключевым атрибутом производительности памяти. В любом случае CPU всегда вынужден ждать результат чтения памяти, перед тем как он сможет продолжить обработку. Для того чтобы прочитать бит устройство должно проверить запомненную конденсатором величину. Эту работу выполняет усилитель считывания, затрачивая на определение заряда конденсатора небольшое время.

Электрический заряд очень маленькой емкости в ячейке DRAM стекает за несколько десятков миллисекунд, поэтому каждый бит информации

должен обновляться (регенерировать) для предотвращения разрушения данных. Во время выборки усилитель считывания, определяя заряд емкости, растрачивает его. Таким образом, и при обращении к памяти по чтению данные разрушаются, поэтому необходимо заботиться об их восстановлении после чтения.

Другим ключевым атрибутом производительности DRAM является полная ширина полосы пропускания или пропускная способность, которая значительно увеличилась по сравнению с первыми образцами за счет использования конвейеризации операций записи и чтения. Рис. 1.11 иллюстрирует повышение производительности в DDR (DDR – вариант современной SDRAM с удвоенной скоростью обмена) за счет конвейеризации, поддерживающей множественные параллельные запросы доступа с двумя раздельными линиями запроса и ответа. На рисунке обозначено: Tl – время задержки доступа (латентное время), Rq – запрос Rs – ответ.

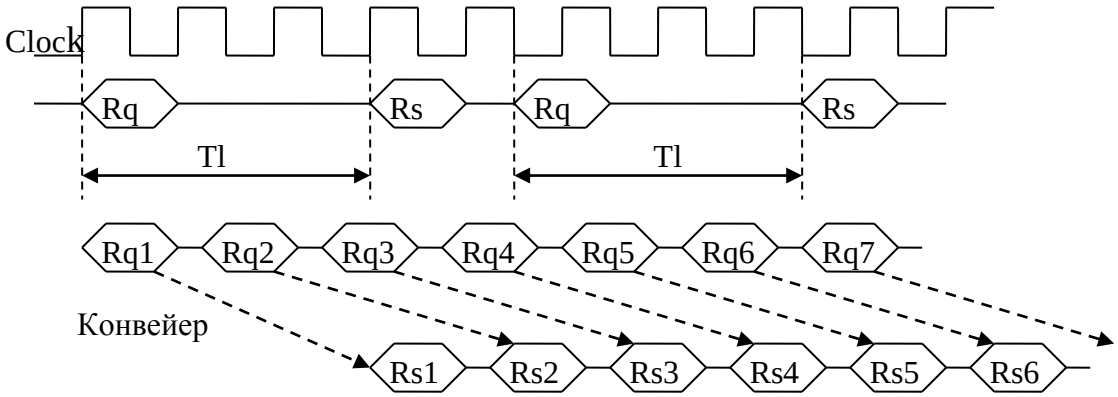


Рис. 1.11. Увеличение производительности DDR с помощью конвейера

Сегодня DDR2 SDRAM являются доминирующими для платформ встроенных систем, а DDR3 завоевывают компьютеры общего назначения. Важным моментом является стандартизация интерфейса, позволяющая многим разным производителям выпускать совместимые устройства. Стандарты для DDR1, DDR2 и DDR3 поддерживает организация JEDEC. Стандарт для DDR устанавливает что данные поставляются/потребляются и по фронту и по срезу тактового сигнала CLK. Ячейки памяти внутри DRAM организованы в виде матрицы из строк и столбцов (рис. 1.12). Интерфейс CPU не позволяет прямое взаимодействие с DRAM. Для этих целей необходим посредник – DRAM контроллер, реализующий необходимые транзакции соответствующие стандарту JEDEC.

Рис. 1.12. Структура DDR

Транзакции состоят из двух фаз. Первую называют Row Address Strobe (RAS) – строб адреса строки (или команда активизации – Activate). В этой фазе вся строка адресованная битами A0 – A13 выбирается из матрицы памяти. Все биты строки посылаются на усилители считывания, а затем в буфер строки (так же называемый страницей). Процесс чтения строки данных разрушителен для зарядов емкостей, поэтому в некоторый момент необходимо их перезаписать.

Вторую фазу называют Column Address Strobe (CAS) – строб адреса колонки. Перед входом во вторую фазу контроллер памяти должен выждать время tRCD – RAS CAS задержка. В этой фазе необходимое слово выбирается из ранее выбранной строки. Латентное время (задержка) CAS (tCL) есть время от установки адреса колонки до получения данных контроллером памяти. Это время измеряется не в наносекундах, а количеством тактов сигнала Clock. Когда транзакция полностью завершилась, контроллер должен инициировать команду Precharge (пердзарядка), если при следующей транзакции будет обращение к другой строке. Precharge закрывает строку памяти, которая была использована. RAS Precharge Time (tRP) – время между командой Precharge и следующей командой активизации. Таким образом, следующая транзакция не может начаться, пока это время не истечет. Как мы видим общее латентное время выполнения транзакции состоит из трех латентных времен. Если мы будем продолжать использовать загруженную в усилители считывания строку, то можно получать данные за много меньшее латентным временем. Для этого конечное устройство памяти и контроллер памяти поддерживают команду пакетной или групповой передачи (Burst). Это тот случай, когда соседние данные поставляются с минимальной задержкой. Обычный размер пакета равен 4 двойным словам (двойное слово равно 32 бита). Групповая передача широко используется контроллером для подкачки кэш-памяти.

Поддержка пакетной передачи предполагает, что стартовый адрес выровнен по размеру пакета. Например, если 32-разрядная DRAM с размером пакета 4, то стартовый адрес для пакета был бы равен 0xXXXXXX00, а последующие адреса – 0xXXXXXX04, 0xXXXXXX08 и 0xXXXXXX0C. Если, однако, процессор ожидал последнее двойное слово из пакета по адресу 0xXXXXXX0C (при подкачке кэш-памяти), то использование контроллером команды пакетной передачи уменьшило бы производительность процессора ожидающего результат чтения. Для преодоления этого контроллер поддерживает операцию называемую Critical Word First (CRW – Критическое Слово Первым). Операция CRW заставляет процессор ждать только первое слово пакета, а затем контроллер возвращает остальные слова пакета. Для нашего примера контроллер вернул бы данные в следующем порядке адресов: 0xXXXXXX0C, 0xXXXXXX00, 0xXXXXXX04, 0xXXXXXX08. Это уменьшило бы общее латентное время чувствительного процессора, но только в случае, если данные необходимые для подкачки кэш-памяти локализованы в пределах активной строки.

Другая оптимизация подсистемы DRAM связана с политикой удержания активной строки (открытой страницы) как можно дольше. Выше было отмечено, что последовательный доступ к открытой странице занимает меньше времени, чем требуется для загрузки новой. Замечено, что программы часто имеют высокую степень пространственной и временной локализации, поэтому для контролера наилучшим является сохранение страницы открытой до наступления переключения на новую страницу. Задержка возникает при открытии новой страницы. Действие по сохранению пока неиспользуемой страницы открытой, известно как политика открытой страницы. Если контроллер закрывает действующую страницу сразу после завершения транзакции, то такое действие называют политикой закрытия страницы.

Размер страницы относительно большой. Например, в Embedded Atom SoC контроллер поддерживает память с 1 К, 2 К и 4 К страницами. Для реализации политики открытой страницы DRAM разбивается на независимые банки (расслоение памяти), каждый со своей поддержкой страницы (свой буфер строки). Контроллер в SoC, например, поддерживает 4 и 8 банков. С таким количеством банков контроллер памяти теперь может сохранять несколько открытых страниц, увеличивая шансы пакетной передачи без задержек для различных адресных потоков. Встроенные системы часто обладают большей гибкостью и в управлении выделением и разделением памяти. Увеличение производительности может быть достигнуто распределением различных структур по различным банкам. Например, увеличение примерно на 5% в некоторых случаях может быть достигнуто в простом сетевом роутере, в котором назначение памяти для сетевых пакетов, коду операционной системы и приложениям происходит для каждого из них в разных банках.

Устройства памяти выпускаются различного объема и разной разрядностью шины данных (x4, x8, x16, x32). Требуемый объем памяти может быть получен с помощью различных их комбинаций. Например, можно

использовать одно 32-разрядное устройство объемом 256 M или два 16-разрядных устройства объемом по 128 M. Необходимо в любом случае конфигурировать контроллер DRAM для обеспечения корректной работы. Выбор рационального решения связан с разрядностью элементов платформы, площадью монтажа и необходимым объемом. Решение имеет также экономическое измерение: элементы памяти большего размера более дорогие.

Каждая строка DRAM должна проходить цикл регенерации с частотой, гарантирующей предотвращение разряда емкостей ячеек. Современные DRAM должны постоянно полностью регенерировать с периодом 64 ms. Если устройство содержит 8192 строки, то цикл регенерации строки составляет 7.8 mks. Цикл регенерации состоит из последовательности фаз CAS перед RAS, в которой адрес строки определяет строку, которая должна быть регенерирована. Некоторые DDR устройства имеют собственные счетчики регенерируемых строк, которые обновляют адрес строки во время цикла регенерации. В этом случае контроллер DRAM должен только запускать цикл регенерации, но не отслеживать адреса строк. Контроллеры поддерживают само-регенерацию, когда для снижения потребляемой энергии система переходит в состояние сна (понижается тактовая частота, а, следовательно, и потребление энергии в этом состоянии). На сохранение заряда емкости ячейки влияет температура. Если память должна эксплуатироваться в расширенном диапазоне температур, частота регенерации должна быть увеличена.

Даже если DRAM регенерируется с требуемой частотой, существует небольшая вероятность того что емкость самопроизвольно изменит свою величину или может произойти ошибка при чтении бита или нейтроны космической радиации нанесут удар по группе ячеек. Эти ошибки известны как сбои (soft error). Для уменьшения эффекта от таких сбоев в устройства добавляют избыточные разряды. Они используются для реализации либо только для обнаружения ошибок (проверка четности: 1 избыточный разряд на 8 информационных разрядов) либо обнаружения двукратной и исправления однократной ошибки (Error Correcting Code – ECC). В первом случае, когда контроллер обнаруживает ошибку четности при чтении данных из DRAM, он генерирует сигнал исключения для процессора. В случае когда обнаруживается однократная ошибка ECC, она исправляется «на лету» и процессор получает корректные данные. Ошибка может продолжать находится в памяти, поэтому контроллер автоматически записывает скорректированное значение, обратно в память. В некоторых системах контроллер генерирует сигнал исключения, а программное обеспечение должно само позаботиться о записи прочитанных данных обратно в память. В случае двукратной ошибки она только обнаруживается и генерируется сигнал исключения. Когда в память записываются новые данные, контроллер вычисляет новый ECC и записывает его вместе с данными. После включения питания нет гарантии того, что избыточные разряды памяти соответствуют данным. Программа загрузчика или в некоторых случаях контроллер памяти должны записать во все ячейки корректные значения ECC до их

первого чтения. Так как ECC исправляет однократную ошибку, становится разумным организовать фоновый процесс известный как очищение (scrubbing). Периодически читается вся память, и восстанавливаются ошибочные разряды. Это препятствует возникновению двукратных ошибок, а, следовательно, сбоям памяти.

В соответствии со стандартом JEDEC интерфейс SDRAM должен состоять их следующих сигналов:

- Clock – тактовая частота;
- A0…An – адрес строки/колонки;
- D0…Dm – вход/выходные данные;
- B0…Bk – номер банка;
- /RAS – строб адреса строки;
- /CAS – строб адреса колонки;
- /WE – операция чтения/записи;
- /CS – выборка кристалла;
- DQM0…DQMm/8 – маска байт данных (разрешение выдачи соответствующих бай слова данных памяти).

Процессор формирует полный адрес памяти не структурированный на строки, колонки и банки. Поэтому контроллер DRAM должен сформировать сигналы A0…An для строки и столбца из полного адреса.

Существует два варианта чередования адресов строк, столбцов и банков в полном адресе для SDRAM. Первый называют чередованием страницы. Полный адрес ячейки памяти структурируется следующим образом (слева старшая часть адреса):

номер строки – номер банка – номер колонки – номер байта.

Второй вариант называют чередованием банка:

номер банка - номер строки – номер колонки – номер байта.

1.3.1.3. Энергонезависимая память

Все встроенные системы требуют тех или иных форм энергонезависимой памяти. Сегодня используются в основном две технологии: твердотельная память – наиболее распространенная для встроенных систем, и магнитная память в форме жестких дисков.

Современную твердотельную память обычно называют флэш-памятью (медленнее DRAM, но быстрее жестких дисков). В качестве запоминающего элемента используется МОП транзистор с плавающим затвором. Существуют два различных типа устройств флэш-памяти: NOR и NAND. Они получили такие название благодаря логическим элементам, используемым в конструкции ячеек памяти, и различаются по многим аспектам. Ключевое отличие состоит в том, что NAND-флэш более емкая (мегабайты для NOR и гигабайты для NAND). Большинство коммерческой внешней твердотельной памяти, такой как USB и SD-карты, используют NAND-флэш устройства вместе контроллером доступа к ним. В случае встроенных систем устройства флэш-памяти подключаются через соответствующий интерфейс и напрямую припаиваются к печатной плате.

1.3.1.3.1 NOR флэш-память [1]

Микросхемы NOR-флэш состоят из нескольких банков. Каждый банк содержит несколько секторов. Эти сектора можно индивидуально стирать. Обычно можно читать из одного банка, в то время как идет программирование или стирание другого. После включения питания в микросхеме NOR-флэш устанавливается режим чтения. В этом режиме данные доступны по чтению точно также как и в RAM. При выполнении в микросхеме операций стирания и записи программа должна записать специфические команды по специальным адресам и образцовые данные в регистры устройства. Перед программированием данных для части флэш прежде всего ее необходимо стереть. Можно стереть сектор или всю флэш. После стирания все биты устанавливаются в 1.

Ниже приведен пример программы на Си, которая стирает сектор и записывает слово (для микросхемы типа WORD – 16-разрядная шина данных). Это базовая операция и поэтому существует много вариантов увеличения ее производительности.

```
//Стирание сектора (Sector Erase)
// ww(address,value): подпрограмма записи слова
unsigned short sector_address;
ww(sector_address + 0x555), 0x00AA); // Запись 1 цикла разблокировки
 ww(sector_address +0x2AA), 0x0055); // Запись 2 цикла разблокировки
ww(sector_address + 0x555), 0x0080); // Запись команды setup
ww(sector_address + 0x555), 0x00AA); // Запись 1 дополнительного цикла разблокировки
ww(sector_address + 0x2AA), 0x0055); // Запись 2 дополнительного цикла разблокировки
ww(sector_address) ,0x0030); // Запись команды sector erase
// Выполнение полинга для установления завершения выполнения команды
// Программирование слова data по адресу program_address
ww(sector_address+ 0x555), 0x00AA); // Запись 1 цикла разблокировки
ww(sector_address+0x2AA), 0x0055); // Запись 2 цикла разблокировки
ww(sector_address+ 0x555), 0x00A0); // Запись команды write program setup
ww(program_address), data); // Запись слова для программирования
// Выполнение полинга для установления завершения выполнения команды
```

Типичный асинхронный параллельный интерфейс микросхемы NOR-флэш содержит следующие сигналы:

- A0…A20 – адрес ячейки;
- D0…D7 (D0…D15 для устройств типа WORD) – вход/выходные данные;
- WE (Write Enable) – инициализация записи данных;
- OE (Output Enable) – инициализация чтения данных;
- CE (Chip Enable) – сигнал выборки устройства;
- WP (Write Protect) – защита записи: если сигнал активен, то это препятствует стиранию или записи. С целью предосторожности он подключается

к переключателю на плате или подключается к линии порта общего назначения для программного управления.

Для получения сведений о флэш-памяти (размер, тип и производительность) многие устройства поддерживают стандарт «Common Flash Interface» (JEDEC 137-A и JESD68.01), разработанный для поддержки системной интеграции различных устройств.

Во многих микросхемах выделяется загрузочный сектор. Он по сравнению с остальными имеет более робастную защиту от записи. Обычно имеется отдельный вывод для установки определенного уровня напряжения, разрешающего запись в загрузочный сектор. Загрузочный сектор обычно содержит код необходимый для загрузки платформы (первый код, который выбирает процессор). Загрузочный сектор содержит достаточный объем для поддержки продолжения загрузочной последовательности и нахождения области оставшейся части программы загрузки и что важно содержит программу восстановления позволяющую перепрограммировать оставшуюся часть флэш.

Начальное программирование микросхемы флэш-памяти, припаянной к плате, обычно выполняется через JTAG-цепочку, к которой кабелем подключается JTAG-программатор. Он выполняет программирование, эмулируя поведение процессора при взаимодействии с флэш через тот или иной интерфейс при операциях запись/чтение. Микросхемы флэш-памяти могут поддерживать как параллельный (приведен выше) так и последовательный интерфейсы доступа.

Микросхема NOR-флэш обладает большей достоверностью и менее подвержена искажению бит, чем NAND-флэш. В результате микросхемы NOR-флэш рассматриваются как наиболее безопасные для запоминания начальной программной последовательности процессора (хотя NAND-флэш снабжаются средствами для подавления искажения бит путем дублированием программы загрузчика или избыточным кодированием ECC).

Благодаря возможности произвольного доступа в NOR-флэш, ей естественно поддержать операцию XIP (eXecute in Place), когда программа может напрямую исполняться из флэш минуя копирования данных из флэш в DDR.

Современные микросхемы флэш поддерживают свыше 100000 циклов перезаписи до выхода из строя. Когда микросхема флэш используется для файловой системы или сохранения логов часто меняющихся данных, сектора часто стираются и программируются. Программные драйверы должны обеспечить такой порядок при котором вся активность не фокусировалась на небольшом числе секторов. Эта стратегия известна как «равномерный износ» и все файловые системы поддерживают этот принцип.

В дополнении к перепрограммируемой памяти многие микросхемы содержат однократно программируемую область (OTP – One Time Protect/Programmable). Она используются для хранения постоянной информации, такой как IMEI/ESN, защищенный загрузочный код или SIM-блокировку. OTP программируется при производстве и не может быть больше обновлена.

1.3.1.3.2. NAND флэш-память [1]

В отличие от NOR-флэш NAND-флэш не поддерживает интерфейс произвольного доступа к памяти. Микросхемы NAND-флэш работают в режиме страница/блок (page/block). Перед чтением драйвер устройства сначала запрашивает страницу для доступа точно также как и драйвера других устройств вторичной памяти и это прямой путь для построения файловой системы на базе NAND-флэш. Стирание и программирование выполняются в блочном режиме. В отличие от NOR-флэш программирование блока выполняется последовательно в порядке возрастания адресов. Каждый блок имеет несколько страниц. Размер страницы обычно 2 К (рис. 1.13). Время необходимое для загрузки страницы в регистр страницы относительно велико по сравнению с временем последовательного чтения из регистра страниц. Для увеличения производительности многие микросхемы содержат несколько регистров страниц.

Микросхемы NAND-флэш могут содержать плохие страницы, которые должны отслеживаться контроллером (программным или аппаратным), обеспечивающим интерфейс с микросхемой. Для увеличения выхода годных микросхем они могут выпускаться с маркировкой плохих блоков. Микросхемы NAND-флэш могут начинать свою жизнь с плохими блоками и в дальнейшем деградировать в течение жизни. Найденные во время производства плохие блоки обычно маркируются в специальных битах (Spare) внутри каждого блока. Пользователь не должен стирать или программировать страницы помеченные как плохие. Увеличение плотности NAND-флэш сопровождается увеличением возможности спонтанного изменения значения бит. Для повышения достоверности вводится избыточное кодирование (ECC).

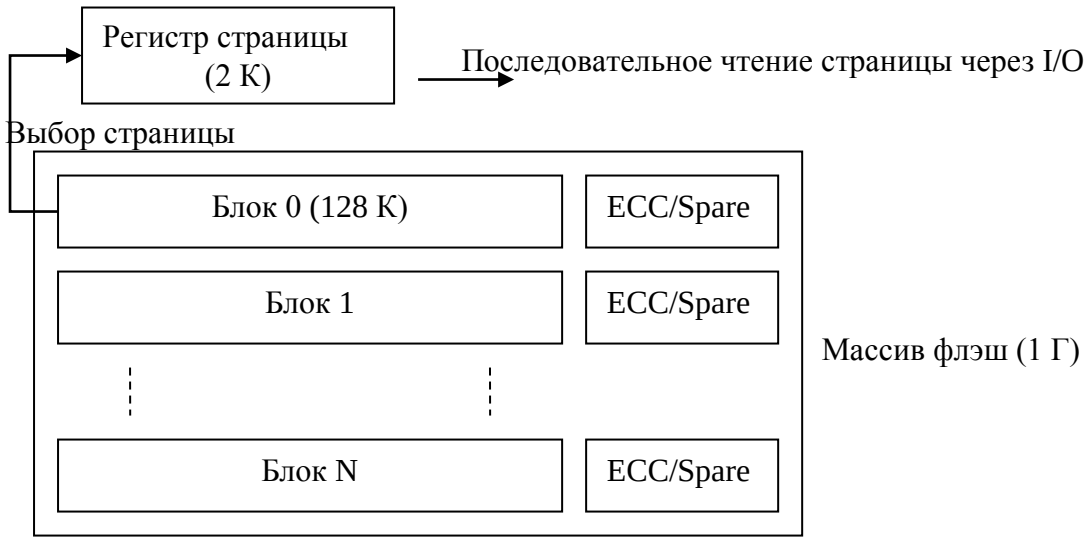


Рис. 1.13. структура NAND-флэш

«Родной» интерфейс микросхем NAND-флэш содержит следующие сигналы:

• CE (Chip Enable) – сигнал выборки устройства;
• Write – индикация направления доступа по записи/чтение;
• Read – индикация направления доступа по чтению;
• Command Latch Enable – защелка команды на выводах I/O;
• Address Latch Enable – защелка адреса;
• I/O0 … I/O7 (I/O0 … I/O15 для устройств типа WORD) – мультиплексированные входы/выходы для команд, адресов и данных;
• WP (Write Protect) – защита записи: если сигнал активен, то это препятствует стиранию или записи;
• Busy – индицирует возможность принять команду к исполнению.

Операция чтения состоит из загрузки страницы массива из флэш и затем чтения данных из регистра страницы. Базовый процесс чтения содержит следующие шаги:

1. Установка на линии I/O[0:7] команды чтение с кодом 0x00 и сопровождение ее сигналами Command Latch Enable и Write.

2. Установка бай за байтом на линии I/O[0:7] адреса и сопровождение каждого байта адреса сигналами Address Latch Enable и Write.

3. Установка второго цикла чтения подачей на I/O[0:7] значения 0x30 и сопровождение его сигналами Command Latch Enable и Write.

4. Линия Busy должна оставаться активной пока выполняется чтение данных из массива в регистр страницы.

5. Теперь данные байт за байтом могут быть выданы на I/O[0:7] в сопровождении импульсов сигнала Read.

Для уменьшения необходимости иметь в системе и NOR-флэш и NAND-флэш, некоторые микросхемы NAND-флэш обеспечиваются специальным загрузочным блоком. Загрузочным блоком разрабатывается таким же надежным, как и для микросхем NOR-флэш (используется ECC или уменьшается плотность, или создается область NOR-флэш). Загрузочная страница автоматически загружается в буфер чтения страницы, таким образом, процессор может начать произвольный доступ к этой странице при первой загрузке системы.

Рабочая группа ONFi (Open NAND Flash Interface) с 2006 года разработала ряд спецификаций интерфейса для NAND-флэш (http://www.onfi.org):
• ONFi 1.0 определяет электрический интерфейс и протокол, включающий базовое множество команд.
• Block Abstracted NAND 1.0 определяет управляемое решение, которое использует «сырой» интерфейс.
• ONFi 2.0 определяет высокоскоростной (удвоенная скорость) синхронный интерфейс до 133 М/сек.
• NAND connector 1.0 определяет физический соединитель для поддержки DIMM формфактора, облегчающего обновления.
• ONFI 2.1 увеличивает скорость передачи до 200 М/сек.
• ONFI 2.2 включает прерываемые операции чтение/стирание для поддержки высокоприоритетных чтений.

Некоторые SoC обеспечиваются «родным» интерфейсом NAND-флэш. Во многих случаях SoC включают контроллер NAND-флэш и микросхемы NAND-флэш подключаются по стандарту ONFi. В других случаях (главным образом асинхронные модели) используют низкоуровневое программирование для управления транзакциями. В простейшем случае (самый низко производительный) интерфейс может управляться через универсальные порты ввода-вывода.

Интерфейс для NAND-флэш является сложным, поэтому требуется контроллер, особенно если необходима высокая производительность. Контроллер NAND-флэш обычно встраивается в SoC. Он обычно поддерживает устройство прямого доступа к памяти (DMA), для которого сообщается адрес блока и указатель на область буфера данных. Затем контроллер выдает команды микросхеме флэш-памяти, используя механизм ONFi, читает регистр NAND-флэш, размещает результаты в буфере памяти и формирует сигнал прерывание, уведомляя о том, что страница прочитана.

Во многих случаях SoC не содержат контроллера NAND-флэш или прямой поддержки микросхем стандарта ONFi. В этом случае SoC обычно обеспечена альтернативным интерфейсом MMC (Multi-Media Card). Память MMC обычно принимает форму MMC карты. Определение в этом случае стандартизуется под управлением JEDEC. Определение включает контроллер флэш-памяти и микросхемы флэш-памяти. Для встроенных приложений существует спецификация eMMC (Embedded MMC – http://www.jedec.org/standards-documents/docs/jesd84-a441). в некоторых случаях микросхемы eMMC могут быть припаяны к печатной плате. микросхемы eMMC состоят из контроллера и памяти. Контроллер предлагает высокоуровневый прикладной интерфейс. Интерфейс SD (Secure Digital) более поздний стандарт, основанный на MMC. Стандарт для интерфейса хост-контроллера MMC/SD определяет SD Association. Большинство контроллеров покрывают операции SD и MMC/eMMC микросхем/карт.

1.3.2. Иерархия памяти

Многие приложения требуют значительного количества памяти, больше чем содержится на кристалле микроконтроллера, поэтому встроенные системы используют иерархию памяти, объединяющую различные технологии для увеличения общей емкости, а также для оптимизации стоимости, задержки и потребления энергии.

Обычно относительно небольшого объема SRAM на кристалле процессора используется вместе с памятью DRAM большого объема вне кристалла процессора. Эта память в дальнейшем может быть объединена с памятью третьего уровня, такой как твердотельная память, имеющая очень большую емкость, но без произвольного доступа, что увеличивает время записи и чтения. Прикладной программист может и не знать, что память так фрагментирована. Использование рассмотренной выше схемы виртуальной памяти делает эти разнообразные технологии памяти такими, что на взгляд программиста он имеет дело с непрерывным адресным пространством.

Операционная система и/или аппаратные средства обеспечивают механизм трансляции адресов, который преобразует логический адрес в физический адрес доступной технологии памяти. Эта трансляция часто выполняется специализированной частью аппаратуры под названием TLB (translation look-aside buffer или буфер быстрого преобразования), ускоряющей преобразование адресов. Для разработчиков встроенных систем эта техника может создать серьезные проблемы, т.к. трудно предсказать или понять как будет выполняться доступ к большой памяти. Разработчикам встроенных систем обычно необходимо более глубоко понимать системы памяти. чем программистам, создающих программы общего назначения.

1.3.2.1. Распределение или карта памяти

Карта памяти процессора определяет, как адреса отображаются на аппаратуру. Общий размер адресного пространства ограничивается разрядностью адреса процессора. 32-разрядный процессор, например, может адресовать 2^{32} ячеек или 4 Гбайт, полагая, что каждый адрес относится к одному байту. Разрядность адреса обычно совпадает с разрядностью слова (данных) за исключением 8-разрядных процессоров, у которых разрядность адреса обычно выше (16 бит). На рис. 1.14 в качестве примера приведена карта памяти процессора с архитектурой ARM CortexTM - M3.

Рис. 1.14. Карта памяти процессора с архитектурой ARM CortexTM - M3

Эта архитектура Гарвардского типа, поэтому область **A** может быть доступна по своей магистрали, а области **B** и **D** – по своей. Это удваивает полосу пропускания.

Некоторые реализации в кристалле этой архитектуры ограничивают ее карту памяти. Например, микроконтроллер Luminary Micro LM3S8962

имеет 256 KB flash памяти вместо допустимой flash памяти 0.5 GB. Эта память отображается на пространство 0x00000000 - 0x0003FFFF. Оставшееся пространство 0x00040000 - 0x1FFFFFFF осталось резервным. Это означает, что целевой компилятор не будет его использовать.

LM3S8962 имеет 64 KB SRAM, отображенные на 0x20000000 - 0x2000FFFF, маленькую область в **B**. Он также содержит периферию (устройства), доступную через область **C** (0x40000000 - 0x5FFFFFFF). К этим устройствам относятся таймеры, ADC, GPIO, UART и другие устройства ввода-вывода. Каждое из устройств занимает небольшое число адресов памяти для реализации регистров, отображаемых на память. Процессор микроконтроллера может записывать данные в некоторые из них для конфигурирования устройств или выдачи данных на их выходы. Некоторые из регистров могут быть прочитаны для ввода данных из периферии. Небольшое количество адресов из области магистраль частной периферии используется для доступа к контроллеру прерываний

К LM3S8962 можно подключить дополнительно внешние память DRAM (в область **E**) и устройства, такие как LCD дисплей, слот MicroSD и USB интерфейс. При этом остается много неиспользуемого адресного пространства. В архитектуре ARM введен механизм под названием bit banding (привязка бит), который позволяет через неиспользуемые адреса осуществлять доступ к индивидуальным битам в памяти и периферии.

1.3.2.2. Блокнотная и кэш память

Многие встроенные системы используют в своем составе память различных технологий. Некоторая память может быть доступна перед другой памятью. Говорят, что эта первая память является закрытой процессором. Например, закрытая память (SRAM) обычно используется для сохранения рабочих данных временно, пока процессор оперирует с ними.

Если закрытая память имеет отдельное множество адресов и программа отвечает за перемещение данных в них или из них и удаленную память, то такую память называют сверхоперативной или блокнотной (scratchpad).

Если закрытая память дублирует данные в удаленной памяти, автоматически копируя их туда и обратно, такую память называют кэш памятью.

Для встроенных приложений с ограничениями реального времени кэш память представляет значительный препятствие, т.к. ее временное поведение может сильно изменяться и это трудно предсказать. С другой стороны ручное управление данными в блокнотной памяти могут быть трудоемкими для программиста, а автоматические методы управления на уровне компиляторы находятся в начальной стадии развития.

Пусть процессор поддерживает виртуальную память и оборудован MMU для трансляции адресов, имеет карту памяти как на рис. 1.14 и пусть некоторому процессу может быть разрешено использовать логические адреса из области **D** (1 GB). MMU может реализовать кэш в области **B** достаточного объема. Когда программа выдает адрес памяти, MMU определя-

ет находится ли он в области **B** и если да, то транслирует адрес и выполняет выборку. Если нет, то фиксируется промах кэш и MMU управляет выборкой данных из вторичной памяти (область **D**) в кэш (область **B**). Если ячейки нет и в области **D**, то MMU фиксирует ошибку страницы, в результате программное обеспечение управляет перемещением данных из диска в память. Так программа создает иллюзию огромной памяти. Расплата за это - трудность в предсказании времени доступа.

Итак, кэш-память – это быстродействующая память буферного типа, заполняемая с умеренной скоростью из ОП. Кэш-память по сравнению с другими видами памяти MPS имеет минимальное время доступа и используется для хранения только тех данных, которые могут понадобиться CPU в ближайшее время.

В МП находится небольшая кэш-память для команд и небольшая кэш-память для данных (разделенная кэш-память первого уровня). Кэш-память второго уровня расположена вне МП и соединена с ним через высокоскоростную магистраль. Эта кэш-память обычно не является разделенной и содержит смесь данных и команд. Ее размер гораздо больше. Кэш-память третьего уровня – это статическое оперативное запоминающее устройство (ОЗУ) в несколько мегабайтов, работающее гораздо быстрее, чем динамическое ОЗУ. Обычно все содержимое кэш-памяти первого уровня находится в кэш-памяти второго уровня, а все содержимое кэш-памяти второго уровня находится в кэш-памяти третьего уровня.

Во всех типах кэш-памяти используется следующая модель. ММ разделяется на блоки фиксированного размера, которые называются строками кэш-памяти. Строка кэш-памяти состоит из нескольких последовательных байтов (обычно от 4 до 64). В любой момент несколько строк находятся в кэш-памяти. Когда происходит обращение к кэш-памяти, контроллер кэш-памяти проверяет, есть ли нужное слово в данный момент кэш-памяти. Если есть (попадание), то можно сэкономить время, требуемое на доступ к ОП. Если данного слова в кэш-памяти нет (промах), то какая-либо строка из нее удаляется, а вместо нее помещается нужная строка из ММ или из кэш-памяти более низкого уровня.

1.3.2.3. Кэш-память прямого отображения

Кэш-память прямого отображения является самым простым типом кэш-памяти. На рис. 1.15 приведен пример одноуровневой кэш памяти, содержащей 1024 блока по 32 байта.

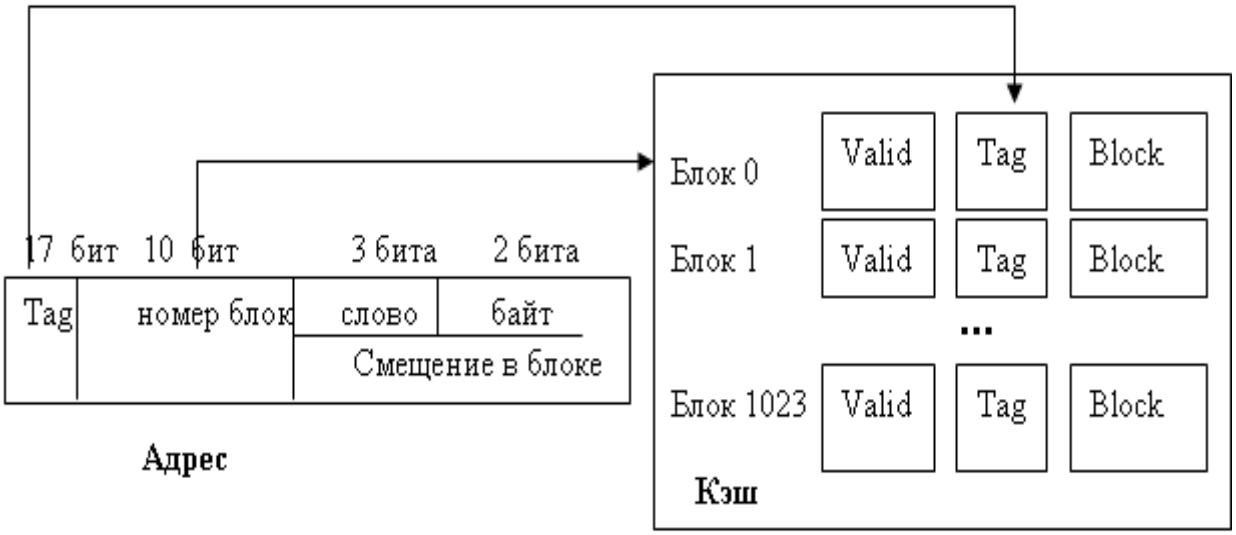


Рис.1.15. Кэш-память прямого отображения и формат адреса

Каждый элемент кэш-памяти состоит из трех частей:

1) Поле Tag (бирка) содержит старшие 17 разрядов адреса памяти, из которого поступил блок данных (адрес блока).
2) Поле Valid (действительно) содержит сведения о достоверности данных.
3) Поле Block содержит копию блока данных памяти.
4) В кэш-памяти прямого отображения данный блок может храниться только в одном месте по номеру блока. Когда процессор выдает адрес памяти, контроллер кэш-памяти выделяет из этого адреса 10 бит номера блока и использует их для поиска в кэш-памяти одного из 1024 элементов. Если этот элемент действителен, то производится сравнение поля Tag адреса и поля Tag кэш-памяти. Если поля равны, то попадание, иначе – промах. В случае попадания слово считывается из кэш-памяти. При промахе или недействительности, блок вызывается из памяти и сохраняется в кэш-памяти, заменяя тот блок, который там был.

1.3.2.4. Ассоциативная по множеству кэш-память

В кэш-памяти прямого отображения одноименные блоки ММ конкурируют за право занять одну и ту же область в кэш-памяти. Если программе часто требуются ячейки с адресами 0x00000000 и 0x0000800000, то будут иметь место постоянные конфликты и каждое обращение повлечет за собой замещение нулевого блока. Чтобы разрешить эту проблему, нужно сделать так, чтобы одноименные блоки помещались в нескольких местах (множествах) кэш-памяти.

На рис. 1.16. приведена ассоциативная по множеству кэш-память. Такая память сложнее предыдущей, поскольку требуется проверить все множества, чтобы узнать, есть ли нужный блок. Память для хранения тэгов адресов реализуется как ассоциативная, что позволяет исключить перебор при поиске места блока.

Если нужно поместить новый блок в кэш-память, то какой из старых блоков нужно заместить? Для этого хорошо подходит рассмотренный выше алгоритм LRU.

Обобщим параметры кэш памяти:

m – разрядность адреса памяти ;

$M = 2^m$ – величина адресного пространство в байтах;

$S = 2^s$ – число множеств кэш;

E – число линий (входов) в множестве;

$B = 2^b$ – размер блока в байтах;

t = m - s- b – число бит тега;

C – общий размер кэш в байтах.

Таким образом, кэш может быть охарактеризована четверкой (m,S,E,B), а общий размер кэш составляет C = S*E *B байт.

1.3.2.5. Обновление кэш-памяти

При работе с кэш-памятью одновременно могут существовать две копии одних и тех же данных: одна в кэш, а другая в основной памяти. В ММ данные может изменять не только процессор, но и устройства ввода-вывода, работающие по каналу прямого доступа в память. Поэтому необходимо поддерживать их непротиворечивость или когерентность. В MPS с несколькими МП, разделяющих общую ММ, также необходимо поддерживать когерентность, но уже и кэш этих МП.

Во всех решениях контроллер кэш-памяти разрабатывается так, чтобы кэш-память могла перехватывать запросы на магистрали, контролируя все запросы магистрали от других процессоров и устройств ввода-вывода, и предпринимать необходимые действия. Эти устройства называются кэш-памятью с отслеживанием (snooping cashe). Набор правил, которые выполняются кэш-памятью, процессорами и ММ, чтобы предотвратить появление различных вариантов данных в нескольких блоках кэш-памяти, формирует протокол когерентности кэширования. Единица передачи и хранения кэш-памяти называется строкой или блоком кэш-памяти (32 или 64 байта).

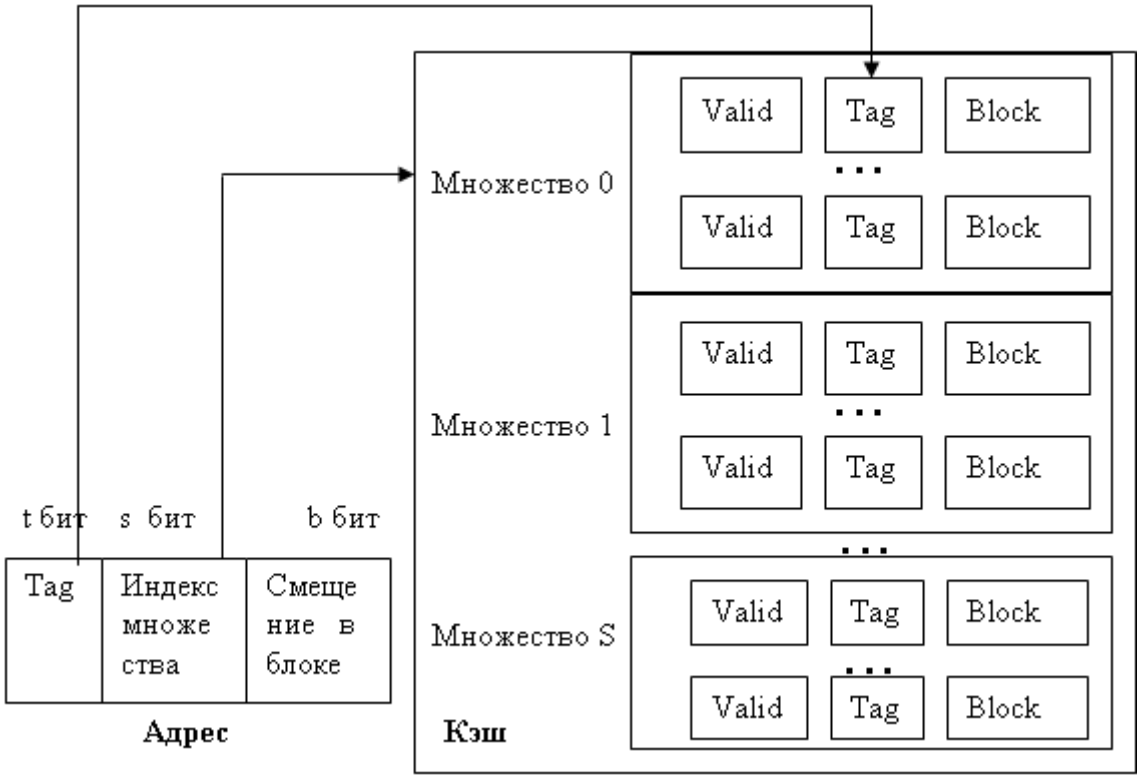


Рис. 1.16. Ассоциативная по множеству кэш и формат адреса

Самый простой протокол когерентности кэширования называется сквозной записью (write through). В случае промаха кэш-памяти при записи слова, которое было изменено, оно записывается в ММ. Строка, содержащая нужное слово, не загружается в кэш-память. В случае попадания при записи кэш обновляется, а слово плюс ко всему записывается в ММ. Всегда при обновлении кэш-памяти одновременно обновляется и основная память.

Другой вариант поведения при промахах по записи – загрузка кэш-памяти (политика заполнения по записи – write-allocate). Заполнения по записи эффективно, когда высока вероятность повторного обращения к слову, вызвавшему промах.

В устройствах с отслеживанием все кэши отслеживают все запросы магистрали, и всякий раз, когда записывается слово, оно обновляется в кэш-памяти инициатора запроса, обновляется в ММ и удалится из всех остальных кэшей.

1.3.2.6. Протокол когерентности кэширования с обратной записью

При большом количестве процессоров не эффективно каждый раз перезаписывать ОП. Вместо этого используется обратная запись (write back). Используется бит изменения в поле состояния кэша. Этот бит устанавливается, если блок был обновлен новыми данными и является более поздним, чем его оригинальная копия в основной памяти. Перед тем как перезаписать блок в кэш-памяти, контроллер проверяет состояние этого бита. Если он установлен, то контроллер переписывает данный блок и основную память перед загрузкой новых данных в кэш-память.

Один из популярных протоколов с обратной записью MESI. MESI – аббревиатура состояния кэш-памяти:

Invalid – блок кэш-памяти содержит недействительные данные.

Shared - в нескольких кэшах содержаться одинаковые блоки, основная память обновлена.

Exclusive – блок только в этой кэш-памяти, основная память обновлена.

Modified - блок действителен, основная память недействительна, копий блока не существует.

При загрузке процессора все элементы кэш памяти помечаются как недействительные I. При первом считывании из ММ нужная строка вызывается в кэш-память и помечается как E. Другой процессор (процессор 2) может вызвать ту же строку и поместить ее в кэш, но при отслеживании исходный держатель строки (процессор 1) узнает, что он уже не единственный, и объявляет, что у него есть копия. Обе копии помечаются состоянием S.

Процессор 2 производит запись в строку кэш-памяти в состоянии S и помещает сигнал о недействительности на магистраль, сообщая всем другим процессорам, что нужно отбросить свои копии. Соответствующая строка в процессоре 2 переходит в состояние M.

Процессор 3 считывает эту строку. Процессор 2, который в данный момент содержит строку, знает, что копия в ММ недействительна, поэтому он передает на магистраль сообщение, чтобы процессор 3 подождал, пока он запишет строку, обратно в память. После записи строки в ММ процессор 3 вызывает из памяти копию этой строки, и в обоих кэш строка помечается как S. Затем процессор 2 записывает в эту строку снова, что делает недействительным копию в кэш-памяти процессора 3.

Процессор 1 записывает слово в эту строку. Процессор 2 видит это и передает на шину сигнал, который сообщает процессору 1, что нужно подождать, пока строка не будет записана в ММ. После завершения этого действия процессор помечает собственную копию как недействительную, поскольку он знает, что другой процессор собирается изменить ее. Возникает ситуация, в которой процессор 1 записывает что-либо в некэшируемую строку. Если применяется политика write-allocate, строка будет загружена в кэш-память и пометится как M. В противном случае строка в кэш-памяти сохранена не будет.

1.3.2.7. Команды поддержки когерентности памяти

Устройства ввода-вывода MPS, работающее по каналу прямого доступа к ММ не содержат кэш-памяти, а следовательно и рассмотренных выше механизмов поддержки когерентности памяти. Вся ответственность ложится на CPU и реализуется программным путем. Для этой цели в системе команд процессора предусмотрены команды flush и invalidate.

Пусть программа процессора формирует данные для отправки через устройство ввода-вывода, помещая их в кэшируемую область ММ в виде

буфера выходных данных (кэш-память работает в режиме обратной записи). Это означает, что новые данные будут сформированы в кэши, а не в ММ. Устройство же ввода-вывода берет данные из ММ. Поэтому перед разрешением на передачу, процессор должен переместить подготовленные данные из кэш-памяти в ММ, выполнив команду flush. По этой команде данные из кэш-памяти перезаписываются в ММ. В команде flush указывается адрес и размер перезаписываемых данных.

Пусть устройство ввода-вывода принимает данные и помещает их во входной буфер ММ, минуя процессор. В кэш-памяти процессора могут остаться данные входного буфера от предыдущей обработки, помеченные как действительные. По завершении приема устройство ввода-вывода информирует процессор о готовности входного буфера. Для работы с этими новыми данными процессор должен объявить недействительными строки кэш-памяти, содержащие старые значения входного буфера. Поэтому процессор выполняет команду invalidate для соответствующих строк кэш-памяти. Команда invalidate объявляет недействительными строки кэш, содержащие данные старого входного буфера. В команде invalidate указывается адрес и размер недействительных данных.

1.4. Магистраль микропроцессорной системы

Каждая микросхема процессора содержит набор выводов, через которые происходит обмен информацией с внешним миром. Эти выводы подразделяются на три типа: адресные, информационные (данные) и управляющие, которые и являются основой трех шинной магистрали микросистемы. Эти выводы называют шиной адреса (AB - Address Bus), шиной данных (DB – Data Bus) и шиной управления (CB – Control Bus). Чтобы, например, выбрать команду, CPU помещает на шину адреса адрес команды. Затем формирует сигналы на одной или нескольких линиях шины управления, чтобы сообщить памяти о выполнении операции чтения. В ответ память помещает на шину данных требуемое слово и посылает сигнал о том, что это сделано. Когда CPU получает данный сигнал, он записывает выставленное слово в регистр команд.

Число адресных выводов и число выводов шины данных – два ключевых параметра, определяющих производительность CPU. При наличии m адресных линий можно обратится к 2*m ячейкам памяти. Обычно m=16,20,32,64. CPU, содержащий n линий шины данных, может считывать или записывать n-битное слово за одну операцию. Обычно n=8,16,32,64.

Линии шины управления регулируют и синхронизируют поток данных, а также выполняют другие разнообразные функции. Все шины управления содержат линии питания, “земли” и синхронизирующего сигнала. Остальные выводы разнятся от одной шины к другой. Тем не менее, линии шины управления можно разделить на несколько основных групп:

1. Управление шиной.
2. Прерывание.
3. Арбитраж шины.

4. Состояние.
5. Разное.

1.4.1. Циклы обращения к магистрали

Обмен данными через магистраль выполняется словами или байтами в виде следующих друг за другом обращений. За один цикл обращения к магистрали между CPU, MM и I/O передается от одного до нескольких байт. Существует несколько типовых циклов обмена. Среди них чтение памяти и запись в память. В случае архитектуры гарвардского типа, когда память программ и данных физически разделены, вводится также цикл чтения памяти программ. Рассмотрим простую магистраль со следующим набором сигналов управления в манере Intel:

RD (Read) – строб чтения памяти;

WR (Write) – строб записи в память;

READY – готовность к обмену.

Временные диаграммы передачи данных через магистраль однотипны и имеют вид, представленный на рис. 1.17. На диаграммах выходные данные памяти истинны в момент окончания строба RD, тогда как формируемые CPU выходные данные в течение действия сигнала WR.

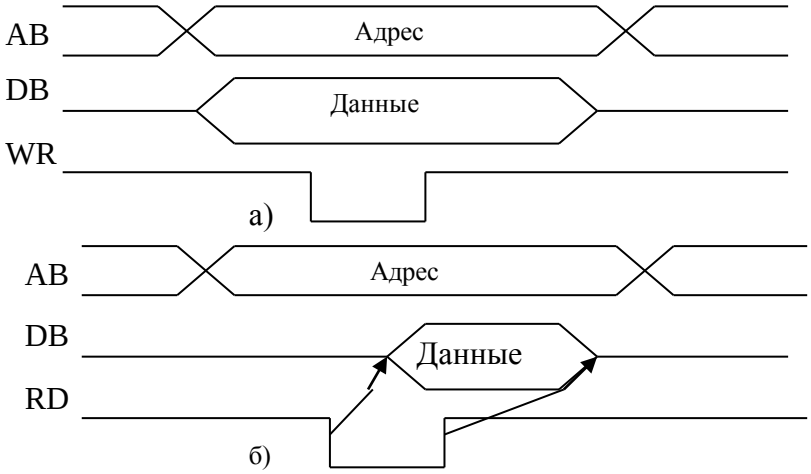


Рис.1.17. Циклы записи (а) и чтения (б) магистрали

В некоторых случаях, например, когда в MPS используются медленно работающие компоненты или некоторые из модулей еще не готовы к обмену по ряду причин, не зависящих от CPU, длительность стробов WR и RD могут оказаться недостаточными для правильного обмена со стороны компонента. Тогда для организации надежного завершения магистральной операции в состав магистрали вводят специальную линию READY.

В каждом цикле обращения к магистрали перед окончанием строба RD или WR CPU проверяет линию READY. При подтверждении обмена CPU завершает операцию на магистрали, в противном случае он переходит в состояние ожидания подтверждения, в котором остается до установления сигнала READY (рис. 1.18).

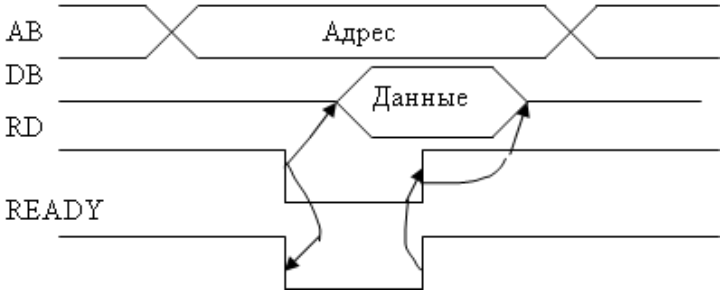


Рис.1.18. Цикл чтения с подтверждением обмена

Рассмотрим простую магистраль со следующим набором сигналов управления в манере Motorola:

STB – строб операции на магистрали;

W/R – тип операции запись/чтение;

ACK – подтверждение операции.

На рис. 1.19. приведена временная диаграмма цикла чтения в манере Motorola. По сигналу ACK CPU осуществляет прием данных с шины и завершает цикл магистрали.

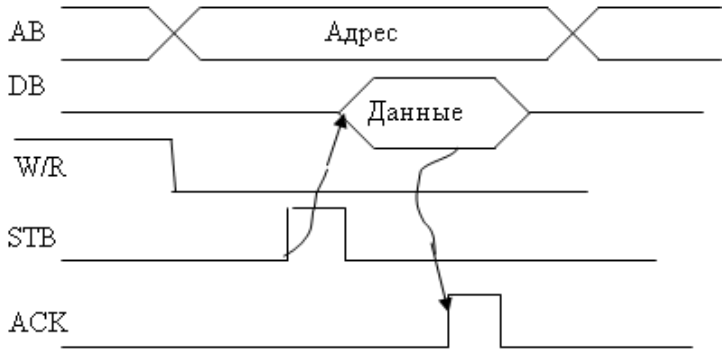


Рис.1.19. Циклы чтения с подтверждением обмена в манере Motorola.

1.4.2. Двухшинная магистраль

В CPU с целью сокращения ширины магистрали вводят совмещенную шину адреса/данных (AD), по которой передаются как адреса, так и данные. Этап передачи адресной информации отделен по времени от этапа передачи данных и стробируется специальным сигналом ALE (Address Latch Enable), который включен в состав СВ. На рис.1.20 представлена временная диаграмма работы такой магистрали.

Каждый модуль с двухшинной магистралью содержит локальный адресный регистр для запоминания адресной информации по сигналу ALE. Для фиксации адресной информации может быть использован и один общий регистр, в результате MPS с двумя шинами преобразуется в MPS с тремя шинами, как показано на рис. 1.21.

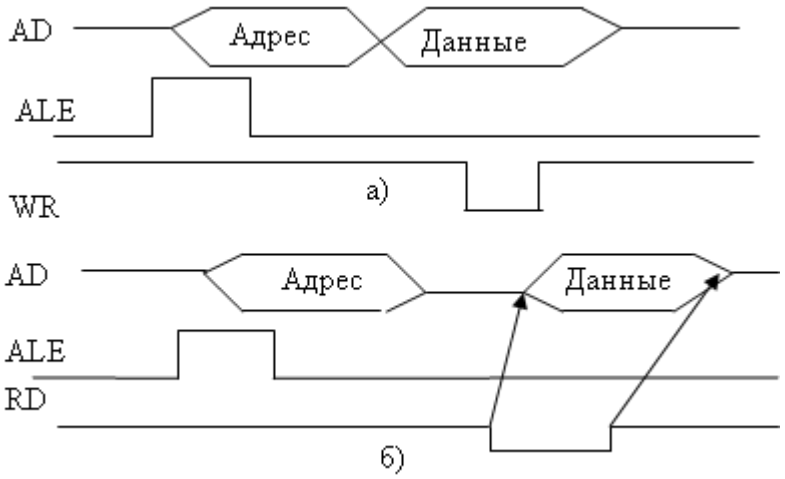


Рис.1.20. Циклы записи (а) и чтения (б) двухшинной магистрали

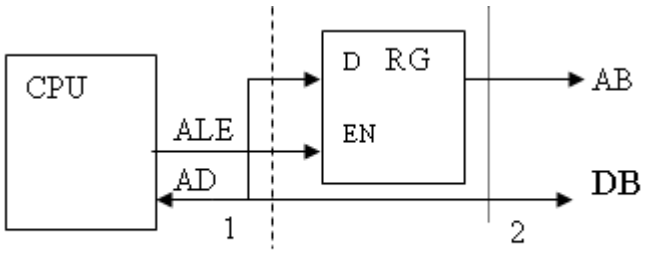


Рис.1.21. Преобразование магистрали: 1- двухшинная, 2-трехщинная магистраль

1.5. Контроллеры прерываний, устройств и интерфейсов устройств ввода-вывода

1.5.1. Контроллер прерываний [1]

Процессору необходима способность взаимодействовать с окружением через набор устройств ввода-вывода. Эти устройства обычно требуют быстрой реакции процессора для обслуживания событий реального мира. Прерывания обеспечивают этот механизм и исключают для процессора необходимость постоянно опрашивать (polling) устройства для обнаружения их внимания к себе. Очень часто имеется множество источников прерывания на некоторой данной платформе, поэтому необходим контроллер. Контроллер прерываний компонент, который собирает все аппаратные события от SoC и платформы и предоставляет их процессору. На этом фундаментальном уровне контроллер прерываний передает события процессорному ядру для обработки. Контроллер прерываний способствует идентификации события вызвавшего прерывания, так что механизм обработки исключения (exception) процессора может передать управление подходящей функции обработки. На рис. 1.22 приведена упрощенная возможная структура контроллера прерываний.

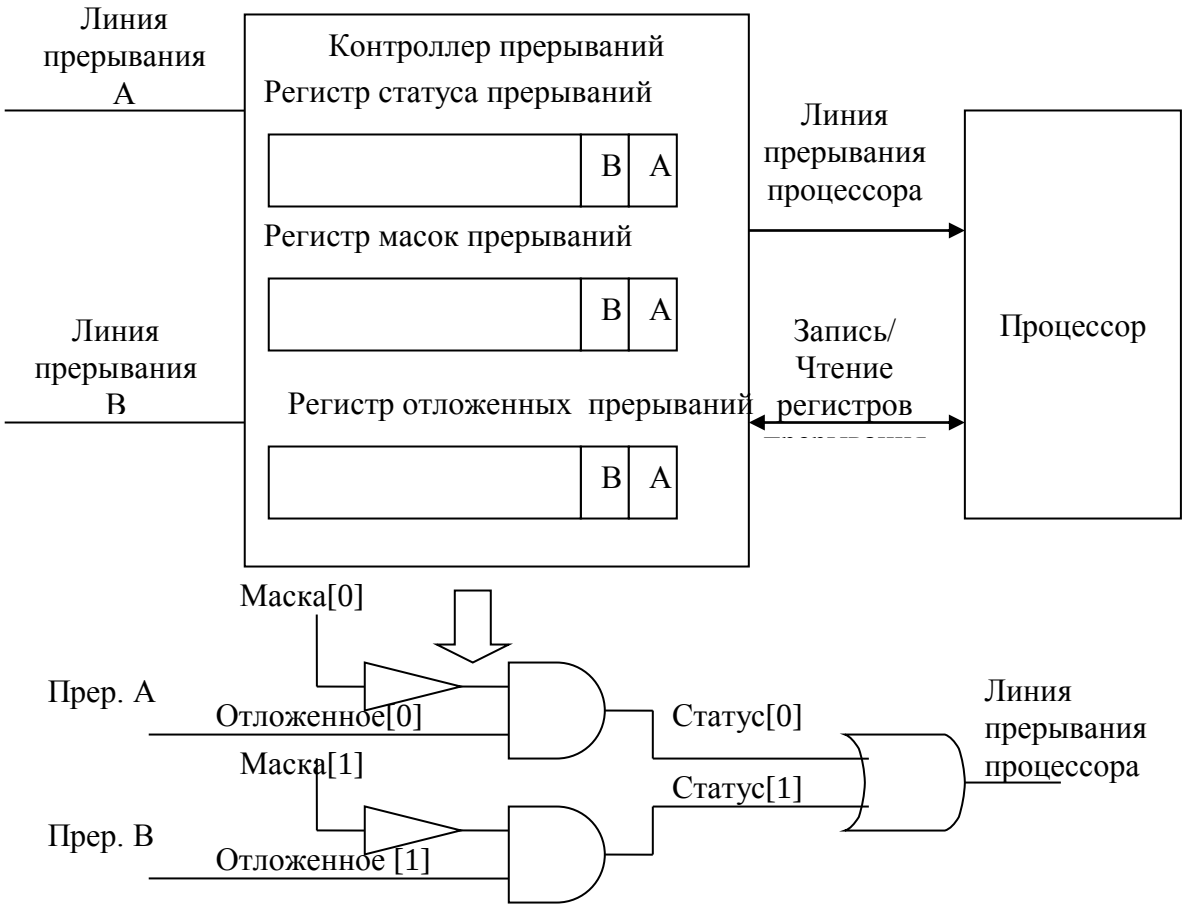


Рис.1.22.

Он содержит три регистра, которые может читать и записывать процессор. Регистры содержат битовые поля с одиночными битами, выделенными для каждого источника прерывания. Регистр статуса прерываний отражает текущее состояние входящих линий прерываний. Он устанавливается, когда запрос прерывания активен и очищается (0), когда нет отложенного прерывания. Второй регистр – регистр маскирования (разрешения) прерываний. Отложенные прерывания устройств могут быть не допущены до выборки процессорным ядром с помощью маски прерывания. Если в регистре маски бит установлен в 1, соответствующее прерывание не доберется до линии прерывания процессора. Регистр статуса прерывания показывает состояние немаскированной линии прерывания. Объединившись по ИЛИ, активные немаскированные прерывания генерируют прерывание для процессора. В рассмотренном примере есть только одна линия прерывания процессора. Когда линия прерывания становится активной, процессор сохраняет свое состояние и передает управление вектору внешнего прерывания. Это характерно для устройств с архитектурой ARM. Обработчик прерывания читает регистр статуса и анализирует активные биты согласно приоритету. Обычная реализация предполагает, что младшие биты, имеют более высокий приоритет. Поскольку этот процесс программно управляемый, то можно легко варьировать порядком проверки бит.

Регистр отложенных прерываний часто выполняется как регистр-защелка. Когда немаскированное прерывание становится активным, соот-

ветствующий бит в регистре отложенных прерываний устанавливается в 1. Затем даже если сигнал прерывания становится пассивным, бит в регистре отложенных прерываний продолжает оставаться установленным в 1. Начиная обслуживание прерывания, обработчик должен записать 1 в соответствующий разряд регистра для подтверждения обслуживания. Это называют «Запись единицы для очистки» (Write One to Clear).

Обработка прерывания с поиском приоритетного источника путем сканирования бит регистра увеличивает время идентификации. Во встроенных системах обычно пытаются уменьшить накладные расходы на идентификацию прерывания требующего обслуживания. Для этого вводят аппаратный блок, который берет немаскируемое активное прерывание и генерирует число в соответствии с аппаратной приоритетной схемой. Это число (номер прерывания) отражает высший приоритет отложенного прерывания. Номер прерывания сохраняется в регистре вектора с высшим приоритетом и используется для быстрого нахождения и выполнения соответствующего обработчика прерывания. Номер прерывания может быть получен CPU с помощью одного из двух механизмов. В первом механизме обработчик прерывания читает регистр вектора с высшим приоритетом. Прочитанное значение используется в качестве индекса программной таблицы векторов, содержащей указатели функций обработки. Обработчик прерывания просматривает таблицу и вызывает для входящего вектора требуемую функцию. Эта схема часто используется в устройствах на базе процессора ARM с одной линией прерывания.

Во второй схеме CPU сам генерирует цикл подтверждение прерывания, в котором автоматически возвращается номер прерывания. Номер прерывания автоматически преобразуется в вектор прерывания, которому передаст управление процессор. Обработчик прерывания вызывается в обеих схемах. В первой схеме есть программное воздействие на контроллер, которое исключается во второй схеме. Процесс чтения номера вектора прерывания из контроллера прерываний выполняемый программно ли, аппаратно ли, формирует свою характерную последовательность подтверждения прерывания. Эта характерная последовательность информирует контроллер, что процессор принял прерывание к обработке. Контроллер прерываний затем заново вычисляет прерывание высшего приоритета и обновляет регистры. В других реализациях контроллера устройство-источник прерывания должно быть очищено первым, после чего источник высшего приоритета обновится автоматически. На рис 1.23 приведены рассмотренные схемы.

Сигналы A и B могут быть сгенерированы как внутренней периферией так и внешними входами портов ввода-вывода. Линии прерывания обычно могут оперировать в следующих режимах (типы прерывния):

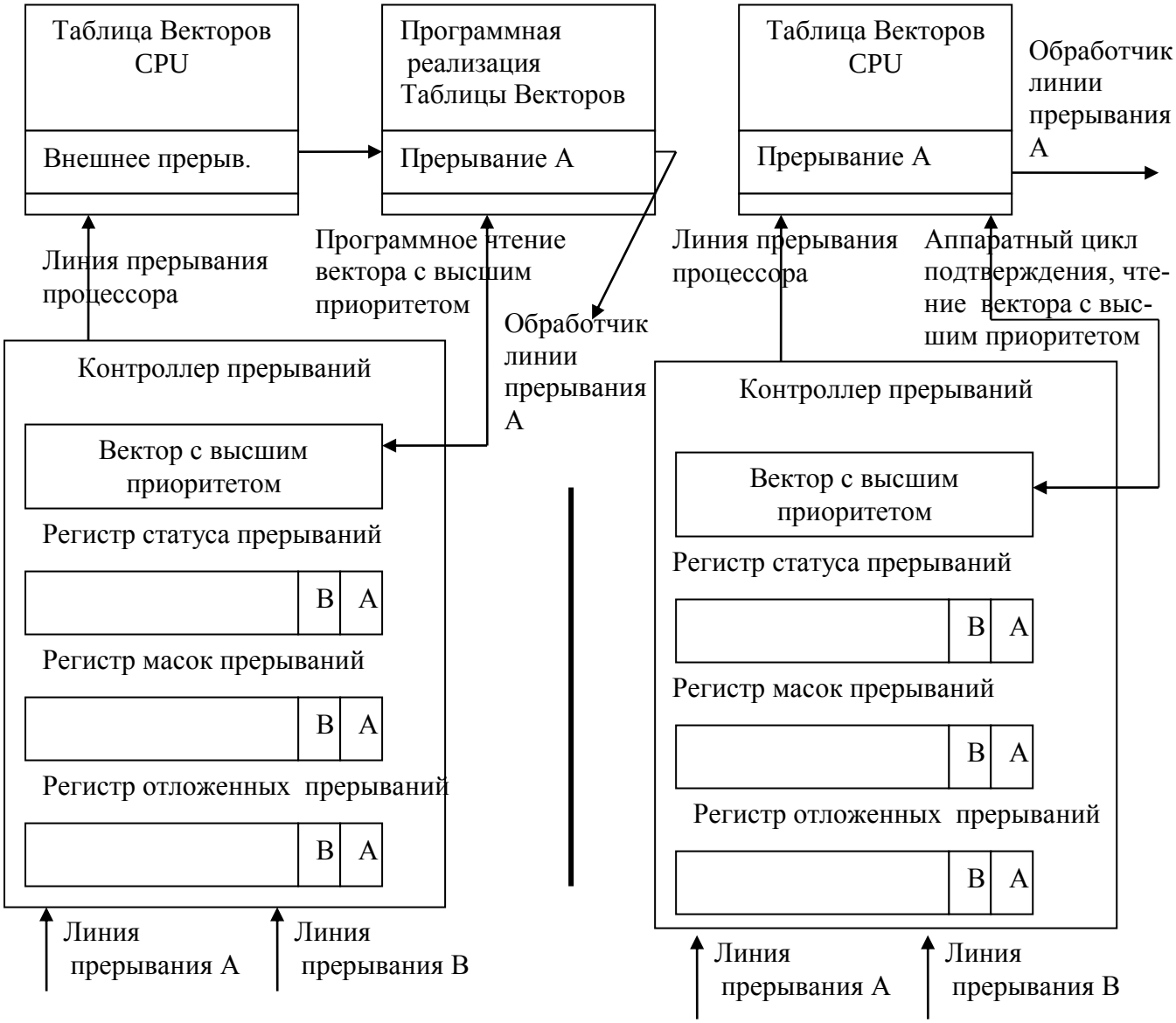


Рис.1.23.

• С запуском по уровню сигнала (level-triggered) – с активным низким или высоким уровнем. Это относится к тому как интерпретируется логический уровень. Если сигнал высокий и сконфигурирован как активный, то он индицирует запрос прерывания как активный, а когда низкой – как отсутствие прерывания. Такой режим часто используют, когда много устройств разделяют одну линию. Выходы прерываний от нескольких источников соединяют вместе на электрическом уровне.

• С запуском по перепаду сигнала (edge-triggered) – по восходящему или нисходящему, или по обоим переключениям сигнала. В этом случае факт переключения сигнала прерывания индицирует его активность. Если, например, сигнал прерывания переключается с уровня логического нуля на уровень логической единицы и сконфигурирован как восходящий, то переключение индицирует, что генерирующее его устройство запрашивает прерывание.

Система (и драйверы устройств) должны отдельно обрабатывать каждый тип прерывания по-разному. Уровневое прерывание остается активным

и отложенным для процессора пока уровень сигнала не переключится в пассивное состояние. Драйвер устройства обрабатывающий уровневое прерывание должен выполнить специальные действия, чтобы обеспечить возврат сигнала в неактивное состояние. Иначе устройство будет постоянно требовать прерывание. Прерывание с запуском по перепаду наоборот самоочищается. Во многих случаях устройства имеют несколько внутренних событий, заставляющих генерировать прерывания. Для прерываний с запуском по перепаду обработчик должен обеспечить рассмотрение всех случаев. В случае уровневых прерываний наоборот другие события автоматически прерывают процессор, если драйвер не очищает сразу все внутренние источники прерываний. Уровневые прерывания могут разделять общую линию с помощью монтажного ИЛИ.

.

1.5.2. Контроллеры устройства ввода-вывода

Подсистема ввода-вывода ответственна за связь с устройствами ввода-вывода I/O. Каждое I/O в общем случае состоит из двух частей: одна из них называется контроллером, а другая представляет собой само I/O.

С точки зрения программиста подсистему ввода-вывода можно представить или в виде отдельного пространства ввода-вывода (изолированное пространство ввода-вывода) со своими командами для доступа к нему или в виде определенной части адресного пространства памяти (совмещенное пространство ввода-вывода). В любом случае пространство ввода-вывода организовано в виде набора n-разрядных ячеек – портов (n=8,16,32) и линейно упорядочено.

Между CPU и I/O происходит обмен информацией двух типов: служебной и собственно данных. Служебная информация от CPU инициирует действия, связанные с обменом данных, и представляется с помощью управляющих слов CW (Control Word). Служебные сообщения от I/O, информирующие CPU о его текущем состоянии, называются словами состояния SW (Status Word). В отличие от них данные передаются с помощью слов данных DW (Data Word).

Объем служебной информации, которой обмениваются I/O и CPU , а также ее интерпретация зависят от типа I/O. Для наиболее простых устройств, служебная информация не нужна, а для других – управляющая информация и данные о состоянии УВВ могут иметь значительный объем.

Размер пространства доступа (совокупность портов) в общем случае не зависит от объема информации, которой обмениваются I/O и CPU . Распространена практика последовательной передачи массива информации через один и тот же порт. Это связано не только с экономией пространства ввода-вывода, но и с минимизацией ширины физического интерфейса I/O, а также с его стандартизацией. Существует соглашения об обмене информацией между I/O и CPU , называемые протоколами обмена. Эти протоколы являются основой для разработки драйверов (набор подпрограмм) I/O, организующих обмен данными.

В тех случаях, когда процедуры обмена информацией с I/O инициируются и выполняются непосредственно программой, реализуемой CPU, говорят о программно-управляемом обмене. Программно-управляемый обмен не является единственным типом обмена. Но судя по аппаратным затратам, это наиболее эффективный тип обмена, поэтому он находит самое широкое применение в MPS.

В наиболее простом виде процесс программно-управляемого ввода-вывода выполняется независимо от состояния I/O. Такой вид обмена назван прямым или безусловным. Процессы прямого ввода-вывода в чистом виде возможны только при условии, что I/O всегда готовы к обмену. К тому же они являются составными элементами более сложных процессов программно-управляемого обмена, к числу которых относится условный ввод-вывод. Существует два типа условного ввод-вывод: с занятием цикла (рис. 1.24,а) и совмещенный (рис. 1.24,б).

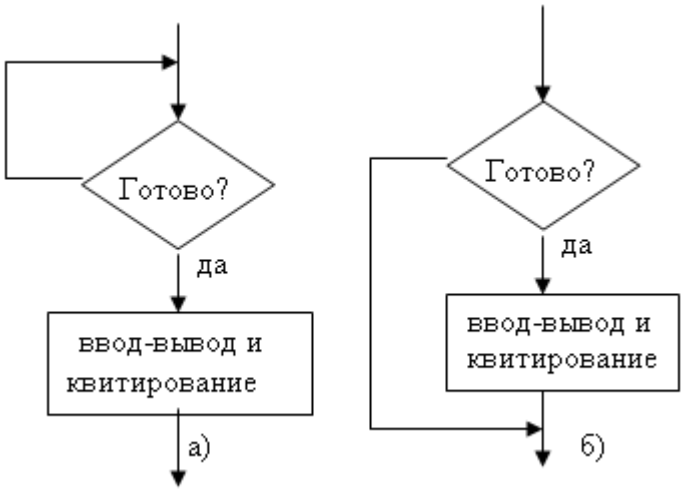


Рис. 1.24. Процедуры программно-управляемого обмена:
б – условного с занятием цикла: в – условного совмещенного

В первом случае MPS «зависает» в цикле ожидания готовности, тратя на это все процессорное время. Во втором случае, если I/O не готово к обмену, CPU возвращается к основной задаче без выполнения операции ввода-вывода. Однако он может снова проверить готовность УВВ к обмену и при удачном исходе выполнить его.

После завершения операции обмена сигнал готовности I/O должен быть снят и выставлен заново только при новой готовности к обмену. С этой целью I/O следует проинформировать об окончании операции, для чего используется включенный в одно из управляющих слов CW сигнал подтверждения. Протокол обмена служебной информации такого типа называется квитированием. Он обеспечивает надежную асинхронную передачу данных со скоростями, определяемыми I/O.

1.5.2.1. Порты ввода-вывода общего назначения

Порты ввода-вывода общего назначения (GPIO) - это группы выводов (чаще по 8 линий на порт) через которые μC получает сигналы (будь то аналоговые или цифровые) от внешних устройств (для их последующей обработки): управляет или обменивается данными с внешними устройствами, сигнализирует о проделанной работе и т.д. Режим общего назначения означает запись/чтение двоичных значений на линии порта в отличии от разнообразных альтернативные функции (АФ) каждого вывода. На рис. 1.25 в качестве примера приведена упрощенная электрическая схема одного вывода порта популярного микроконтроллера AVR.

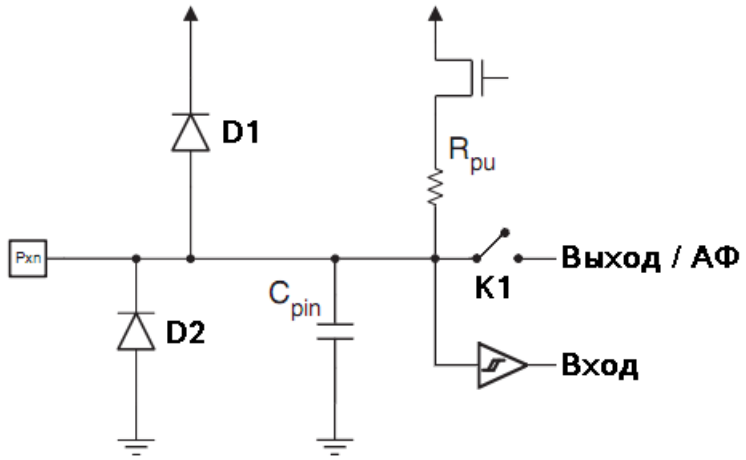


Рис. 1.25. Электрическая схема вывода линии порта AVR микроконтроллера

Диоды D1 и D2, предназначены для защиты вывода от напряжений, превышающих напряжение питания (электростатика - ESD и т.п.), а емкость Cpin - это паразитная емкость вывода. Rpu – встроенный подтягивающий (pull up) резистор, который может быть отключен от плюса источника питания полевым транзистором, что предаст линии порта свойство входа с открытым коллектором. Срабатывание ключа K1подключает к выводу порта Pxn двоичный источник или альтернативную функцию. Входной сигнал оценивает пороговый элемент с гистерезисом. Итак, любой вывод μC может быть настроен как на вход, так и на выход, причем читать состояние вывода Pxn можно в любой момент, даже, когда вывод настроен как выход или АФ.

Каждый GPIO имеет регистры данных и конфигурации (управления). Через эти регистры осуществляется доступ по чтению к входам, доступ по записи к выходам, а также позволяют настраивать индивидуальный вывод как вход или выход, подключать встроенные подтягивающие резисторы (pull up/pull down), входной гистерезис, задавать выходную нагрузочную способность и т.д.

Конфигурация выводов GPIO как вход или выход обычно выполняется через регистр, который называют регистром направления (DDR - Data Direction Register). Запись 1 в соответствующий бит DDR конфигурирует линию порта как выход, а 0 – как вход.

Наряду с величиной выходного напряжения GPIO важным является вытекающий ток при выходе равном «1» и втекающий ток при выходе равном «0». Если выход GPIO управляет входом другого цифрового устройства эти токи обычно очень маленькие (микроамперы), но когда управляет светодиодом (LED), биполярным транзистором (BJT) или реле токи становятся значительными (миллиамперы). Выходной ток GPIO для различных μC распределен в диапазоне 2 – 20 mA .

Рассмотрим подключения различных элементов к GPIO [12]. На рис. 1.26а показано подключение LED к выходу GPIO, возбуждаемое высоким уровнем сигнала (ток протекает через LED, когда выход имеет высокий уровень напряжения), а на рис. 1.26в - подключение, возбуждаемое низким уровнем сигнала (ток протекает через LED, когда выход имеет низкий уровень напряжения).

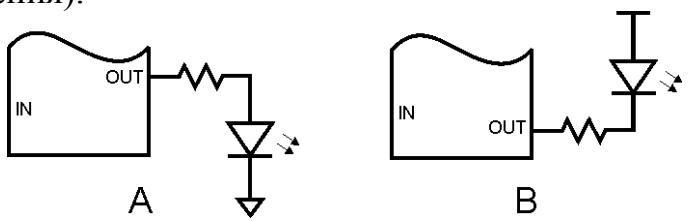


Рис. 1.26. Подключение светодиода к выходу GPIO

Формула для вычисления значения сопротивления: R = (Vcc – Vled)/Iled,где Vled и Iled - падение напряжения на LED и ток свечения LED соответственно.

На рис. 30 а и в показано подключение p-n-p BJT и n-p-n BJT к выходу GPIO по схеме с общим коллектором соответственно. Подключение на рис. 1.27а возбуждается высоким уровнем сигнала, при котором ток протекает через нагрузку RL , когда выход имеет высокий уровень напряжения. Подключение на рис. 1.27в, возбуждается низким уровнем сигнала, при котором ток протекает через нагрузку RL , когда выход имеет низкий уровень напряжения.

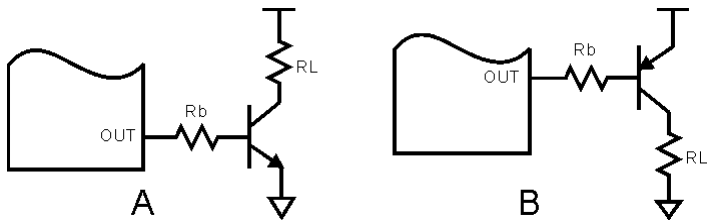


Рис. 1.27. Подключение биполярного транзистора к выходу GPIO

На рис. 1.28 а и в показано подключение N-канального/P-канального MOSFET к выходу GPIO возбуждаемое высоким/низким уровнем сигнала соответственно.

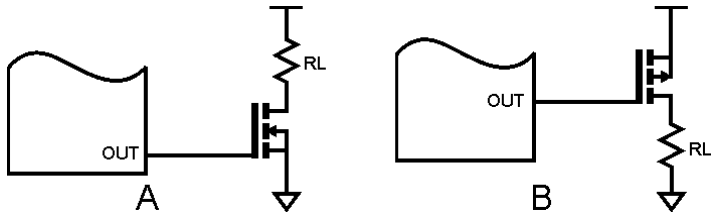


Рис. 1.28. Подключение MOSFET транзистора к выходу GPIO

На рис. 1.29а показано подключение переключателя к входу GPIO с подтягивающем к земле резистором. Входной сигнал является возбуждаемым высоким уровнем. Это означает, что вход принимает напряжение высокого уровня, когда переключатель замыкает контакты.

На рис. 1.29в показано подключение переключателя к входу GPIO с подтягивающем к питанию резистором. Входной сигнал является возбуждаемым низким уровнем. Это означает, что вход принимает напряжение низкого уровня, когда переключатель замыкает контакты.

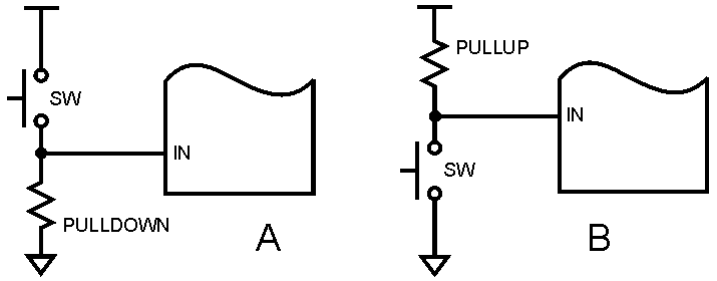


Рис. 1.29. Подключение переключателя SW к входу GPIO

1.5.2.2. Таймер-счетчик

Почти каждый μC имеет в своем составе несколько (иногда очень много) таймеров-счетчиков (для краткости будем говорить таймер, T/C), занимающих по полезности для встроенных систем второе место после GPIO. Название таймера отражает тот факт, что это устройство может быть сконфигурировано для подсчета регулярных тактовых импульсов (работает как таймер) или нерегулярных импульсов, выражающих некоторые события (работает как счетчик событий).

Обычно таймер содержит предварительный делитель частоты (prescaler), N-разрядный регистр таймера-счетчика (N = 8, 16 или 32), один или более N-разрядный регистр захвата (capture) и один или более N-разрядный регистр сравнения (compare). Так же могут быть регистры управления и статуса для управления и наблюдения за таймером-счетчиком.

Предварительный делитель частоты получает базовую тактовую частоту (это может быть тактовая частота CPU или более высокая или более низкая частота) и делит ее на некоторую величину перед подачей в таймер в соответствии с конфигурацией регистра предварительного делителя частоты. Величины предварительного делителя частоты могут иметь не-

сколько фиксированных значений или значений в диапазоне 1 - 2^P, где P – разрядность предварительного делителя частоты.

Назначение предварительного делителя частоты – обеспечить таймер нужной частотой для получения требуемой разрешающей способности по времени или максимального периода переполнения таймера.

Регистр таймера – это обычно N-разрядный счетчик, работающий на увеличение с возможностями записи или чтения текущего значения, остановкой и сбросом.

Регистр захвата – это регистр, который может автоматически загружаться текущим значением таймера при наступлении некоторого события (обычно изменение значения на некотором входе GPIO). Регистр захвата таким образом используется для получения «моментального снимка» таймера в момент появления события. Событие захвата может генерировать же прерывание. Регистр захвата может также использоваться для измерения интервала между двумя импульсами, определения времени импульса или паузы и определения времени между двумя различными входными сигналами.

Регистры сравнения (иногда называют совпадения) сохраняют величину, с которой постоянно автоматически сравнивается текущее значение таймера. Регистр сравнения используется для фиксации события, когда величина в таймере совпадает с величиной регистра сравнения. Если таймер-счетчик сконфигурирован как таймер, использование регистра сравнения дает возможность генерировать события (выход GPIO изменяется, и/или возникает прерывание, и/или сбрасывается таймер) в известное и точное время. Если таймер-счетчик сконфигурирован как счетчик, регистр сравнения генерирует события при достижении счетчика заданной величины.

1.5.2.3. Импульсно-кодовая модуляция

Современные μC имеет в своем составе аппаратные средства, позволяющие формировать сигналы импульсно-кодовой модуляции (PWM - Pulse Width Modulation). Такие сигналы широко используются для управления различными устройствами, например, изменение скорости вращения двигателей постоянного тока, поворот на требуемый угол шагового двигателя, изменение яркости свечения ламп или LED (диммеры).

PWM – это формирование последовательности импульсов прямоугольной формы с постоянным периодом и изменяемой величиной длительности импульсов. Такой сигнал характеризуется коэффициентом заполнения (duty cycle) равным отношению длительности импульса к периоду сигнала в процентах. С изменением коэффициента заполнения меняется среднее значение сигнала PWM и это можно рассматривать как вариант цифро-аналогового преобразования, где цифра это дискретные значения длительности импульса, а аналог – среднее значение напряжения.

Часто реализация функции формирования сигнала PWM ложится на рассмотренные выше таймеры-счетчики μC. Пусть имеется два числа, называемые «нижнее» (BOTTOM) и большее по величине – «верхнее»

(TOP). Пусть таймер стартует со значения BOTTOM, затем под действием тактовой частоты он достигает значения TOP и все начинается сначала, как на рис. 1.30. Получаем сигнал пилообразной формы.

Запишем в регистры сравнения таймера значение COMPARE. Если текущее значение «пилы» меньше чем COMPARE значение выхода GPIO остается низким, иначе высоким, как на рис.1.31. Сформировался сигнал с коэффициентом заполнения 50%.

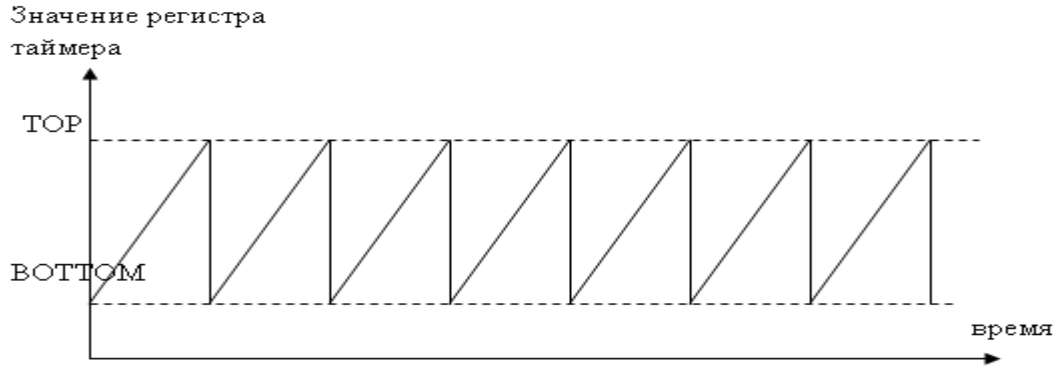


Рис. 1.30. Сигнал пилообразной формы таймера-счетчика

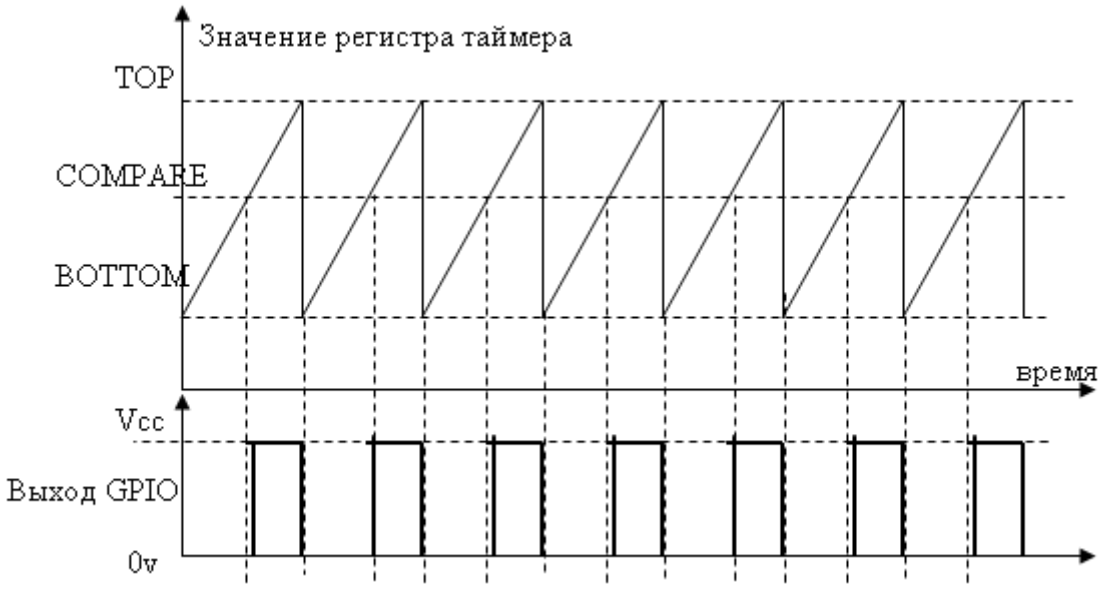


Рис. 1.31. Сигнал пилообразной формы, формируемый таймером-счетчиком

1.5.2.4. Многоканальный аналого-цифровой преобразователь

Как правило, μC, используемые во встроенных системах, имеет в своем составе многоканальный аналого-цифровой преобразователь (АЦП или ADC). Такой ADC преобразует аналоговый сигнал, поступающий с одного из M входов GPIO, в N-разрядный двоичный код.

На рис. 1.32 приведена упрощенная структурная схема многоканального ADC . Через Аналоговый MUX (мультиплексор) один и аналоговых входов ADCi подключается к схеме выборки/хранения (S/H), выходное значение которой преобразуется ADC в двоичный код. Преобразование за-

пускается по команде программного обеспечения, возвращающего затем полученный результат.

Компонент S/H схеме выполняет захват (ключ замкнут конденсатор C_INAD через резистор Z_INAD заряжается до величины входного значения) и хранение величины входного напряжения на время выполнения преобразования (ключ разомкнут и конденсатор C_INAD не имеет возможности быстро разрядится). На время захвата влияет выходное значение источника сигнала: чем оно выше, тем больше времени необходимо для захвата.

Существует большое количество методов преобразования. Базовый принцип, используемый в них, основан на применении компаратора для определения может или нет, тот или иной бит входить в результат преобразования со значением 1.

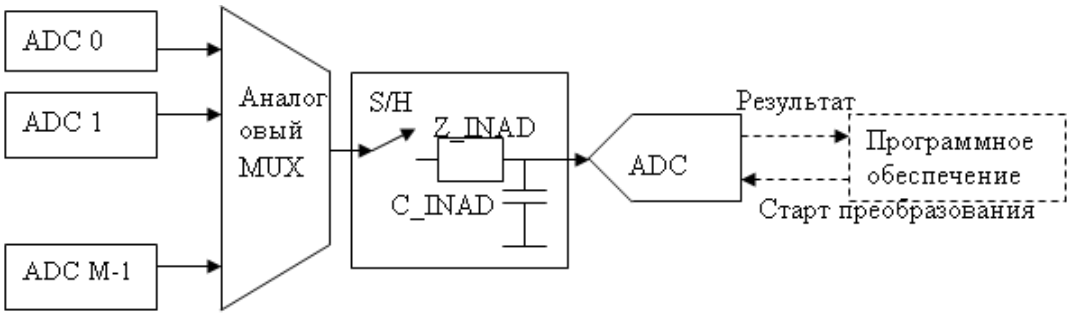


Рис. 1.32. Схема многоканального аналого-цифрового преобразователя

К наиболее используемым методам относятся: сравнения с пилообразным сигналом, последовательного приближения (или с поразрядным уравновешиванием) и параллельное прямое преобразования (Flash ADC).

В первом методе цифро-аналоговый преобразователь (ЦАП или DAC) увеличивает напряжение на своем выходе согласно последовательному увеличению значения двоичного N-разрядного счетчика, подключенного к его входу. Как только компаратор зафиксирует совпадение выходного значения DAC и аналогового входа, двоичный счетчик останавливается, сохраняя свое значение как результат преобразования. Этот метод грешен переменным временем преобразования, и оно тем больше чем больше величина входного аналогового сигнала.

Второй метод в среднем быстрее первого, время преобразования постоянно и равно N-тактам. В каждом такте преобразования решается вопрос, какое значение должен принять очередной i-разряд N-разрядного регистра результата на основании сравнения значения выхода DAC, к входу которого подключен регистр результата, и аналогового входа. Вначале преобразования наиболее старший значащий бит регистра последовательного приближения устанавливается в 1, остальные в 0. Это цифровое значение затем преобразуется DAC в аналоговую величину, соответствующую половине максимального входного значения. Если измеряемая входная величина превосходит величину DAC наиболее старший значащий бит регистра последовательного приближения остается в 1, иначе сбрасывается в 0. Этот процесс повторяется со следующим битом. Он останется в 1,если входная

величина находится во второй или четвертой четверти диапазона. И так для всех остальных бит.

Третий метод самый быстрый, но требует своего компаратора для каждого уровня квантования, т.е. 2^N-1 компараторов (например, при N=10 необходимо 1023 компаратора). Схема кодирования посредством таблицы истинности преобразует значения выходов компараторов в N-разрядный двоичный код. Поэтому обычно N=8, но это отрицательно сказывается на точности. ADC различных микроконтроллеров могут различаться такими свойствами, как:

– разрешающая способность (разрядность, может быть переменной);
– значением интегральной нелинейности:
– абсолютная погрешность;
– время преобразования (может быть переменным);
– производительность (может быть переменной);
– простой /дифференциальный аналоговый вход, количество входов;
– наличие входного усилителя с переменным коэффициентом усиления;
– допустимый диапазон изменения входного сигнала:
– набор опорных напряжений;
– режимы преобразования (однократный режим, режим с автозапуском, режим свободного хода);
– прерывание по окончанию преобразования;
– наличие механизмов шумоподавления.

1.5.3. Контроллеры интерфейсов устройств ввода-вывода

Во многих случаях SoC нуждаются в дополнительных специализированных устройствах ввода-вывода. Эти устройства могут быть подключены с помощью различны параллельных (высокопроизводительных) или последовательных интерфейсов (магистралей), контроллеры которых должны бать интегрированы в SoC. К таким интерфейсам предъявляются различные требования, которые должны поддерживать их контроллеры.

Последовательная передача и прием данных означает передачу и прием по одной линии только одного бита за раз. Это уменьшает до минимума количество необходимых проводников для передачи данных в отличие от параллельной передачи (высокопроизводительной), кода прием и передача выполняется одновременно по нескольким линиям – по биту на линию. Различаются последовательные интерфейсы тем, как биты данных передаются и принимаются.

Схема последовательной передачи называется *симплексной*, если поток данных передается и затем принимается только в одном направлении.

Схема последовательной передачи называется *полудуплексной*, если поток может передаваться и приниматься в одном из двух направлений, только в одном направлении в каждый момент времени.

Схема последовательной передачи называется *дуплексной*, если поток может передаваться и приниматься в одном из двух направлений одновременно.

Последовательная передача, выполняемая как непрерывный поток с регулярными временными интервалами, определяемыми тактовой частотой CPU, называется *синхронной передачей*, а с нерегулярными временными интервалами – *асинхронной передачей*.

Контроллеры интерфейсов в общем случае должны обладать следующими способностями:

• Захватывает транзакции процессора (запись или чтение) к адресу памяти выделенной для интерфейса и трансляция ее в соответствующую транзакцию интерфейса.

• Для высокопроизводительного взаимодействия через интерфейс важно иметь возможность внешним устройствам читать и писать в ресурсы SoC главным образом в память. Это часто называют мастерингом (mastering) интерфейса. Контроллер должен уметь преобразовывать адрес в интерфейсе в системный адрес и обратно для использования каналов прямого доступа в память.

• Высокопроизводительными взаимодействие с устройствами требуют механизма направления прерываний от устройства к процессору. Это может быть сделано просто индикацией событий на портах ввода-вывода общего назначения, сконфигурированных для генерации прерываний.

1.5.3.1. Асинхронный старт-стопный интерфейс UART

Устройство, реализующее стартстопную последовательную передачу данных, называют UART (Universal Asynchronous Receiver Transmitter). Передатчик разделяет порцию данных на символы (от 4до 8 бит или от 5 до 9 бит на символ). Каждый из символов инкапсулируется в кадр для раздельной передачи. Перед отправкой в линию каждый символ дополняется битом START в начале символа и битом/битами STOP (может бать 1, 1.5 и более бит) в конце символа. На рис. 1.33 приведена временная диаграмма старт-стопной передачи данных.

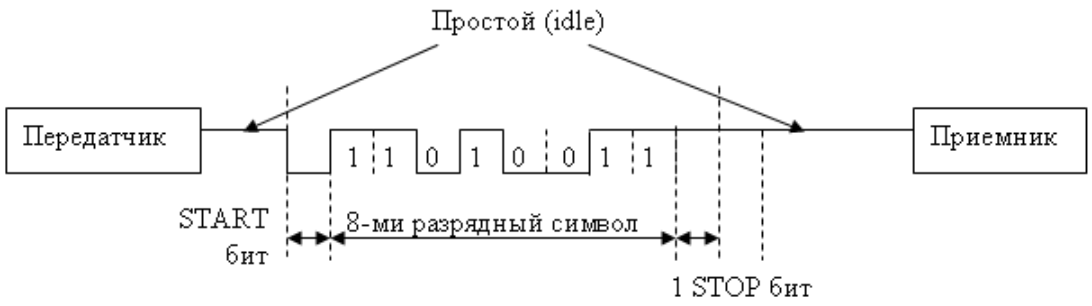


Рис. 1.33 Временная диаграмма старт-стопной передачи данных

При настройке UART назначаются:

– число STOP-битов (регулирует минимальный интервал между соседними символами, обычно выбирают равным значению 1 или 2, реже 1,5 битовых интервала);

– число бит данных (5…8);

– наличие или отсутствие проверочного разряда P (дополнение до четности или нечетности);

– скорость передачи V бит/сек .

Приемник на уровне символа синхронизируется переходом линии из состояния простой (1) в 0 (START-бит). Зная длительность битового интервала T=1/V и формат символа, приемник запускает тактовый генератор и последовательно считывает передаваемые биты, анализируя линию связи посередине битового интервала.

Синхронизация на уровне бит, основана на стабильности работы и точности задания частоты передающего и приемного генераторов на коротком интервале приема символа.

Асинхронность передачи вытекает из того, что приемник не знает в какой момент времени поступит очередной символ. Отсутствует единая для всего потока данных синхронизация, т.е. она имеет локальный характер – устанавливается заново всякий раз при обнаружении приемником начала очередного старт-бита.

Через UART к встроенной системе обычно подключаются микросхемы модемов различных беспроводных коммуникационных технологий, GPS-приемники. Это связано с историей возникновения UART как интерфейса для передачи данных между компонентами телекоммуникационного оборудованием DTE (Data Terminal Equipment - оконечное оборудование данных или компьютер) и DCE (data communication equipment - оборудование передачи данных или модем) и являющегося элементом стандарта RS-232.

В этом стандарте вход данных UART называн RxD, а выход TxD и ведены многочисленные служебные сигналы. Сейчас из них используется лишь следующие сигналы. RTS (Request To Send – запрос на передачу данных) и CTS (Clear To Stnd – готовность к передаче данных). Их первоначальное назначение (для отображения запроса и готовности передачи данных от DTE к DCE) в настоящее время зачастую игнорируется – эти сигналы могут альтернативно трактоваться как равноправные признаки готовности устройств DTE и DCE к приему данных от устройства-партнера. В асинхронном режиме сигналы RTS и CTS обслуживают оба направления передачи данных, что выходит за рамки стандарта и отражает некий “стандарт де-факто”. Поэтому наименования сигналов не соответствуют (и даже противоречат) их назначению. Выходной сигнал RTS теперь рассматривается как готовность DTE принять данные RxD от такого же устройства DTE. Аналогично входной сигнал CTS теперь рассматривается как разрешение на передачу данных для DTE.

На рис. 1.34 приведена схема соединения двух DTE (нуль-модемное соединение). Логика работы такова: передача данных в ту или иную сторону возможна только при условии, что приемник готов эти данные принять.

Если обнаружена неготовность приемника, то источник данных приостанавливает работу, ждет появления готовности , возобновляет передачу и т.д. Это – так называемое аппаратное управление потоком данных (hardware flow control)

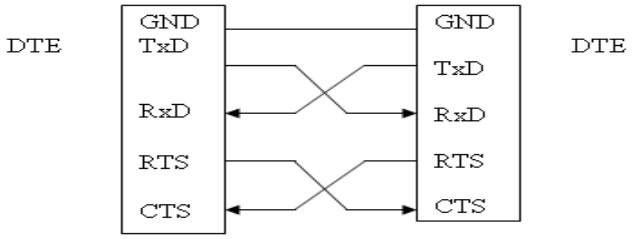


Рис. 1.34. Схема соединения двух DTE

Современные микроконтроллеры могут содержать несколько UART. Это позволяет достаточно просто реализовывать во встроенных системах коммуникационные функции.

1.5.3.2. Последовательный интерфейс SPI

SPI (Serial Peripheral Interface) – последовательный синхронный интерфейс предназначенный для связи CPU с компонентами MPS, расположенных в пределах печатной платы, по принципу ведущий-ведомый (Master - Slave), как показано на рис.1.35.

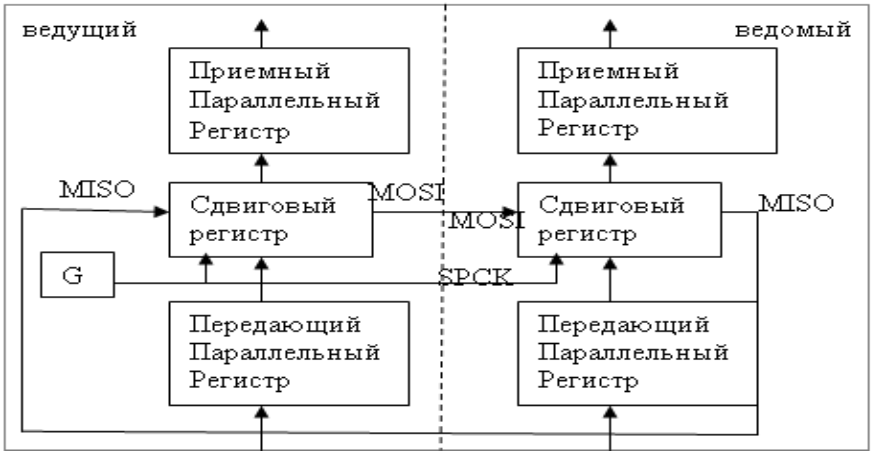


Рис.1.35. Взаимодействие через интерфейс SPI

Через SPI обеспечивается связь с такими компонентами встроенных систем как:

– датчики (температуры, давления, ADC, сенсорный экран);

– управляющие устройства (аудио кодеки, цифровые потенциометры, DAC);

– flash и EEPROM;

– MMC и SD карты.

SPI основан на взаимодействии двух сдвиговых регистров, соединенных в кольцо. Перед началом передачи данные передающих параллельных регистров переписываются в соответствующие сдвиговые регистры. По окончании передачи данные из сдвиговых регистров переписываются в парал-

лельные регистры. На рис. 1.36 приведена временная диаграмма работы интерфейса. Под действием тактовых импульсов генератора ведущего данные из сдвигового регистра ведущего перемещаются в сдвиговый регистр ведомого.

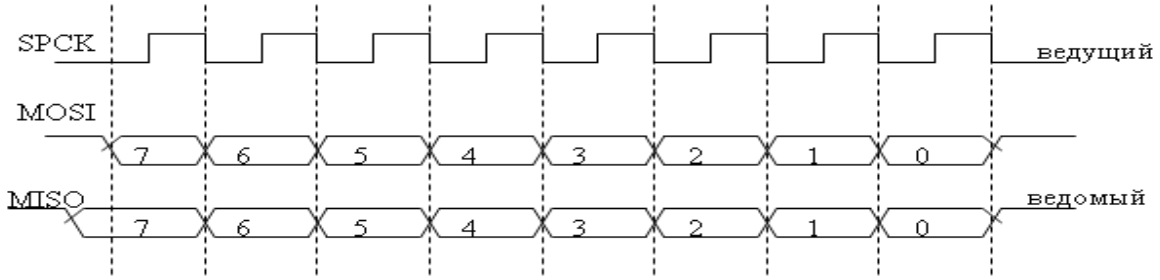


Рис.1.36. Временная диаграмма работы SPI

1.5.3.3. Интерфейс TWI

Также как и SPI TWI (Two-wire Interface – двухпроводный интерфейс) или I2C (Inter Integrated Circuit) – последовательный синхронный интерфейс предназначенный для связи CPU с компонентами MPS, расположенных в пределах печатной платы, по принципу ведущий-ведомый.

TWI находит применение в устройствах, предусматривающих простоту разработки и низкую себестоимость изготовления при относительно неплохой скорости работы. Через него обеспечивается связь с такими компонентами встроенных систем как:

– модули NVRAM;

– низкоскоростные ADC/DAC;

– регулировка контрастности, насыщенности и цветового баланса мониторов;

– регулировка звука в динамиках;

– управление светодиодами, в том числе в мобильных телефонах;

– чтение информации с датчиков мониторинга и диагностики оборудования, например, CPU или скорость вращения вентилятора охлаждения;

– чтение информации с часов реального времени (кварцевых генераторов);

– управление включением/выключением питания системных компонент;

– информационный обмен между микроконтроллерами;

Интерфейс TWI является байт-ориентированным со скоростью передачи до 2 Мбит/сек. На рис. 1.37 приведена схема взаимодействие через интерфейс I2C. Каждая передача кадра начинается со стартовой позиции S и заканчивается стоповой P, как показано на рис. 1.38. На рис.1.39 приведены временные диаграммы операций записи а) и чтения б). Первый байт кадра состоит из 7-битного адреса ведомого и бита R/W типа операции. Значение следующего бита (ACK) =0 выставляет ведомый, распознавший свой адрес. При записи ведущий в ответ выставляет байт данных (старшими разрядами вперед). Следующим битом (ACK)=0 ведомый подтверждает принятый байт. В случае операции чтения бит R/W=1 и вслед за

ACK ведомый выставляет на линию SDA байт данных, который следующим битом подтверждает ведущий.

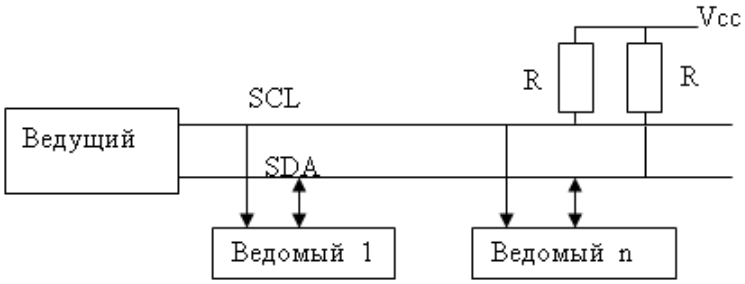


Рис. 1.37. Взаимодействие через интерфейс TWI

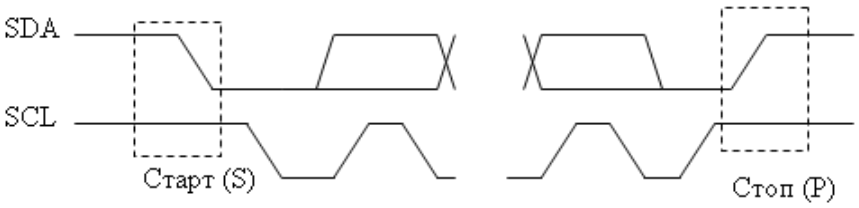


Рис. 1.38. Условия начала и конца кадра интерфейса TWI

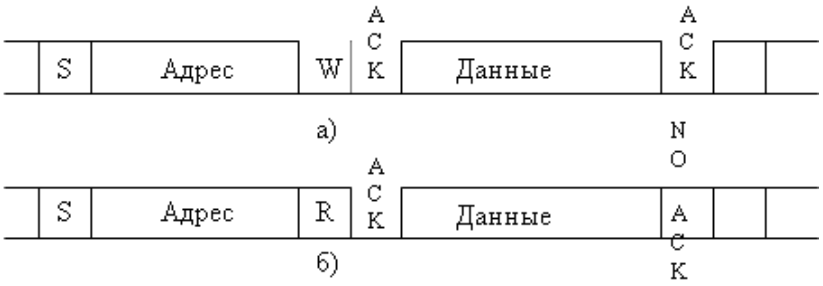


Рис. 1.39. TWI: а) операция записи, б) операция чтения

1.6. Разработка структурной схемы аппаратной платформы встроенной управляющей систем реального времени

Разработка структурной схемы аппаратной платформы встроенной управляющей систем реального времени представляет собой процесс выбора среди множества различных платформ той, которая наилучшим образом удовлетворяет заданным требованиям.

Выбор платформы для реализации прикладной программы является общей задачей. Решение делается на базе опыта прошлых проектов и знания экспертами предметной области. Выбор подвержен влиянию многих факторов [13]. Главное то, что аппаратная платформа должна позволить реализовать функциональные возможности, заданные функциональными требованиями.

Основным атрибутом (характеристикой) аппаратуры, определяющим функциональные возможности, являются «Уровень функциональности», а в случае систем реального времени и атрибут «Уровень производительности». «Уровень функциональности» определяет величину функциональных возможностей (число функций, размер структуры данных) которые могут

быть реализованы определенной платформой. «Уровень производительности» определяет, как быстро функции выполняются.

Кроме функциональных требований в процессе выбора участвуют и нефункциональные требования: от уровня надежности, обслуживаемости до потребляемой мощности и конкурентоспособности.

Сильная взаимосвязь между программным и аппаратным обеспечением встроенных систем требует, чтобы проектировщик имел дело с комбинацией обоих компонент. Выбор аппаратной платформы обычно происходит перед или совместно с разработкой программного обеспечения. Для того чтобы создать систему, удовлетворяющую заданным требованиям важно не только рассмотреть аппаратные свойства, но и также влияние на них определенных свойств программного обеспечения. Таким образом, процесс выбора платформы требует знаний свойств аппаратного обеспечения, включая влияющие свойства программного обеспечения и умения отображать эти свойства на требования проектируемой системы.

1.6.1. Связывание атрибутов системы со свойствами аппаратного обеспечения

Рассмотрим структуру «Дерево атрибутов аппаратного обеспечения» (функциональных и нефункциональных) [13], которые задают свойства системы (рис. 1.40). На этом этапе необходимо установить, как эти атрибуты влияют на аппаратное обеспечение. Эта задача очевидна для некоторых свойств, другие требуют знаний экспертов. Представим на абстрактном уровне для нескольких атрибутов эти зависимости в графическом (рис. 1.41 – 1.49).

На атрибут «Производительность» влияет архитектура процессорного узла (что можно сделать за один цикл) и тактовая частота (насколько быстр один цикл) (рис. 1.41). Архитектура процессорного узла подвержена влиянию разрядности (сколько бит может одновременно изменяться), доступностью команд (что обрабатывается одной командой), уровня параллелизма (как много действий могут выполняться параллельно) и интегрированной периферией (какие операции выполняют специализированные аппаратные модули).

«Уровень функциональности» (рис. 1.42) определяется доступной памятью (сколько программного кода может быть запомнено) и/или объемом кристалла (как много синтезировано функциональных узлов) завися от типа платформы. Уровень функциональности определяется также интегрированной периферией (допускается ли параллельная работа специализированных модулей), свойствами подсистемы ввода-вывода (могут ли быть подключены все требуемые внешние устройства) и свойствами абстракции (как управляется сложная система).

В отличие от предыдущих атрибутов «Конкурентоспособность» подвержена влиянию факторов наиболее далеких от реальной целевой платформы (рис. 1.43). Один их таких факторов – время выхода изделия на рынок. Время разработки устройства должно быть коротким, чтобы сэконо-

мить деньги и достигнуть успехов среди компьютерных компаний. Свойства аппаратуры влияющие на время выхода на рынок это основные затраты на функциональность определенного типа платформы (как долго реализовывать функциональность этой аппаратуры), экспертиза команды разработчиков (проводила ли команда эксперименты с подобным типом аппаратуры) и внешняя поддержка проекта. Поддержка в дальнейшем может быть разделена на программную и аппаратную части. Обе включают такие факторы как качество доступного инструментария, качество доступной "горячей линии", сетевых конференций и наличие примеров реализации.

Рис. 1.40. Дерево аппаратных свойств

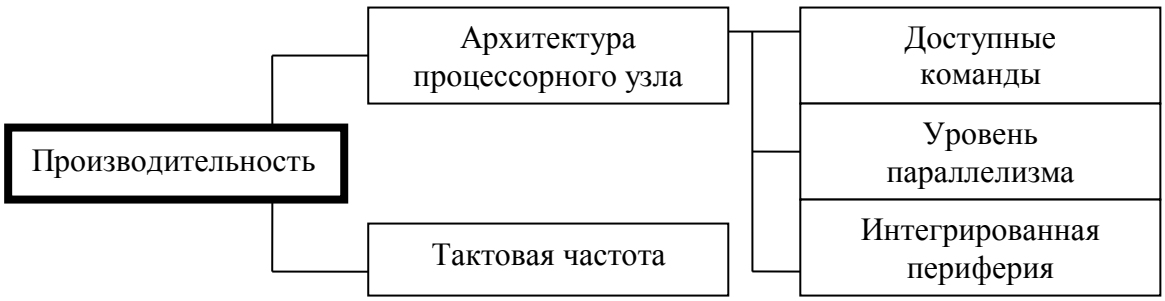


Рис. 1.41. Атрибут производительности

Другой важный фактор – результирующие расходы на проект платформы. Основные из них это собственно расходы на целевую платформу и расходы на физическую интеграцию устройства с системой (специальные требования к схеме размещения). Степень влияния на процесс выбора зависит главным образом от количества узлов, которые необходимы, а также от их наличия на рынке.

Затраты на платформу в потребительских товарах (например мобильные телефоны) вероятно оказывают большее влияние чем затраты для уникального производственного объекта. Другой фактор это затраты на среду разработки необходимую для успешной реализации требуемой функциональности. Среда разработки включает такие программные средства как компиляторы, симуляторы и определенные инструменты для отладки, аппаратные средства, такие как устройства для программирования и отладки. Наконец важным фактором конкурентоспособности платформы является ее полезность (availability). Если прекращается развитие проекта для платформы и прекращается производство соответствующего устройства и если система не может быть легко перенесена на другую платформу, то много усилий затраченных на проект будет потеряно.

Свойства аппаратуры по их влиянию на атрибуты можно разделить на прямые и косвенные. Например, на «Устойчивость» («Робастность») прямо влияют такие свойства аппаратуры как структура кристалла и свойства подсистемы ввода-вывода. Это отличается от атрибута «Конкурентоспособности». На этот атрибут оказывают влияние такие факторы как, например, готовность команды разработчиков к экспертизе, которая включает знание аппаратуры (прямое влияние), но также и способность разрабатывать программное обеспечение для нее (непрямое влияние).

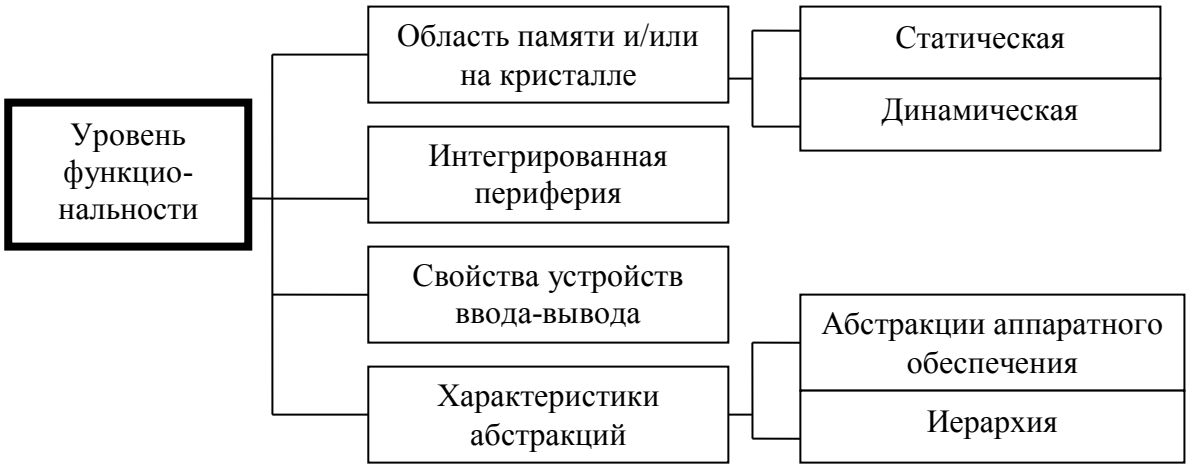


Рис. 1.42. Атрибут уровня функциональности

Рис. 1.43. Атрибут конкурентоспособности

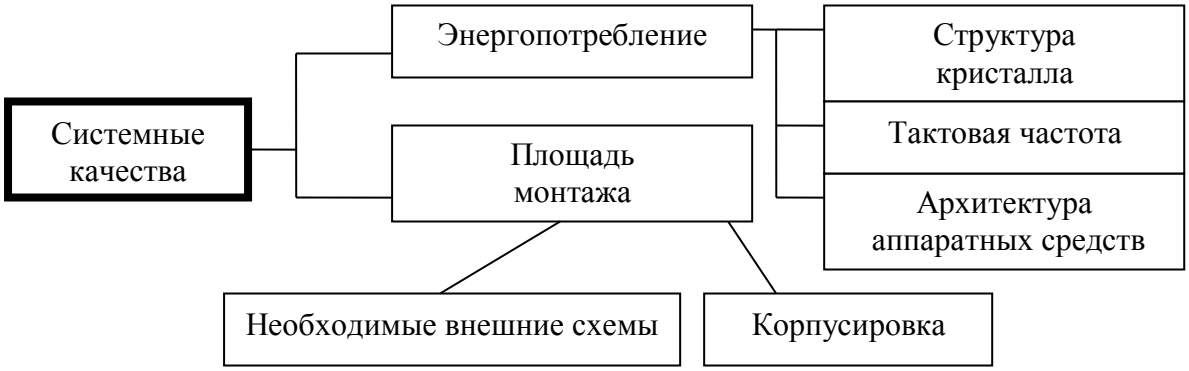


Рис. 1.44. Атрибут системных качеств

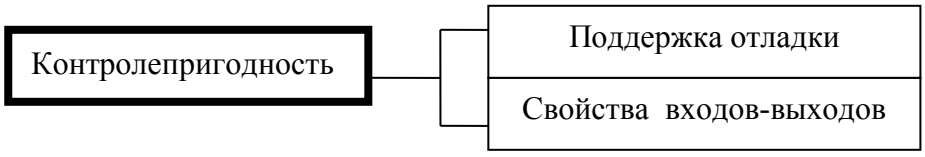


Рис. 1.45. Атрибут контролепригодности

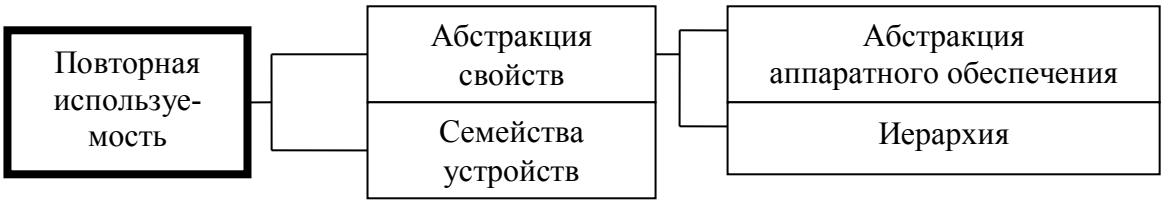


Рис. 1.46. Атрибут многократной используемости

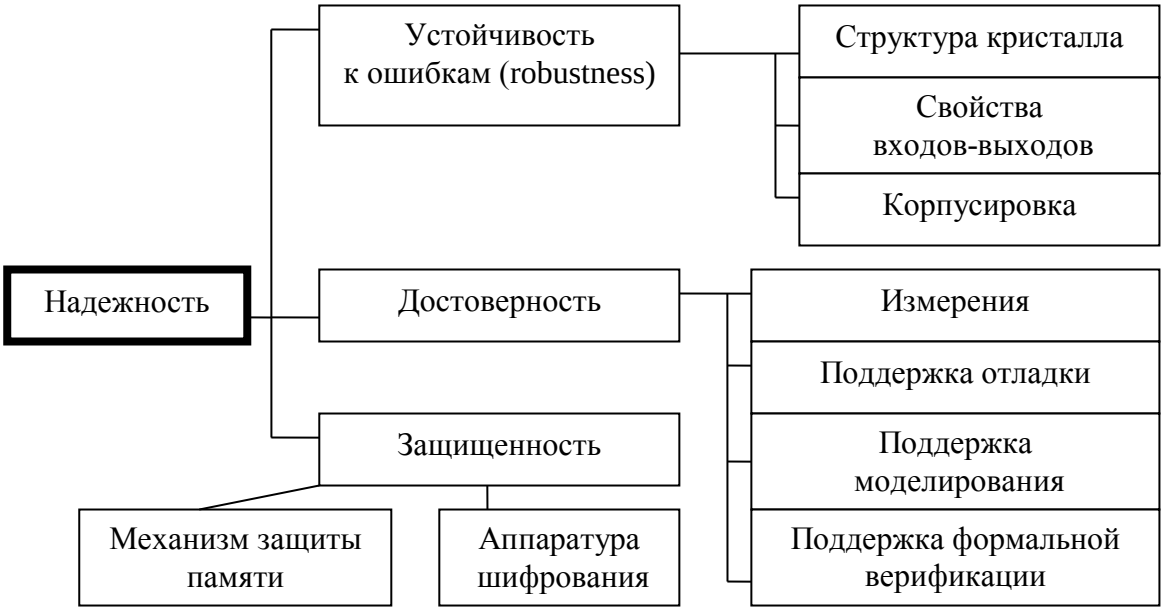


Рис. 1.47. Атрибуты надежности

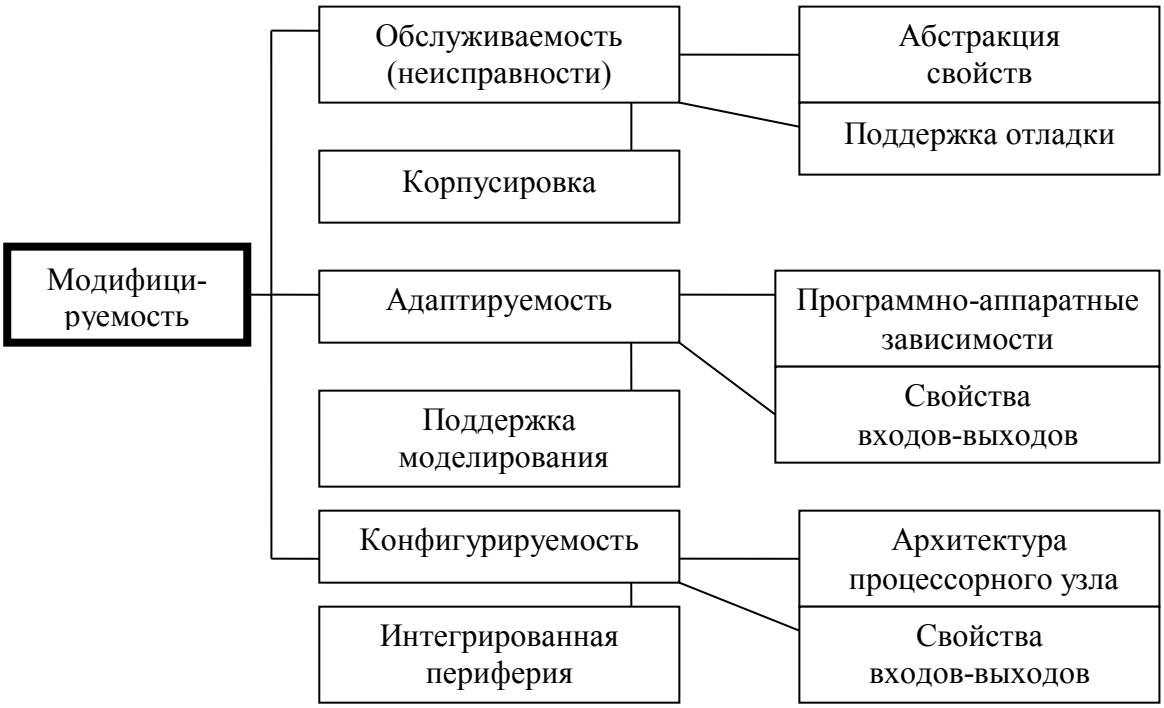


Рис. 1.48. Атрибуты модифицируемости

1.6.2. Подробное представление аппаратных свойств

Для успешного выбора оптимальной аппаратной платформы важно иметь детальную информацию об индивидуальных свойствах аппаратуры (например, система команд). Эта информация может быть задана в форме таблицы, позволяющей подчеркнуть схожесть и различие разных типов платформ. В качестве примера рассмотрим информацию о влиянии аппаратных свойств на атрибут «Адаптируемость» (табл. 1.1). В структуре таблицы информация о каждом аппаратном свойстве разбита на три подкате-

гории. Первая категория содержит общую информацию, применимую ко всем платформам. Вторая и третья категории содержат информацию специфичную для одной из двух групп встроенных систем на базе CPU и программируемых логических схемах (PLD). Это представление позволяет быстро получить информацию о схожести и различии этих двух групп. В дальнейшем при необходимости могут быть добавлены новые группы, начиная с описания соответствующего имени устройства (например, DSP: специальные алгоритмы для потока данных). Для ясности в таблицу 1.1 включена только общая информация. За другой информацией необходимо обратится к соответствующим публикациям (например, статье, разделу книги, технической документации).

Пусть известны требуемые качества и функциональность разрабатываемой системы. Представим графические структуры атрибутов аппаратуры разрабатываемой системы как в подразделе 1.6.1. Для каждого атрибута с помощью таких структур зададим влияющие аппаратные свойства. Более детальная информация об этих свойствах по отношению к различным платформам может быть задана в таблицах аналогичных таблице 1.1. На базе этой информации становится возможным выбор структуры разрабатываемой системы. Обычно необходимы компромиссы из-за того что различные платформы конкурируют друг с другом. Важно заметить, что рассмотренные структуры не обеспечивают получения оптимального решения, но формируют базу для систематического сравнения различных опций систем.

Таблица 1.1. Влиянии аппаратных свойств на атрибут «Адаптируемость»

Факторы «Адаптируемости»	CPU	PLD	Описание
HW/SW зависимости	X X	X X	Усилия необходимые для переноса программного обеспечения от одного устройства к другому. Перенос, если требования не могут быть удовлетворены данным устройством.
	X X X		Семейства микроконтроллеров, в которых легко выполняется миграция с одного MCU на другой в пределах семейства (микроконтроллерное семейство имеет один и тот же CPU, разную периферию/память/упаковку). Специальные требования к приложению для помощи процессу миграции. Наличие абстракций аппаратура/операционная система для уменьшения зависимости между аппаратурой и высшими уровнями программного обеспечения
		X X	Описание функциональности модуля на HDL (поведенческое описание) легко преобразуется в PLD. В этом случае преобразование включает только новые назначения выводов для устройства. Структурное описание функциональности модуля легко преобразуется только в платформу с подобной структурой (одинаковые базовые элементы).
Свойства ввода-	X	X	Легкость реализации новые требования к I/O.
	X		Отображение определенной функциональность I/O на спе-

вывода (I/O)	X		циальные выводы. Отображение некоторой функциональности I/O на различные выводы.
	X		Свободное отображение функциональности на I/O (только небольшое количество MCU предлагают это).
	X		Внешние шины (системная шина, SPI, I2C, …) легко интегрируют дополнительные периферию/память в систему.
		X	Одинаковые свойства/опции (вход, выход, подтяжка, …) для всех выводов I/O.
		X	Ввод сигналов тактовой частоты в устройство через специализированные выводы I/O.
		X	Программное отображение функциональности на выводы I/O обеспечивает максимальную гибкость.
		X	Влияние распределения на область кристалла, используемую для проекта.
Интегрированная периферия	X	X	Увеличение модифицируемости многоцелевой интегрированной периферии.
	X		Задание функциональности интегрированной периферии через регистры специального назначения.
	X		Увеличение модифицируемости с помощью интегрированной периферией с большим количеством опций (и сложностью).
		X	Задание почти всей (цифровой) функциональности через программное обеспечение.
		X	Доступность ранее спроектированных на HDL модулей (ядер) в качестве общих компонент.
		X	Использована периферия с жесткими связями (например, делитель частоты), если она подходит для решения задачи.
		X	Интегрированная периферия обычно не содержит аналого-цифрового преобразователя
Дополнительные влияющие свойства могут быть получены из атрибутов «Производительность» и «Уровень функциональности»			

1.6.3. Пример разработки платформы четырехканального генератора частоты.

Необходимо спроектировать встроенную систему, выполняющую функции четырехканального генератора, программируемого через интерфейс RS232 (кадр содержит 12 байт сообщения + CRC) с минимальным интервалом между сообщениями 100 ms. Описание задачи включает несколько функциональных требований, таких как уровни производительности и функциональности. Это, прежде всего, должно быть использовано в процессе выбора. Для полного же представления о системе необходимо задание и нефункциональных требований (табл. 1.2).

Проанализируем следующие платформы: MCU низкой стоимости, MCU средних размеров; MCU низкой стоимости + небольшое CPLD; средних размеров FPGA.

Реализация 1: MCU низкой стоимости. Прежде всего, необходимо установить удовлетворение функциональным требованиям. Предполагаем, что необходимый для RS232 контроллер интегрирован в MCU. Следова-

тельно, последовательный коммуникационный порт может быть реализован без особых усилий с минимальным вмешательством MCU (чтение входных 12 байт каждые 100 ms). В основном потребуется время для подсчета CRC. Генерация сигнала требуемой частоты более проблематична т.к. нет подходящей периферии на кристалле MCU. Генерацию сигнала можно было бы реализовать программно, но должна быть очень высокой тактовая частота MCU, обеспечивающая одновременную работу программ для RS232 и четырех каналов генератора. Для достижения необходимой производительности программы вероятнее всего необходимо было бы писать на ассемблере. Возможная тактовая частота для MCU низкой стоимости всего 16 MHz. Поэтому система не может бать реализована на этой платформе.

Таблица 1.2. Нефункциональны требования для четырехканального генератора частоты

Требования	Приоритет
Робастность	Высокий
Достоверность	Высокий
Безопасность	Низкий
Обслуживаемость	Низкий
Модифицируемость	Средний
Масштабируемость	Средний
Конфигурируемость	Низкий
Многократная используемость	Низкий
Контролепригодность	Высокий
Время выхода на рынок	Высокий
Стоимость	Высокий
Площадь монтажа	Средний
Энергопотребление	Низкий

Реализация 2: MCU средних размеров. Проблему рассмотренную выше может решить более быстрый MCU или MCU с интегрированной периферией, подходящей для генерации сигналов по четырем каналам. Рассмотрим последний вариант. MCU должен обрабатывать входящие сообщения из RS232, подсчитывать CRC и обновлять генерирующую частоту периферию. Так как большинство этих задач реального времени выполняется периферийными устройствами кристалла, такой MCU пригоден для требуемой функциональности.

«Робастность» системы зависит главным образом от аппаратуры MCU (рис. 1.46), а достоверность зависит от нескольких факторов. Такие средства как сторожевой таймер, или более продвинутый встроенный CPU, или мониторы памяти усиливают достоверность работы MCU.

В MCU средних размеров встраиваются средства поддержки отладки (JTAG, трассировщик, …) и доступны поддерживающие программные инструменты. Это позволяет наблюдать изнутри встроенную систему с целью верификации. Поддержка симуляции и формальной верификации дает возможность выполнить валидацию и верификацию функциональности целевой платформы перед выполнением на ней программ. Симуляция для MCU

средних размеров в реальном времени не поддерживается. Но так как функциональность реального времени реализуется периферией на кристалле, это не является препятствием. Формальная верификация для MCU средних размеров не поддерживается.

На атрибут «Модифицируемость» (рис. 1.48) часть аппаратуры оказывает влияние напрямую (насколько гибкой может быть встроенная периферия), а часть косвенно (программно-аппаратные зависимости, как легко происходит миграция на другие платформы). «Модифицируемость», вероятно, улучшается благодаря использованию языков высокого уровня. Дополнительная функциональность главным образом ограничивается факторами, влияющими на производительность и уровень функциональности (рис. 1.41, 1.42).

Влияние платформы на «Время выхода на рынок» непрямое (рис. 1.43). Время, затрачиваемое на разработку основного приложения очевидно небольшое, т.к. оно лишь инициализирует и координирует встроенную периферию. Время, затрачиваемое на разработку аппаратуры, зависит от количества и сложности внешних схем, необходимых для работы платформы. В любом случае влияющим фактором является качество технической документации и внешняя поддержка (подраздел 1.6.1). Затраты определяются стоимость целевой платформы и стоимость среды разработки. Планируемый объем производства определяется важностью этих двух издержек.
площадь монтажа определяется упаковкой целевой аппаратной платформы и количеством внешних схем, необходимых для работы (рис. 1.44).

Реализация 3: MCU низкой стоимости + небольшое CPLD. В этой комбинации MCU занимается управлением коммуникационным портом и подсчетом CRC, в то время как критичную по производительности часть по формированию требуемого частотного сигнала выполняет сложная программируемая логическая интегральная схема (CPLD). MCU и CPLD связаны 8-разрядной шиной адресов/данных и 3 линиями управления. Генератор частоты может быть реализован в CPLD на базе делителя тактовой частоты. Такая реализация может удовлетворить функциональные требования.

«Робастность» реализации зависит как от MCU и CPLD, так и магистрали системы. CPLD имеет легко повреждаемые входные цепи соответствующие минимальным показателям (свойства I/O). Следовательно, необходимы дополнительные схемы защиты для четырех выводов. Достоверность приложения теперь зависит от нескольких устройств, уменьшающих достоверность по сравнению с реализацией на одном кристалле. С другой стороны различная функциональность строго отделена друг от друга, что может уменьшить посторонние эффекты в программном обеспечении.

Усложняются симуляция и отладка, т.к. появляется два устройства, которые необходимо анализировать совместно. Начинает проявляться влияние свойств «Контролепригодности». Однако т.к. интерфейс MCU-CPLD является простым и хорошо определенным это дает возможность раздельной симулировать и тестировать части системы.

«Масштабируемость» по отношению к числу частотных сигналов главным образом зависит от размера CPLD. «Масштабируемость» по отношению к максимальной тактовой частоте зависит от максимальной тактовой частоты CPLD.

«Модифицируемость» по отношению к дополнительной функциональности генератора частоты зависит главным образом от размера CPLD (табл. 1.1). Новые требования к коммуникационному порту зависят от MCU (переход от RS232 к USB).

«Время выхода на рынок» зависит от серии экспертиз командой разработчиков и дополнительной поддержки. Реализация включает MCU, CPLD, взаимосвязь двух устройств и разработку объединительной платы. Затраты включают стоимость обеих платформ, обеих сред разработки, стоимость более сложной печатной платы и схем защиты. Площадь монтажа больше чем для варианта на одном чипе.

Реализация 4: средних размеров FPGA. Так как кажется, что программируемая логика подходит для реализации генератора частоты, представляет интерес интеграция всей функциональности в большой PLD (средних размеров FPGA). Для порта RS232 доступно IP-ядро. Должен быть интегрирован алгоритм CRC. Генератор частоты делается как для 3-й реализации.

«Робастность» зависит от FPGA и особенно от свойств I/O. Для повышения «Робастности» необходимы четыре схемы защиты. Из-за отсутствия системной шины достоверность лучше, чем в предыдущем варианте реализации.

Возможна симуляция реального времени для этого устройства. «Контролепригодность» улучшается из-за возможности маршрутизации промежуточных сигналов на выводы кристалла или внутренний тестовый модуль без влияния на основную функциональность.

«Масштабируемость» по отношению к числу каналов и «Модифицируемость» относительно дополнительной функциональности зависит главным образом от размера FPGA. Переход от RS232 к USB означает замену одного IP-ядро на другое (при условии его доступности и наличия места на кристалле).

Во всех случаях «Время выхода на рынок» зависит от серии экспертиз командой разработчиков. В противоположность предыдущему варианту дело имеют только с одним устройством, а также отсутствуют взаимосвязи. Однако реализовать RS232 и CRC быстрее на MCU чем PLD. Издержки на платформу 4 такие же или чуть больше чем для второго варианта, а стоимость среды разработки простирается от 0 до очень большой величины. Как и во втором варианте, площадь монтажа определяется упаковкой целевой платформы и числом внешних схем необходимых для работы.

Выбор платформы. Для выбора необходим обзор удовлетворения требованиям различных платформ, например, в форме таблицы. Эта таблица как минимум должна включать функциональные и нефункциональные требования, рассматриваемые как важнейшие. Для примера генератора все сведено в табл. 1.3. «++» означает очень хорошо, « - » представляет неудо-

влетворенность. «Уровню функциональности» удовлетворяют только варианты 2, 3. 4.

Таблица 1.3. Сравнение альтернатив

Требования	1	2	3	4
Производительность	-	+	+	+
Уровень функциональности	-	++	+	++
Робастость	+	+	+	+/-
Достоверность	+/-	++	+	++
Модифицируемость	-	+/-	+	+
Масштабируемость	-	+/-	+	++
Контролепригодность	+/-	+	+/-	++
Время выхода на рынок	+/-	++	+/-	+/-
Стоимость	++	+	+/-	+
Площадь монтажа	++	+	-	+

Платформы 2 и 4 кажутся лучшими. Платформа 4 кажется более подходящей в фокусе «Модифицируемости», тогда как платформа 2 кажется лучшей в фокусе скорости разработки (предполагается, что команда разработчиков имеет опыт работы и с микроконтроллерами и с программируемой логикой). Так как в нашем примере скорость разработки имеет более высокий рейтинг, чем «Модифицируемость», должна быть выбрана 2-ая платформа. На рис. 1.49. приведена структурная схема 2-й аппаратной платформы четырехканального генератора частоты.

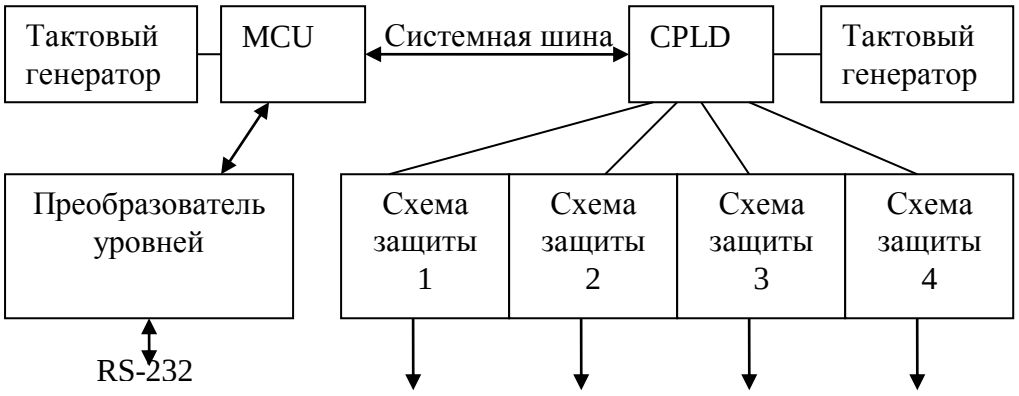


Рис. 1.49. Структурная схема аппаратной платформы четырехканального генератора частоты

.

Вопросы для самоконтроля

1. К архитектуре какого типа (Гарвардской или фон Неймана) относится процессор, изображенный на рис.4?
2. Какие элементы входят в понятие архитектуры процессора?
3. В чем разница между регистрами адреса, данных и общего назначения?

4. Какую цель преследовали разработчики архитектуры некоторых процессоров, когда ввели механизм регистровых окон?
5. С какими особенностями процессоров связано понятие когерентности памяти?
6. Как соотносятся прерывания и исключения?
7. Зачем в системе прерываний появилась вторичная система приоритетов?
8. Для каких целей были придуманы механизмы виртуальной памяти и сегментации памяти?
9. В чем особенность коммуникационных процессоров?
10. Какова роль пузырей в конвейере?
11. Какой вариант чередования внутренних банков SDRAM более производительный?
12. В чем отличия блокнотной и кэш памяти?
13. Чему равно значение параметра E для кэш-памяти с прямым отображением?
14. На что направлено обеспечение когерентности кэш-памяти?
15. В чем суть различия асинхронных магистралей в манерах Intel и Motorola?
16. Какую роль в аналого-цифровом преобразователе играет схеме выборки/хранения.
17. Как в UART выполняется синхронизация на уровне бит?
18. Возможно ли взаимодействие через интерфейс SPI более двух устройств, если да то как?

2. МОДЕЛЬНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРИКЛАДНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ВСТРОЕННЫХ УПРАВЛЯЮЩИХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Традиционная методология создания встроенных систем основана на модели фон Неймана – поочередное выполнение последовательности примитивных вычислений представленных на традиционных императивных языках программирования (зачастую это язык Си). Для критичных по безопасности встроенных систем проект должен быть детерминированным, безопасным, корректным и полным. Большинство же традиционных языков программирования привносят в проект операции являющиеся недетерминистскими, непредсказуемыми и небезопасными [14]. Эти операции классифицируют в три категории: небезопасные, недетерминированные и операции завершения.

Небезопасные операции это такие операции, которые могут привести к катастрофическим ошибкам или утечке безопасности в процессе выполнения программы. Вот некоторые из таких операций:

– Неструктурное программирование. Использование в программах оператора перехода ***goto***, может привести к их непредсказуемому выполнению.

– Манипуляция с указателями. Использование указателей часто требует понимания архитектуры процессора. Программы, которые используют указатели трудно понимать, а значит анализировать.

– Динамическое выделение памяти. В случае встроенных систем выделение памяти в процессе выполнения программы часто невозможно из-за малого ее количества. Программисты часто забывают освобождать выделенную память. Компиляторы и операционные системы часто ошибаются при полном восстановлении освобождаемой памяти. В результате возникают «загадочные» ошибки.

– Неявная и явная типизация данных. Типизация обеспечивает использование и присвоение данных только для совместимых типов. В случае неявной типизации компилятор автоматически выполняет приведение типов (например, float f = 4;). В случае явной типизация программист должен сам указывать приведение типов переменных (например, *float f = (float)4.0;*).

– Неограниченное выполнение. Рекурсии являются очень трудными для анализа и тестирования. Также неподходящее использование циклов, таких как for, while может создать проблемы, часто приводящие к непредсказуемому поведению в реальном времени.

Недетерминированные операции. Недетерминизм возникает когда одно и тоже множество входов системы ведет себя по-разному. Вот некоторые из таких операций:

– Одновременность. Большинство языков программирования поддерживают одновременность. Поведение двух одновременных программ с одинаковыми приоритетами не может быть предопределено заранее.

– Прерывания. Прерывания с одинаковыми приоритетами вносят недетерминизм во время работы программы.

– Прекращение операций. Большинство встроенных систем проектируются для бесконечного постоянного взаимодействия с окружающей средой. Завершение во встроенных системах рассматривается как плохое поведение. Вот некоторые из основных операций завершения:

– Выход (Exit). Выход из программы в некоторой точке может быть пагубным в случае критичных по безопасности систем.

– Исключения и ошибки при выполнении программы. Это может вызвать непредсказуемые ситуации или необычные результаты. Большинство такого типа ошибок трудно для обработки (например, исключение по выходу индекса массива за ограничение).

Для устранения неправильного поведения были разработаны синхронные языки. Эти языки используют ограниченные конструкции традиционных языков для исключения недетерминизма и небезопасных операций. Главным образом синхронные языки поддерживают только временное ограничение выполнения операций. Они работают в соответствии с концепцией глобальной длительности временного интервала, означающей, что все в системе могут выполнять свою требуемую задачу в этом интервале.

Модельно-ориентированное проектирование является другой методологией создания встроенных систем. Изначально оно использовалось для систем комбинированного управления (complex control), цифровой обработки сигналов и телекоммуникационных систем. Сейчас модельно-ориентированное проектирование также используется для управления движением, в промышленном оборудовании, в аэрокосмических и автомобильных приложениях. Ключевая идея подхода состоит в построении абстрактной визуальной модели системы и изучения на этом уровне ее свойств до получения программной или аппаратной реализации.

Парадигма модельно-ориентированного проектирования систем означает, что помимо языков той или иной предметный области используются точные и практические описания важных сущностей, в терминах объекта и процесса проектирования.

Программная или аппаратная реализация (код на том или ином языке программирования) генерируется автоматически только после детального изучения поведения модели (верификации модели).

Существуют различные среды разработки встроенных систем, основанные на методологии модельно-ориентированного проектирования.

Компания MathWorks [14] разработала интегрированную среду разработки на платформе **MATLAB/Simulink**. Это позволило объединить в непрерывный рабочий процесс разные этапы разработки системы, такие как формирование спецификаций и системных требований, имитационное моделирование, разработку системы, отладку и тестирование. Модельно-ориентированное проектирование помогает координировать работу раз-

личных групп разработчиков и позволяет выявлять ошибки на ранних стадиях, значительно сокращая время разработки и повышая эффективность проектирования.

Технология MathWorks содержит следующие компоненты.

Real Time Workshop автоматически генерирует кода из Simulink для интерактивной работы и отладки.

xPC Target обеспечивает проверку на прототипах в режиме реального времени и аппаратную проверку на основе персонального компьютера.

Real Time Workshop Embedded Coder генерирует высокоэффективный код, который может быть подогнан так, чтобы выглядеть как код, написанный вручную.

Target Support Packages облегчает применение автоматически созданного кода на целевых процессорах.

Пакеты Real Time Workshop и Real Time Workshop Embedded Coder делают возможным применение автоматически созданного кода на любом микропроцессоре, так как они генерируют стандартный Си (ANSI/ISOC).

Моделируемые спецификации обеспечивают отсутствие двусмысленности, тесное взаимодействие команды разработчиков, компонентное моделирование, определение входных и выходных параметров систем, моделирование на требуемом уровне детализации и автоматическое документирование. Проектирование совместно с моделированием позволяет выполнить моделирование от математических уравнений до эмпирических данных, компонентов различной физической природы (механики, электричества, СВЧ, событийно управляемых систем, пользовательских компонентов), использовать интегрированные средства разработки законов управления, IEEE стандарт с плавающей точкой и точное моделирование систем с фиксированной точкой.

Автоматическая генерация кода обеспечивает генерацию ANSI/ISO C кода из моделей Simulink для применения в конечном продукте и ускорения имитации (для микропроцессоров, DSP), а также быстрого создания прототипов и тестирования в аппаратном режиме. Тестирование и верификация осуществляются на основе технических требований, прослеживаются требования в коде, осуществляется непрерывное тестирование в процессе имитационного моделирования, создания прототипов, программ, на уровне микроконтроллера и аппаратном уровне.

Компания Agilent Technologies предлагает САПР **SystemVue** предназначенный для автоматизации проектирования на уровне ESL (electronic system-level, системный уровень электроники) [16]. SystemVue вводит такие новшества как архитектура системы и разработка алгоритмов для проектирование физического уровня беспроводных и аэрокосмических телекоммуникационных систем, обеспечивая их реализации в виде RF (радиочастотные узлы), DSP и FPGA/ASIC. SystemVue заменяет такие среды разработки как проектирование цифровых устройств общего назначения, проектирование аналоговых узлов и выполнения математических расчетов. SystemVue

(называют просто RF) сокращает разработку и время верификации физического уровня в двое. Ключевыми достоинствами SystemVue являются:

– Лучшая в классе средств разработки RF узлов для работы в основной полосе, позволяющая виртуализацию проектирования.

– Более высокая интеграция с тестированием ускоряет степень завершенности разработки и рационализирует последовательность модельно-ориентированного проектирования от Архитектуры до Верификации.

– Для групп разработчиков, объединенных в сеть, максимизирует повторное использование результатов проектирования и капитализирует совместную деятельность.

Компания Esterel Technologies [17] создала среду разработки **SCADE** для получения законченных решений разработчиками прикладного программного обеспечения **критических по безопасности** встроенных систем. Она состоит из следующих компонент.

Комплект программ **SCADE Suite** – это основанный на модели набор инструментов (tool-chain) для разработки прикладного программного обеспечения систем управления, интеграционную роль, в которой играет собственный (native) язык Scade и его формальная нотация. На рис. 2.1 приведена структура SCADE Suite. Это компоненты для создания проектов, моделирования и верификация проектов, генерации кода на Си и Ada и средства поддержки функциональной совместимости с инструментами разработки других производителей.

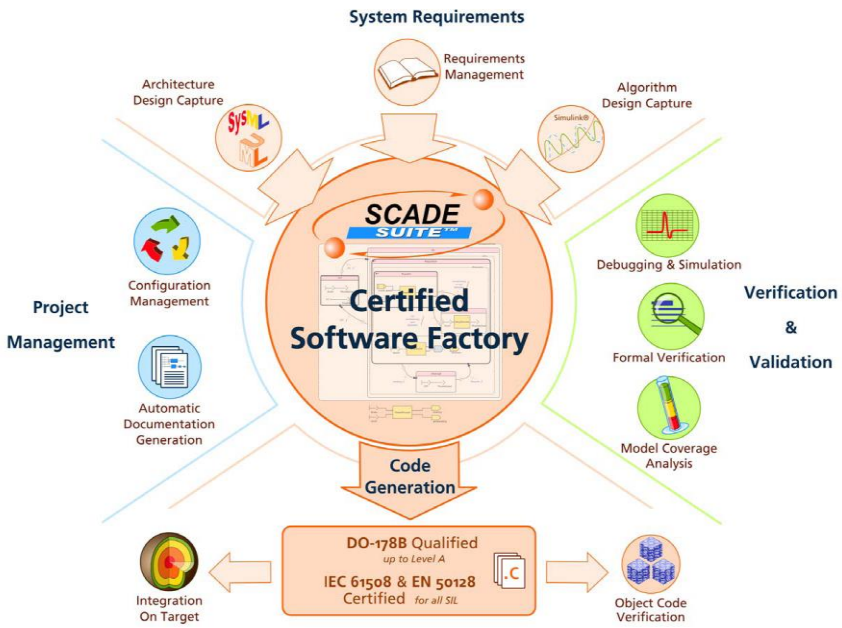


Рис. 2.1. Структура SCADE Suite®

Унифицированная Методология Разработки Моделей в системе SCADE позволяет объединять алгоритмические описания, сделанные с помощью потоков данных, с описаниями, сделанными с помощью конечных автоматов, на любом уровне иерархии проекта. SCADE обеспечивает проектировщика технологией однократного безитерационного ввода информа-

ции, которая позволяет выполнить реализацию требований без избыточности или неоднозначности и построить более точные программные модели. В библиотеке SCADE содержится более трехсот дискретных операторов.

Эффективное Системное Проектирование. Шлюз к системам управления требованиями SCADE Requirements Management (RM) Gateway позволяет создавать связи прослеживаемости (traceability links) между системными требованиями, моделями, созданными в SCADE Suite и в SCADE Display, структурным дизайном, тестовым планом и проектной документацией.

Система создания сертифицированного ПО SCADE предоставляет возможность ввода и обработки проекта системного уровня – спецификации требований, описания архитектуры на языке SysML/UML.

Шлюз SCADE SysML Gateway импортирует архитектуру, разработанную системными экспертами. Шлюз может работать с системами Telelogic Rhapsody и Artisan Studio™, а также легко адаптирован к другим средствам проектирования на языке SysML/UML.

Импорт из систем Simulink ™ и Stateflow выполняется в строгом соответствии с требованиями обеспечения безопасности, которые накладываются Унифицированной Методологией Разработки Моделей SCADE.

Верификация и Валидация на стадии проектирования. Симулятор SCADE Simulator исполняет встраиваемый код.

Верификатор проекта SCADE Suite Design Verifier предоставляет проектировщику возможность всестороннего исследования для поиска «краевых» ошибок, практически не детектируемых традиционными методами тестирования, и исправления их на самой ранней стадии проекта.

Тестовое Покрытие Моделей SCADE Model Test Coverage (MTC) оценивает полноту тестовой процедуры, разработанной на основе требований, и помогает достичь 100%-ого покрытия по MC/DC. MTC является средством верификации, квалифицированным по DO-178B

Масштабируемость до многопользовательских, территориально-распределенных проектов. Модульность SCADE-моделей обеспечивает безопасное и управляемое разделение труда между несколькими разработчиками или коллективами разработчиков.

Библиотеки позволяют совместное использование данных и функциональных блоков несколькими проектами, SCADE Suite обеспечивает согласованность в итоговых моделях.

SCADE Suite тесно интегрирован с системами управления требованиями, производимыми другими фирмами.

Генератор отчетов SCADE Reporter автоматически создает проектную документацию, обеспечивая ее постоянное обновление. Генерация отчетов может быть адаптирована к специальным требованиям путем настройки, управляемой из языка TCL. SCADE Reporter является средством разработки, квалифицированным по DO-178B.

Генерация эффективного, компактного и мобильного кода промышленного качества. Кодогенератор SCADE Code Generator производит код языка C стандарта ANSI, легко читаемый и прослеживаемый. Код незави-

сим от целевого процессора, не требует никаких специальных исполняемых библиотек или библиотек, специфичных для конкретного процессора. Код может исполняться на любом целевом микропроцессоре под управлением ОСРВ или без нее. В SCADE Suite кодом, можно задать автоматическое оформление для исполнения в среде операционных систем Integrity-178B™ (Green Hills Software), μC/OS-II (Micrium), PikeOS™ (Sysgo), VxWorks™ 653 (Wind River).

Кодогенерация, квалифицированная по DO-178B. Кодогенератор SCADE Suite Code Generator (KCG) является средством разработки, квалифицированным по DO-178B до Уровня А. Квалификационный комплект кодогенератора Qualification Kit предоставляет все артефакты (материалы), требуемые для средств разработки (DO-178B 8110.49).

Эта квалификация позволяет разработчику исключить низкоуровневое тестирование сгенерированного кода и, поэтому, выполнять обновления дизайна быстро и без потери безопасности кода. Сгенерированный с помощью SCADE код используется более чем в 30-ти программах по сертификации и уже квалифицирован более чем в 20-ти программах.

Всесторонняя Сертификация Объектного Кода. Комплект верификации компилятора SCADE Compiler Verification Kit (CVK) позволяет провести верификацию применяемого компилятора языка С на корректность компиляции кода, сгенерированного в SCADE KCG. Комплект верификации подтверждает корректность работы сгенерированного кода по утвержденной методике, описанной в руководстве CAST 12.

Средство **SCADE LifeCycle** расширяет функциональность SCADE в направлении администрирования жизненного цикла критических приложений. Он интегрируется со всеми продуктами SCADE для обеспечения эффективности, полноты и долговременного обслуживания приложений SCADE. Особенностями комплекта является наличие в его составе компонент «Управления требованиями» (Requirements Management), автоматической «Генерации документов» (Documentation Generation), «Планов сертификации» (Certification Plans для сертификационного процесса DO-178B).

Средство **SCADE System** это инструмент разработки открытой архитектуры критической системы. Средство SCADE System базируется на стандартах SysML (Systems Modeling Language) и Eclipse. Используя SCADE System вместе с SCADE Suite, SCADE Display и SCADE LifeCycle, системные инженеры и программисты могут работать в одной инфраструктуре (framework), чтобы исключить дублирования усилий и несоответствия между описанием структуры системы и поведенческого описания ПО.

Модуль **SCADE System Designer** – инструмент моделирования проектов систем на архитектурном уровне позволяет системным инженерам моделировать проекты системных компонент и структуры с использованием блок-диаграмм SysML. Модуль также позволяет экспортировать структурные компоненты из какой-нибудь модели направлять их группам разработчиков подсистем.

Средство **SCADE Display** это гибкая графическая среда проектирования и развития для дисплеев и HMI (Human-Machine Interface, интерфейс

«человек – машина») критических систем. Вместе с собственной поддержкой OpenGL (Open Graphics Library, графическое API) SC (Safety Critical, критическая безопасность) и стандартом ES (Embedded System), SCADE Display является новым поколением инструментов для разработки графики, макетирования остова, проектирования отображения, моделирования, верификации и валидации и квалифицированной генерации кода (KCG).

2.1. Модели вычислений

Встроенные системы относятся к **реагирующим системам** (reactive system) которые определяют как вычислительные системы, непрерывно взаимодействующие с физическим окружением и это окружение не способно логически синхронизироваться с системой (например, не может ждать). Этот класс систем был предложен для разграничения с **трансформационными системами** (transformation system), т.е. классическими программами, чьи входные данные полностью доступны перед началом исполнения и которые обеспечивают результат по завершению работы, а также – с **интерактивными системами** (interactive system), которые непрерывно взаимодействуют с окружением и обладают способностью к синхронизации (например, операционные системы).

Реагирующие системы в основном применяются в системах автоматического управления и мониторинга, цифровой обработки сигналов, телекоммуникационных системах, системах человеко-машинных интерфейсов, которые требуют очень малого времени отклика. Следующие особенности являются определяющими для таких систем.

– **Параллелизм** во взаимодействии системы с окружающей средой. Систему часто делают распределенной для повышения производительности, отказоустойчивости и функциональности. Кроме того может оказаться легче представить систему как композицию параллельных модулей, достигающих в кооперации требуемого поведения, нежели как централизованную реализацию.

– **Временные ограничения** относится к частотным характеристикам входных данных и времени отклика системы, По определению они диктуются окружающей средой и верифицируются как важней пункт корректности полученной в процессе проектирования системы.

– **Надежность** является важнейшей характеристикой критических систем, какими часто являются реагирующие системы. Поэтом при проектировании таких систем приоритет за инструментами разработки, поддерживающие формальные методы верификации.

Очевидны два простых пути реализации реагирующих систем:

– Управление (event-driven) по событиям означает вычисление выходов системы при каждом новом событии – изменение в окружающей среде.

– Дискретизация (sampling) означает вычисление выходов системы для каждого дискретного момента времени.

Характерны модели вычислений будут рассматриваться именно для реагирующих систем.

Модель вычислений (MoC – Model of Computation) описывает механизм, предназначенный для выполнения вычислений [2]. В общем случае необходимо представить систему, состоящую из компонент, взаимодействующих через коммуникационные протоколы. Компоненты и организация вычислений в них это в, частности, процедуры, процессы, функции, конечные автоматы. Коммуникационные протоколы описывают методы взаимодействия компонент, такие как асинхронная передача сообщений и рандеву.

Компоненты выполняют вычисления (отображают входную последовательность данных в выходную последовательность данных). Вычисления реализуются на языках программирования. Типичные вычисления содержат итерации. В каждом цикле итерации компоненты получают входные данные, обрабатывают их, генерируют выходные данные.

Связи между компонентами могут быть зафиксированы в виде графа, вершины которого и представляют компоненты. В таких графах к вычислениям относят процессы или задачи и соответственно называют графами задач или сетями процессов. Ребра графа представляют отношения между компонентами. Наиболее очевидным отношением между компонентами является причинная зависимость (вычисления могут быть выполнены только после завершения выполнения других вычислений). Она фиксируется графом зависимости. На рис. 2.2 приведен пример граф задач, где T1, T2, T3, T4, T5 – задачи (T1, T2, T3 являются независимыми). (1,8], например, обозначает, что 1 – время начала выполнения задачи, а 8 – предельное время исполнения. Специально помеченные узлы обозначают доступ к ресурсам (ввод/вывод).

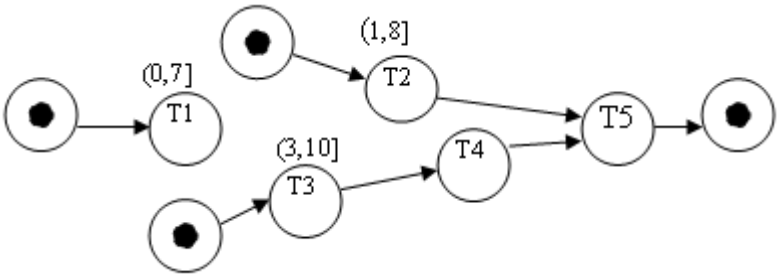


Рис. 2.2. Пример графа зависимости (граф задач)

Модели взаимодействия часто основываются на двух парадигмах: разделяемая память и передача сообщений.

В случае разделяемой памяти взаимодействия происходят через доступ всех компонент к одной и той области хранения данных. Доступ к разделяемой памяти должен находиться под защитой за исключением доступа только по чтению. Если выполняется запись, то на это время должен бать гарантирован исключительный доступ компонента к памяти. Сегмент кода программы выполняющего такой доступ называют *критической секцией*. Были предложены различные механизмы для гарантированного исключительного доступа, например, семафоры. Механизм разделяемой памяти

очень быстр, но трудно реализуем для микропроцессорных систем, если нет доступной общей физической памяти.

При передаче сообщений используются «случайные» области памяти. Этот способ легко реализуется, даже если нет доступной общей памяти. Однако передача сообщений происходит медленнее. Рассмотрим различные техники передачи сообщений.

Асинхронная передача сообщений (или взаимодействие без блокировки). Компоненты взаимодействуют между собой посылкой сообщений через канал, оборудованный буфером для сообщений. Отправителю не надо ждать готовности получателя для приема. Это аналогично отправке обычного или электронного письма. Потенциальная проблема кроется в переполнении буфера сообщений. Существует несколько схем взаимодействия, такие как взаимодействующие конечные автоматы (в языке SDL) и модель потока данных.

Синхронная передача сообщений (или взаимодействие с блокировкой). Компоненты взаимодействуют с помощью немедленных неделимых действий называемых рандеву. Компонент, достигший точки взаимодействия первым, должен дождаться достижения партнером своей точки взаимодействия. Это аналогично физической встрече людей или телефонным вызовам. Нет риска переполнения, но производительность ниже. Такое взаимодействие включено, например, в язык ADA.

Расширенное рандеву, удаленный вызов. В этом случае отправителю разрешается продолжить работу только после получения подтверждения от приемника. Приемник не должен посылать подтверждение немедленно после приема сообщения. Он может сделать ряд предварительных проверок перед отправкой подтверждения.

Организация вычислений в компонентах базируется на нескольких моделях.

Модель фон Неймана основана на последовательном выполнении цепочек примитивных вычислений.

В *Модели дискретных событий* события переносят строго упорядоченные штампы времени, показывающие моменты возникновения событий. Симулятор дискретных событий обычно содержит глобальную очередь событий, упорядоченную во времени. Записи очереди обрабатываются в соответствии с этим порядком. Недостаток модели в том, что она полагается на глобальное понятие очереди событий, делающей трудным отображение семантики модели на параллельную реализацию. Такую модель поддерживают языки проектирования аппаратуры VHDL, SystemC и Verilog.

Модель конечного автомата или машина с конечным числом состояний (Finite State Machine – FSM) основана на понятии конечного множества состояний, множествах входов и выходов, а также множества переходов между состояниями. Несколько таких автоматов могут взаимодействовать между собой, образуя *сообщающиеся автоматы* или *Сообщающиеся Машины с Конечным числом Состояний* (CFSM) .

Дифференциальные уравнения уравнения способны моделировать аналоговые электрические схемы и физические системы. Поэтому они находят применение для моделирования кибер-физических систем.

Модель потока данных используют, в случае если вычисления предпочтительнее представить не как синхронизированные события, а в виде зависимости реакции одних компонент от реакции других компонент.

Комбинированные модели. Реальные языки программирования обычно объединяют определенные модели взаимодействия с организацией вычислений в компонентах. Например, SDL объединяет машины с конечным числом состояний с асинхронной передачей сообщений. Языки ADA и CSP объединяют модель фон Неймана с синхронной передачей сообщений.

Каждая из МоС успешна в той или иной прикладной области. Выбор «лучшей» МоС для конкретного приложения может быть очень трудным. Преодолению дилеммы помогает применение смешанных МоС.

2.1.1. Потоковая модель вычислений

В парадигме потокового моделирования вычисления представляются в виде ориентированного графа. Вершине (блоку или компоненту) сопоставлен вычислительный модуль или иерархически вложенный подграф (подсеть). Ребру графа (соединению) сопоставлен FIFO-буфер, через который перемещаются выборки данных (отсчеты) от блока-источника к блоку-приемнику. Ребро может иметь сопоставленную с ним неотрицательную целую задержку, величина которой определяет начальное количество отсчетов, которые буферируются в соединении перед стартом моделирования.

Граф потока данных функционирует на основе выполнения вычислений управляемых данными (data-driven): блок выполняется (возбуждается – fire) только когда он имеет необходимое количество отсчетов на всех его входных соединениях. При возбуждении блока он берет определенное количество отсчетов из своих входных соединений, выполняет вычисления и производит определенное количество отсчетов для своих выходных соединений.

Синхронная потоковая модель (Synchronous Data Flow – SDF) является наиболее зрелой из потоковых моделей [16]. В SDF число отсчетов, вводимое в соединение *e* при возбуждении блока-источника *src(e)*, ограничивается целой положительной константой, которая должна быть известна перед началом моделирования. Эту константу называют производительностью соединения *e – prd(e).* Подобным образом число отсчетов извлекаемых из соединения *e* блоком-приемником *snk(e)* называют нормой потребления соединения *e – cns(e)*. Говорят, что соединение представляет многоскоростное соединение, если *prd(e) /= cns(e).*

Перед моделированием выполняется планирование работы (составляется расписание – schedule) SDF-графа, т.е. определяется последовательность и количество возбуждений блоков в цикле моделирования. SDF-граф называют согласованным (не содержит взаимоблокировок - deadlock) если

есть положительное целое решение уравнений баланса для каждого соединения SDF-графа:

*prd(e) * x[src(e)] = cns(e) * x[snk(e)]*, где *x[v]* означает количество возбуждений узла *v* в полном расписании.

Уравнение баланса легко интерпретируется следующим образом: для каждого соединения общее число поступающих отсчетов равно общему числу потребляемых отсчетов в одной итерации полного расписания возбуждения блоков. Такой баланс производства и потребления позволяет постоянно выполнять вычисления в границах выделенной памяти.

В SDF-графе на рис.2.3 [2] узел А выполняет умножение, а узел В сложение.

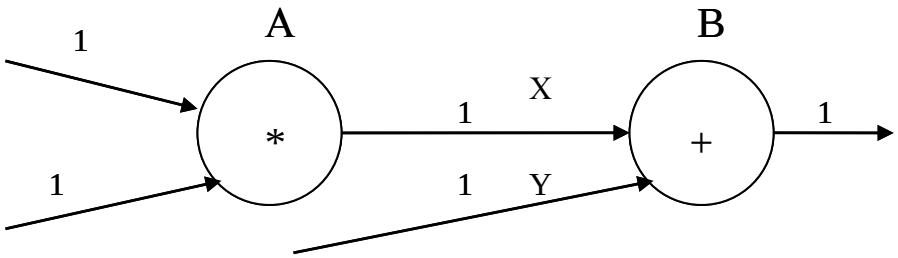


Рис 2.3. SDF-граф

Предполагается, что входы графа образуют бесконечную последовательность отсчетов. Соединения помечены количеством возбуждений узлов за цикл моделирования (1). Пусть X = (0,1,2,3,4,...) и Y = (2,4,6,8,10,...), тогда X+Y=(2,5,8,11,14,...).

SDF-граф может содержать задержки (D) на соединениях как на рис. 2.4.

Правильное расписание возбуждений для узлов SDF-графов из приведенных примеров – это повторяющаяся последовательность (A,B). Последовательность (A,A,B) когда А выполняется в два раза чаще В, является неправильной, т.к. это потребует аккумуляции бесконечного числа отсчетов в FIFO-буфере между узлами А и В.

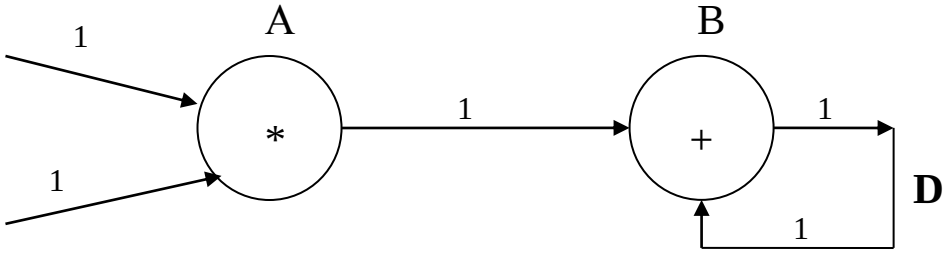


Рис. 2.4. SDF-граф с задержкой

Как база для языков высокого уровня потоковая модель обладает рядом преимуществ.

1. Потоковая модель является функциональной, а, следовательно, математически чистой и без сложных побочных эффектов. Это делает ее хорошо приспособленной для формальной верификации и безопасной транс-

формации, т.к. функциональные связи через поток данных могут рассматриваться как инварианты времени.

2. Потоковая модель является параллельной моделью, в которой некоторые ограничения порядка следования и синхронизации вытекают из зависимости данных. Эта особенность дает возможность получить естественную параллельную реализацию.

2.1.2. Автомат с конечным числом состояний

Начальное представление проектируемой системы на более детальном уровне требует более точной модели, основанной на представлении поведения с помощью состояний. Граф переходов, представляющий автомат с конечным числом состояния, является классическим способом представления состояний. Будем рассматривать детерминированные FSM, в которых в каждый момент активным является только одно состояние. Узлы графа сопоставлены состояниям. Ребра графа сопоставлены переходам между состояниями. Метки ребер представляют события. Предположим, что некоторое состояние FSM активно, происходит событие, которое соответствует одному из исходящих ребер. FSM изменит свое текущее активное состояние на состояние указанное этим ребром. Если FSM полностью тактируема, то ее называют синхронной. FSM может также генерировать выход.

FSM недостаточны для задания встроенных систем, тат как не моделируют время и не поддерживают совместного поведения нескольких FSM. Поэтому были предложены следующие усовершенствования классических автоматов .

Расширенный конечный автомат (EFSM) – автомат, пополненный переменными, которые могут быть прочитаны и записаны как часть перехода между состояниями. Введение переменных решает проблему быстрого увеличения числа состояний классического автомата для моделирования реальных объектов.

2.1.2.1. Временной автомат

Для моделирования времени в EFSM ввели действительные переменные, которые моделируют логические часы системы. Переменные времени инициализируются значением 0 при старте системы, а затем синхронно увеличиваются с одной и той же скоростью. Автомат с такими переменными называют *временным автоматам* (timed automata – TA). Переменные времени являются элементами охранных условий ребер. Переходы будут выполняться, когда значения часов удовлетворят охранным условиям ребер. Часы могут быть сброшены в 0 во время перехода [18]. На рис. 2.5 приведен пример модели «Автоответчик» в виде временного автомата.

Автоответчик обычно находится в начальном состоянии «Старт». При поступлении вызова часы X устанавливаются в 0 и выполняется переход в состояние «Ожидание». Если вызываемый абонент поднимает трубку,

то происходит разговор до опускания трубки. В противном случае выполняется
переход в состояние «Проигрывание текста», если время достигнет значения 4. По завершению проигрывания текста выполняется переход в состояние «Бип-сигнал». Часы Y обеспечивают однократную генерацию бип-сигнала. После бип-сигнала часы X сбрасываются снова в 0 и автоответчик готов к записи сообщения. Когда время достигает величины 8 или вызывающий абонент замолчал, генерируется следующий однократный бип-сигнала. После второго бип-сигнала выполняется переход в состояние «Отключено».

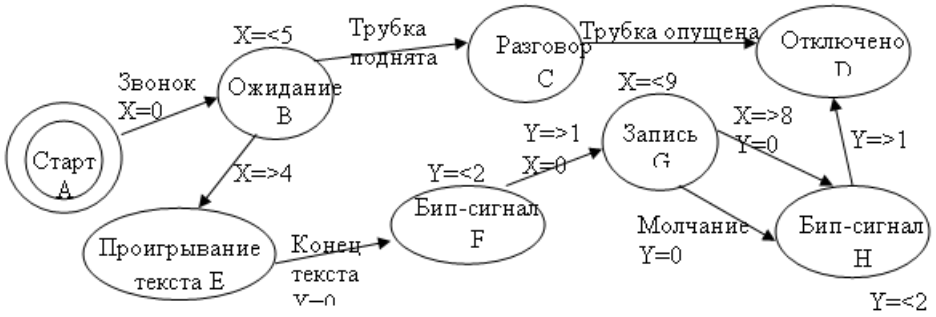


Рис. 2.5. Модель обслуживания входящих вызовов Автоответчиком в виде временного автомата

В этом примере переходы выполняются или под воздействием входов (таких как «Трубка поднята») или так называемых временных ограничений (clock constraints). Временные ограничения описывают переходы, которые могут, но не обязаны быть. Для того чтобы обеспечить реальный переход по этим условиям дополнительно определяются инварианты местоположения (location invariants). Инварианты местоположения X <= 5, X <= 9 и Y <= 2 используются в примере для того чтобы переходы выполнялись не позже чем через время при котором условия инвариант становятся истинными.

Формально временной автомат определяется следующим образом. Пусть C есть множество действительных не отрицательных переменных, представляющих часы. Пусть Σ есть конечный алфавит возможных входов.
Определение. Временное ограничение есть коньюктивная формула элементарных ограничивающих условий в форме:
$x \circ n$ или $(x-y) \circ n$ для $x, y \in C$, $\circ \in \{\leq,<,=,>,\geq\}$ и $n \in N$.

Пусть B(C) есть множество временных ограничений.
Определение. Временной автомат есть кортеж (S, s0,E, I), где
S – конечное множество состояний.
S0 – начальное состояние.

$E \in S \times B(C) \times \Sigma \times 2^C \times S$ – множество ребер. B(C) представляет конъюнктивное условие, которое должно иметь место и Σ – входы необходимые для перехода. 2^C выражает множество временных переменных, которые сбрасываются во время перехода. $I: S \rightarrow B(C)$ – множество инвариант для каждого состояния. B(C) представляет инварианту, которая должна

иметь место для определенного состояния. Эта инварианта описывается коньюктивной формулой.

Временной автомат хоть и расширяет классический автомат, однако в своей стандартной форме не выражает таких свойств как иерархия и одновременность работы. Это учитывают в различных языках проектирования, ориентированных на те или иные модели вычислений.

2.1.2.2. PLC-автомат

Целью автоматизации в промышленности является управление и оптимизация производственных процессов для обеспечения выпуска высококачественной продукции с минимальными материальными, стоимостными и энергетическими затратами. Промышленные системы автоматизации основываются на интеллектуальных датчиках (сенсорах), исполнительных механизмах (актюаторах) и другом промышленном оборудовании таком как роботизированные и электронно-механические (мехатронные) компоненты. Открытые и стандартизованные коммуникационные сети используются для передачи данных, конфигурации и управления различными компонентами автоматизации.

Стандартная архитектура состоит из программируемых логических контроллеров (PLC– Programmable Logic Controllers) или распределенных систем (DCS – Distributed Control Systems), полевых систем и ПК в качестве человеко-машинного интерфейса. Полевые системы собирают сигналы от сенсоров и актюаторов производственного процесса с помощью полевых интерфейсов, передают их распределенным или централизованным управляющим устройствам таким как PLC. Стандарт IEC 61131-3 поддерживает ряд форм записи программ для реализации на PLC. Они включают схемы подключения, список команд, функциональные схемы, а также графические и текстовые формы, называемые схемой последовательной функции (SFC – sequential function chart) и текстовой структурой. Сегодня разработка программного обеспечения для автоматизации производственных процессов ведется с использования этого стандарта и различных инструментов, которые обеспечивают производители PLC.

Проблема в том, что различные производители PLC используют свой собственный вариант стандарта с различными синтаксическими, семантическими и инструментальными наборами. Так же подходы, основанные на стандарте, не могут удовлетворить требования для распределенных приложений и приложений с жестким реальным временем. Вклад в решение этой проблемы вносит модель вычислений под названием *PLC-автомат* [24]. Эта модель обладает следующими важными свойствами:

• Реализуемость. PLC-автомат может быть автоматически скомпилирован в программу реального времени (исходный код), которая выполнима на PLC и других аппаратных платформах.

• Семантика. В качестве операционной семантики PLC-автомата может выступать временной автомат, описывающий, как ведет себя аппаратура PLC во время выполнения скомпилированного кода.

• Верифицируемость. Использование семантики временного автомата дает возможность автоматической верификации свойств реального времени, т.е. может быть установлено, что PLC-автомат удовлетворяет данным требованиям реального времени. Предполагая, что компилятор корректен, такое доказательство подразумевает, что исходный код, полученный из PLC-автомата, удовлетворяет требованиям.

Типичная область применения PLC это системы управления производственными линиями и транспортом. Аппаратура PLC разрабатывается как устойчивая к нагреванию, охлаждению, пыли и вибрации. Приложение реального времени работает в операционной среде реального времени (ОСРВ), установленной в PLC. С целью безопасности ОСРВ не может быть нарушена приложениями, что гарантирует минимальную функциональность в случае программных сбоев. Для заданного приложения ОСРВ выполняет следующий цикл из трех фаз:

• Опрос (Polling). Входные интерфейсы читают результаты и копируют их в зарезервированные области памяти PLC. Эта фаза выполняется автономно ОСРВ и не может подвергаться манипулированию со стороны приложения.

• Обработка (Computing). ОСРВ выполняет приложение один раз. Приложение допускает выполнение произвольных вычислений и имеет доступ к памяти с входными и выходными данными. Приложение может использовать таймера для работы со временем. Таймеры реализуются операционной системой, а приложение может их устанавливать, читать и сбрасывать.

• Обновление (Updating). В последней фазе цикла на выходные интерфейсах копируются значения из зарезервированной области памяти.

Рассмотрим модель PLC-автомата на простом примере из системы управления транспортом. На железной дороге используются контактные сенсоры, определяющие прохождение состава по некоторому ее участку (например, в районе переезда). Эти сенсоры могут генерировать «дребезг контактов» – переключаться несколько раз из одного положения в другое при прохождении состава. Это опасно, так как система управления может неправильно интерпретировать такой сигнал как прохождение нескольких составов. Предположим, что сенсор гарантирует возникновение дребезга спустя 4 сек после первого срабатывания, а минимальная дистанция по времени между составами 6 сек. Идея фильтрации дребезга состоит в том чтобы игнорировать переключения в течении короткого промежутка времени, скажем в 5 сек. Это требование свидетельствует, что вся прикладная программа является приложением реального времени. Возможное решение может быть представлено в виде PLC-автомата (рис. 2.6).

Автомат состоит из двух состояний q1 и q2 с выходными значениями N (нет состава) и T (состав). Автомат реагирует на входной сигнал со значениями no_tr (нет состава) и tr (состав) соответствующими переходами. Состояние q2 должно быть стабильным не менее 5 сек. Для реализации такого автомата на PLC предположим что в течении каждого цикла система реагирует только раз для чтения входной величины. В случае задержки в

состоянии q2 PLC игнорирует входную величину, пока не истечет время задержки. Отсюда реализация на PLC будет вести себя следующим образом:

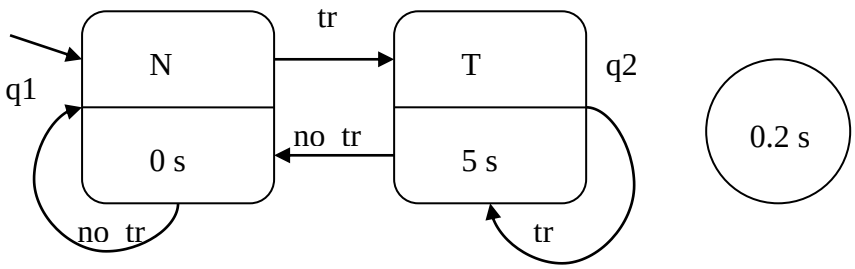


Рис. 2.6.

• Начальное состояние – q1.
• Если находимся в состоянии q1 и прочитано входное значение tr, то переходим в состояние q2, иначе остаемся в состоянии q1.
• В состоянии q2 система будет оставаться в течении 5 сек независимо от результата опроса входного значения. После этого будем оставаться в этом состоянии, пока опрос возвращает значение tr, иначе перейдем в q1.

Предположим, что сенсор из предыдущего примера генерирует сигнал ошибки Error, свидетельствующий о его неисправности. Необходимо модифицировать автомат на рис. 2.6 так, чтобы он реагировал на этот сигнал немедленно, генерируя выходной сигнал X (исключение). Проблема в после перехода в состоянии q2, когда будут игнорироваться входы в течении 5 сек. Для решения этой проблемы определим множество входов на которые распространяется задержка (рис. 2.7).

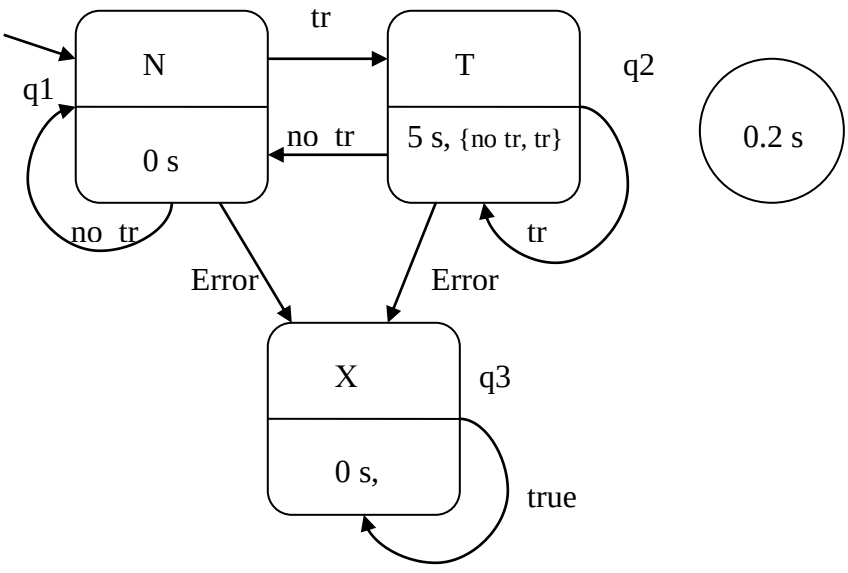


Рис. 2.7.

Идея этого автомата в том, что Error-переход из q2 в q3 будет выполняться без проверки истекло ли время задержки в 5сек. При переходе же из q2 в q1 необходимо проверять, истекло ли время задержки. Итак, формальное определение PLC-автомата выглядит следующим образом.

PLC-автомат это структура A = (Q,Σ, δ, q0, ε, St, Se,Ω, ω) где:

• Q – конечное множество состояний, q – типовой элемент;
• Σ – непустое конечное входов , σ – типовой элемент;
• δ – функция перехода типа $Q \times \Sigma \rightarrow Q$;
• $q0 \in Q$ – начальное состояние;
• $\varepsilon > 0$ – верхняя граница времени выполнения цикла;
• St – функция типа $Q \rightarrow R_{\geq 0}$ назначает задержку каждому состоянию;
• Se – функция типа $Q \rightarrow 2\Sigma$ назначает множество задержанных входов для каждого состояния;
• Ω – непустое конечное множество выходов;
• ω – функция типа $Q \rightarrow \Omega$ – назначает выход каждому состоянию.

Это определение представляет только синтаксис PLC-автомата, т.е. его структурные компоненты. Семантика была неформально представлена при рассмотрении примеров. В них автомат был представлен графически. Каждое состояние q изображено как прямоугольник с двумя компонентами. Верхний компонент отражает выходную величину ω(q). нижний компонент представляет время задержки St(q) и множество Se(q), которое отражает задержанные выходы. Переход δ(q, σ) = q_ представляется стрелкой из q в q_ отмеченный входной величиной σ. Ограничение времени цикла показано в отдельной окружности.

PLC-автомат транслируется в программу на языке ST (Structured Text). ST – Pascal-подобный императивный язык, определенный в стандарте IEC 61131-3 для PLC. Операционная система PLC реализует циклическое поведение с помощью бесконечного цикла WHILE TRUE, чередующего три фазы Polling, Computing и Updating ((* …. *) – комментарий):

```
WHILE TRUE DO
      (* Polling-фаза *)
      (* Computing-фаза *)
      (* Updating-фаза *)
END
```

Транслятор PLC-автомата должен только выполнить преобразование состояний для Computing-фазы. Для этого используются оператор IF. Ниже представлена ST-программа фильтра сенсора из рассмотренного выше примера.

```
1: PROGRAM  PLC_PRG_FILTER
2: VAR
3:      state : INT := 0; (*коды состояний 0:=N, 1:=T, 2:=X  *)
4:      tmr : TP;
5: ENDVAR
6:
7: IF state=0 THEN
8:       %output:=N;
```

```
9:      IF %input = tr THEN
10:             state:=1;
11:             %output:=T;
12:     ELSIF %input = Error THEN
13:             state:=2;
14:             %output:=X;
15:     ENDIF
16: ELSIF state=1 THEN
17:     tmr(IN:=TRUE,PT:=t#5.0s);
18:     IF (%input = no_tr AND NOT tmr.Q) THEN
19:             state:=0;
20:             %output:=N;
21:             tmr(IN:=FALSE,PT:=t#0.0s);
22:     ELSIF %input = Error THEN
23:             state:=2;
24:             %output:=X;
25:             tmr(IN:=FALSE,PT:=t#0.0s);
26:     ENDIF
27: ENDIF
```

Строки 1-5 программы представляют заголовок программы, в котором декларируются переменные state типа INT(целое) и timer стандартного типа TP. Таймер tmr может быть установлен в значение d единиц времени. Установленный таймер сохраняет на выходе tmr.Q величину true в течении d единиц времени. После этого выход tmr.Q переключается на false и остается таким до следующей операции установки (строка 17).

Строка 7: оператор IF начинает распознавание текущего состояния PLC-автомата.

Строка 8: для начального состояния (0) устанавливаем выходную величину (в N). Для всех других состояний будем обновлять значение выхода, когда меняем состояние. Символ % используется для адреса зарезервированной памяти. Псевдокоды с именами %input и %output используются для представления программного интерфейса к сенсорам и актюаторам соответственно. Эти имена объявляются неявно и могут быть использованы как ординарные переменные.

Строка 9: проверяем опрошенную входную величину в состоянии (с выходом) N. Проверяются только те переменные (tr и Error), которые могут вызвать переключение в другое состояние.

Строки 10, 11: PLC в состоянии N и после опроса входной величины tr. Следуя функции перехода PLC-автомата, присваивается значение 1 переменной state и устанавливается выходное значение T.

Строки 16 - 26: относятся к состоянию с выходом T.

Строка 17: это состояние имеет задержку в 5 сек и запускаем таймер tmr. Это делается с помощью вызова соответствующей процедуры tmr(IN:=TRUE,PT:=t#5.0s) с двумя параметрами. Параметр IN представляет стартовый флаг, а параметр PT желаемую длительность. Если только опе-

рационная система увидит восходящий фронт стартового флага, она запустит таймер с величиной указанной параметром длительности. Иначе вызов не будет иметь эффекта. Другими словами таймер установится, только если стартовый флаг имеет значение true, а предшествующий вызов имел значение стартового флага false. Изначально стартовый флаг подразумевается равный false.

Строка 18: проверяем, может ли быть запущен переход в N. Это произойдет в случае, если опрошенная входная величина равна no tr и таймер tmr истек. Последнее состояние представляет выход таймера tmr.Q.

Нет оператора реализующего верхнее временное ограничение (в примере 0.2 сек) цикла PLC. Это ограничение на самом деле представляет предположение что аппаратура PLC достаточно быстра, чтобы оставаться в заданных пределах в каждом цикле. Есть два пути разгрузки этого предположения:

1. Определить наихудший случай времени выполнения (Worst Case Execution Time – WCET) ST-программы на аппаратуре PLC и проверить, не превышает ли оно установленной границы. Для кода из примера это сделать реально.

2. Можно вначале сообщить операционной системе PLC о верхней границе цикла. Если она обнаружит нарушение этой границы во время выполнения кода, то перейдет в состояние ошибки и просигнализирует определенным значением выхода.

В процессе трансляции любого PLC-автомата необходим один таймер. Это связано с тем, что в каждом состоянии PLC-автомата нужно контролировать только один таймер. После выхода из состояния таймер становится ненужным для него и может быть повторно использован в другом состоянии.

PLC-автомат может быть реализован и на других аппаратных платформах отличных от PLC. Необходимо реализовать бесконечный цикл содержащий ввод значений сенсоров, обновление состояния в соответствии с величиной таймера и вывод значений актюаторов.

2.2. Язык Lustre/Scade

Среды разработки встроенных систем, основанные на методологии модельно-ориентированного проектирования, ориентируясь на ту или иную предметную область, используют различные MoC. Эти MoC реализуются на своих языках программирования. В SCADE Suite используется собственный язык Scade, в основе которого лежит синхронный декларативный язык Lustre для описания синхронных потоковых моделей. В отличии от императивных языков, которые отвечают на вопрос «как надо сделать», декларативные языки отвечают на вопрос «что надо сделать».

2.2.1. Потоки и тактирование

В Lustre/Scade под *потоком* (flow) понимают пару, образованную переменной ***x*** в общем случае с бесконечной последовательностью значений определенного типа и *тактированием* (clock) $\boldsymbol{b_x}$, представляющим последовательность моментов времени: *x* определяется в момент ***I***, если $\boldsymbol{b_x(i)=true}$. Входные переменные определены в каждый момент времени. Поэтому их тактирование называют *базовым тактированием* (basic clock). Поток с базовым тактированием принимает n-ое значение в n-ом цикле вычисления.

Таблица. 2.1. иллюстрирует масштабы тактирования, определяемые потоком C с базовым тактированием и потоком C‘ с тактированием определяемым потоком C, где обозначено T – true, F – false.

Таблице. 2.1. Булевы потоки и тактирование

Базовый масштаб времени	1	2	3	4	5	6	7	8
Поток C	T	F	T	T	F	T	F	T
Масштаб времени потока C	1		2	3		4		5
Поток C’	F	F	T	F	F	T	F	T
Масштаб времени потока C’			1			2		3

Концепция тактирования не ограничивается физическим временем. Базовое тактирование надо рассматривать, как задание минимальной частицы времени программы, в течение которой не воспринимаются поступающие внешние события и которая соответствует времени отклика системы. При необходимости реального времени оно может быть реализовано как входной булев поток, например, поток, истинные значения которого индицирует событие – «прошла миллисекунда» (или «прошла микросекунда» …). Это реализует многозначную концепцию времени в Lustre.

В каждом такте выполняется чтение очередных значений из входных потоков, вычисление реакции и формирование значений для выходных потоков. Все это происходит с длительностью равной 0.

2.2.2. Переменные, равенства, выражения, утверждения

Переменные объявляться вместе со своими типами, а переменные, которые не соответствуют входам, должны быть заданы только одним определением в форме равенства в математическом смысле: равенство $X=E$ определяет переменную X как идентичную выражению E. Обе имеют одинаковые значения и тактирование. Из такого определения вытекает принцип замещения: X может быть замещен в программе на E и наоборот. Как следствие равенства могут быть записаны в любом порядке и дополнительные переменные могут быть созданы для именования подвыражениq без изменения смысла программы.

Lustre содержит небольшое количество типов: ***bool, integer, real*** и один тип конструктора – *tuple* (кортеж). Однако могут быть импортированы составные типы из базовых языков и обработаны как абстрактные типы.

У констант те же типы. Соответствующие им потоки имеют неизменные последовательности величин и базовое тактирование.

На базовых типах определены:

– арифметические операторы +, -. *, /, ***div, mod***;

41. булевы операторы ***and, or, not***;

– операторы отношения =, <, >, <=, >=;

– условный оператор ***if then else***;

– из базовых языков могут быть импортированы функции.

Эти операторы называют *операторами над данными (data operators).* Они действуют только с операндами, разделяющими общее тактирование, и выполняют это поэлементно над последовательными значениями из соответствующих потоков. Например, если выражения *X* и *Y* с базовым тактированием и соответствующими последовательностями величин (*x1, x2, ..., xn, ...*) (*y1, y2, ..., yn, ...*), то выражение ***if*** *X>0* ***then*** *Y+1* ***else*** *0* представляет поток с базовым тактированием, n-ая величина которого равна *yn+1* если *xn >0* иначе 0.

Кроме этих операторов имеются четыре темпоральных (временных) оператора:

• ***pre*** (previous – предыдущий) действует как память: если (*e1, e2, ..., en, ...*) есть последовательность величин выражения *E* , то *pre€* имеет то же тактирование, что и *E* и последовательность величин (*nil, e1, e2, ..., en, ...*), где *nil* обозначает неопределенное значение, соответствующее неинициализированной памяти.

• –> (followed by – за которым следует): если выражения *E* и *F* с одинаковым тактированием и соответствующими последовательностями (*e1, e2, ..., en, ...*), (*f1, f2, ..., fn, ...*), то выражение *E –> F* это выражение с тем же тактированием что и для *E* и *F* и последовательностью значений (*e1, f2, ..., fn, ...*). Другими словами выражение *E –> F* всегда равно *F* за исключением первого момента времени.

В таблице 2.2 приведены результаты работы следующих двух темпоральных операторов:

• ***when*** выполняет «выборку» (sampling) для более медленного тактирования (аналогично децимации в цифровой обработке сигналов). Если *E* выражение и *B* булево выражение, то *E when B* такое выражение, тактирование которого определяются *B* и чья последовательность сохраняет только те значения *E*, индексы которых соответствуют значениям *true* последовательности *B*. Другими словами это последовательность значений *E*, когда *B* принимает *true*.

• ***current*** «интерполирует» выражение, тактирование которое идет медленнее, чем необходимо для нового выражения. Пусть *E* выражение, тактирование которого не являются базовыми и *B* булево выражение с базовыми тактированием. Тогда выражение *current E* имеет такое же тактирование С, как и *B* и его значение в любой момент времени тактирования С есть значение *E* в последний момент времени, когда *B* было *true*.

Таблица 2.2. Операторы выборки и интерполяции.

B	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*
X	x1	x2	x3	x4	x5	x6	x7	x8
Y=X when B		x2		x4			x7	x8
Z=current Y	*nil*	x2	x2	x4	x4	x4	x7	x8

Допустимы только ациклические равенства. *X=E* является ациклическим, если *X* не присутствует в *E* за исключением только как операнд *pre*:

X = X ***and pre****(X)* – недопустимое циклическое равенство;
X = Y ***and pre****(X)* – допустимое ациклическое равенство.

Утверждение (assertion) – это обобщающее выражение, состоящие из булевого выражения, которое всегда истинно. Его основное назначение состоит в индикации компилятору о возможности выполнения оптимизации кода, когда окружающая среда программы обладает известными свойствами. Например, если известно, что два входных события, представленные булевыми переменными *x* и *y*, никогда не происходят одновременно, можно написать ***assert not*** *(x and y)*. Подобным образом утверждение ***assert*** ***(true->not*** *(x* ***and pre****(x)))* говорит о том, что событие *x* никогда не может возникнуть дважды подряд. Кроме оптимизации кода утверждения играют важную роль в верификации программ.

2.2.3. Структура программы

В Lustre выражения могут быть представлены графически в виде схемы из операторов. Например, выражение *n=0->****pre****(n)+1;* представляющее счетчик с базовым тактированием. Компания Esterel разработала SCADE – графический формализм для Lustre. На рис. 2.8 приведена схема этого счетчика в SCADE. Подобная схема может быть введена как новый многократно используемый оператор, который называют ***node*** (узел).

Декларация узла содержит спецификацию интерфейса – входные и выходные параметры с их типами и возможно с их тактированием. Необязательно могут быть объявлены переменные. В теле узла выражения и утверждения выполняют преобразования значений входов в выходы и внутренние переменные. Например, следующий узел определяет счетчик общего назначения, имеющий входы, начальное значение и значение после события сброса, а также значение шага приращения:

node *COUNTER(val_init, val_incr:* ***int****; reset:* ***bool****)* ***return*** *(n:* ***int****)*
let
n = val_int –> ***if*** *reset* ***then*** *val_init* ***else pre****(n)+val_incr;*
tel.

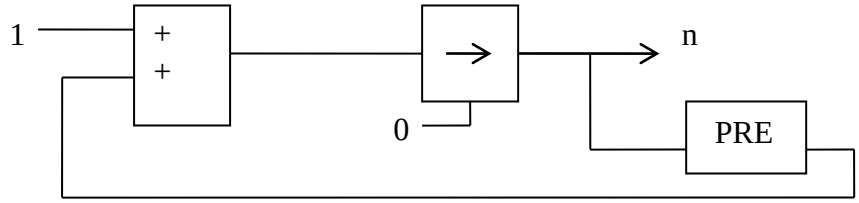


Рис. 2.8. Схема счетчика в SCADE

Такой узел может быть функционально введен в другие выражения, например следующие счетчики

even=COUNTER(0, 2, false);
modulo5= COUNTER (0,1, ***pre****(modulo5)=4);*

определяют последовательность четных чисел и циклическую последовательность чисел по модулю 5 с базовым тактированием. Аналогично, если *gamma* является ускорением, выраженным в м/сек^2 и его тактовая частота *opersecond*, то можно написать

speed=COUNTER(0, gamma, ***false****);*
position= COUNTER(0, speed, ***false****);*

Согласно принципу замещения это эквивалентно

position= COUNTER(0, COUNTER(0, gamma, ***false****),* ***false****);*

Узел может иметь несколько выходов. В этом случае выход представляется как кортеж, например

node *D_INTEGRATOR(gamma:* ***int****)* ***returns*** *(speed, position:* ***int****)*
let
speed=COUNTER(0, gamma, false);
position= COUNTER(0, speed, false);
tel.

Кортеж вводится как *(v, x) = D_INTEGRATOR(g);*

Что касается тактирования, то с точки зрения потока данных базовое тактирование узла определяется входами. Например, выражение *COUNTER((0, 1,* ***false****)* ***when*** *B)* выполняется, когда *B* равно ***true***. Здесь оператор *when* применяется к кортежу *(0, 1,* ***false****)*. В таблице 2.3 приведены результаты моделирования выражений *COUNTER(0, 1, false), COUNTER((0, 1,* ***false****) when B)* и *COUNTER(0, 1,* ***false****)* ***when*** *B* для иллюстрации разницы применения оператора ***when*** к входам и выходам.

Таблица 2.3. Узлы и тактирование

B	*true*	*false*	*true*	*false*	*true*

(0, 1, false) when B	*(0, 1, false)*		*(0, 1, false)*		*(0, 1, false)*
COUNTER((0, 1, false) when B)	0		1		2
COUNTER(0, 1, false)	0	1	2	3	4
COUNTER(0, 1, false) when B	0		2		4

Узел допускает входные параметры с различным тактированием. Базовое тактирование является самым быстрым для узла и все остальное тактирование должно быть задекларировано в списке входов. В узле
node *N(millisecond:**bool**; (x: **int**; y: **bool**)* ***when*** *millisecond)* ***return*** *...* вход *millisecond* имеет базовое тактированием, тактирование же входов *x* и *y* определяется значениями *millisecond.*

Выходы узла могут иметь тактирование отличное от базового тактирование, которое должно быть видимым за пределами узла. Безусловно, это тактирование медленнее базового.

Все выражения узла выполняются одновременно, поэтом порядок их записи не имеет значения.

Недопустимы неинициализированные выражения.

Тактирование должно быть согласованным.

Допустимы только статические рекурсии, т.е. полностью развернутые.

Недопустимы рекурсивные вызовы узла.

2.2.4. Конечный автомат в Scade

Тело состояния автомата образуется следующими упорядоченными элементами [19]:

• Возможно пустой список строгих (strong) переходов вводимых ключевым словом ***unless*** (пока не). При строгом переходе сначала проверяется условие перехода в выбранном состоянии.

• Возможно пустая область определения.

• Возможно пустой список нестрогих (weak) переходов вводимых ключевым словом ***until*** (до тех пор). При нестрогом переходе сначала выполняются действия активного состояния, а затем проверяется условия перехода. Поэтому в условиях перехода могут принимать участие локальные переменные, определенные в данном состоянии.

Переходы также содержат ключевые слова ***restart*** или ***resume*** (возобновление). В случае строгого перехода тело и нестрогая часть целевого состояния сбрасываются во время этого же такта, тогда как строгая часть будет сброшена в следующем такте. В случае нестрогого перехода все части целевого состояния сбрасываются в текущем такте. Сброс состояния означает, что оно рассматривается как снова находящееся в состоянии инициализации значений переменных: операторы ***fby*** или -> выполняют левый аргумент. Однако оператор ***pre*** (***last***) по-прежнему вычисляет последнее значение переменной. В случае ***resume*** значения переменных целевого состояния сохраняются.

Переходы могут сопровождать действия. Действие (action) – список сигналов возникающих в процессе перехода. Каждый сигнал сопровождает ключевое слово ***emit.***

Сигналы (signals) это локальные переменные, которые вводятся в рассмотрение ключевым словом ***sig***. Идентификаторы сигналов не могут напрямую использоваться в частях равенства. Они могут появляться только в форме с кавычками. Не требуется объявления типа сигнала. Сигналом Анипулируют как с булевым потоком: значение *true* означает наличие сигнала, а *false* – его отсутствие.

Пример автомата со строгим переходом и его входные и выходные потоки:

```
node expect  (X: bool)
returns (O: bool)
let
automaton SM_expect
initial state S1
unless if X restart S2;  - строгий переход со сбросом
  let
    O = false;
  tel
state S2
   let
     O = true;
   tel
returns ..;
tel
```

X	false	false	true	false	false	true	...
expect X	false	false	true	true	true	true	...

Пример автомата с нестрогим переходом и его входные и выходные потоки:

```
node expect  (X: bool)
returns (O: bool)
let
automaton SM_expect
initial state S1
  let
    O2 = false;
  tel
until if X do restart S2; - нестрогий переход со сбросом
state S2
  let
    O = true;
```

tel
returns ..;
tel

X	*false*	*false*	*true*	*false*	*false*	*true*	...
expect X	*false*	*false*	*false*	*true*	*true*	*true*	...

С автоматом связано определение двух глобальных потоков: потока выбранного состояния (selected state) и потока активного состояния (active state). Выбранное состояние идентифицирует состояние, в котором проверяется условие строгого перехода. Активное состояние это такое состояние, в котором проверяются тело и условия нестрогого перехода. Активное состояние совпадает с выбранным состоянием, если только имеет место строгий переход в другое состояние. Следующее выбранное состояние совпадает с текущим активным состоянием, если только имеет место нестрогий переход в другое состояние.

Поведения автомата подчиняется следующим правилам.

1. В каждом такте выполнение стартует с выбранного состояния.
2. В каждом такте существует только одно активное состояние.
3. В каждом такте запускается только один переход.

После того как становится известным выбранное состояние поведение автомата складывается из следующей последовательности шагов:

• Вычисляются условия всех строгих переходов (если они имеют место быть) из выбранного состояния. Первое по тексту истинное условие запускает переход, ведущий к идентификатору активного состояния. В случае если ни один строгий переход не запускается или они отсутствуют вообще, выбранное состояние становится активным.

• Вычисляются операции тела активного состояния.

• Если в текущем такте не запускается строгий переход, вычисляются условия нестрогих переходов. Первое по тексту истинное условие запускает переход, ведущий к идентификатору следующего выбранного состояния. В случае если ни один нестрогий переход не запускается или они отсутствуют вообще, следующим выбранным состоянием становится текущее активное состояние.

Благодаря третьему правилу поведения автомата чередование нестрогих и строгих переходов не приводит к выполнению в одном такте операций двух состояний. Рассмотрим случай, когда после нестрого перехода из S1 в S2 выполняется строгий переход из S2 как на рис. 2.9.

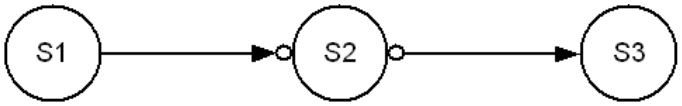


Рис. 2.9. Пример чередования нестрогого и строго переходов

Если нестрогий переход запускается из активного состояния S1, то в следующем такте будет выбрано состояние S2. Затем в этом следующем

такте, если строгое условие истинно, S2 не выполняется, а становится активным S3.

Рассмотрим случай, когда после строго перехода из S1 в S2 выполняется нестрогий переход из S2 как на рис. 2.10.

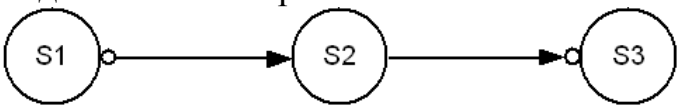


Рис. 2.10. Пример чередования строгого и нестрого переходов

Если строгий переход запускается из выбранного состояния S1, то S2 стает активным состоянием. Но так как существует запущенный строгий переход, нестрогий переход из S2 не вычисляется, а состояние S2 будет выбрано в следующем такте.

Следующий пример дает представление о полном синтаксисе автомата.

```
Node StateMachine_1(bI1: bool)
returns (bO1: bool default = true; iO2: int default = 0; bO3: bool default =
false)
let
  automaton SM1
  initial state ST1
  unless if bI1 resume ST2;   – строгий    условный переход с возобновлением
  sig                         – объявление сигнала
    sig1;
  var                         – объявление переменной
    iV1: int;
  let                         – начало действий в состоянии
    iV1 = 10;
    emit 'sig1;               – возбуждение сигнала
    bO1 = 'sig1;
    iO2 = iV1 -> pre iO2 + 2;
  tel                         – окончание действий в состоянии
  state ST2
  sig
    sig1;
  var
    bV1: bool;
  until if true do let emit 'sig1; – нестрогий    безусловный переход с
                                     возбуждением сигнала,
    bV1 = 'sig1;              – преобразование его в переменную,
    bO3 = bV1;                – подключение ее к выходному потоку и
  tel
  restart ST1;                –  рестартом
  returns ..;
tel
```

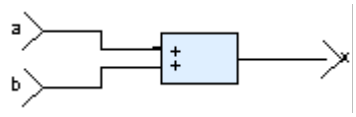
Следующий пример иллюстрирует декларацию ***default.*** Вместо использования предыдущего значения переменой в состоянии, где нет объявления этой переменной, используется значение из ***default.***

```
Node StateMachine_073(iI1: int; bI2: bool)
returns (iO1: int default = 10; iO2: int default = 5 * iO1)
let
  automaton SM1
  initial state ST1
    unless if bI2 resume ST2;
    let
      iO1 = iI1;
    tel
    until if true restart ST2;
  state ST2
   let
    iO2 = 0 -> pre iO1 + iI1;
   tel
  until if true restart ST1;
  returns ..;
tel
```

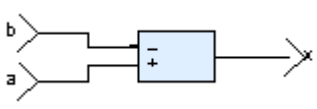
Переменные/такты	1	2	3	4	5
iI1	1	1	1	2	0
bI2	false	false	true	false	true
Активное состояние	ST1	ST2	ST2	ST1	ST2
iO1	1	10	10	2	10
iO2	5	0	11	10	10

2.2.5. Соотношение текстового и графического представления операторов в SCADE [19]

2.2.5.1 Арифметические операторы

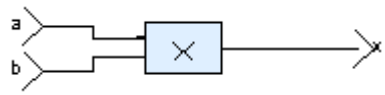


x = a + b ;

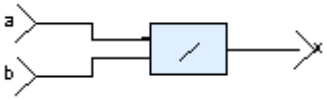


x = a – b ;

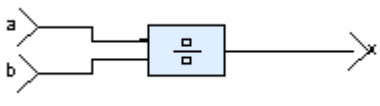
c = - (a) ;



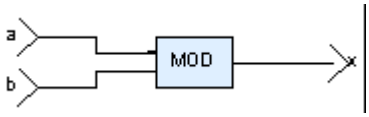
*x = a **b ;*



x = a / b ;

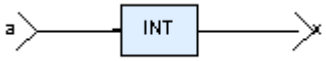


*x = a **div** b ;*



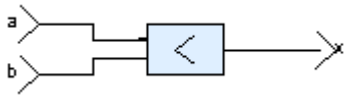
x = a ***mod*** *b ;*

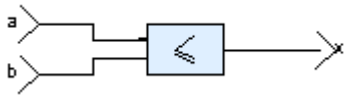
2.2.5.2 Операторы преобразования типов

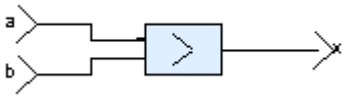


x = ***int****(a) ;* *x =* ***real****(a) ;*

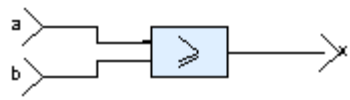
2.2.5.3 Операторы отношения

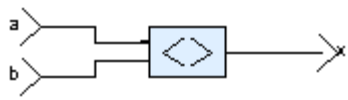


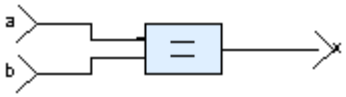




x = a < b ; *x = a <= b ;* *x = a > b ;*

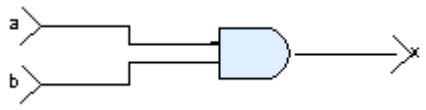


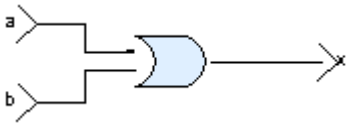


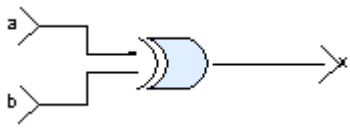


x = *a > =b ;* *x = a <> b ;* *x = a = b ;*

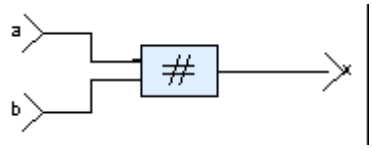
2.2.5.4. Логические операторы



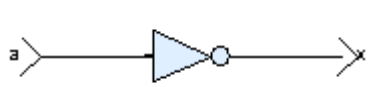




x = a ***and*** *b ;* *x = a* ***or*** *b ;* *x = a* ***xor*** *b ;*

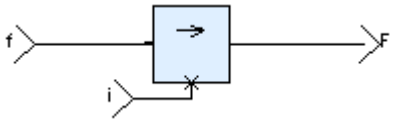


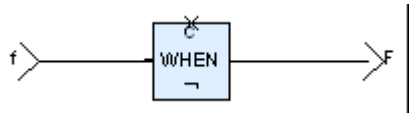
x = #(a,b) ; (исключение: выход принимает истинное значение когда только один входов принимает истинное значение)

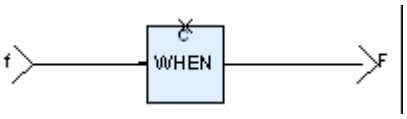


x = ***not****(a) ;*

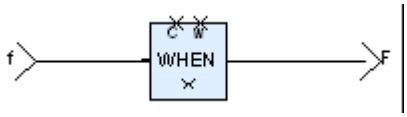
2.2.5.5. Временные операторы



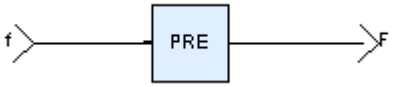




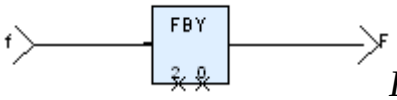
F = I -> f ; *F = (f)* ***when*** *C ;* *F = (f)* ***when*** *(****not*** *C) ;*



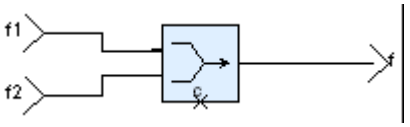
F = *(f)* ***when*** *(C* ***match*** *v1) ;*



F = ***pre****(f) ;*



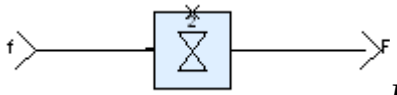
F = ***fby*** *(f; 2;0) ;* Оператор не является примитивом. В данном случае это 0 ->pre (0 -> pre (f)), т.е. задержка f на два такта и начальными значениями 0.



f = ***merge*** *(c; f1; f2);* (объединение, где c – А жетификатор тактирования). Оператор позволяет создать поток с более высоким тактированием из стыкующихся потоков с низким тактированием. Таблица 2.4 иллюстрирует его поведение.

Таблица 2.4. Оператора ***merge***

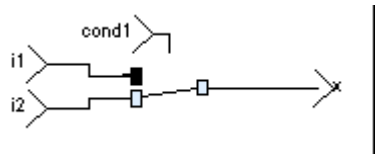
c	true	true	false	true	false	false	…
f1	f1_1	f1_2	f1_3	fi1_4	f1_5	f1_6	…
f2	f2_1	f2_2	f2_3	f2_4	f2_5	f2_6	…
f1 when c	f1_1	f1_2		f1_4			…
f2 when not c			f2_3		f2_5	f2_6	…
merge(c; f1 when c; f2 when not c)	f1_1	f1_2	f2_3	f1_4	f2_5	f2_6	…



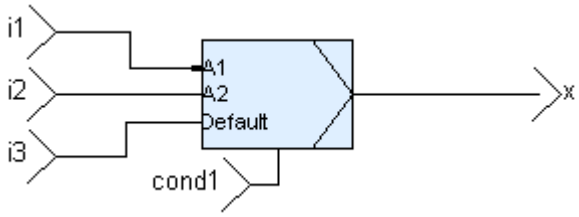
F = 2 ***times*** *f ;* Оператор ***times*** (счетчик заданного количества условий) не является примитивом. Его поведение определяет следующий узел:

node *times_behavior (n :* ***int****; c :* ***bool****)* ***returns*** *(o :* ***bool****)*
var
v3, v4 : ***int****;*
let
v4 = n -> ***pre*** *(v3);*
v3 = ***if*** *(v4 < 0)* ***then*** *v4* ***else*** *(****if*** *c* ***then*** *v4 – 1* ***else*** *v4);*
o = c ***and*** *(v3 = 0);*
tel

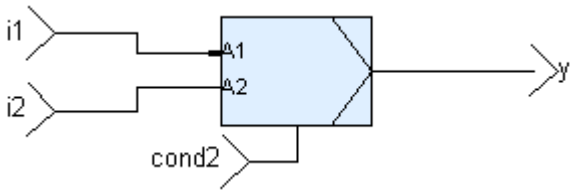
2.2.5.6. Операторы выбора



*x = **if** cond1 **then** (i1) **else** (i2) ;*

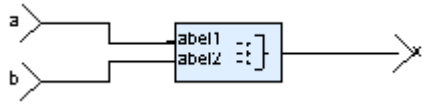


*x = (**case** cond1 **of** | A1 : i1 | A2 : i2 | : i3) ;*

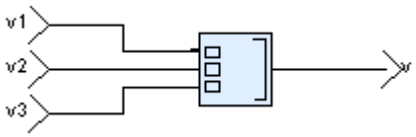


*y = (**case** cond2 **of** | A1 : i1 | A2 : i2) ;*

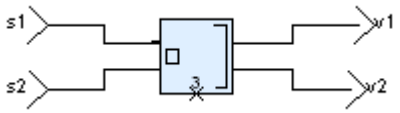
2.2.5.7. Операторы структуры и массива



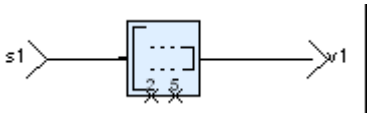
x = { label1 : a, label2 : b}: – построение структуры из списка величин.



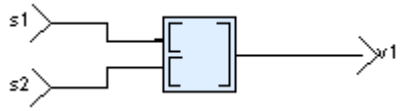
v = [v1, v2, v3]; – конструктор массива, перечисление значений.



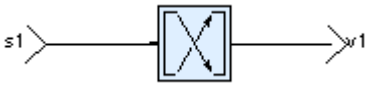
v1 , v2 = (s1^3 , s2^3); – конструктор массива, повторение значения.



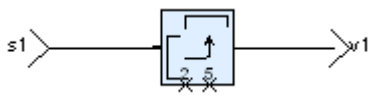
v1 = w1[2 .. 5] ; – доступ к массиву: элементы со 2 по 5.



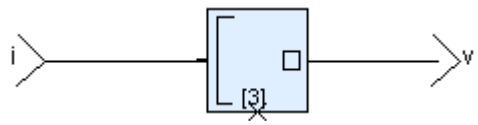
*v1 = s1 **@** s2* ; – конкатенация массивов.



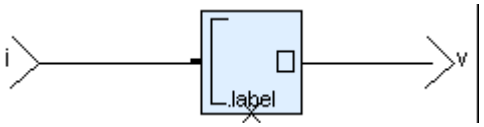
*v1 = **reverse** s1 ;* – реверс массива.



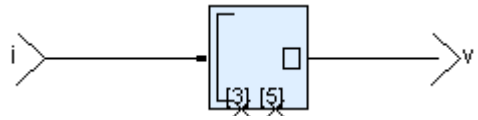
*v1 = **transpose** (s1;2;5) ;* – транспозиция массива: v1 равен s1 с обменом 2 и 5 элементов.



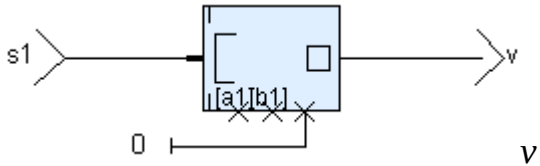
v = i[3] ;



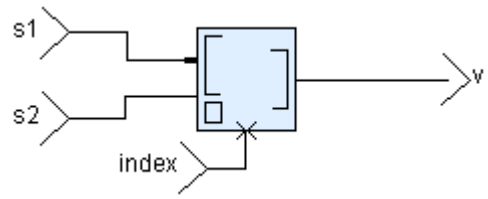
v = i.label ;



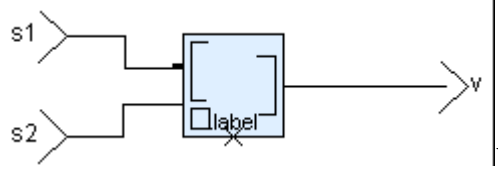
v = i[3][5] ;



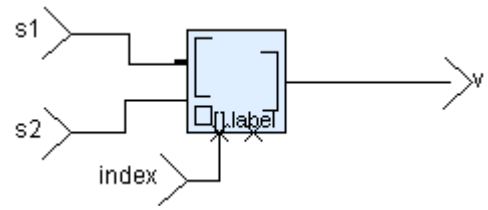
v = (s1.[a1][b1] ***default*** *0) ;* – динамическое индексирование при доступе к массиву.



v = (s1 ***with*** *[index] = s2) ;* – копирование массива с модификацией: v[i]=s2[i], если i= index, иначе v[i]=s1[i].



v = (s1 ***with*** *.label = s2) ;* – копирование структуры с модификацией.

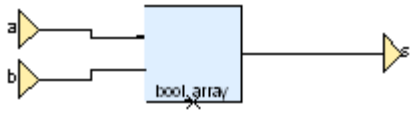


v = (s1 ***with*** *[index].label = s2) ;* – копирование комбинации структуры и массива с модификацией.

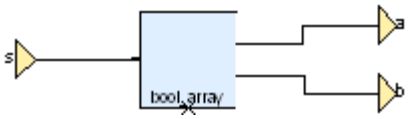
Сборка и разборка массива/структуры (Make и Flatten)

Предположим следующий тип и переменные:

type bool_array = [label1 : bool , label2 : int^2] ;
var a : bool ;
b : int^2 ;
s : bool_array ;



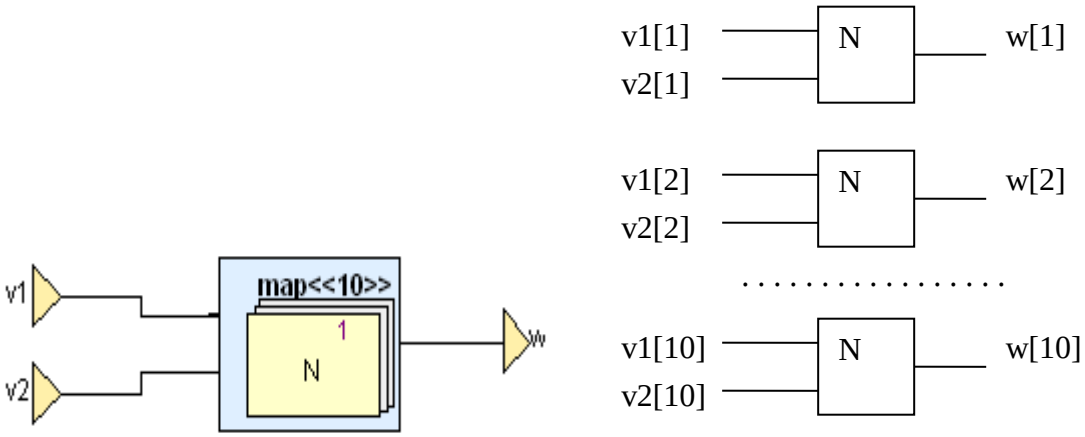
*s = (****make*** *bool_array)(a, b) ;*



*a, b = (****flatten*** *bool_array)(s) ;*

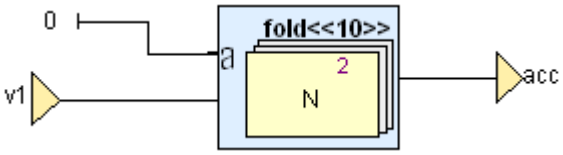
2.2.5.8. Операторы высшего порядка

Итератор отображения (map):

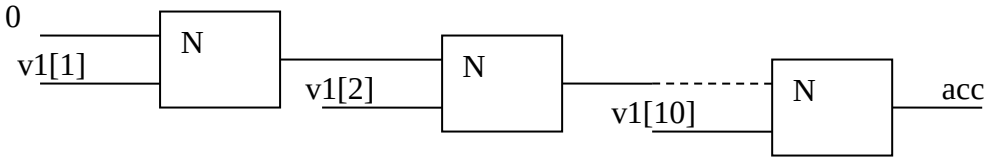


*w = (**map** N << 10 >>) (v1 , v2) ;*

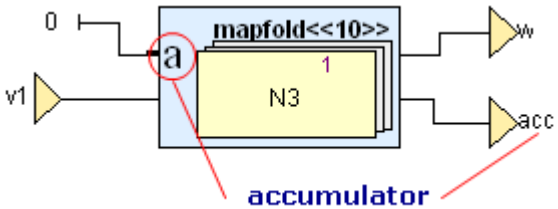
Итератор свертывания (fold):



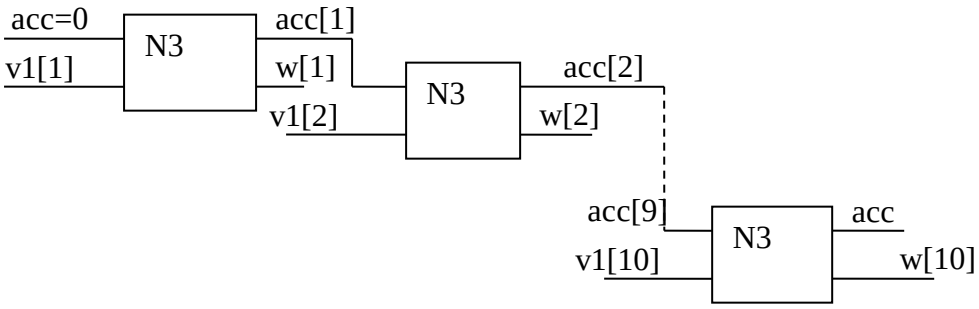
*acc = (**fold** N << 10 >>) (0, v1) ;*



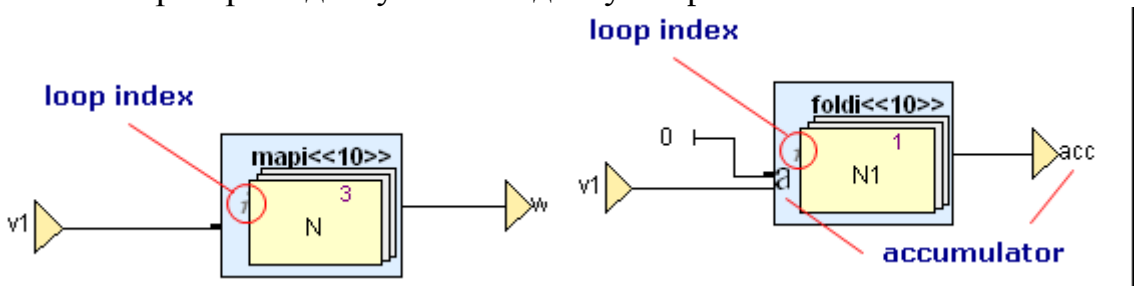
Итератор отображения-свертывания



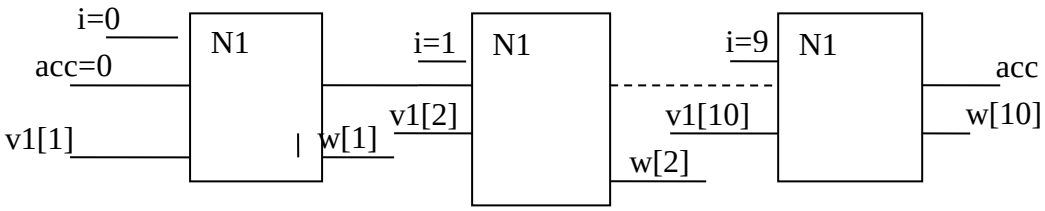
(mapfold):



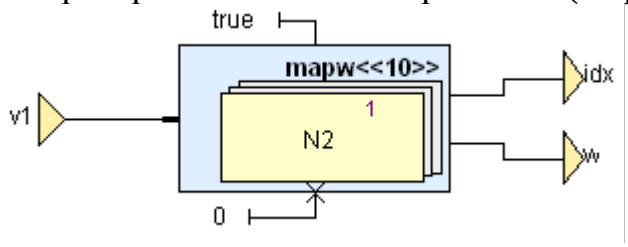
Итераторы с доступом к индексу mapi foldi:



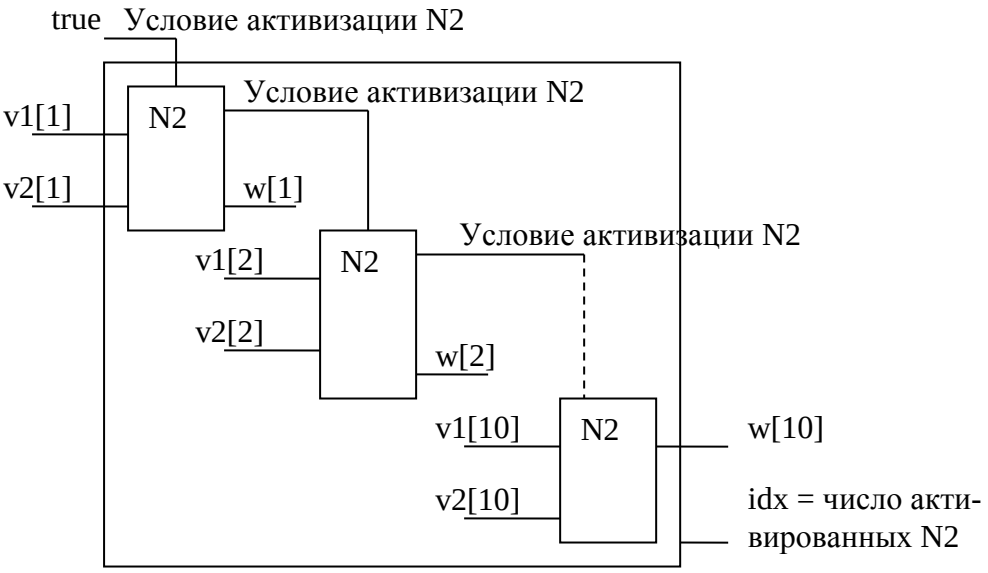
*w = (**mapi** N << 10 >>) (v1) ;* *acc = (**foldi** N1 << 10 >>) (0, v1) ;*



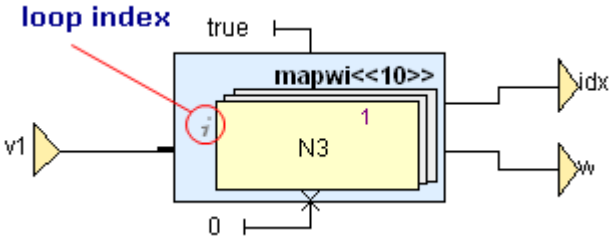
Итератор частичного отображения (mapw):



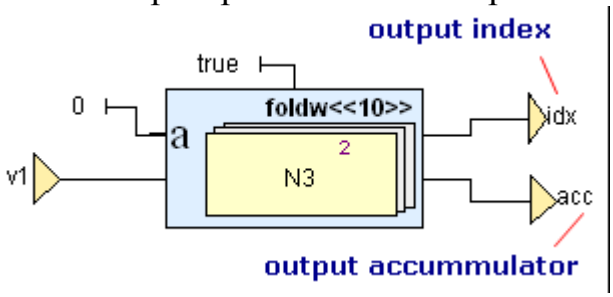
*idx , w = (**mapw** N2<< 10 >> **if true default** 0) (v1);*



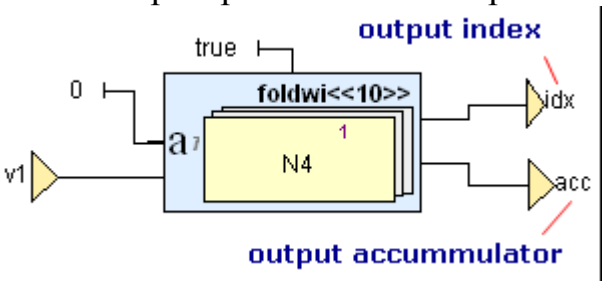
Итератор частичного отображения с доступом к индексу (mapwi):



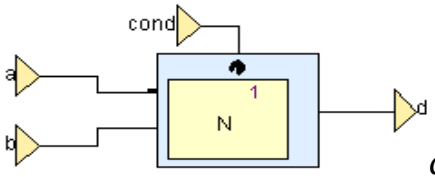
Итератор частичного свертывания (foldw):



Итератор частичного свертывания с доступом к индексу (foldwi):

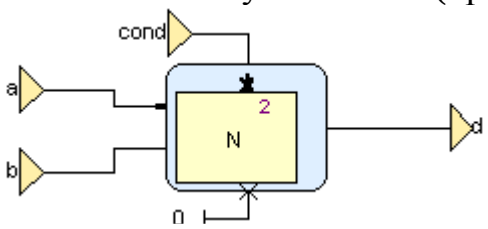


Создание экземпляра перезапускаемого узла:



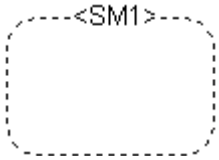
*d = (**restart** N **every** cond) (a, b) ;*

Создание экземпляра узла с условной активизацией и начальными значениями по умолчанию (предполагает наличие памяти):

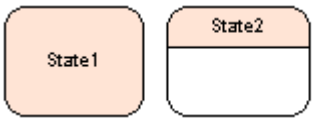


*d = (**activate** N **every** cond **initial default** 0) (a, b) ;*

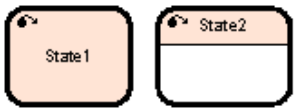
2.2.5.9. Автоматы



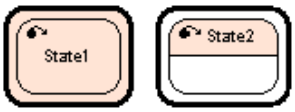
automaton *SM1*
returns .. ; - автомат



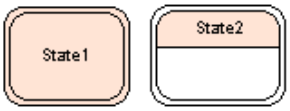
state *State1* – состояния
state *State2*
let
tel



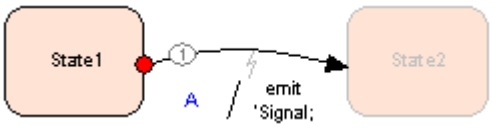
initial state *State1*
initial state *State2*
let
tel



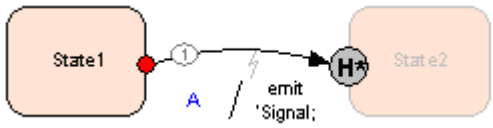
initial final state *State1*
initial final state *State2*
let
tel



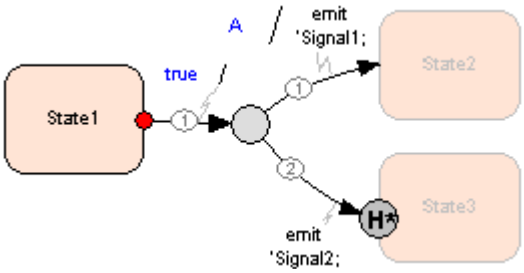
final state *State1*
final state *State2*
let
tel



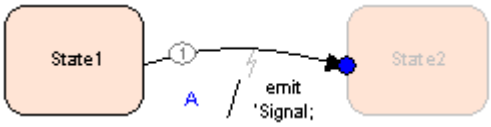
state State1
unless if *A* ***do let emit*** *'Signal; tel*
restart *State2*



state State1
unless if *A* ***do let emit*** *'Signal; tel*
resume *State2 ;*

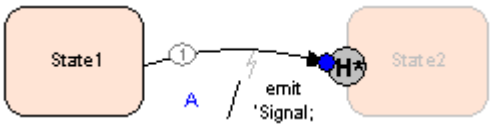


state State1
unless
if *true*
if *A* ***do let emit*** *'Signal1; tel* ***restart*** *State2*
else do let emit *'Signal2; tel* ***resume*** *State3*

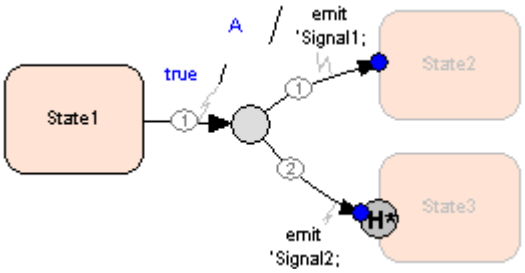


state State1
until if *A* ***do let emit*** *'Signal; tel*
restart *State2 ;*

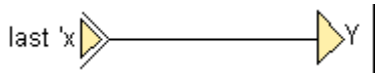
end;



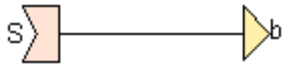
state State1
***until if** A **do let emit** 'Signal; tel*
***resume** State2 ;*



state State1
until
***if** true*
***if** A **do let emit** 'Signal1; tel **restart** State2*
***else do let emit** 'Signal2; tel **resume** State3*
end;

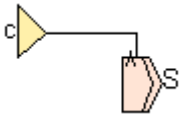


*Y = **last** 'x ;*



b = 'S ;

***emit** 'S ;*



***emit** 'S **if** c ;*

2.3. Разработка прикладного программного обеспечения управляющей системы реального времени в интегрированной среде ориентированной на модель

2.3.1. Линейные системы

Преобразование дискретных линейных систем в программы на Lustre является очевидной задачей. Если система представлена в виде выражений z-преобразования, то преобразование сводится к замене z^-1 на *0.0 –>pre( ).*

Пусть фильтр второго порядка представлен импульсной передаточной функцией:

$$H(z)=(a*z^2+b*z+c)/(z^2+d*z+e).$$

Выход $y = H(z)*x$ может быть записан как:

$$y=a*x+(b*x-d*y)*z^-1+(c*x-e*y)*z^-2=$$
$$a*x+(b*x-d*y+(c*x-e*y)*z^-1)*z^-1,$$

а соответствующая программа как:

const** a,b,c,d,e: **real.

```
Node SECOND_ORDER(x: real) returns (y: real)
var u,v: real;
let
  y=a*x+(0.0 –>pre(u));
 u=b*x-d*y+(0.0 –>pre(v));
 v=c*x-e*y;
tel.
```

Запишем выражение $y = H(z)*x$ как:

$$y=a*x+b*x* z^{-1}+ c*x*z^{-2} – d*y*z^{-1} –e*y*z^{-2}.$$

На рис. 2.11 дано схемное представление этого равенства в SCADE.

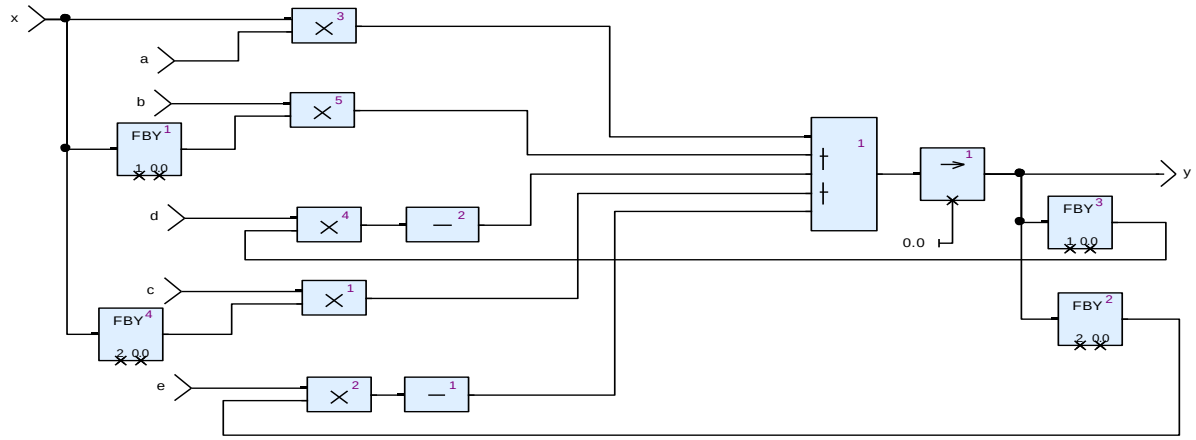


Рис. 2.11. Схема фильтра второго порядка в SCADE.

Приведенный фильтр второго порядка станет нестационарным, если вместо констант *a,b,c,d* и *e* использовать параметры. Также легко ввести нелинейность, например:

```
y=rho*cos(theta0–>pre(theta));
```

2.3.2. Логические системы

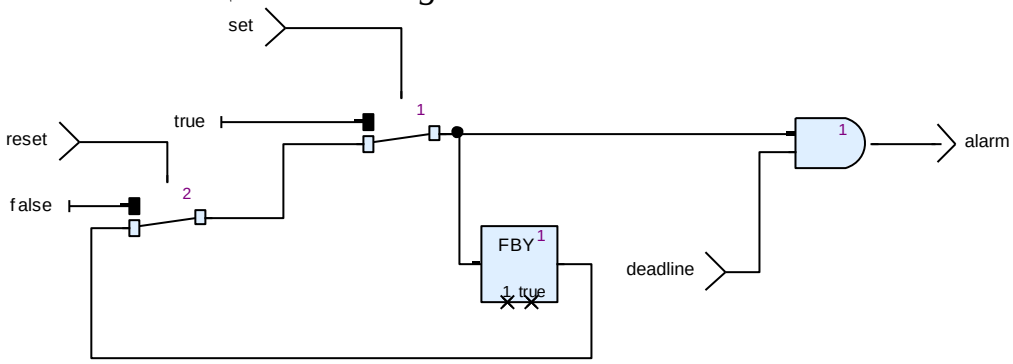
В приведенном выше примере потоковой программы для обработки сигналов выражения специфицируют исключительно динамику системы. Однако многие системы имеют важные логические компоненты и некоторые из них, например, системы мониторинга являются существенно логическими системами. Такие системы часто описываются в терминах автоматов, частичных автоматов, сетей Петри и т.д., т.е. императивными формализмами, которые представляют состояния и переходы между ними. Вопрос адекватности потоковой парадигмы, позволяющей легко представлять непрерывные системы дискретного времени, проверим для логических систем на следующем примере. Рассмотрим три варианта watchdog (сторожевого таймера), устройства для мониторинга времени отклика.

В первом варианте три входа: команды *set* и *reset* и событие наступление предельного срока (*deadline*). Выход – сигнал тревоги (*alarm*) должен возникать всякий раз, когда событие *deadline* появляется после команды *set*. Обычно событие представляется булевой переменной, значение *true*

которой индицирует наличие события. Положим, что команды *set* и *reset* не могут приходить одновременно, тогда (порядок следования выражений не имеет значения):

node *WatchDog1(set, reset,* deadline*:**bool**) **returns**(alarm: **bool**);*
var *is_set: **bool**;*
let
*alarm = deadline **and** is_set ;*
*is_set =set –>**if** set **then true else if** reset **then false else pre**(is_set);*
***assert not**(set **and** reset);*
tel.

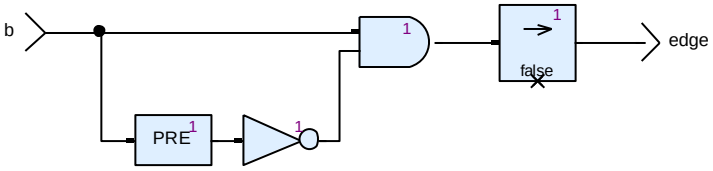
Реализация *WatchDog1*в SCADE:



Во втором варианте те же команды, но *alarm* наступает после того как команда *reset* не появляется спустя определенное время после команды *set*. Это время задается количеством базовых тактов. Новая программа будет использовать *WatchDog1* для обеспечения необходимого параметра для события *deadline*: приход *set* устанавливает начальное значение регистра, которое затем декрементируется. Предельный срок наступает, когда значение регистра уменьшается до 0. Вспомогательный узел *EDGE* возвращает значение *true*, когда обнаруживает изменение входного сигнала от *false* к *true*:

node *EDGE (b:**bool**) **returns**(edge: **bool**);*
let
*edge = **false** –>(b **and not pre**(в));*
tel.

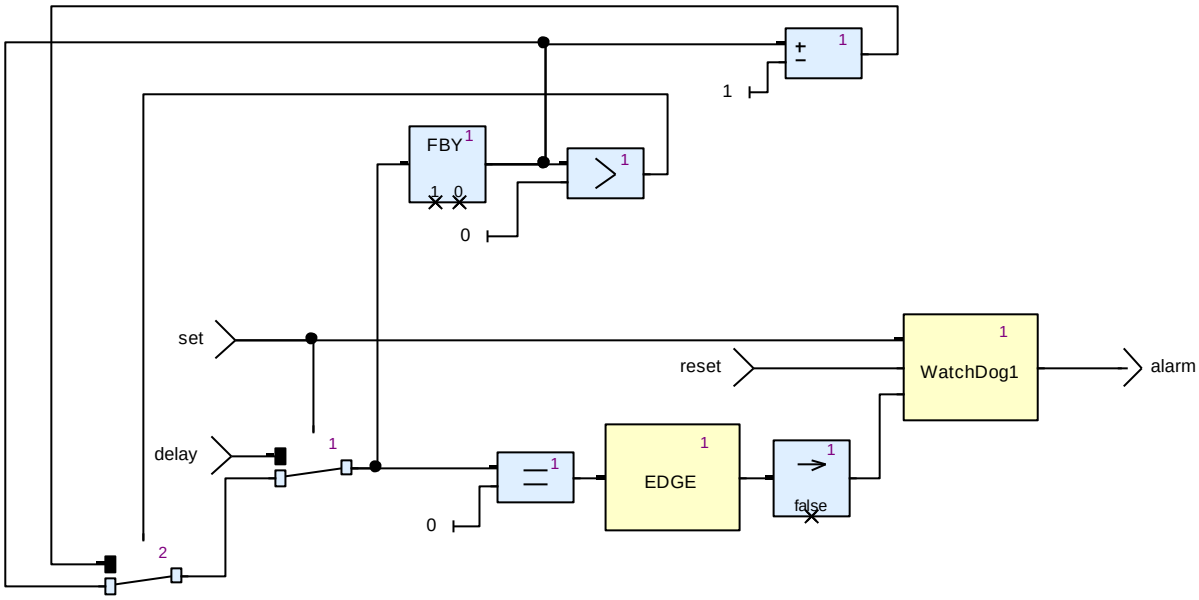
Реализация *EDGE* в SCADE:



node *WatchDog2(set, reset:**bool**; delay:**int**) **returns**(alarm: **bool**);*
var *remain: **int**; deadline: **bool**;*
let
alarm = WatchDog1(set, reset, deadline*);*
*deadline =**false** –> EDGE(remain=0);*

*remain=**if** set **then** delay **else if pre**(remain)>0 **then pre**(remain)-1 **else** **pre**(remain);*
tel.

Реализация *WatchDog2*в SCADE:



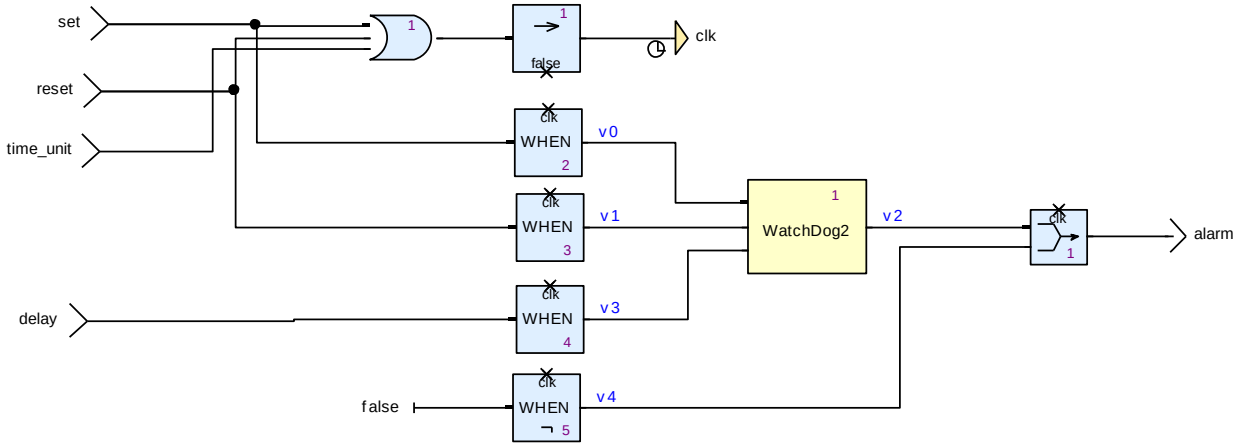
Предположим, что задержка дается в заданном временном масштабе, т.е. с шагом *time_unit*. Необходимо поэтому вызывать *WatchDog2* не во всех, а только в определенных тактах, задаваемых истинными значениями булевого потока тактирования *clk* (понижение скорости тактирования), отсюда получаем третий вариант. Восстановление исходной скорости тактирования (повышение скорости тактирования) выполняется оператором ***current***.

***Node** WatchDog3(set, reset, time_unit :**bool**; delay:**int**) **returns**(alarm: **bool**);*
***var** clk: bool;*
let
*alarm = **current**(WatchDog2((set, reset, delay) **when** clk);*
*clk =**true** –> (set **or** reset **or** time_unit);*
tel.

Оператор ***current*** не реализован в Scade, поэтому повышение скорости тактирования выполним с помощью оператора ***merge*** из Scade.

***Node** WatchDog3(set, reset, time_unit :**bool**; delay:**int**) **returns**(alarm: **bool**);*
***clock** clk: bool;*
let
*alarm = **merge (**clk; WatchDog2((set, reset, delay) **when** clk) **when** clk; false*
*__**when not** clk**)***__
*clk =**true** –> (set **or** reset **or** time_unit);*
tel.

Реализация *WatchDog3* в SCADE:



2.3.3. Смешанные системы

Моделирование смешанных систем с обработкой сигналов и логическими преобразованиями так же довольно простая задача. Компоненты обработки сигналов вычисляют логические значения с помощью логических выражений и операторов отношения, а логические компоненты вычисляют значения потоков управления с помощью условного оператора ***if then else*** и темпоральных операторов ***when*** и ***current*** (***merge***).

Рассмотрим модель смешанной системы на примере цифровых часов с секундомером (digital stopwatch), как на рис. 2.12. [20]

Рис. 2.12. Цифровые часы с секундомером

По аналогии с разработкой аппаратуры часы можно представить моделью программно-управляемого автомата (ПУА) [21]. ПУА состоит из устройства управления (УУ) и операционного устройства (ОУ), как показано на рис. 2.13.

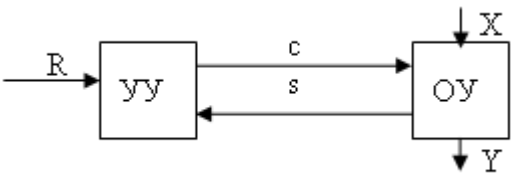


Рис.2.13. Программно-управляемый автомат

Первое свойство этой модели гласит, что операционное устройство задается на уровне регистровых передач (в нашем случае преобразованием потоков данных), а устройство управления задается на алгоритмическом уровне (в нашем случае конечным автоматом). Второе свойство утверждает, что ПУА является рекурсивной моделью. Это означает, что УУ может представлено в виде ПУА1 (ОУ1, УУ1), УУ1 в виде ПУА2 (ОУ2, УУ2) и так далее.

На ОУ определено множество микроопераций (МО, в нашем случае узлов), выполняющих преобразование входных сигналов (потоков) X из окружающей среды в выходные сигналы (потоки) Y для окружающей среды. Выполнение той или иной МО инициируется выходными сигналами (потоками) с УУ, зависящими от состояния УУ, его входов R и s. Сигналы s (потоки) характеризуют результаты выполнения операций в ОУ, а R – управляющие воздействия (потоки) из окружающей среды.

Часы с секундомером не что иное, как набор из счетчиков долей секунд по модулю 100, секунд по модулю 60, минут по модулю 60 и часов по модулю 24 – это и есть ОУ. Счетчики отличаются только модулями, поэтому можно создать параметризованную модель *ModCount* с заданием модуля счета, как на рис. 2.14.

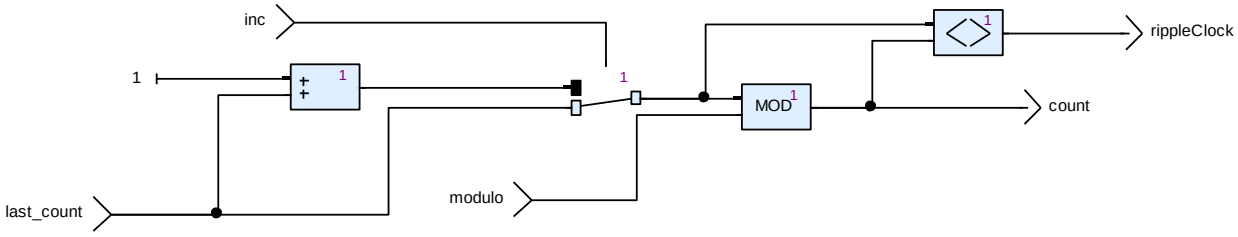


Рис.2.14.

На вход *last_count* поступает предыдущее значение потока счета, на выходе *count* формируется текущее значение потока равное *last_count+1* по модулю *modulo*, если вход *inc* принимает значение *true*, иначе присваивается предыдущее значение. На выходе *rippleClock* (перенос) формируется значение *true*, если текущее значение счетчика достигает величины равной *modulo*. Рис. 2.15 изображена модель часов как набор из четырех последовательно взаимодействующих счетчиков каждый со своим модулем. Эта модель получена с помощью итератора ***mapfold*** с параметром 4, который представляет каскадное соединение четырех счетчиков *ModCount* (выход *rippleClock* очередного каскада соединяется со входом *inc* следующего каскада). Переменная *clk* – массив из 4 элементов, каждый из которых представляет текущее значение счетчиков долей секунд, секунд, минут и часов соответственно.

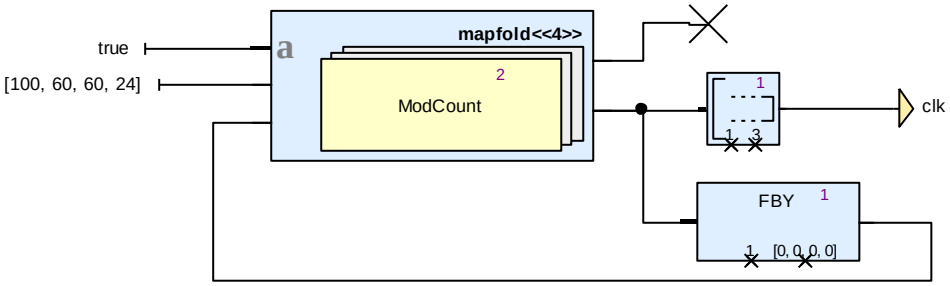


Рис.2.15.

С точки зрения управления (часть УУ ПУА) часы с секундомером это конечный автомат на два состояния: *stopped* (останов) и *running* (работа). Кнопка *start_stop* последовательно переключает часы между этими состояниями. Кнопка «reset» сбрасывает счетчики секундомера только в состоянии *stopped*, а часы функционируют независимо от состояния устройства управления.

На рис. 2.16 изображен граф переходов этого автомата под названием *chrono*. Каждый автомат в SCADE дожжен иметь начальное состояние. Состояние *Stopped* назначено таковым, о чем свидетельствует черная окантовка и маленькая пиктограмма в левом верхнем углу.

В состоянии *running* считает секундомер полученный итератором ***mapfold*** из трех экземпляров счетчика *ModCount* (доли секунд, секунды, минуты).

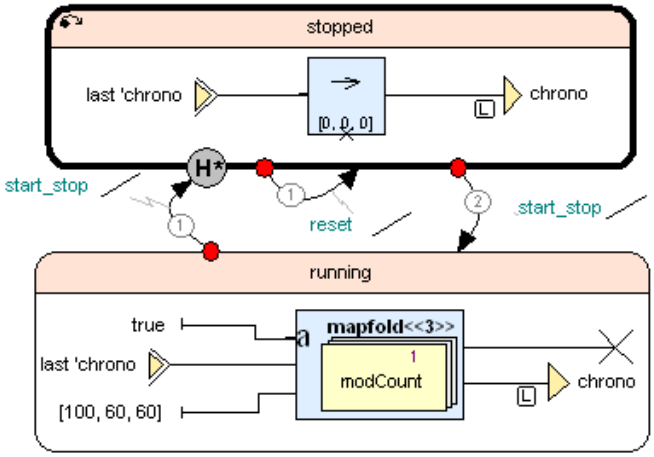
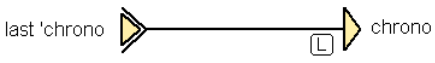


Рис.2.16.

Выходная составляющая локальной переменной *chrono* представляет текущее значение компонент секундомера, а его входная составляющая *last'chrono* – предыдущее значение.

Когда переменная определяется в нескольких состояниях, состояния, где она точно неопределенна, переменная сохраняет свое старое значение. Поэтому если оставить состояние *stopped* пустым, то это будет то же самое, если заполнить его равенством:

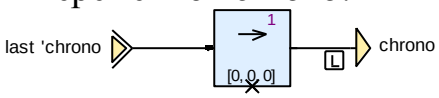


а в поле "Last" свойств переменной *chrono* ввести [0,0,0].

Реализации сброса секундомера в начальное значение по кнопке *reset* в состоянии *stopped* основано на использовании строгого перехода с

restart, при котором происходит сброс содержимого состояния как и в первом такте.

Поэтому в состояние *stopped* введен оператор ***init*** с начальным А жениием (0, 0, 0) для переменной *chrono*:



Переход из состояния *running* в состояние *stopped* по кнопке *start_stop* реализован с ***resume***, что позволяет избежать рестарта *chrono* в *stopped*. Далее необходимо добавить в модель часы с секундами, минутами и часами. Часы в отличие от секундомера никогда не останавливаются. Более того необходимо переключать индикацию значений между часами и секундомером, которые могут функционировать одновременно. Для этого в модель к секундомеру добавим параллельный поток моделирующий часы как на рис. 2.17.

Есть две локальные переменные *chrono* и *clk*, которые необходимо поочередно индицировать. Для этого введем выход *display* и автомат *display* с состояниями *display_clock* и *display_chrono* (рис. 2.18). Роль автомата *display* состоит в выборе какую из переменных выводить на экран часов при нажатии кнопки *mode*. В состоянии *display_clock* индицируется значение *clk* (выходу *display* присваивается переменная *clk*), а когда нажимается кнопка *mode*, то автомат переходит в состояние *display_chrono* и индицируется значение *chrono* (выходу *display* присваивается переменная *chrono*).

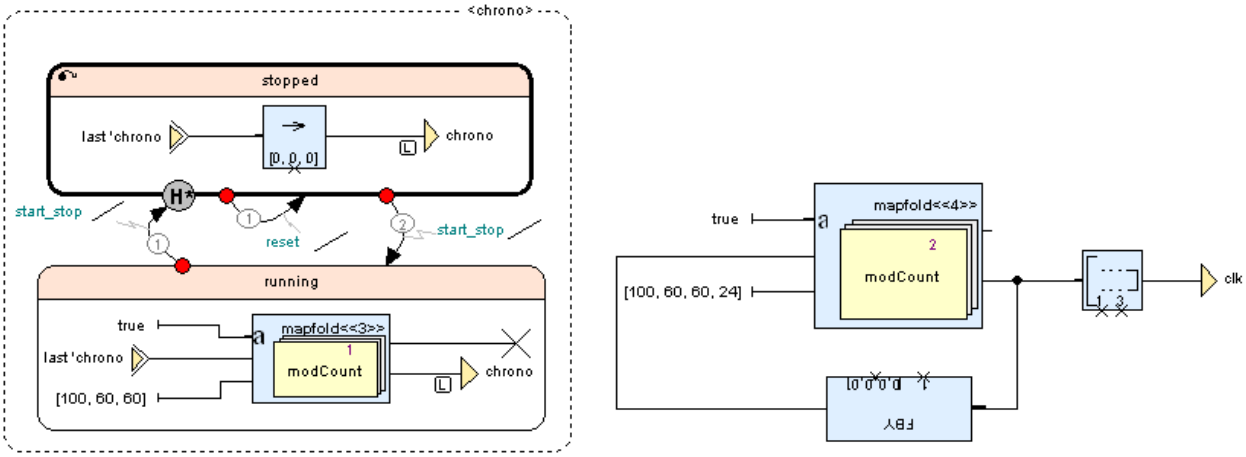


Рис.2.17. Параллельная работа автомата и потока

Для правильной работы необходимо блокировать переход из состояния *stopped* в состояние *running* (запуск секундомера), когда автомат *display* находится в состоянии *display_clock*. Поэтому в состоянии *display_clock* постоянно формируется сигнал *clk_mode*, который входит в условие перехода в состояние *running*. На рис. 2.19 приведена окончательная модель часов с секундомером.

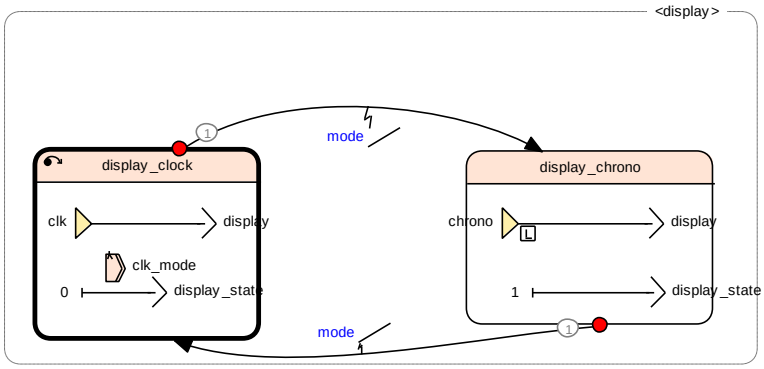


Рис.2.18. Параллельная работа автомата и потока

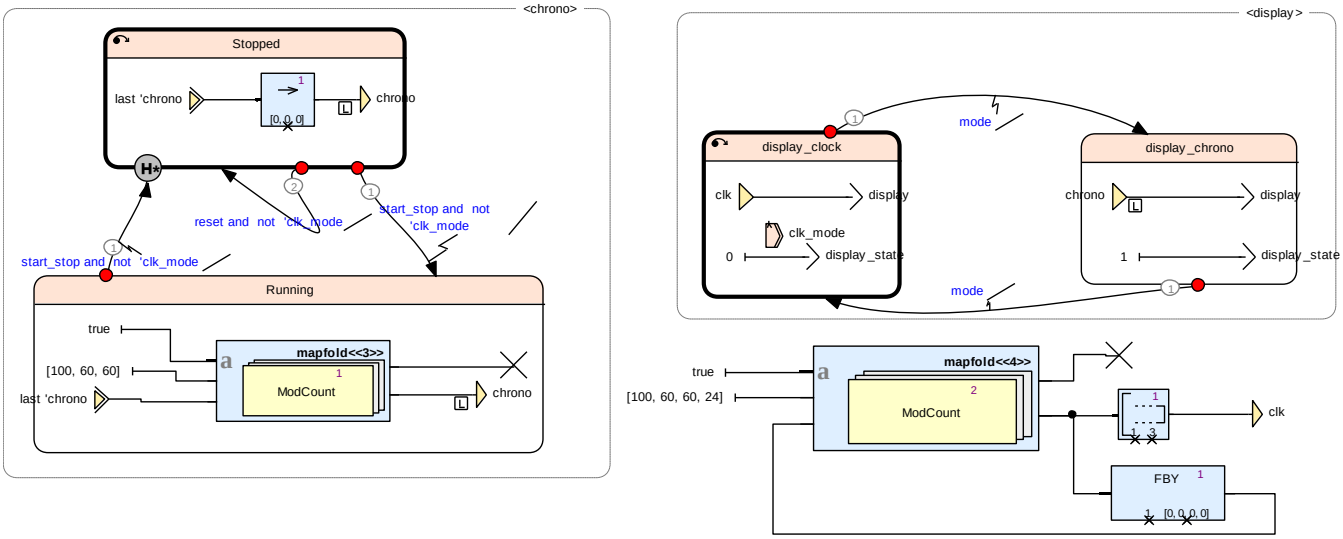


Рис.2.19. Модель часов с секундомером

2.3.4 OFDM модулятор

OFDM (Orthogonal Frequency-Division Multiplexing) – мультиплексирование с ортогональным частотным разделением каналов является одной из цифровых схем модуляции.

Две вещественные функции $\varphi_1(t)$ и $\varphi_2(t)$ на интервале $[a, b]$ называются ортогональными если

$$\int_a^b \varphi_1(t)\varphi_2(t)\,dt = 0$$

Для комплексных функций вводится комплексное сопряжение одной из функций под интегралом.

OFDM разбивает имеющийся широкополосный канал на N узкополосных каналов (обычно 100-8000) каждый со своей поднесущей. Эти поднесущие делают ортогональными друг к другу. Это означает, что каждая из них имеет целое число периодов на интервале длительности символа OFDM как на рис. 2.20.

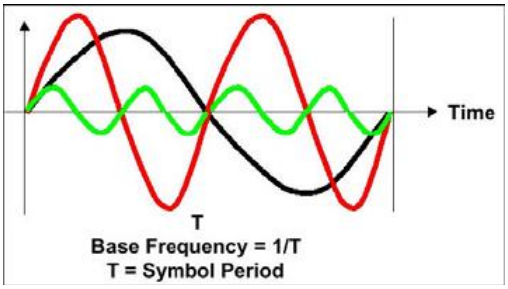


Рис.2.20. Множество ортогональных сигналов

Таким образом, спектр каждой поднесущей в системе имеет значение «0» в центре каждой другой поднесущей в системе (рис. 2.21). Это уберегает поднесущие от интерференции (взаимовлияния).

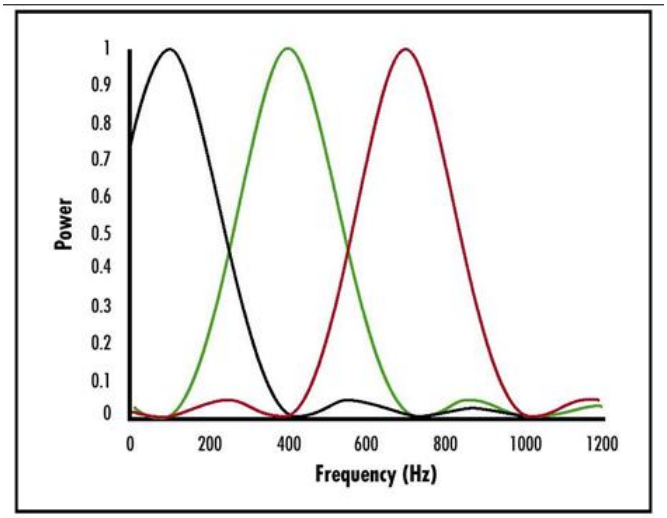


Рис.2.21. Ортогональность поднесущих

Пусть выбрана схема модуляции для каждого канала (например, BPSK или QPSK или QAM или …). Обычно схема модуляции одна для всех каналов и пусть она преобразует m-разрядный элементарный символ в соответствии с модулирующим созвездием. Очередные N*m бит последовательного информационного потока, поступающего со скоростью $1/T_b$ (где T_b – длительность битового интервала) преобразуются в параллельный код из N m-разрядных элементарных символов (OFDM символ). Каждый элементарный символ модулируется (X_0, X_1, …X_{N-1}) и переносится с помощью обратного быстрого преобразования Фурье (ОБПФ, IFFT, FFT^-1) на свою поднесущую, складываясь друг с другом (рис. 2.22). Итак, исходный высокоскоростной поток разбивается на N низкоскоростных потоков с временем передачи элементарного символа $T_S = N*m*T_b$. Перед отправкой в среду передачи суммарный сигнал ОБПФ модулирует по схеме I/Q модуляции (*Re, Im*) повышающую несущую f_c. Полученный сигнал *S(t* излучается антенной в эфир.

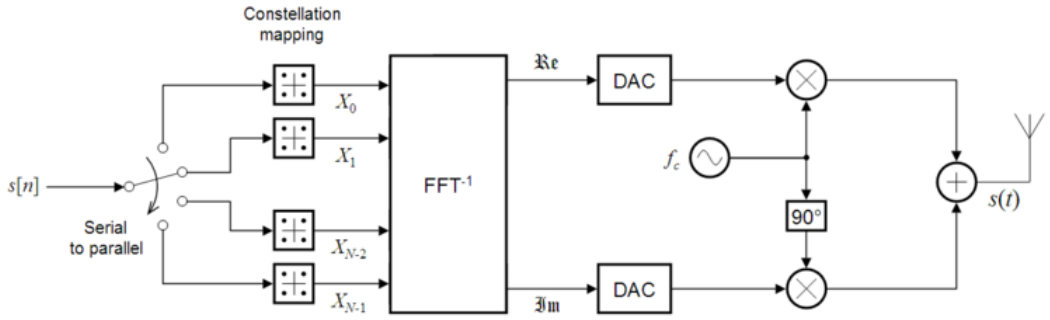


Рис.2.22. OFDM модулятор

Использование защитного интервала (циклический префикс - cyclic prefix) между символами позволяет рассеять следы из-за многолучевого приема от предыдущего OFDM символа перед приемом текущего символа, т.е. устранить межсимвольную интерференцию. В качестве циклического префикса используется конечная часть очередного OFDM символа (рис. 2.23).

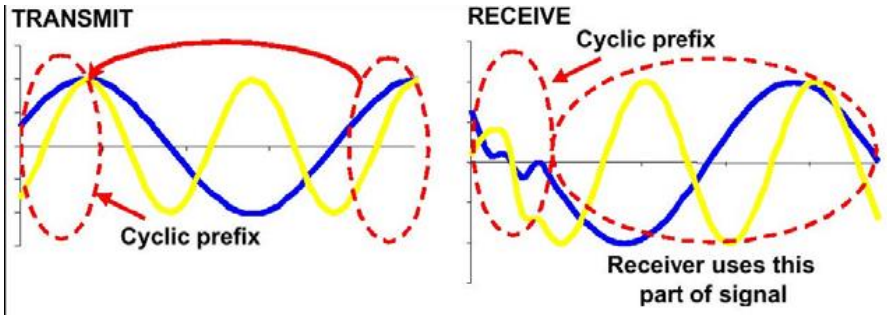


Рис.2.23. Циклический префикс

Рассмотрим SCADE модель OFDM модулятора до этапа переноса I/Q модулированного сигнала на f_c. Пусть:

• N = 64 – число поднесущих;

• T_b=4e-6 с – длительность битового интервала исходного потока;

• PAM модуляция элементарных двухразрядных символа (00->-3в, 01->-1в, 10->1в, 11->3в);

• T_S=N*2*T_b=128e-6 – время передачи OFDM символа;

• F_{SA}=4e6 – частота дискретизации при генерации sin/cos для обратного дискретного преобразования Фурье (ОДПФ, IDFT);

• NB=F_{SA}*T_S=512 – число отсчетов OFDM символа;

• PreGuard=40 – количество отсчетов защитного интервала (cyclic prefix это отсчеты OFDM символа с 472 по 511);

• PreGuard+NB=552 отсчетов.

Константы SCADE модели OFDM модулятора:

N=64,

NB=512,

Pi=3.141592654,

PRE_GUARD=40.

Для отладки входной испытательный двоичный (булев) поток данных будем формировать с помощью генератора псевдослучайной последовательности (ПСП, PRBS). PRBS реализован в виде 8-разрядного регистра с линейными обратными связями (LFSR Фибоначчи) представляющий полином y(x)=x^8+x^4+1 (рис 2.24).

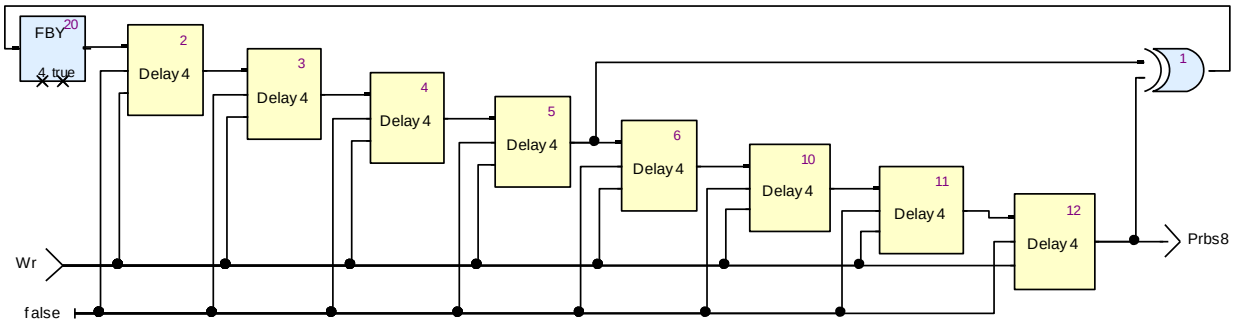


Рис.2.24. SCADE модель Фибоначчи LFSR для полинома y(x)=x^8+x^4+1

В примере каждый OFDM символ переносит 128 бит. На это тратится 512 отсчетов. Поэтому в SCADE модели триггеры LFSR моделируется задержкой в 4 такта, которая обеспечивает одинаковые значения четыре соседних элемента в потоке Prbs8. В начале каждого OFDM символа передается защитный префикс из 40 отсчетов. В это время LFSR не должен работать (сохраняет последнее состояние). Для этого в модель каждого разряда введены библиотечные элементы памяти *MEM* , которые по сигналу *Wr* запоминают состояние ячейки (рис 2.25).

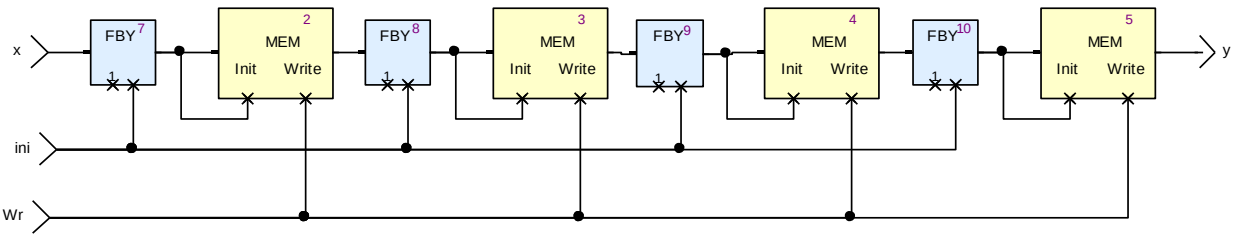


Рис.2.25. SCADE модель ячейки LFSR Фибоначчи

PAM модулятор десериализует входной поток и преобразует значения двух соседних элементов двоичной последовательности в уровень напряжения согласно схеме:

-3в соответствует комбинации 00,
-1в соответствует комбинации 01,
1в соответствует комбинации 10,
3в соответствует комбинации 11.

Он реализован с помощь итератора *mapfold* над N операторами *DeserPAM* (рис 2.26). Две задержки на 4 такта каждая в *DeserPAM* представляют два соседних бита в потоке, которые с помощью библиотечной функции *BoolVect2int* преобразуются в целое число. Это значение выступает в качестве селектора для оператора *case*, реализующего схему преобразования PAM. Через 512 тактов на выходе PAM модулятора будут одновременно присутствовать N целых значений уровней элементарных символов PAM.

Итератора mapfold для DeserPAM

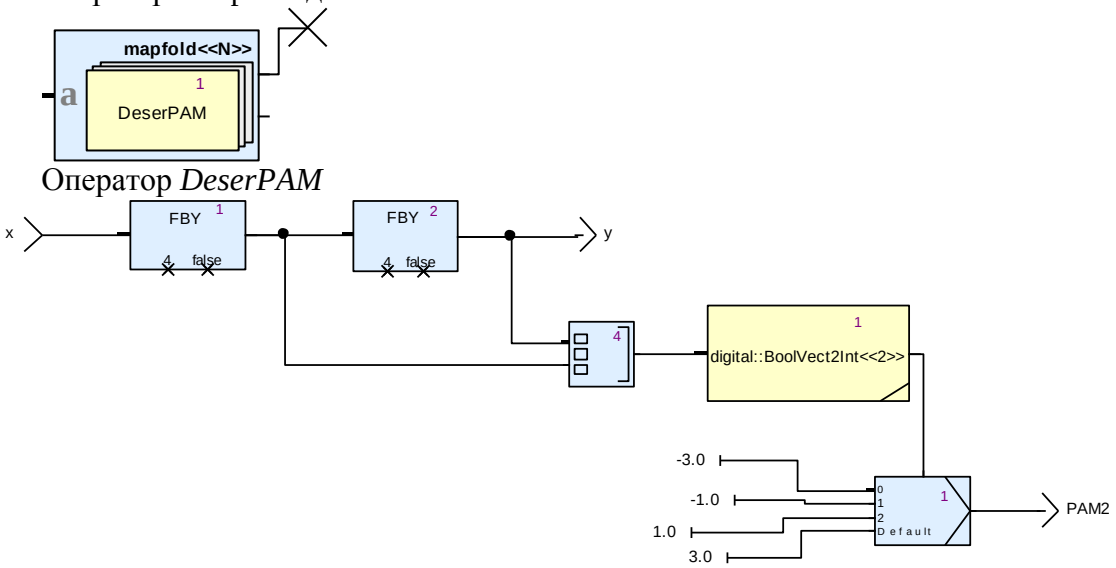


Рис.2.26. SCADE модель PAM модулятора

Каждый из N уровней моделирует амплитуду своей гармонической поднесущей, представленной NB отсчетами. Одноименные отсчеты всех поднесущих суммируются (обеспечивает итератор *foldi* для IDFT) – это реализация дискретной формы обратного преобразования Фурье (ОДПФ, IDFT, рис. 2.27). Отсчет *k* поднесущей *i* OFDM символа равен:

$$x(k,i)=X(i)* f(2 * PI * k * i / NB),$$

где *X(i)*– амплитуда элементарного символа *i* после PAM модуляции, а *f* – функция *sin(x)/cos(x)* (в примере для первой половины поднесущих это *cos*, а для второй – *sin*). Отсчет *k* OFDM символа равен:

$$x(k)=Sum_{i=1,\ldots,N} X(i)* f(2 * PI * i * k / NB).$$

Итератора *foldi* для IDFT

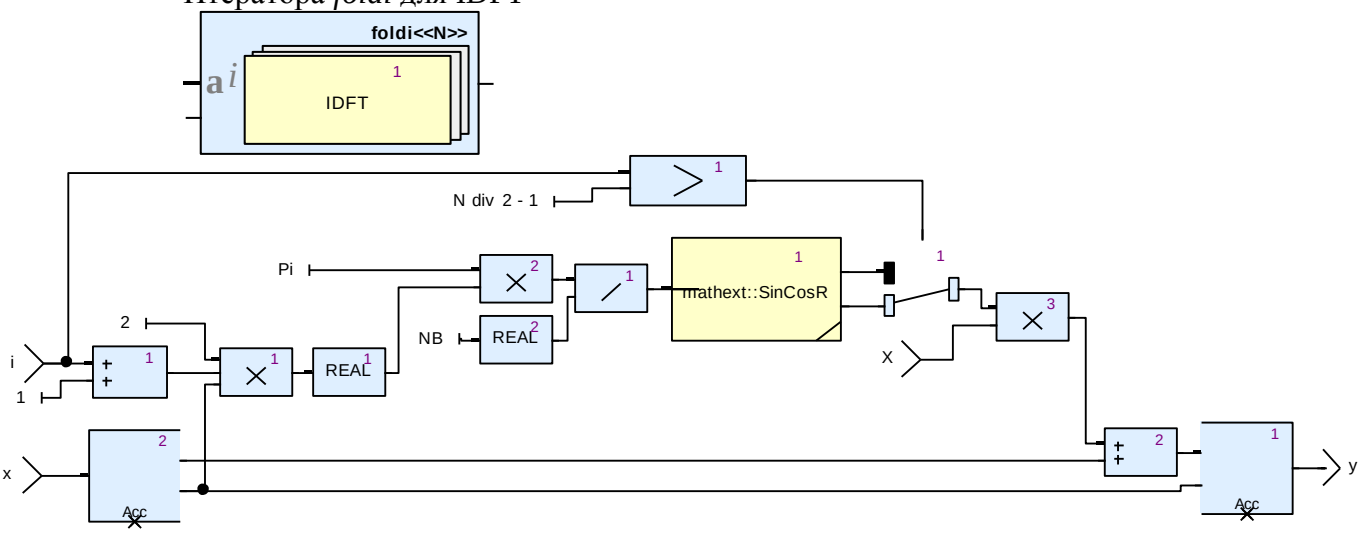


Рис.2.27. SCADE модель IDFT

Одним из главных недостатков сигналов с множеством несущих и OFDM в частности является высокое значение коэффициента PAPR (Peak-to-Average Power Ratio, отношения пикового значения мощности к средне-

му значению мощности, пик-фактор), являющееся результатом суммирования N-го количества поднесущих частот со случайными значениями амплитуд и фаз. Формируется сигнал с большими амплитудными выбросами по сравнению со среднеквадратичным уровнем сигнала. Это вызывает нелинейные искажения усилителей мощности передатчиков, работающих в режиме с отсечкой тока.

Одним из эффективных способов решения указанных проблем является включение после IDFT дополнительного фазового (ФМ) (или частотного (ЧМ)) модулятора, т.е. реализация сигналов OFDM с постоянной огибающей (Constant Envelope OFDM (CE-OFDM)). Применение сигналов с постоянной огибающей позволяет использовать в передатчиках нелинейные усилители мощности, обладающие высоким КПД (порядка 80%), а сами CE-OFDM сигналы, по сравнению с обычными OFDM сигналами, обладают большей помехоустойчивостью в условиях многолучевого приема. В примере реализован следующий вариант (рис. 2.28):

$$x'(k)= x(k)* 0.079 * 0.4+0.7853981,$$
$$x''(k)= exp(j * x'(k)) = cos(x'(k)) + j* sin(x'(k)).$$

Cos(x'(k)) и *sin(x'(k))* соответственно синфазная и квадратурная составляющие для последующей I/Q модуляции формируются с помощью библиотечной функции *SinCosR*. Эти потоки задерживаются операторами *Delay* на NB+PRE_GUARD тактов и запоминаются элементами памяти *MEM*, для того чтобы затем последние PRE_GUARD тактов очередного OFDM символа можно было передать в его начале как cyclic prefix.

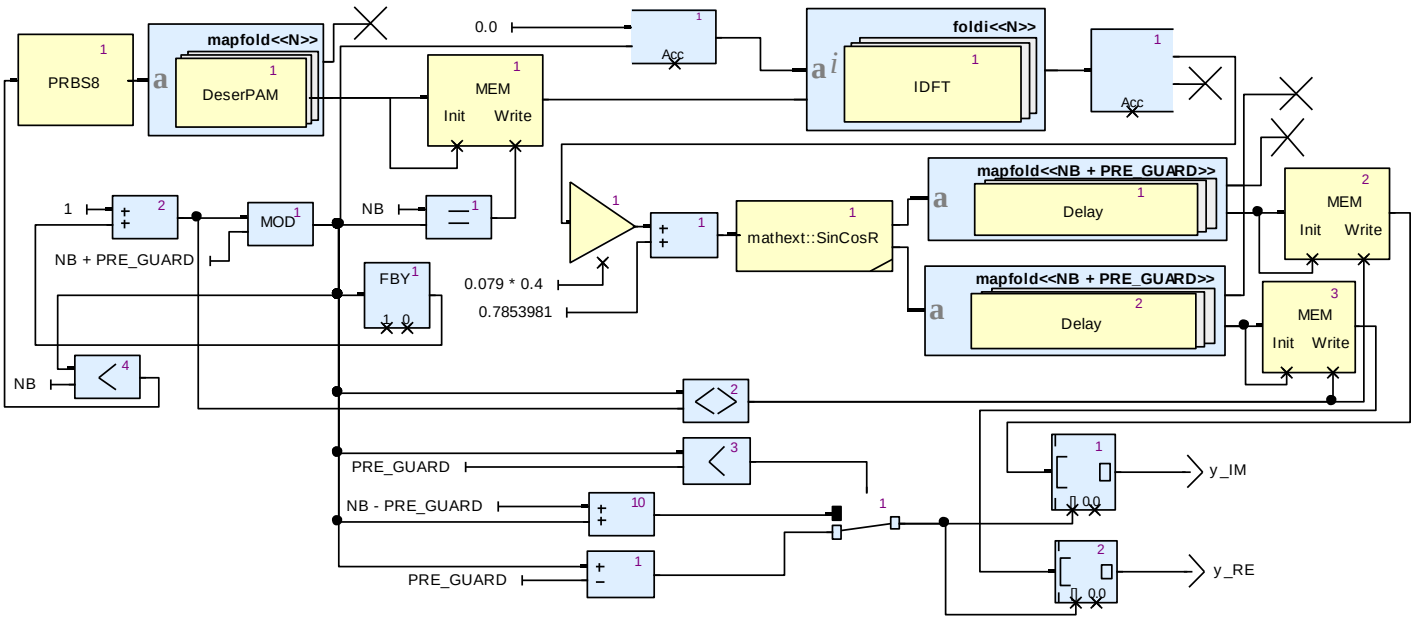


Рис.2.28. SCADE модель CE-OFDM

2.3.5. Нечеткое управление

Рассмотрим модель устройства с нечетким управлением (с fuzzy - регулятором) на простом примере из мобильной робототехники. Предположим, наша цель разработать управление для мобильного робота (рис. 2.29), использующего простой алгоритм. Робот оборудован дальномером, информирующем о расстоянии *d* до стены, вдоль которой он должен постоянно двигаться, удерживая заданное расстояние. На первом шаге создания модели регулятора необходимо определить входы и выходы, т.е. лингвистические переменные. В нашем случае это один вход – расстояние до стены *d* и один выход – угол поворота рулевого управления *a*.

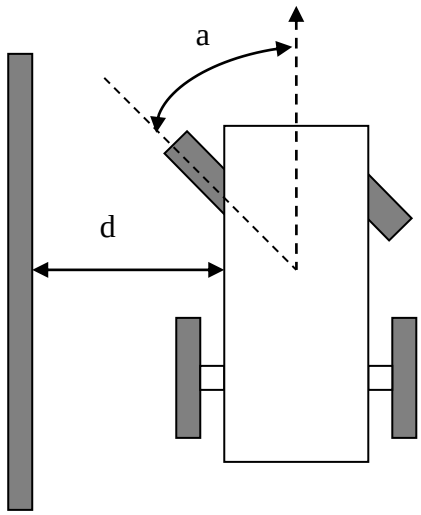


Рис.2.29. Мобильный робот

Для релейного регулятора можно установить следующие лингвистические правила:

• Если расстояние до стены меньше 50 см, то повернуть вправо на -10° (вправо).

• Если расстояние до стены больше 50 см, то повернуть на +10° (влево).

• Если расстояние до стены 50 см, то двигаться прямо.

На рис. 2.30 приведены области рассуждения для расстояния *d* и угла поворота *a* релейного регулятора.

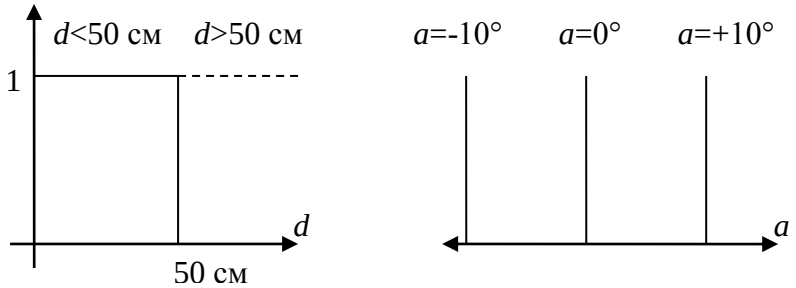


Рис.2.30. Области рассуждения для расстояния *d* и угла поворота *a* релейного регулятора

Так как значения находятся в двоичном домене, функция принадлежности является четкой. Алгоритм работы примитивен и робот будет неминуемо совершать рысканье.

Для нечеткого регулятора определим следующие лингвистические правила:

• Если расстояние до стены меньше 50 см, то повернуть направо (*a* отрицательное).

• Если расстояние до стены больше 50 см, то повернуть налево (*a* положительное).

• Если расстояние до стены около 50 см, то двигаться прямо (*a* около 0).

В соответствии с этими правилами определим три лингвистических значения для лингвистической переменой «расстояние»: «далеко» (больше 50 см), «около» (около 50 см, оптимальное расстояние) и «близко» (меньше 50 см). Для лингвистической переменой «угол поворота» выберем три лингвистических значения: «вправо» для отрицательных углов, 0 для прямолинейного движения и «влево» для положительных углов. С таким регулятором можно ожидать более «интеллектуальное» поведение. На рис. 2.31 приведены области рассуждения для расстояния *d* и угла поворота *a* нечеткого регулятора.

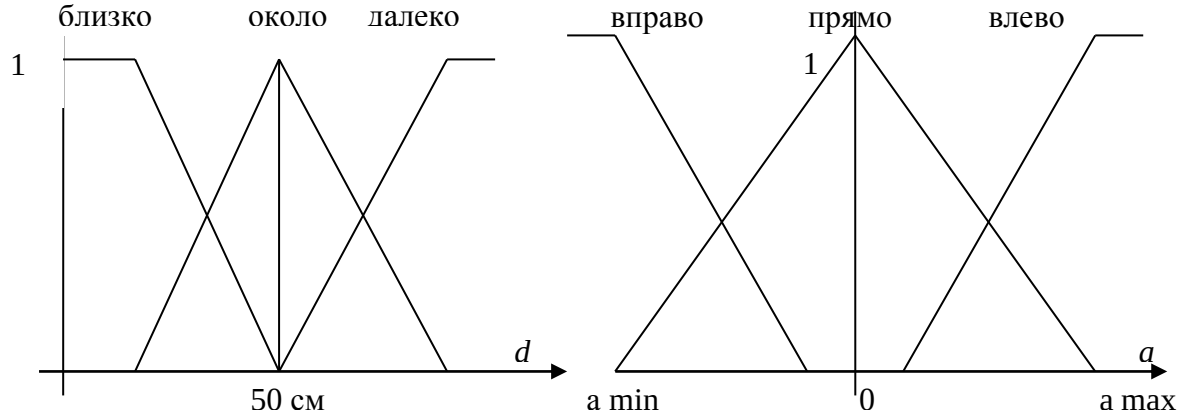


Рис.2.31. Области рассуждения для расстояния *d* и угла поворота *a* нечеткого регулятора

На рис. 2.32 приведен результат фузификации входной переменной *d*.

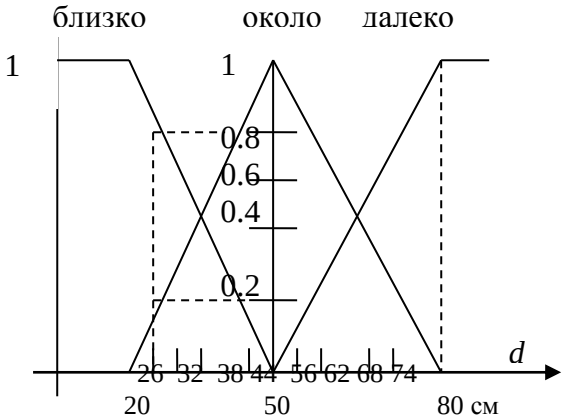


Рис.2.32. Фузификации входной переменной *d*

Предположим, что дальномер дает расстояние 26 см (рис. 2.32). Это расстояние принадлежит нечеткому множеству «близко» с вероятностью 0.8 (80%), а нечеткому множеству «около» с вероятностью 0.2 (20%). Пер-

вое правило удовлетворяется на 80%, третье на 20%, а второе ложно. Следовательно, робот должен повернуть направо на 80%, а двигаться прямо на 20%. Это означает, что необходимо «модулировать» соответствующие множества переменной «угол поворота» этими двумя величинами. Наиболее используемыми методами являются «линейная модуляция» (рис. 2.33) и «модуляция отсечением».

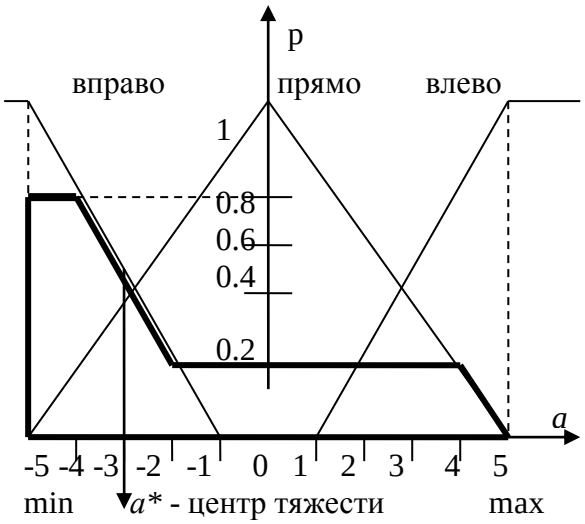
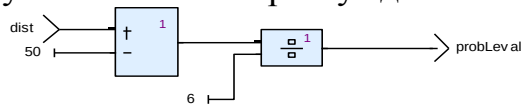


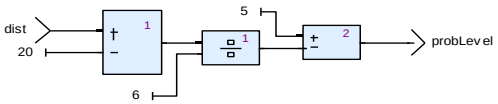
Рис.2.33. Нахождения четкого значения угла *a**

На рис. 2.32. показан также способ нахождения четкого значения угла *a** с помощью определения «центра тяжести» модулированной области (выделена жирными линиями) : $a^* = Sum(i*a_i)/ Sum(a_i)$. Рулевое управление может принимать только 11 значений, обозначенные от -5 до +5. Поэтому в процессе фузификации, рассмотренном выше, приняло участие лишь 11 значений расстояния в диапазоне от 20 до 80 см с шагом 6 см (шаг дальномера равен 1 см).

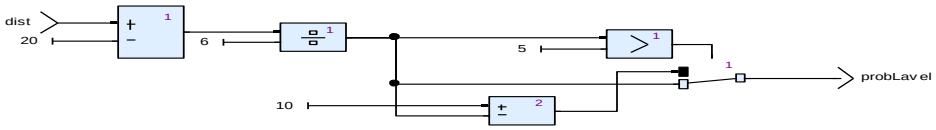
Рассмотрим реализацию модели этого нечеткого контроллера в SCADE suite. Процесс фузификации представим с помощью трех ператоров *fuzzificationFar, fuzzificationNear и fuzzificationNorm* соответствующих лингвистическим значениям «далеко», «близко» и «около» лингвистической переменой «расстояние» (рис 2.34). Каждый из операторов ставит в соответствие значения расстояния *dist* их вероятностям *problevel* для своего участка области рассуждения.



а) fuzzificationFar



б) fuzzificationNear



в) fuzzificationNorm

Рис.2.34 Операторы процесса фузификации

Нахождение центра тяжести реализуются с помощью пар итераторов

NomGravityFar/DenomGravityFar,NomGravityNear/DenomGravityNear (рис. 2.35, рис. 2.36, рис. 2.37, рис. 2.38) . Каждая пара соответствует паре итераторов числитель/знаменатель выражения для вычисления «центра тяжести» соответствующих модулированных областей переменной «угол поворота».

На рис. 2.39 приведена SCADE-модель нечеткого регулятора дистанции до левой стены мобильного робота. В процессе фузификации значений входа *dist* задействованы операторы *fuzzificationFar, fuzzificationNear и fuzzificationNorm* и три конструктора массива. В процессе дефузификации (модуляция переменной «угол поворота» и вычисление «центра тяжести» модулированной области) задействованы четыре итератора *foldi*, сгруппированные попарно. На выходе *angle* они формируют индексы значений угла поворота от -5 до +5 .

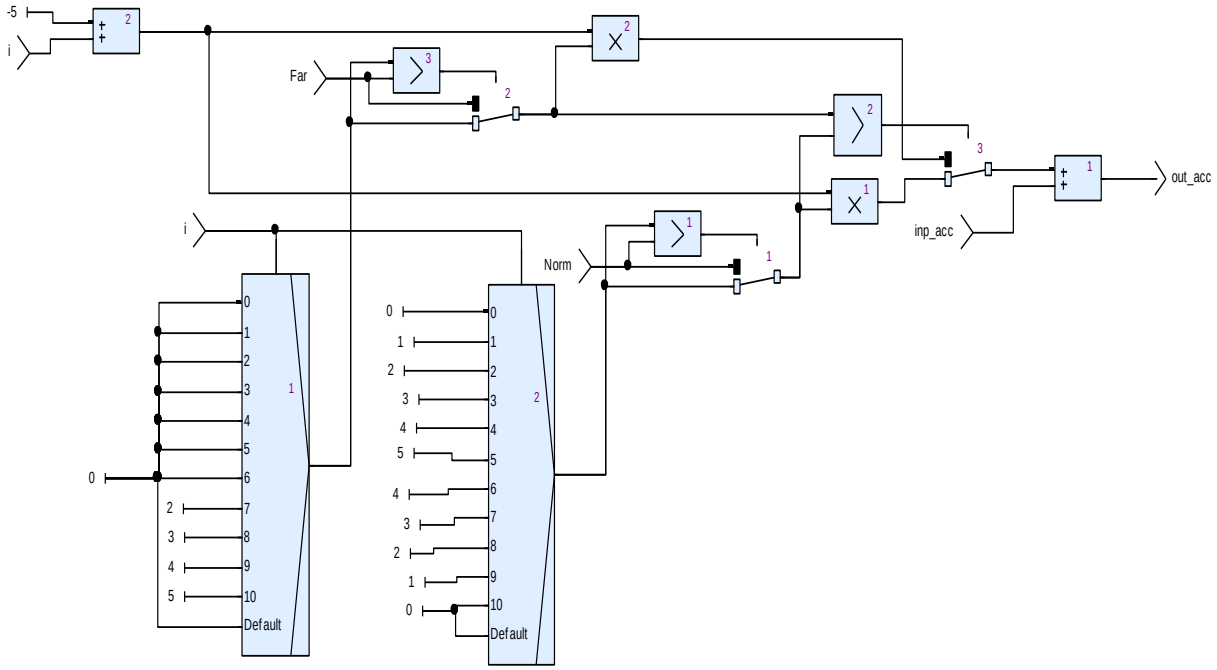


Рис.2.35. Оператор NomGravityFar

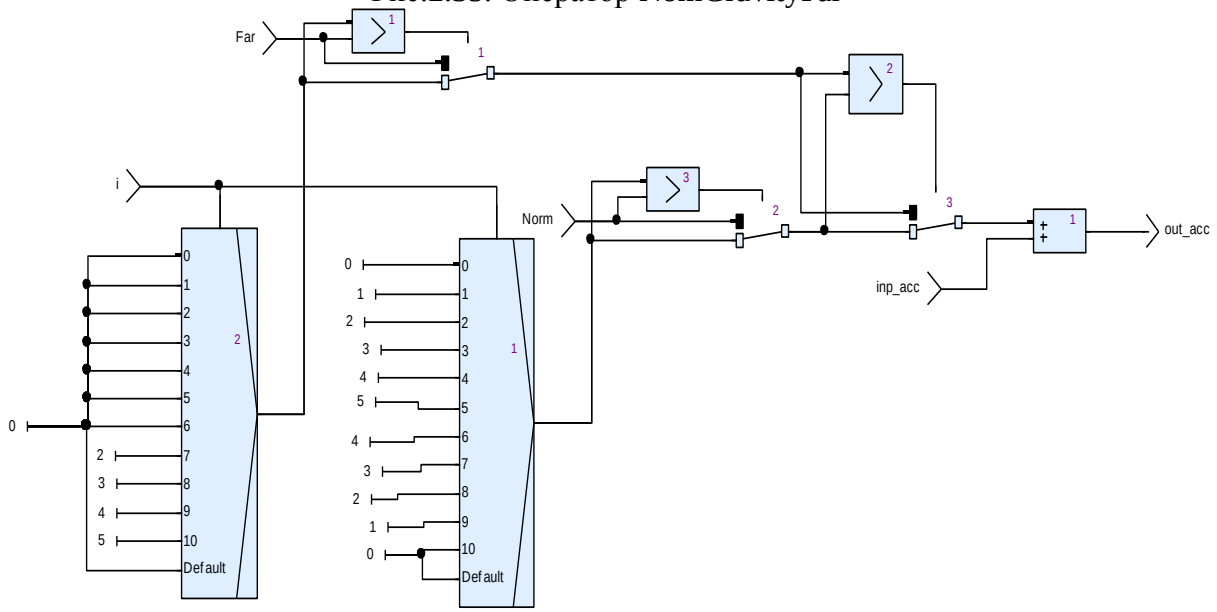


Рис.2.36. Оператор DenomGravityFar

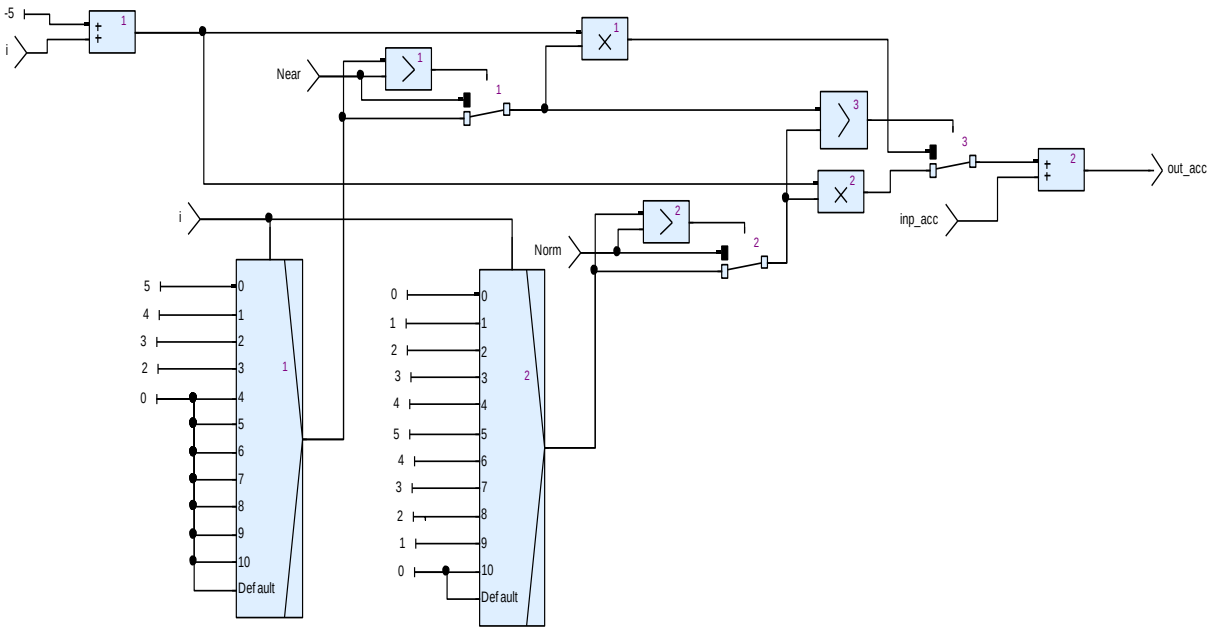


Рис.2.37. Оператор NomGravityNear

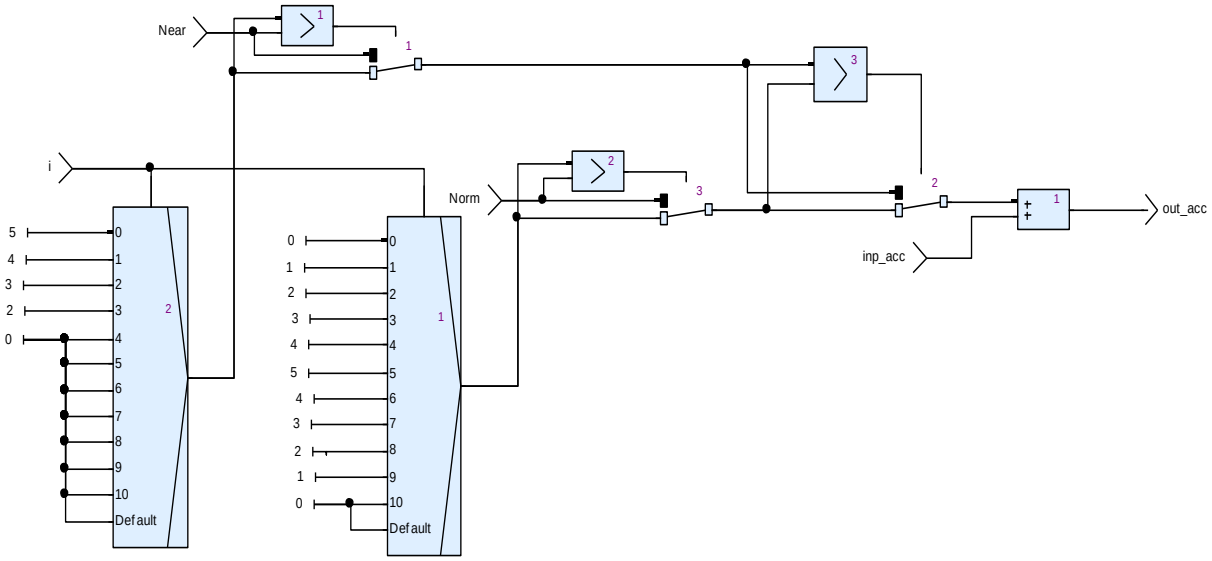


Рис.2.38. Оператор DenomGravityNear

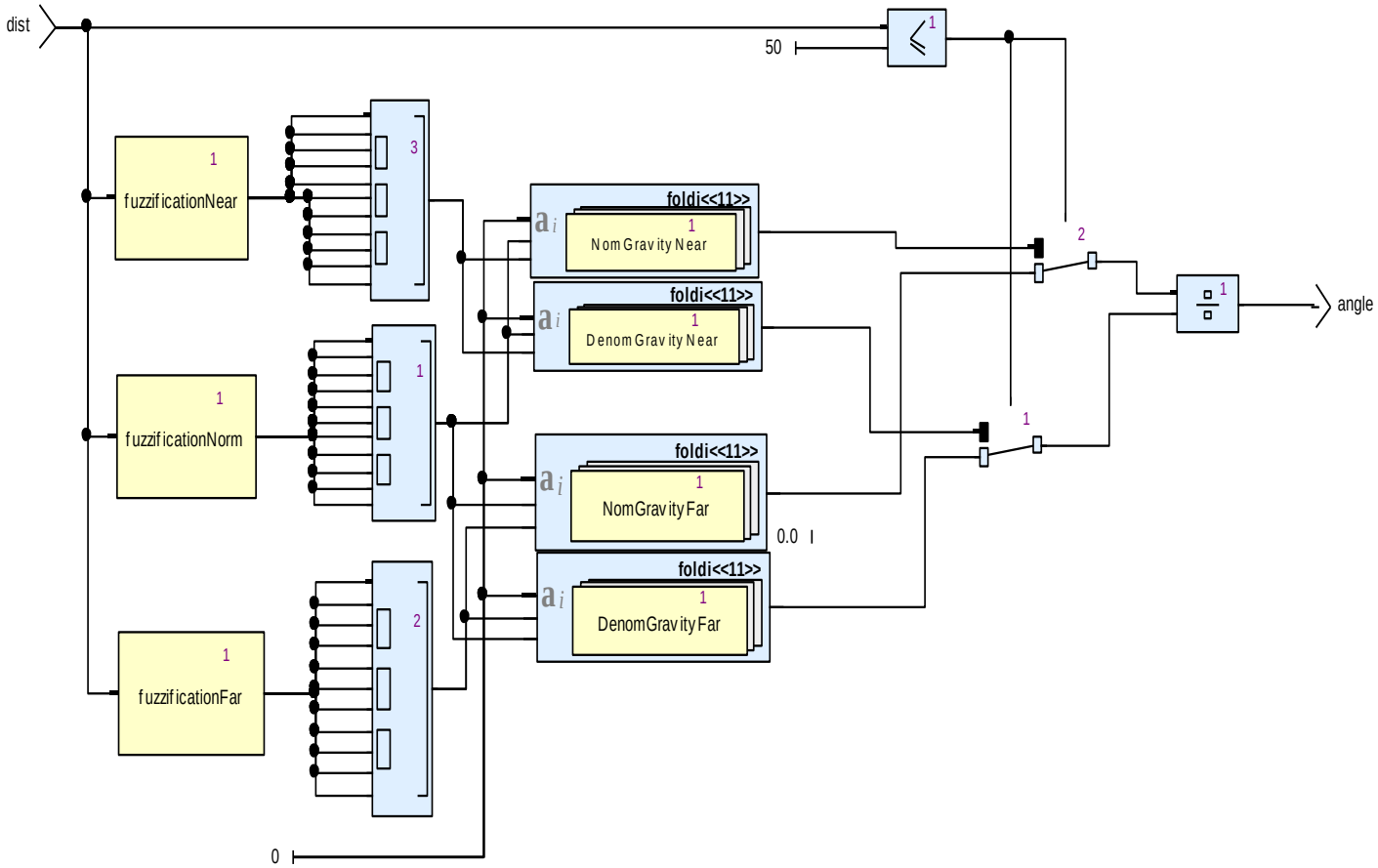
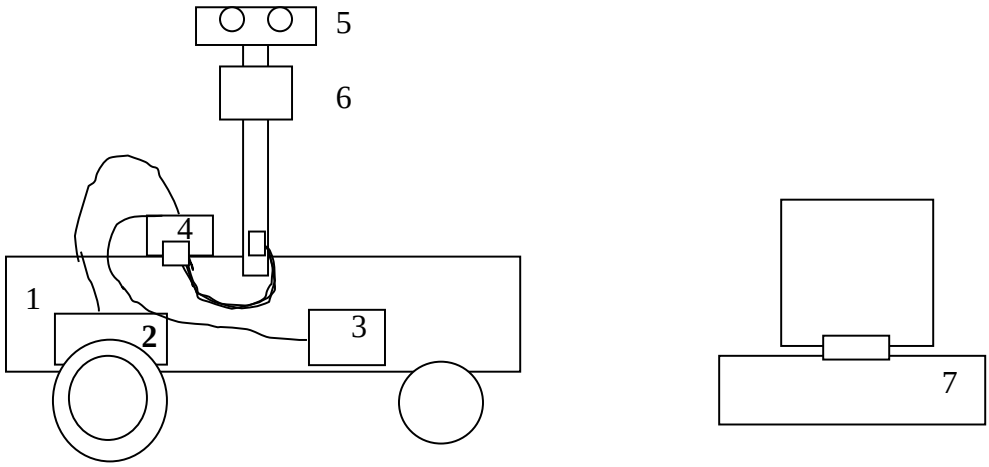


Рис.2.39. Модель нечеткого контроллера мобильного робота

2.3.6. SCADE-модель лабораторного прототипа Робота Исследователя Лабиринта

На рисунке 2.40 приведена схема Робота Исследователя Лабиринта (РИЛ). Подвижный объект РИЛ перемещается в пространстве под управлением программы на персональном компьютере (ПК) разработчика-оператора.



1 – шасси от радиоуправляемой модели стадиум-трака (гоночный машина для автодрома) десятого масштаба;
2 – задний ведущий электропривод постоянного тока модели;
3 – электрический сервопривод с металлическим редуктором для рулевого управления модели;

4 – микроконтроллер AT90CAN128 в качестве УСО (Устройство Связи с Объектом) с Bluetooth модемом;
5 – инфракрасный дальномер (измерение расстояния от 10 до 90 см);
6 – сервопривод дальномера;
7 – персональный компьютер разработчика-оператора с Bluetooth модемом.

Рис.2.40. Схема РИЛ

Запросы по управления «движение вперед», «движение назад», «стоп», «поворот влево», «поворот вправо» и «прямо» передаются от оператора к РИЛ через интерфейс Bluetooth в ответ на полученную от робота измерительную информацию (расстояние до ближайшей стены) по запросу «выдать расстояние». В РИЛ используется один дальномер, который с помощью сервопривода может поворачиваться для измерения расстояния в требуемом направлении. Поэтому существует несколько запросов: «выдать расстояние в направлении слева», «выдать расстояние в направлении спереди», «выдать расстояние в направлении справа» На рис. 2.41 приведена SCADE модель РИЛ верхнего уровня иерархии (блок *MazeExplorar*). При создании SCADE модели важно найти баланс между абстрактным и конкретным представлением функционирования. Так в нашем случае можно создать детальную SCADE модель, отражающую байтную структуру протокола взаимодействия прикладного уровня РИЛ. Однако SCADE не располагает средствами манипуляции с байтами, как это есть в императивных языках. Придется работать со встроенным типом *bool*, представляющий один бит подобно типу *unsigned char* на Си, а это целый байт.

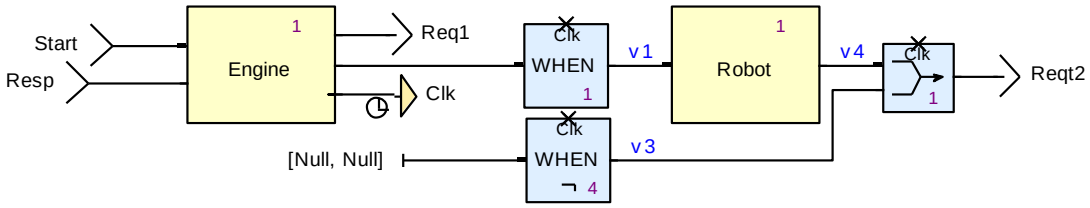


Рис.2.41. Блок *MazeExplorar* SCADE модели РИЛ верхнего уровня иерархии

Заменим детальные форматы запросов и ответов прикладного протокола их именами, т.е. введем перечислимый тип:

type *ReqType=* ***enum*** *{DistL, DistD, DistR, Dist45L, Null, Dir, Left, Right, Forward, Stop, Back};*

где *DistL, DistD, DistR, Dist45L* – группа запросов «выдать расстояние», *Left, Right, Forward, Stop, Back* – группа запросов по управления А жениием, а значение *Null* представляет отсутствие запроса (аналогично пустому символу в канале).

Блок *Engine* инициирует последовательность запросов на получение расстояния до ближайших стен в (рис. 2.42). *Engine* реализован как автомат.

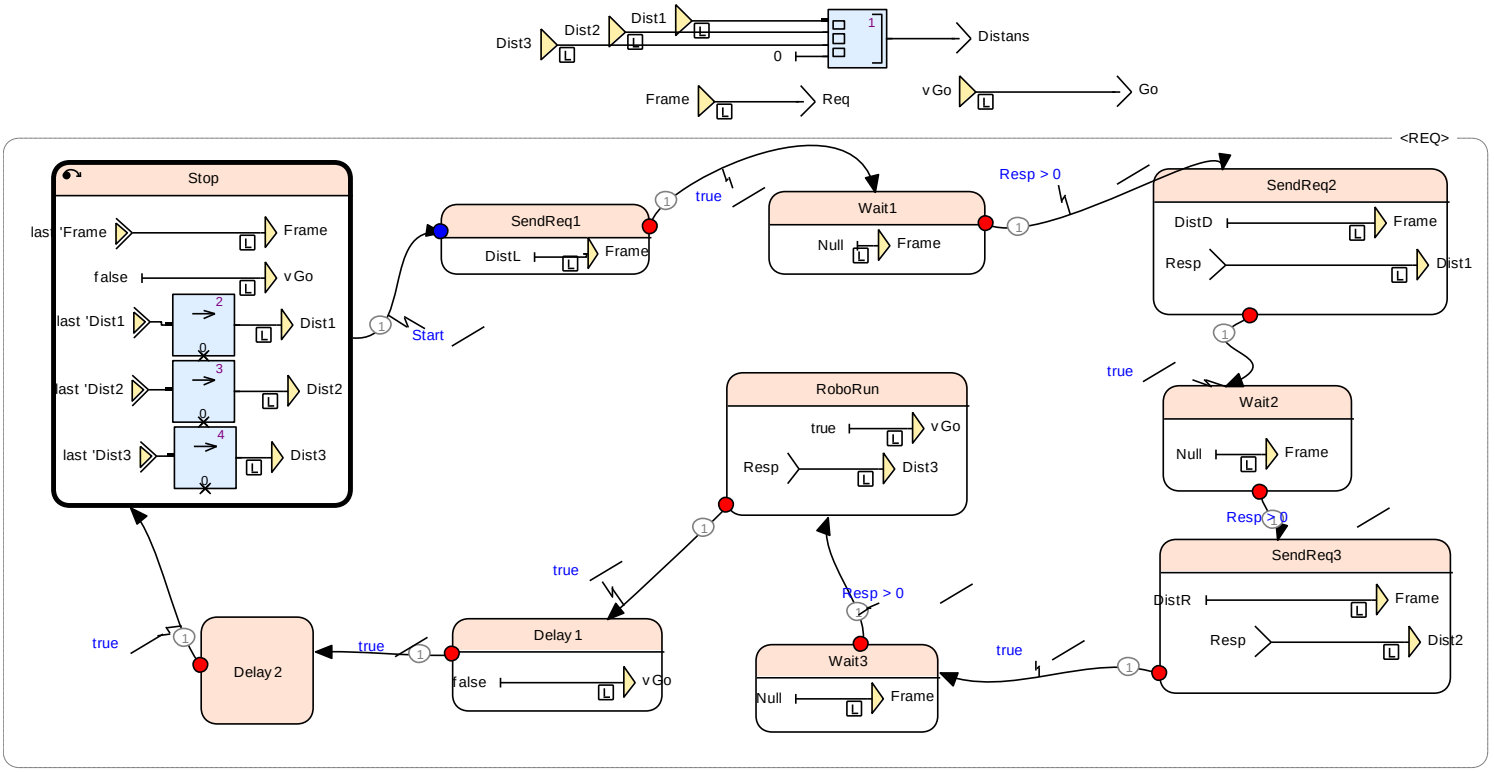


Рис.2.42. SCADE модель блока Engine РИЛ

Входной булев поток *Start* значением *true* запускает РИЛ. Переменная *Frame* типа *ReqType* принимает значения запросов расстояния в состояниях *SendReq1*, *SendReq2*, *SendReq3* по соответствующим направлениям. Поток *Frame* подключается к выходному потоку *Req*. В состояниях *Wait1*, *Wait2*, *Wait3* *Frame* присваивается значение *Null*. В состояниях *SendReq2*, *SendReq3*и *RobotRun* переменные Dist1, Dist2 и Dist3 запоминают полученную от РИЛ измерительную информацию, поступающую через входной поток *Resp*. Значения отличные от 0 в этом потоке соответствуют ответам РИЛ на запрос расстояния. В состоянии *RobotRun* (достигается при получении последнего ответа на запрос расстояния) переменная *vGo* принимает значение *true* (в следующем состоянии Delay1 устанавливается в false), которое через выходной поток *Go* запускает работу блока *Robot* (рис. 2.43), реализующего алгоритм поиска и удержания стены.

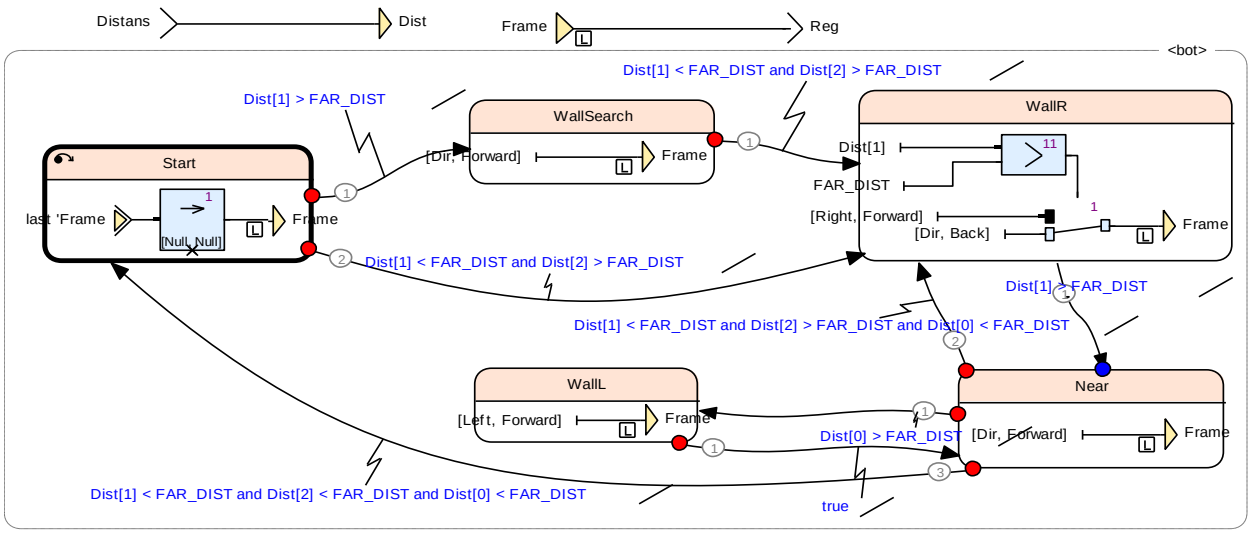


Рис. 2.43. SCADE модель блока Robot РИЛ

Условиями переходов в автомате блока *Robot* являются значения расстояний по различным направлениям, поступающим через входной поток *Distans*. В состояниях формируются значения для выходного потока запросов *Req* по управления движение РИЛ.

Тактирование потоков блока *Robot* ниже базового и определяется значением *true* в выходном потоке *Go* блока *Engine*. Поэтому в SCADE модель верхнего уровня (блок *MazeExplorar*) введены временные операторы *when* для децимации потока с базовой тактировкой и *merge* для интерполяции потока с более низкой тактировкой до базовой. Переменная *Clk* выполняет роль тактирования в блоке *Robot*.

2.4. Верификация

Разработчики проектов в SCADE Suite для проверки их корректности могут воспользоваться инструментом формальной верификации «Верификатор Проекта» (ВП, Design Verifier) [22]. Верификация отвечает на вопрос о том ведет ли себя проект в соответствии с заданными требованиям в отличии от валидации, которая отвечает на вопрос о том ведет ли проект себя в соответствии с требованиями для заданных условий применения.

Традиционная верификация тестированием предполагает, что разработчик пишет набор испытательных программ (test suite или контрольные примеры – test cases) чтобы показать, что система (проект, модель) удовлетворяет данному свойству. Ограничения времени не позволяет достичь полноты, т.к. написание контрольных примеров для каждой комбинации возможных входных значений из реальной жизни, приведет к комбинаторному «взрыву». Поэтому выбираются различные стратегии для ограничения числа контрольных примеров, но вместе с этим теряется полнота. Традиционно это такие стратегии как: подвергать тестированию такие части системы, где наиболее вероятны ошибки или они не требуют больших временных затрат; подвергать тестированию лишь нормальное поведение системы. В обоих случаях целью испытательных программ является нахождение ошибок, которые воображает себе разработчик и не направлено на обнаружение местоположения реальных ошибок.

В отличие от верификации тестированием ВП не требует написания тестов, а выполняет исчерпывающий поиск подтверждений соответствия проекта заданным свойствам. ВП верифицирует свойства, выраженные в виде булевых выражений содержащих условия над булевыми переменными, числовыми переменными и временными задержками.

Рассмотрим проект SCADE Suite *RollControl*, который вычисляет величину крена самолета и формирует тревожные предупреждения. Необходимо выяснить соответствует ли этот проект следующему свойству: тревожные предупреждения о правом и левом крене никогда не возникают одновременно. Это свойство может быть графически представлено «оператором свойства» (property operator) *Proof* как на рис. 2.44.

Оператор *HasNeverBeenTrue* («Никогда не было истиной») принадлежит Библиотеке Верификации (Verification Library):

node *HasNeverBeenTrue (input1:****bool****)* ***returns****(ounput1: **bool**);*
let
*ounput1= **not** input1–>(**not** input1**and** **pre**(input1));*
tel.

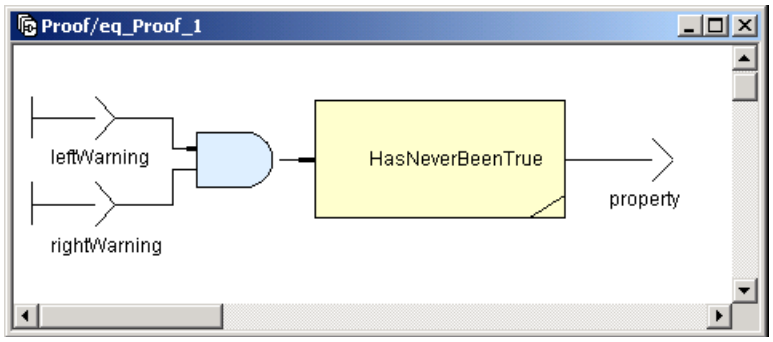


Рис. 2.44. Оператор свойства

leftWarning и *rightWarning* булевы выходы проекта *RollControl* необходимые для анализа свойства.

Соединив оператор *Proof* с проектом *RollControl*, получим новый оператор (рис. 2.45) называемый «оператором наблюдения» (observer operator). Проверка свойства состоит в получении математического доказательства из оператора наблюдения, показывающего, что не имеет значение ковы входы системы, значение булева выхода оператора наблюдения всегда истинно. Если ВП находит доказательство свойства, то разработчик получает высокую степень уверенности в том, что свойство хорошо сформулировано и удовлетворяется. В противном случае, если существует контр пример (т.е. есть значения системных входов, которые фальсифицирует свойство), ВП найдет это и сообщит о свойстве как о фальсификации. Такой контр пример можно воспроизвести Симулятором (Simulator) SCADE Suite для помощи в решении проблемы.

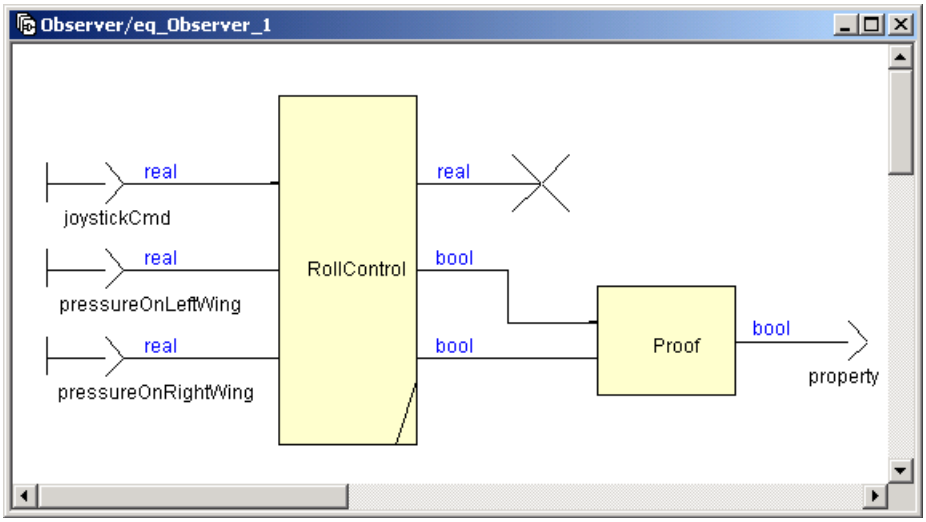


Рис. 2.45. Оператор наблюдения

Библиотека Верификации содержит следующие операторы, раскрытые для понимания в виде реализаций:

• Выход *Ounput1* оператора *AfterNthTick* («После N-го такта» рис. 2.46) равен входу за исключением первых *N* тактов, во время которых выход равен *true* (т.е. первые *N* тактов вход игнорируется).

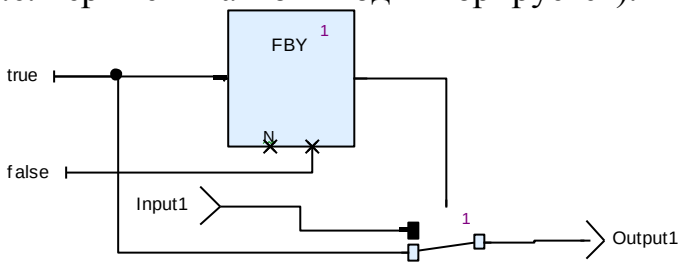


Рис. 2.46. Оператор *AfterNthTick*

• Выход *Ounput1* оператора *HasNeverBeenTrue* (рис. 2.47) становится *false*, как только вход *Input1* становится *true* и после этого навсегда остается *false*.

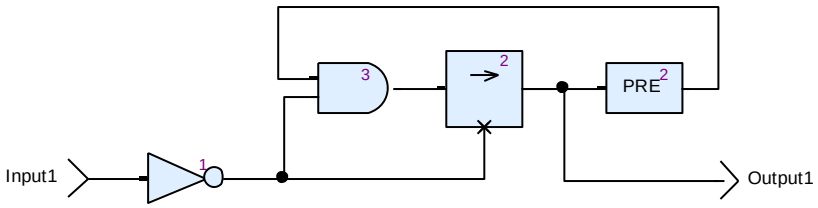


Рис. 2.47. Оператор *HasNeverBeenTrue*

• Выходной поток *Ounput1* оператора *AlwaysAfterFirstCond* («Всегда после первого условия», рис. 2.48) равен входу *Input1*, как только условие Cond станет *true*.

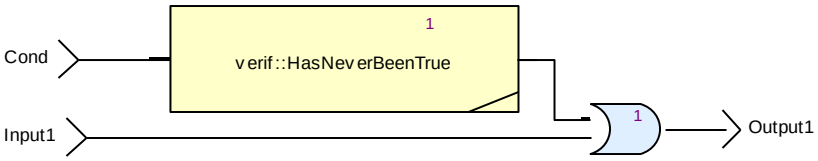


Рис. 2.48. Оператор *AlwaysAfterFirstCond*

• Выходной поток *Ounput1* оператора AtLeastNTick («Не меньше N тактов», рис. 2.49) становится равным входному потоку *Input1* как только он остается *true N* тактов подряд. Перед тактом *N* выход равен *false*.

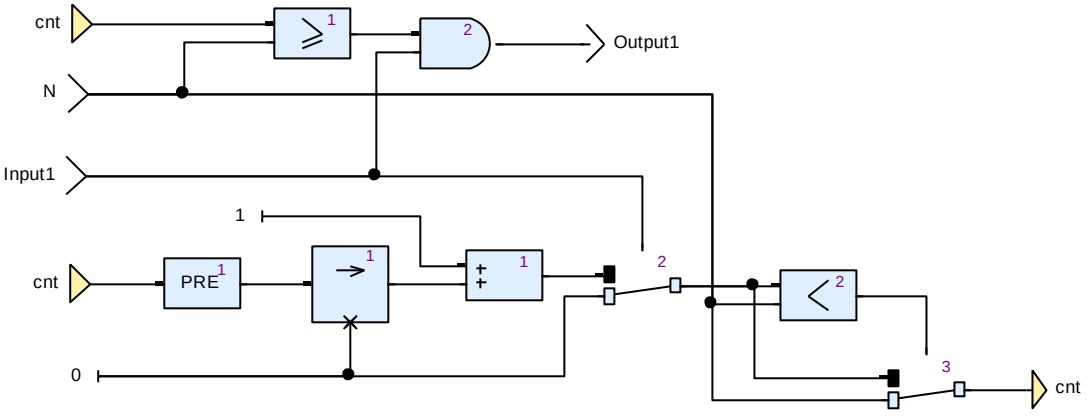


Рис. 2.49. Оператор AtLeastNTick

• Оператора *Implies* реализует логический оператор импликация (рис. 2.50): если A *true* то B *true.*

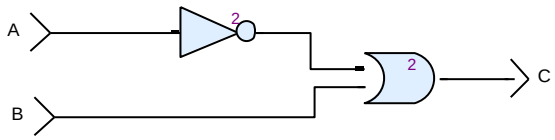


Рис. 2.50. Оператор *Implies*

• Выходной поток *C* оператора ImpliesWithinNTick («Импликация внутри *N* тактов» », рис. 2.51) становится равным «*A Implies B*» спустя *N* тактов, в течение которых вход *A* оставался *true.*

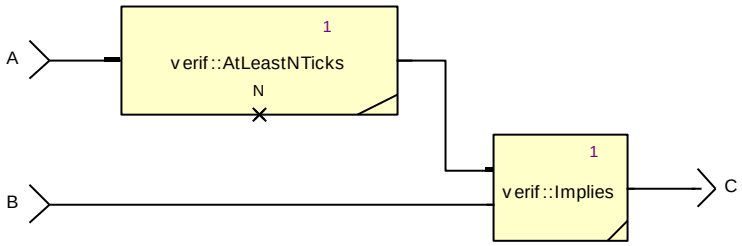
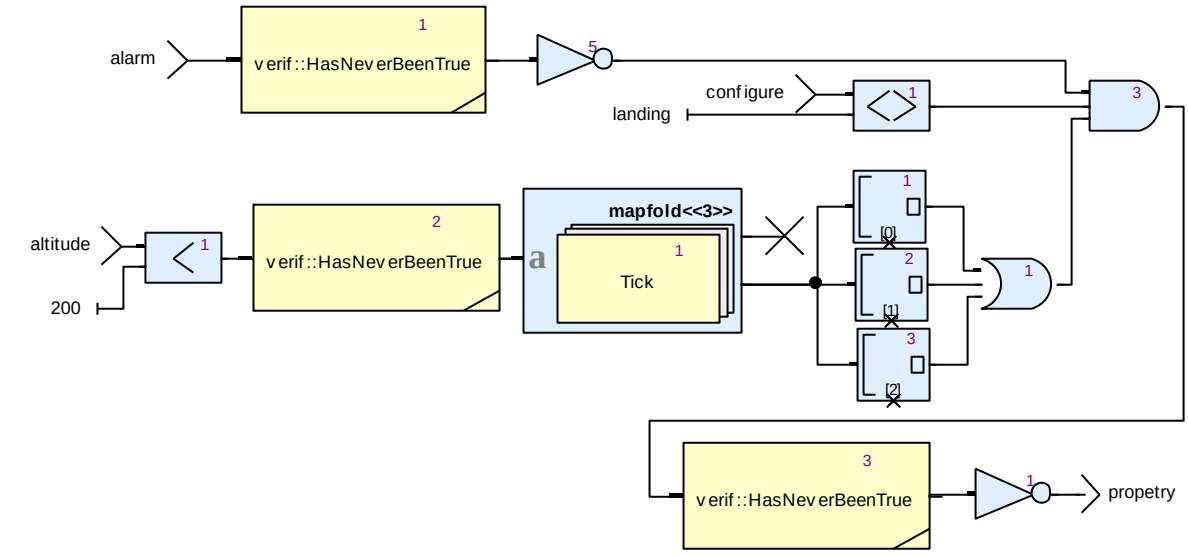


Рис.2.51. Оператор *ImpliesWithinNTick*

Примеры свойств требующих темпоральной логики в Scade:

• Тревожное сообщение (*alarm*) возникает не позже трех тиков дискретного времени после того как высотомер самолета дает показание (*altitude*) меньше 200 м при этом самолет не находится в режиме (*configure*) приземления (*landing*) (рис. 2.52).



Оператор *Tick*

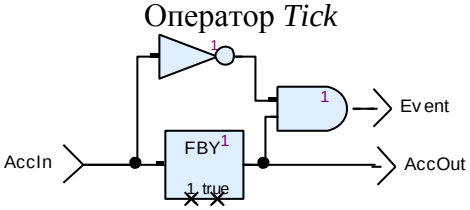


Рис. 2.52. Пример свойства тревожного сообщения о снижении

• Если пилот не сбросит сообщение, оно сохраняется по крайней мере 10 тиков после того как высотомер самолета дает показание меньше 200 м при этом самолет не находится режиме посадки.

При разработке ПО часто происходит повторное появление одних и тех же ошибок. После следующего изменения в программе решение перестаёт работать; при переписывании какой-либо части кода всплывают те же ошибки, что были в предыдущей реализации. Иногда это происходит из-за слабой техники управления версиями или по причине человеческой ошибки при работе с системой управления версиями. Ошибки — когда после внесения изменений в программу перестает работать то, что должно было продолжать работать, — называют регрессионными ошибками (regression bugs). Техника регрессионного тестирования направлена на обнаружение ошибок в уже протестированных участках исходного кода.

ВП может быть также использован для обнаружения регрессионных ошибками с помощью не регрессионного доказательства (доказательство или опровержение свойства эквивалентности двух моделей). Для этого просто создается новый оператор с экземпляром старой модели и оператором сравнения выходов как на рис. 2.53.

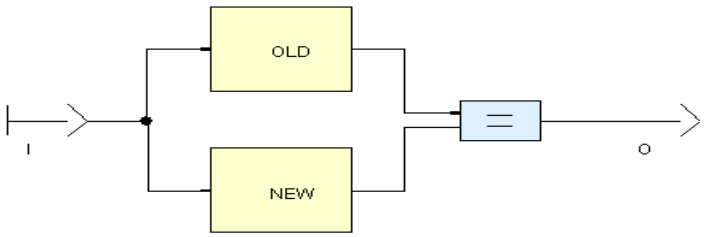


Рис.2.53. Принцип проверки эквивалентности проектов SCADE Suite

Этот же подход может быть использован для анализа состояния деградации (degraded mode analysis). Просто оператор *OLD* заменяется на оператор, моделирующий поведение в состоянии деградации.

2.5 Разработка прикладного программного обеспечения управляющей системы реального времени

Рассматриваемые в разделе модели применяются также и при разработке аппаратного обеспечения. «Водопад» (Waterfall) является классической моделью жизненного цикла программного обеспечения, которая восходит к 80-м годам прошлого века. Она представляет проектирование программного обеспечения в терминах процесс (process) и продукт (product). Каждый процесс из входного продукта производит новый выходной продукт. Процессы является итеративными. На рис. 2.53 приведена модель жизненного цикла программного обеспечения «Водопад». Модель хорошо работает, если разработчики делают мало ошибок.

V-модель является модификацией модели «Водопад» (рис. 2.54). Она позволяет использовать модульность подсистем. Под модульностью пони-

мают организацию программы в виде написанных по определенным правилам взаимодействующих частей – модулей.

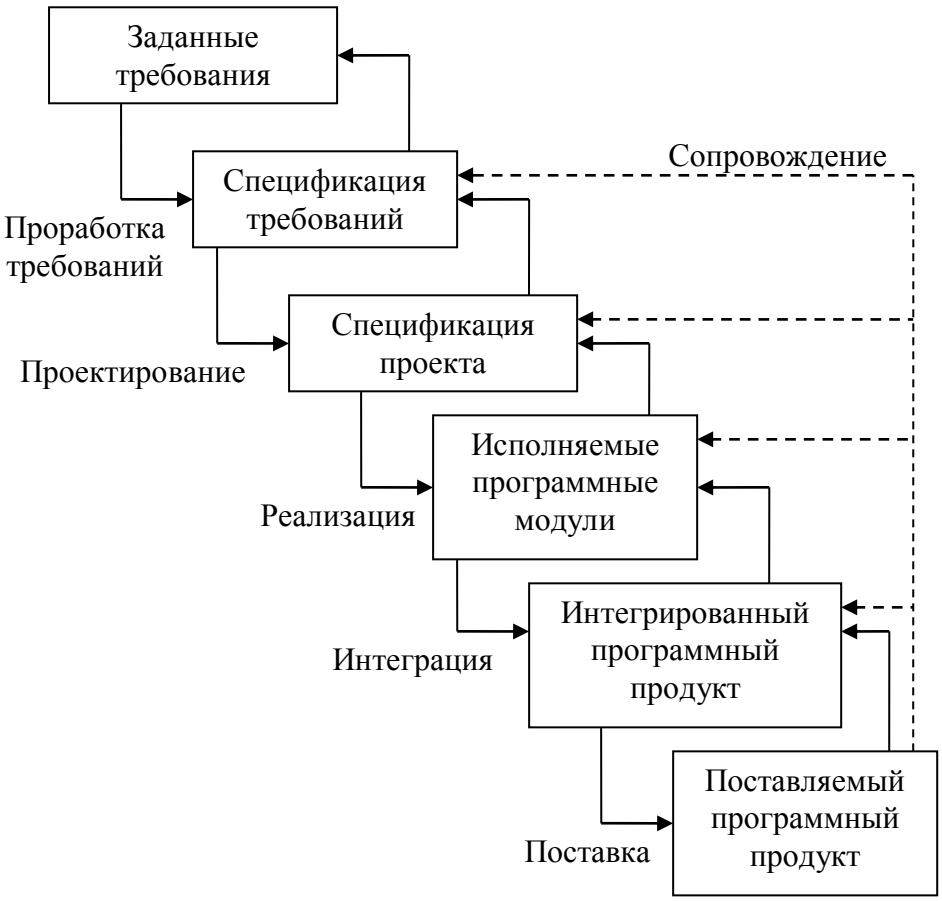


Рис. 2.53. Модель жизненного цикла программного обеспечения «Водопад»

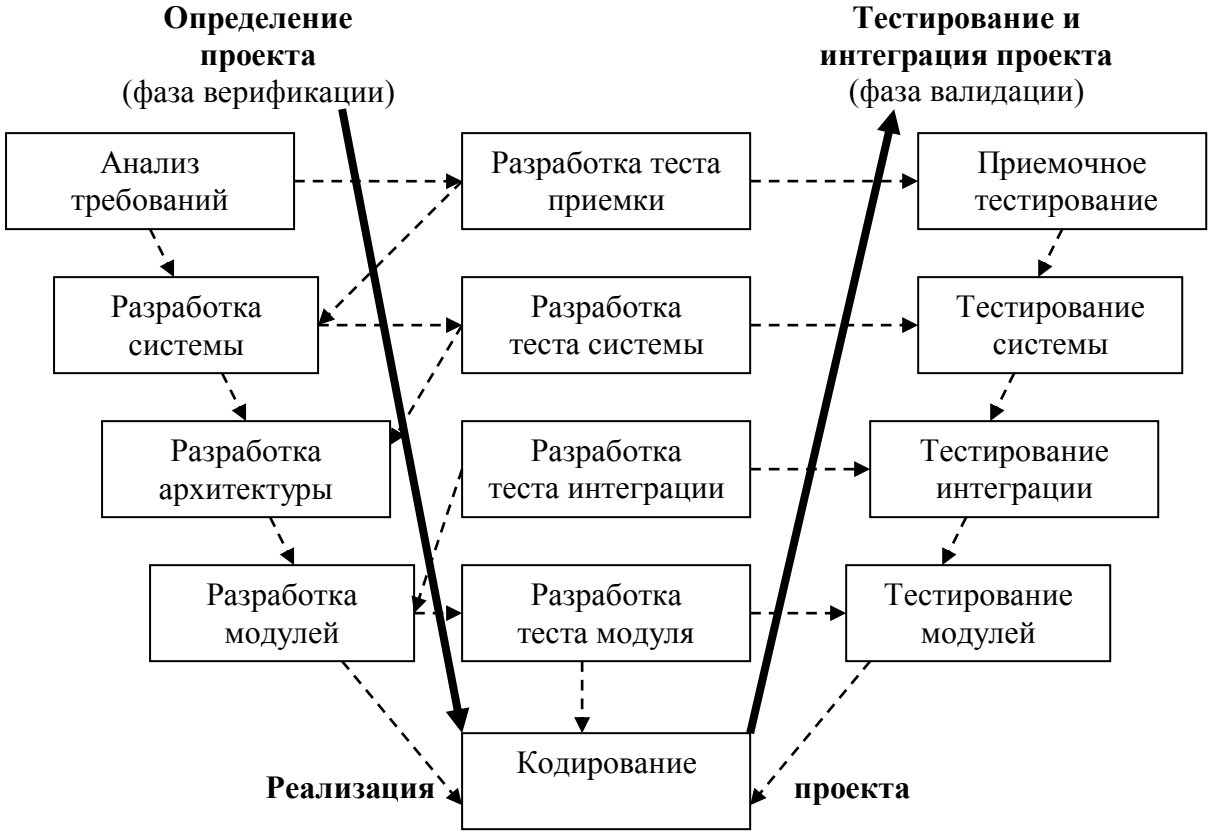


Рис. 2.54. V-модель жизненного цикла программного обеспечения

V-модель регулирует «кто», «когда» и «что» в опытно-конструкторской работе (ОКР). Модель является стандартом IT систем Германии для гражданских и военных областей. Основу модели составляет иерархическая декомпозиция системы на меньшие части, пока не станет возможной их реализация. Верификация и валидация выполняются на каждой стадии. Нет строго оговоренного временного порядка.

Стандарт DO-178B является рекомендацией по производству программного обеспечения для бортовых систем и оборудования. В нем выделяются следующие уровни безопасности разработки:

- A – состояние катастрофического повреждения самолета (разрушение).
- B – состояние опасного повреждения самолета (травмированные персоны).
- C – состояние серьезного повреждения самолета (вышла из строя система управления полетом, переход на ручное управление).
- D – состояние несущественного повреждения самолета (вышла из строя система связи с землей).
- E – нет влияния на функционирование самолета или высокая загруженность пилота в полете (вышла из строя система развлечений).

Стандарт говорит, что сутью является формулировка требований и верификация того, что требования были достигнуты, а цель – обнаружение ошибок, которые были внесены в процессе разработки программного обеспечения. Пути достижения требования могут быть разными. Важно чтобы все требования поддавались проверке и соответствовали требованиям других этапов проектирования. Тестирование является лишь частью процесса верификации, необходимы также экспертиза и анализ. На рис. 2.55 представлен процесс проектирования согласно DO-178B.

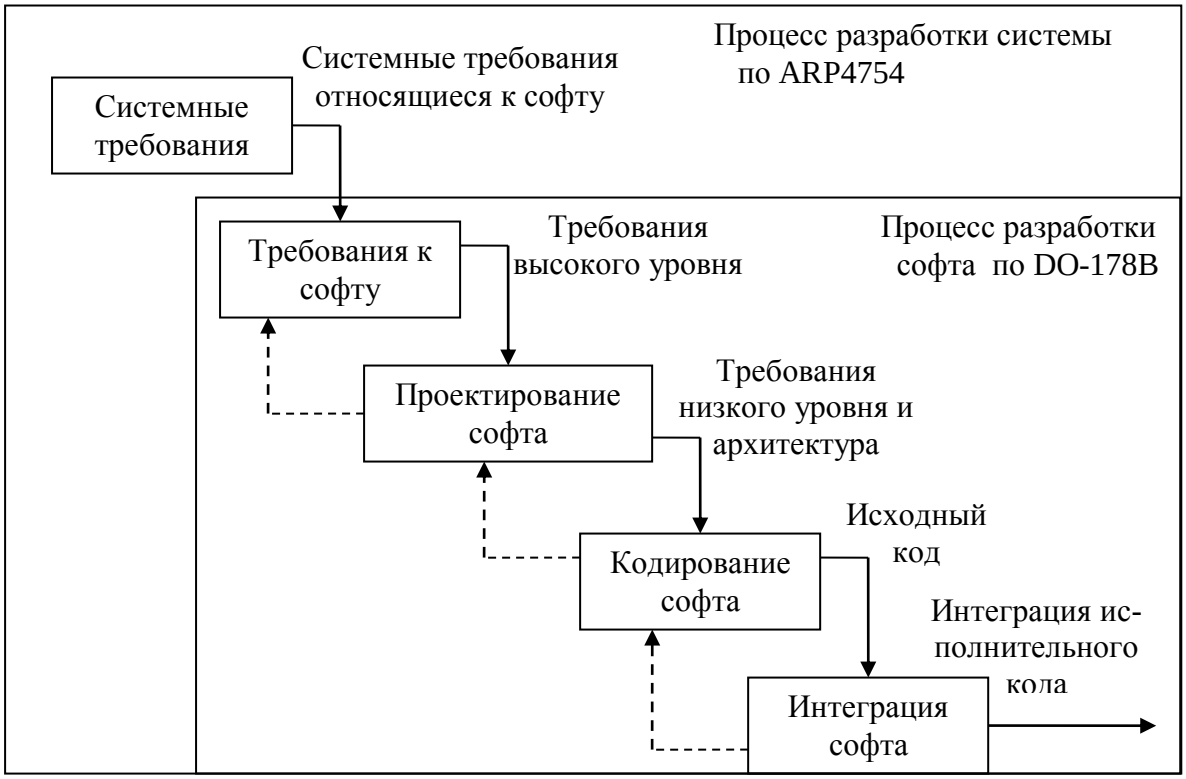


Рис. 2.55. Процесс проектирования в соответствии DO-178B

В [23] используется архитектурный подход к встроенным системам. Он является мощным инструментом для понимания и разработки встроенных систем и решения проблем, возникающих при разработке новых систем. На рис. 2.56. представлена модель жизненного цикла, основанная на архитектуре встроенной системы, в которой основополагающим является этап «Создание архитектуры».

Рис. 2.56. Модель Ноергарда (Noergaard)

Под архитектурой встроенной системы автор понимает абстракцию встроенной системы, в которой не показаны детали реализации, такие как исходный код софта, электрическая принципиальная схема устройства. Вместо этого на уровне архитектуры компоненты аппаратуры и софта представлены как некоторая композиция взаимодействующих элементов. Эти элементы представляются поведенческими абстракциями и интерфейсной информацией. Элементы архитектуры могут быть как внутренне интегрированными, так и существовать вне системы. Таким образом,

встроенная архитектура включает элементы встроенной системы, элементы взаимодействующие со встроенной системой и связи между ними. Информация на уровне архитектуры представляется в виде структуры. Структура одно из возможных представлений архитектуры содержащее множество элементов, свойств и информацию о взаимосвязях. Она является краткой характеристикой (snapshot) аппаратуры и софта системы во время проектирования. Так как очень трудно одной краткой характеристикой охватить сложную систему, обычно это компенсируют несколькими структурами.

Все структуры архитектуры имеют внутренние отношения друг с другом, так что сумма всех структур это и есть архитектура встроенной системы. В таблице 2.4 приведены примеры архитектур.

Таблица 2.4. Примеры архитектур

<table>
<tr><th colspan="3">Тип архитектуры (структуры)</th><th>Описание</th></tr>
<tr><td rowspan="5">Модуль
Элементы определяются как различные функциональные компоненты необходимые для корректной работы системы. Архитектура предстает как модульная структура ОС, процессора, JVM и т.д.</td><td rowspan="4">Использование
Структура представляет взаимовлияние модулей в режиме выполнения (например, как модули используют друг друга)</td><td>Уровни</td><td>Структура, в которой модули распределены по уровням (иерархия). Модули верхних уровней используют модули нижних уровней.</td></tr>
<tr><td>Ядро</td><td>Структура представляет модули, использующие модули (сервисы) ядра ОС или управляемые ядром.</td></tr>
<tr><td>Канальная архитектура</td><td>Структура представляет модули последовательно, показывая преобразования модулей через их использование</td></tr>
<tr><td>Виртуальная машина</td><td>Структура представляет модули, которые используют модули виртуальной машины.</td></tr>
<tr><td>Декомпозиция</td><td colspan="2">Структура, в которой некоторые модули представляются композицией других модулей. Обычно используются для выделения ресурсов, управления проектом (планирование), управления данными (инкапсуляция, приватизации, …).</td></tr>
<tr><td>Класс</td><td colspan="2">Структура представления софта, в которой модули это классы объектно-ориентированного программирования, а связи - отношения наследования.</td></tr>
<tr><td rowspan="2">Компонент и соединительное звено
Структура образуется из компонентов (хард/софт узлы обработки, такие как процессор, JVM) или соединительных звеньев (коммуникационные ме-</td><td>Клиент/ Сервер</td><td colspan="2">Структура системы в режиме выполнения. Компонентами являются клиенты или серверы, а соединительные звенья это механизмы (протоколы, сообщения, пакеты, …) используемые для взаимодействия между клиентом и сервером.</td></tr>
<tr><td>Процесс</td><td colspan="2">Структура системы с ОС. Компонентами являются процессы и/или потоки, а соедини-</td></tr>
</table>

<table>
<tr><td rowspan="6">ханизмы, которые связывают компоненты: аппаратные шины, сообщения ОС, …).</td><td></td><td colspan="2">тельными звеньями – механизмы взаимодействия между процессами (разделяемые данные, каналы. …). Структура полезна для анализа планирования и производительности.</td></tr>
<tr><td rowspan="2">Параллелизм и ресурсы
Структура системы с ОС в режиме выполнения. Компоненты взаимодействуют через параллельно выполняемые потоки. По существу эта структура используется для управления ресурсами и определения
есть ли проблемы с разделяемыми ресурсами, кокой софт может выполняться параллельно.</td><td>Прерывание</td><td>Структура представляет механизмы обработки прерывания в системе.</td></tr>
<tr><td>Планирование (EDF, приоритетность, циклический алгоритм диспетчеризации).</td><td>Структура представляет механизм планирования потоков задачи, показывая распределение ресурсов планировщиком ОС.</td></tr>
<tr><td rowspan="2">Память
Представление в режиме выполнения, содержащее компоненты памяти и данных со схемами выделения и освобождения (соединительными звенья) памяти – по сути, схема управления памятью системы.</td><td>Сборка мусора</td><td>Структура представляет схему выделения памяти под ненужные данные.</td></tr>
<tr><td>Выделение памяти</td><td>Структура представляет схему выделения памяти (статическая или динамическая, размер и т.д.)</td></tr>
<tr><td>Безопасность и надежность</td><td colspan="2">Представление системы в режиме выполнения, в котором избыточные компоненты и механизмы их взаимодействия демонстрируют надежность и безопасность системы при возникновении проблем (способность к восстановлению)</td></tr>
<tr><td>Распределение
Структура представляет взаимоотношения элементов аппаратуры и/или элементов софта и внеш-</td><td>Рабочее задание</td><td colspan="2">Структура определяет назначение ответственных за разработку и развитие модулей системы. Эта структура обычно используется в управлении проектом.</td></tr>
</table>

них элементов окружения.	**Реализация**	Структура софта, показывающая его размещение в файловой системе.
	Внедрение	Структура системы в режиме выполнения, где элементами являются аппаратура и софт, взаимоотношения между элементами это отображение софта на аппаратуру (постоянное хранение, перемещение и т.д).

Наиболее общими проблемами при разработке архитектуры являются:

• определение и сбор данных о системе;

• ограничения стоимости;

• установление свойств целостности системы, таких как надежность и безопасность;

• работа с пределами функциональности элементов.

Архитектура встроенных систем может быть использована для решения этих проблем на ранних стадиях проекта. Без определения или знания некоторых внутренних деталей реализации, архитектура устройства становится первым инструментом анализа и представляется как структурная схема, определяющая инфраструктуру проекта, возможные опции и ограничения. Возможность неформального и быстрого взаимодействия участников проекта с инженерной подготовкой или без нее, занятых планированием проекта или разработкой устройства – это делает архитектурный подход мощной методологией проектирования. Так как архитектура точно обозначает требования системы, она становится прочной базой для анализа и тестирования качеств устройства и его производительности в различных обстоятельствах. Кроме того, если архитектура корректно понята, корректно создана и корректно применяется, она может быть использована для точной оценки и уменьшения стоимости через демонстрацию рисков содержащихся в реализации различных элементов. Различные структуры архитектуры могут быть применены для проектирования будущих продуктов с подобными характеристиками, что приводит к снижению их стоимости.

Вопросы для самоконтроля

1. В чем принципиальное отличие реагирующих и интерактивных систем?

2. В чем суть проблемы синхронизации в потоковых моделях?

3. Какая проблема была снята введением расширенных конечных автоматов?

4. Роль операторов выборки и интерполяции в Scade?

5. В чем суть модели программно-управляемый автомата?

6. Что изменится если в модели на рис.2.41 строгий переход между состоянием *SendReq1* и состоянием *Wait1* заменить нестрогим переходом?

7. В чем разница между формальной верификацией и верификацией с помощью тестирования?

ЗАКЛЮЧЕНИЕ

В учебном пособии раскрываются особенности встроенных систем и этапы их проектирования. Рассмотрены основные компоненты аппаратного обеспечения встроенных систем: процессор, память, устройства ввода вывода и интерфейсы их взаимодействия. Выбор процессоров для встроенных систем имеет важное последствие для программистов. Программисту может потребоваться использовать ассемблер для того чтобы воспользоваться скрытыми возможностями процессора. Для приложений требующих точных временных реакций управление временем в программах может оказаться трудной задачей из-за сбоев конвейера и необходимости учета работы других параллельных ресурсов процессора.

В результате освоения тем учебного пособия разработчик встроенных систем получает знания по технологии и организации памяти целевой платформы. Разработчик получает знания о том, какие части адресного пространства относится к зависимой и энергонезависимой памяти, а также технологии основной памяти и архитектуру кэш памяти. Это необходимо для понимания времени выполнения программ.

Разработчик получает знания аппаратного и программного механизма, используемого для получения процессором данных от датчиков и формирование команд для исполнительных механизмов. В фокусе этого знания лежит мост между последовательной природой программы и параллельным физическим миром. Поэтому рассмотрен интерфейс аналого-цифрового преобразователя в перспективе цифровой обработки сигналов с учетом квантования, дискретизации и шумов.

Рассмотрен верхний уровень абстракции программного обеспечения – модели вычислений, позволяющие разрабатывать как спецификации систем, так и их модели для последующей автоматической генерации объектного кода приложения. Модель вычислений позволяет на ранних стадиях проектирования обнаруживать некорректное задание требований и их реализацию. Рассмотрены как синхронные, так и асинхронные модели одновременного поведения. Изучение этих вопросов сформирует знания по эффективной методологии проектирования, основанной на модели.

Рассмотрены вопросы валидации и оценки проекта. Показаны такие инструменты валидации как моделирование, эмуляция, макетирование и формальная верификация. В качестве составляющих оценки проекта показаны оценки производительности, потребления энергии и выделения тепла.

В последнее теме рассмотрены модели жизненного цикла программного обеспечения, показывающие различные этапы проектирования прикладного программного обеспечения и их взаимодействие друг с другом.

Для снятия ограничений с доступных современных технологий проектирования встроенных систем необходимо совершенствование:

- языков спецификации и моделирования;
- инструментов генерации реализации систем из их спецификаций;

– средства временной верификации;
– системное программное обеспечение;
– операционные системы реального времени.

Необходимо так же четко осознавать проблемы проектирования встроенных систем, связанные с их особенностями:

– аппаратная сложность;
– разнородная система, состоящая из аппаратной и программной частей;
– разнородные компоненты (центральные процессорные узлы, процессоры цифровой обработки сигналов, специализированные интегральные схемы, шины и т.д.);
– разнородные требования: производительность, стоимость, потребление энергии и т.д.
– реализация в виде системы на кристалле;
– укорочение цикла проектирования из-за ограничения на время вывода нового изделия на рынок.

Успешное освоение содержания дисциплины «Проектирование встроенных управляющих систем реального времени» позволяет приобрести необходимые знания для разработки функциональных спецификаций систем управления, опираясь на современные информационные технологии в этой области проектной деятельности.

ЗАДАНИЯ

Задания ориентированы на помощь в формировании компонент компетенции «знать» и «уметь».

1. Рассмотрим таблицу распределения конвейера на рис. 1.6. Предположим, что процессор располагает логикой продвижения, которая способна сказать, что команда A записывает в тот же регистр, из которого читает команда B и, следовательно, результат, записанный командой A, может быть прямо продвинут в АЛУ, не дожидаясь завершения записи. Предположим, что логика продвижения не тратит времени.

Дать исправленную таблицу распределения. Сколько циклов потеряется на пузыри?

2. Рассмотрим функцию compute_variance для вычисления дисперсии целых чисел из массива data.

```
1 int data[N];
2
3 int compute_variance() {
4   int sum1 = 0, sum2 = 0, result;
5   int i;
6
7   for(i=0; i < N; i++) {
8     sum1 += data[i];
9   }
10  sum1 /= N;
11
12  for(i=0; i < N; i++) {
13    sum2 += data[i] * data[i];
14  }
15   sum2 /= N;
16
17   result = (sum2 - sum1*sum1);
18
19 return result;
20 }
```

Предположим, что программа выполняется на 32-разрядном процессоре с кэш прямого отображения с параметрами (m; S; E; B) = (32; 8; 1; 8). Сделаем следующие допущения:

• целое представляется 4 байтами;
• sum1, sum2, result, и I сохраняются в регистрах;
• data сохраняется в памяти, начиная с адреса 0x0;

Необходимо ответить на следующие вопросы:

1) Рассмотрим случай, когда N = 16. Сколько промахов кэш будет в этом случае?

2) Предположим N = 32. Вычислить новое значение числа промахов.

3) Рассмотрим выполнение для N = 16 и 2-входовой кэш с параметрами (m; S; E; B) = (32; 8; 2; 4). Сколько промахов кэш будет в этом случае?

3. Кэш использует среднюю часть бит адреса как множество индексов, а старшую как тэг. Почему так сделано? Как изменится производительность кэш, если средние биты использовать как тэг, а старшие как индекс?

4. Предположим, что 4-х разрядный ADC-преобразователь работает по принципу последовательных приближений. Диапазон входного сигнала от V_{min}=1 В (=0000) до V_{max}=4.75 В (=1111). Какое количество шагов используется для преобразования значений 2.25 В, 3.75 В, и 1.8 В? Изобразить временную диаграмму преобразования этих значений.

5. В модели CE-OFDM (2.3.4) выполнить согласования тактирования генератора псевдослучайной тестовой последовательности и модулятора с помощью прореживания.

6. Произвести изменения в модели нечеткого регулятора (2.3.5) так, чтобы она соответствовала не 11, а 15 различным значениям угла поворота рулевого управления мобильного робота.

СПИСОК ЛИТЕРАТУРЫ

1. Peter Barry,Patrick Crowley. Modern Embedded Computing. Designing Connected, Pervasive, Media-Rich Systems. Morgan Kaufmann Publishers 2012, p.512.

2. Peter Marwedel. Embedded System Design. Embedded Systems Foundations of Cyber-Physical Systems. 2nd Edition. Springer 2011, p. 389.

3. PowerPC™ Microprocessor Family: The Programming Environments For 32-Bit Microprocessors. MPCFPE32B/AD 1/97 REV. 1. Motorolla.

4. EP93xx. User 's Guide. Cirrus Logic, Inc. 2007.

5.C-5 Network Processor Architecture Guide, C-Port Corp., North Andover, MA, May 31, 2001.

6. Lapsley, P., J. Bier, A. Shoham, and E. A. Lee. DSP Processor Fudamentals– Architectures and Features. IEEE Press, New York. 1997.

7. MPC850 Family User's Manual. Integrated Communications Microprocessor. MPC850UM/D, Rev. 1, 1/2001. Freescale Semiconductor, Inc.

8. John Owens. GPU Architecture Overview. UC Davis. 2007. http://gpgpu.org/static/s2007/slides/02-gpu-architecture-overview-s07.pdf

9. Patterson, D. A. and J. L. Hennessy. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 3nd ed, Sun Francisco, CA, 2003, 1136 p,

10. Eden, M. andM. Kagan. The PentiumR processor with MMXTM technology. In IEEE International Conference (COMPCON), IEEE, San Jose, CA, USA, 1997, pp. 260–262.

11. Таненбаум Э. Архитектура компьютера. 5-е изд. (+CD). — СПб.: Питер, 2007. — 844 с: ил.

12. Introduction in Embedded Programming. http://www.scriptoriumdesigns.com/embedded/

13. Salewski, F, Kowalewski, S. Hardware Platform Design Decisions in Embedded Systems – A Systematic Teaching Approach. *WESE* '06, October 26th, Seoul, South Korea, 2006, pp. 27 – 35.

14. An Introduction to Safety Critical Systems: A document by IPL Information Processing Ltd http://www.ipl.com/pdf/p0826.pdf

15. Н.П. Деменков (МГТУ им. Н.Э. Баумана) Модельно ориентированное проектирование систем управления/ http://is.ifmo.ru/misc2/_matlab_simulink.pdf

16. Introduction to Data Flow Simulation. Agilent technologies http://edocs.soco.agilent.com/display/sv201001/Introduction+to+Data+Flow+Simulation/

17. Esterel Technologies, Inc. http://www.esterel-technologies.com/products/

18. Bengtsson, J. and Yi, W. Timed automata: Semantics, algorithms and tools. In: J. Desel, W. Reisig and G. Rozenberg (eds.): ACPN 2003, Springer LNCS, vol. 3098, pp. 87–124, 2004.

19. Scade Language Reference Manual – Copyright © Esterel Technologies SA - Published March 2012. (ScadeLanguageReference_SC-LRM-63.pdf)

20. Scade Language Tutorial – Copyright © Esterel Technologies SA - Published March 2012. (LanguageTutorial_SC-LT-63.pdf)

21. Mano M. Morris, Charles R. Kime. Logic and Computer Design Fundamentals. Upper Saddle River, NJ: Prentice-Hall, 656 p., 2004

22. SCADE Suite User Manual – Copyright © Esterel Technologies SA - Published March 2012. (UserManual_SC-UM-63.pdf)

23. Noergaard, T. Embedded Systems Architecture. A Comprehensive Guide for Engineers and Programmers. Elsevier Inc, 2005, p.640

24. E.-R. Olderog and H. Dierks. Real-Time Systems. Formal Specification and Automatic Verification. Cambridge University Press, 2008, p.311

Учебное издание

ГОНЧАРОВСКИЙ Олег Владленович,

ПРОЕКТИРОВАНИЕ ВСТРОЕННЫХ УПРАВЛЯЮЩИХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Учебное пособие

Редактор и корректор *X.Y. Zzzzzzzz*

Подписано в печать #.##.2013 Формат60x90/16
Усл. печ. л. #,0 В xxxxx. виде Заказ ###/2013

Издательство
Пермского национального исследовательского
политехнического университета
Адрес: 614990, г. Пермь, Комсомольский пр., 29, к.113.

Тел. (342) 219-80-33.