Network-on-Chip & Архитектура SoC

В. Д. Мунистер

Учебно-практическое издание

«Network-on-Chip. Архитектура SoC»

«Network Science»

2021 г.

GLÜCKSRITTER
MUNISTER

§ СОДЕРЖАНИЕ

«Network-on-Chip. Архитектура SoC» В.Д. Мунистер

МОДЕЛИРОВАНИЕ И ПРОЕКТИРОВАНИЕ СРЕДСТВ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ НА ОСНОВЕ СБИС

Практическое занятие 1. Знакомство с процедурой проектирования цифровых устройств в пакете WebPACK ISE

Практическое занятие 2. Создание VHDL-проектов простейших логических элементов

Практическое занятие 3. Создание VHDL-проектов мультиплексора и демультиплексора

Практическое занятие 4. Создание VHDL-проектов полусумматора и сумматора

Практическое занятие 5. Создание проекта счетчика Джонсона

Практическое занятие 6. Создание проекта суммирующего 4-разрядного счетчика

Практическая работа 7. Графический ввод схемы устройства и функциональная симуляция с использованием САПР MAX+plusII

Практическая работа 7. Графический ввод схемы устройства и функциональная симуляция с использованием САПР MAX+plusII

Практическое занятие 8. Ввод описания схемы на языке AHDL, использование монитора иерархии проекта САПР MAX+plusII

Практическое занятие 9. Проектирование комбинационных схем, программирование ПЛИС и анализ размещения схемы на кристалле

СОДЕРЖАНИЕ КУРСА

Учебное пособие посвящено микропроцессорной технике, архитектуре однокристальных систем, содержит основные понятия, и актуальную таксономию и классификацию современной микропроцессорной техники и состоит из трех глав, и практического модуля, в котором рассматривается вопрос разработки VHDL-проектов цифровых устройств.

Оно будет полезно в качестве дополнительного источника информации для студентов, осваивающих образовательные программы среднего профессионального образования, бакалавриата и магистратуры, где основой получения (формирования) профессиональных компетенций является взаимодействие с системно-аналитическими, информационно-управляющими, конструкторско-технологическими, проектирующими технологиями и системами, которые требуют исследования, анализа, синтеза, программирования и управления на основе системно-аналитического подхода.

Содержание учебника построено по принципу приведения знаний студентов высших и средних профессиональных образовательных учреждений к актуальному состоянию и учитывает современные реалии разработки микропроцессорных устройств.

Учебное пособие предназначено для студентов специальностей «Системный анализ и управление», «Управление в технических системах», «Мехатроника и робототехника», «Управление и информатика в технических системах», «Информационные технологии», «Информатика и вычислительная техника», «Информационная безопасность».

В первой главе рассмотрена многоуровневая структура классификации микропроцессорных систем, во второй – аспектология разработки и функционирования SoC, внутренняя классификация, описывается тракт промышленного производства и разработки, а в третьей – декларируется перспективная технология эволюционного развития однокристальных систем. В учебном пособии большое внимание уделяется описанию аспектов разработки синхронных цифровых интегральных схем на уровне передач данных между регистрами, принципов проектирования интегральных схем специального назначения и программируемых логических контроллеров.

В.Д. Мунистер

§ ВВЕДЕНИЕ. ART OF DOING SCIENCE AND ENGINEERING: LEARNING TO LEARN (1995)

Основы цифровой (дискретной) революции.

РИЧАРД УЭСЛИ ХЭММИНГ

Мы приближаемся к финалу, логическому завершению революции, когда использование непрерывных сигналов для передачи данных заменяется прерывными (дискретными), и, вероятно, для этой цели мы откажемся от использования **импульсов** в пользу уединённых волн – **солитонов** (рис.1).

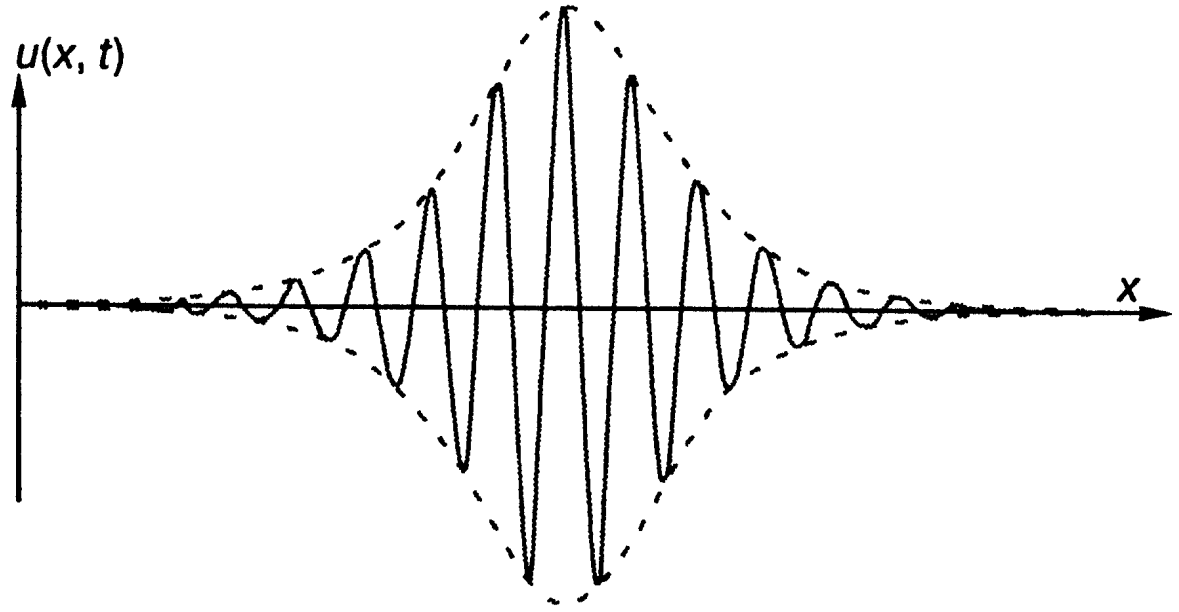


Рис.1 – Солитон - структурно устойчивая уединённая волна, распространяющаяся в нелинейной среде.

В природе многие сигналы являются непрерывными (при условии игнорирования кажущейся дискретной структуры вещей, построенных из молекул и электронов).

В качестве примера непрерывных сигналов можно считать передачу голоса по телефонной связи, музыкальные звуки, высоты и массы людей, пройденное расстояние, скорости, плотности и т.д. Непрерывный сигнал мы почти моментально конвертируем в дискретный, при этом дискретизация обычно выполняется с равными интервалами по времени, а объем сигнала разбивается до сравнительно небольшого количества интервалов и квантуем…

Так каковы были причины и предпосылки этой революции?

1. При передаче непрерывных (аналоговых) сигналов часто приходится компенсировать естественные потери путем усиления сигнала. Любая ошибка, сделанная на одном этапе, до или во время усиления, усугубляется на следующем этапе. Например, телефонная компания, отправляющая голос по всему континенту, может иметь коэффициент усиления 10^{120}. Такой коэффициент может показаться очень большим, поэтому мы быстро выполним простые вычисления чтобы увидеть, разумно ли это. Рассмотрим систему более подробно:

Предположим, что каждый усилитель имеет коэффициент усиления 100, и они разнесены каждые 80 километров. Фактический путь сигнала может быть более 4800 километров, следовательно, на пути размещено около 60 усилителей, поэтому приведенный выше фактор кажется, разумным, теперь мы видели, как он может возникнуть. Должно быть очевидно, что такие усилители должны были быть построены с исключительной точностью, если бы система была пригодна для использования человеком.

Сравним с передачей дискретных сигналов. Нет необходимости усиливать сигнал на каждом этапе, вместо этого используются ретрансляторы. В такой схеме шумы, выявленные на одном из этапов, автоматически удаляются на следующем этапе передачи сигнала. Таким образом, с высокой точностью можно передавать речевой сигнал, и требования к оборудованию и точности его настройки при этом не так высоки. Мы можем использовать, если необходимо, коды обнаружения и исправления ошибок для дальнейшего устранения шума. Наряду с этим мы разработали область цифровых фильтров, которые часто гораздо более универсальны, компактны и дешевле, чем аналоговые фильтры. Следует отметить, что передача сигнала через пространство аналогична передаче сигнала через время (т.е. хранению).

Цифровые компьютеры могут воспользоваться этими функциями и выполнять очень глубокие и точные вычисления, которые недоступны для аналоговых вычислений. Аналоговые компьютеры, вероятно, достигли своего пика производительности, но их нельзя отбрасывать легкомысленно. У них есть некоторые функции, которые, если не требуются большая точность или глубокие вычисления, делают их идеальными в некоторых ситуациях.

2. Изобретение и разработка транзисторов и интегральных схем, особенно цифровых, значительно поспособствовали цифровой революции. Проблема паяных соединений была основной при создании большого компьютера, и микросхемы помогли справиться с большей частью этой проблемы, хотя паяные соединения все еще остаются проблемными. Кроме того, высокая плотность компонентов интегральных схем означает более низкую стоимость и более высокую скорость вычислений (части должны быть близки друг к другу, поскольку в противном случае время передачи сигналов значительно замедлит скорость вычислений). Неуклонное снижение как уровня напряжения, так и тока способствовало частичному решению теплоотдачи.

В 1992 году было подсчитано, что **затраты на межсетевые соединения** составляют примерно:

Взаимосвязь на чипе: $\$10^{-5}$=0.001 цента.
Межчиповая связь: $\$10^{-2}$=1 цент.
Межплатовая связь: $\$10^{-1}$=10 центов.
Межрамочная связь \$1=100 центов.

3. **Общество неуклонно переходит от общества материальных благ к обществу информационных услуг.** Во времена американской революции (около 1780 года), более 90% населения будущих США были в основном фермерами — теперь фермеры составляют очень небольшой процент рабочих. Точно так же до Второй мировой войны большинство рабочих находились на фабриках — теперь их меньше половины. В 1993 году в правительстве (кроме военных) было больше людей, чем в производстве!

Какова будет ситуация в 2030 году? Насколько я понимаю, менее 25% людей в гражданской рабочей силе будут обращаться с вещами, остальные будут обрабатывать информацию в той или иной форме. При производстве фильма или телевизионной программы вы делаете множество действий, хотя, конечно, итоговый продукт имеет материальную форму, содержащий результат организации и обработки информации. Информация, конечно же, хранится в материальной форме, если, например, рассматривать книгу (суть книги — это информация), но информация не является хорошим материалом для потребления, как еда, дом, одежда, автомобиль или поездка на самолете.

Таким образом, информационная революция возникает из вышеупомянутых трех пунктов плюс их синергетического взаимодействия, хотя влияние имеют и следующие пункты:

4. **Компьютеры позволяют роботам делать много вещей, включая большую часть операций на производственных предприятиях**. Очевидно, что компьютеры будут играть значимую роль на предприятиях, хотя нужно быть осторожным, чтобы не требовать, чтобы стандартный компьютер типа фон Неймана был единственным механизмом управления, скорее всего, компьютеры нейронной сети, логика нечеткого набора и вариации будут выполнять большую часть контроля. Оставляя в стороне детский взгляд на робота как машину, напоминающую человека, но, думая об этом как о способе управления и контроля вещей в материальном мире, роботы, используемые в производстве, делают следующее:

A. Производят продукт при более серьезных условиях контроля качества.

B. Производят более дешевый продукт (более низкий по себестоимости).

C. Производят новый продукт, который ранее было производить тяжело или невозможно из-за возможностей человека.

Последний пункт требует особого внимания.

Когда мы впервые перешли от ручной отчетности к машинной, по экономическим соображениям мы сочли необходимым несколько изменить систему отчетности. Аналогичным образом, когда мы перешли от строгой ручной работы к изготовлению машины, мы перешли от винтов и болтов главным образом к заклепкам и сварке.

На практике было тяжело получить точно такой же продукт при производстве вручную, но машины позволяют это сделать.

Действительно, одним из основных элементов преобразования от ручной к машинной продукции является **творческий редизайн** (усовершенствование ресурса целиком, всех его составляющих, от программной части до оформления) эквивалентного продукта. Таким образом, механизация крупной организации может не сработать, если вы попытаетесь сохранять продукт в деталях точно таким же, скорее, должен быть принят крупный компромисс, если мы хотим обеспечить себе значительный успех.

Вы должны понять суть производство, а затем разработать алгоритм производства продукта, его новую форму, а не пытаться механизировать текущую версию и текущие производственные реалии — если вы хотите добиться значительного успеха в долгосрочной перспективе.

Нужно подчеркнуть этот момент; механизация требует, чтобы вы производили эквивалентный продукт, а не идентичный старому. Кроме того, в любом дизайне сейчас необходимо учитывать особенности обслуживания продукта на местах, в долгосрочной перспективе обслуживание нередко перекрывает другие расходы. Чем сложнее разработанная система, тем больше ее обслуживание должно быть главным для окончательной конструкции. Только в том случае, если обслуживание на местах является частью первоначальной конструкции, ее можно безопасно контролировать; нецелесообразно пытаться позже его переносить в другую среду. Это касается любых организаций, механизированных и пользующихся ручным трудом.

5. Влияние компьютеров на науку очень велико и, вероятно, продолжится с течением времени. Мой первый опыт в крупномасштабных вычислениях заключался в разработке оригинальной атомной бомбы в Лос-Аламосе. При разработке не было возможности провести эксперимент на малом масштабе чтобы проверить, есть ли у вас критическая масса или нет, и, следовательно, в то время вычислительные средства были единственным практическим подходом. Мы моделировали на примитивных машинах бухгалтерского учета IBM различные предлагаемые проекты, и они постепенно сводились к дизайну для тестирования в пустыне в Аламагордо, штат Нью-Мексико.

После размышлений над полученным опытом в этом проекте я понял, что компьютеры позволят моделировать много разных видов экспериментов. Я воплощал это видение на практике Bell Telephone Laboratories на протяжении многих лет. Где-то в середине 1950-х годов в обращении к президенту и сотрудникам Bell Telephone Laboratories я сказал: «В настоящее время мы делаем 1 из 10 экспериментов на компьютерах и 9 в лабораториях, но прежде чем я уйду, на компьютерах будет производиться 9 из 10 экспериментов».

Тогда они не поверили мне, поскольку они были уверены, что реальные наблюдения были ключом к экспериментам, и я был просто диким теоретиком из математического факультета, но вы все понимаете, что сейчас мы делаем примерно 90-99% наших экспериментов на компьютерах, и лишь остальное в лабораториях. И эта тенденция будет продолжаться! Гораздо дешевле делать симуляции, чем реальные эксперименты, симуляции гораздо более гибкие в тестировании, и мы можем даже делать то, что нельзя сделать в любой лаборатории. Тенденция неизбежно продолжится в течение некоторого времени. Опять же, продукт изменился!

Но вы же знакомы с пороками схоластики Средних веков — люди делали выводы о том, что произойдет, на основе книг Аристотеля (384-322), а не глядя на природу и реальный характер вещей. Великая идея Галилео Галилея (1564-1642) стала началом современной научной революции: изучение явлений с натуры, а не только из книг! Но что я говорил выше? Мы теперь все больше и больше смотрим в книгах и все меньше и меньше на природу! Очевидно, что мы будем слишком часто заходить слишком далеко, и я ожидаю, что это произойдет в будущем. При всем энтузиазме к компьютерным симуляциям, мы не должны забывать оглядываться на природу, как она есть.

6. Компьютеры также сильно повлияли на Инженерию. Мы не только можем разрабатывать и строить гораздо более сложные вещи, чем мы могли бы вручную, мы можем исследовать еще много альтернативных проектов. Мы также теперь используем компьютеры для управления ситуациями, например, на современном высокоскоростном самолете, где мы строим нестабильные конструкции, а затем используем высокоскоростное обнаружение и компьютеры, чтобы стабилизировать их, так как лишенный посторонней помощи пилот просто не может летать с ними напрямую. Аналогичным образом, теперь мы можем провести нестабильные эксперименты в лабораториях, используя быстрый компьютер для управления нестабильностью. Результат будет заключаться в том, что эксперимент будет измерять что-то очень точно прямо на краю стабильности.

Как отмечалось выше, инженерия приближается к науке, и, следовательно, роль моделирования в неизведанных ситуациях быстро возрастает как в технике, так и в науке. Также верно, что компьютеры часто являются важным компонентом хорошего дизайна/разработки.

В прошлом в инженерии доминировал подход «что мы можем сделать», а теперь подход сформулирован как «что мы хотим делать», так как теперь у нас есть возможность проектировать почти все, что мы хотим. Более чем когда-либо прежде, инженерия — это вопрос выбора и баланса, а не просто то, что можно сделать. И все больше человеческих факторов, которые будут определять хороший дизайн, — тему, которая всегда требует вашего серьезного внимания.

7. Влияние на общество также велико. Наиболее очевидной иллюстрацией является то, что компьютеры предоставили топ-менеджменту возможность микро-менеджмента своей организацией, а высшее руководство практически не проявляло способности сопротивляться использованию этой мощности. Вы можете регулярно наблюдать в газетах статьи о крупных корпорациях, которые занялись децентрализацией, но, когда вы наблюдаете за этой компанией в течение нескольких лет, вы видите, что они просто намеревались сделать это, но не сделали.

Среди других пороков микро-менеджмента: более низкое управление не получает возможности принимать ответственные решения и учиться на своих ошибках, а, скорее, из-за того, что пожилые люди, наконец, уходят в отставку, тогда более низкое управление оказывается в качестве высшего руководства, не имея большого опыта в управлении!

Кроме того, неоднократно было показано, что централизованное планирование дает плохие результаты (например, российский эксперимент или наша собственная бюрократия). Лица, находящиеся на месте, обычно имеют лучшие знания, чем те, кто находится наверху и, следовательно, часто (не всегда) принимают правильные решения, если действия и планы не подвергаются микро-менеджменту. У людей снизу нет глобального взгляда, ну а люди сверху не имеют локального представления обо всех деталях, многие из которых часто могут быть очень важными, так что любая крайность дает плохие результаты.

Далее, идея, которая возникает на местах, основанная на непосредственном опыте людей, выполняющих эту работу, не может попасть в централизованно контролируемую систему, так как менеджеры сами не думали об этом**. Синдром «изобретено не нами»** (NIH) является одним из основных проклятий нашего общества, и компьютеры, способные поощрять микро-менеджмент, становятся существенным фактором.

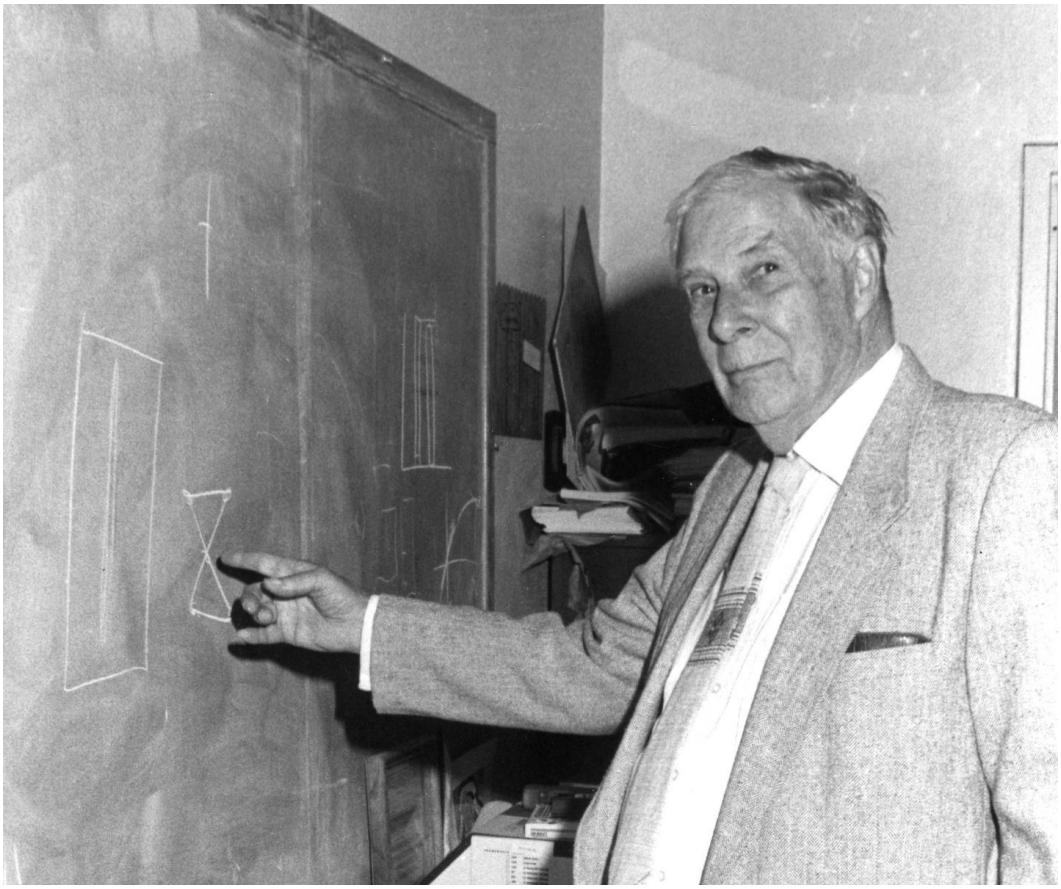
Постепенно надвигается тенденция использования **микро-менеджмента***. Возникают свободные связи между небольшими, несколько независимыми организациями. Например, в брокерском бизнесе одна компания обязалась продавать свои услуги другим небольшим абонентам, например, компьютерным и юридическим сервисам. Это делает брокерские решения для местных менеджеров клиентов близкими к передовым позициям, иными словами очень актуальными. Аналогичным образом, в фармацевтической области некоторые слабо связанные компании организуют внутреннюю торговлю со своими правилами. Я считаю, что вы можете ожидать гораздо большего от этой свободной ассоциации между небольшими организациями в качестве защиты от микро-менеджмента сверху, что происходит так часто в крупных организациях. В организациях всегда была какая-то независимость подразделений, но власть сверху на уровне микро-менеджмента, по-видимому, разрушала обычные линии и автономию принятия решений, и я сомневаюсь в способности большинства топ-менеджеров долгое время сопротивляться микро-менеджменту. Я также сомневаюсь, что многие крупные компании смогут отказаться от микро-менеджмента; большинство из них, вероятно, будут заменены в долгосрочной перспективе небольшими организациями без затрат (накладных расходов) и ошибок высшего руководства. Таким образом, компьютеры влияют на самую структуру того, как общество делает свой бизнес, и на данный момент, по-видимому, в худшую сторону.

8. **Компьютеры уже вторглись в развлекательную зону**. Неофициальный опрос показывает, что средний американец тратит гораздо больше времени на компьютер и телевизор, чем на еду — снова информация имеет приоритет над базовыми потребностями в питании! Многие рекламные ролики и некоторые программы теперь полностью создаются на компьютере. Как далеко машины будут влиять на общество, это вопрос спекуляций, который открывает множество дверей к проблемам, если не обсуждать это открыто! Поэтому я должен оставить это в своих фантазиях относительно того, что, используя компьютеры на чипах, можно сделать в таких областях, как брак, спорт, игры, «путешествие в домашних условиях через виртуальные реальности» и другие виды человеческой деятельности.

*Микро-менеджмент — это стиль управления персоналом, при котором руководство использует чрезмерный и постоянный контроль над сотрудниками, не допуская никакой самостоятельности в принятии решений.

Сколько трендов, предсказанных выше, в направлении микроменеджмента и вне его, будет широко применяться и снова станет самой подходящей для вас темой, как для будущих работников и служащих?

Неизвестно, но вы будете дураком, если не будете уделять этому свое и постоянное внимание. Я предлагаю вам переосмыслить все, что вы когда-либо узнали по этому вопросу, задать вопрос о каждой успешной доктрине из прошлого и, наконец, решить для себя свою будущую применимость. Будда сказал своим ученикам: «Не верьте ничему, независимо от того, где вы это прочитали, или кто это сказал, даже если это сказал я, если это не согласуется с вашим собственным рассудком и вашим собственным здравым смыслом». Я говорю то же самое — вы должны взять на себя ответственность за то, во что верите. В том числе — и в перспективы и реалии вашей профессии.



«Цель этого курса — подготовить вас к вашему техническому будущему»

Ричард Хэмминг

§ РАЗДЕЛ 1: ВВЕДЕНИЕ В АРХИТЕКТУРУ И ТАКСОНОМИЮ ЭВМ

Архитектура компьютера: Принципы фон Неймана

Архитектура компьютера — набор типов данных, операций и характеристик каждого отдельно взятого уровня. Она описывает общую модель компьютера. Аспекты реализации (например, технология, применяемая при реализации памяти) не являются частью архитектуры

Основы учения об архитектуре вычислительных машин заложил **Джон фон Нейман** в 1944 году, когда подключился к созданию первого в мире лампового компьютера **ЭНИАК**.

В процессе работы над ЭНИАКом в Институте Мура в Пенсильванском Университете во время многочисленных дискуссий фон Неймана с его коллегами Джоном Уильямом Мокли, Джоном Эккертом, Германом Голдстайном и Артуром Бёрксом возникла идея более совершенной машины под названием EDVAC. Исследовательская работа над EDVAC продолжалась параллельно с конструированием **ЭНИАКа.**

Отметим архитектурные достоинства машина ENIAC:

- **SIMD-архитектура (см. Раздел «Таксономия Флинна»)**, распределенность и иерархия средств управления, смешанный синхронно-асинхронный способ управления вычислениями;
- параллелизм при обработке данных (допускалась одновременная работа нескольких вычислительных устройств и параллельная обработка десятичных разрядов чисел);
- ручная реконфигурируемость структуры (ручное программирование «неспециализированной» машины под структуру решаемой задачи);
- однородность, модульность и масштабируемость (варьируемость количества устройств).

Итак, машина ENIAC обладала совокупностью архитектурных свойств, которые присущи современным высокопроизводительным параллельным вычислительным системам. Проект ENIAC опережал возможности элементной базы (ламповой электроники).

Если исходить из характеристик элементной базы 1940-х годов (а в то время ламповые элементы были самыми быстродействующими), то можно указать на следующие недостатки машины ENIAC:

- ручное («механическое») трудоемкое программирование ВМ под структуру решаемой задачей (такое программирование длилось несколько часов или даже дней);
- низкая надежность, обусловленная применением большого числа ламп, электромагнитных реле, механических переключателей и кабелей, а также и ручным программированием структуры машины;
- малая емкость оперативной памяти (334 десятиразрядных десятичных чисел);
- громоздкость и дороговизна машины (18000 электронных ламп, 486000 долларов!):
- аппаратурная избыточность.

Машина ENIAC – эта первая электронная ВМ, которая нашла практическое применение и была для своего времени инструментом решения сложных задач.

В 1945 году группой Д. Мочли выполнялись работы по конструированию машины EDVAC. В разработке с 1945 года принимал участие Дж. фон Нейман в качестве консультанта. В 1947 г. группа Д. Мочли распалась, тем не менее, другие специалисты Электротехнической школы Мура завершили проект. **Машина EDVAC** вступила в строй в 1950 г. (хотя усовершенствования вносились до 1952г.)

Отметим некоторые показатели EDVAC: тактовая частота – 1 МГц (на порядок выше, чем в ENIAC); быстродействие – 1000 операций в секунду над 32-разрядными двоичными числами; емкость оперативной памяти – 32768 байт; количество электронных ламп – 3000.

В марте 1945 года принципы логической архитектуры были оформлены в документе, который назывался «Первый проект отчёта о EDVAC» — отчёт для Баллистической лаборатории Армии США, на чьи деньги осуществлялась постройка ЭНИАКа и **разработка EDVACa**. Отчёт, поскольку он являлся всего лишь наброском, не предназначался для публикации, а только для распространения внутри группы, однако Герман Голдстайн — куратор проекта со стороны Армии США — размножил эту научную работу и разослал её широкому кругу учёных для ознакомления.

Так как на первой странице документа стояло только имя фон Неймана, у читавших документ сложилось ложное впечатление, что автором всех идей, изложенных в работе, является именно он.

Документ давал достаточно информации для того, чтобы читавшие его могли построить свои компьютеры, подобные EDVACу на тех же принципах и с той же архитектурой, которая в результате стала называться «архитектурой фон Неймана». После завершения Второй мировой войны и окончания работ над ЭНИАКом в феврале 1946 года команда инженеров и учёных распалась, Джон Мокли, Джон Экерт решили обратиться в бизнес и создавать компьютеры на коммерческой основе. Фон Нейман, Голдстайн и Бёркс перешли в Институт перспективных исследований, где решили создать свой компьютер «IAS-машина», подобный EDVACу, и использовать его для научно-исследовательской работы.

Функциональная структура машины EDVAC

Машина EDVAC состояла из центрального арифметического устройства (АУ), оперативного запоминающего устройства (ОЗУ), внешних запоминающих устройств (ВЗУ), входного и выходного узлов (УВх, УВых) и центрального управляющего устройства (УУ). **В отличие от ENIAC данная ЭВМ была последовательной машиной, она не могла выполнять двух логических или арифметических операций одновременно.** В то время это было технико-экономически обосновано.

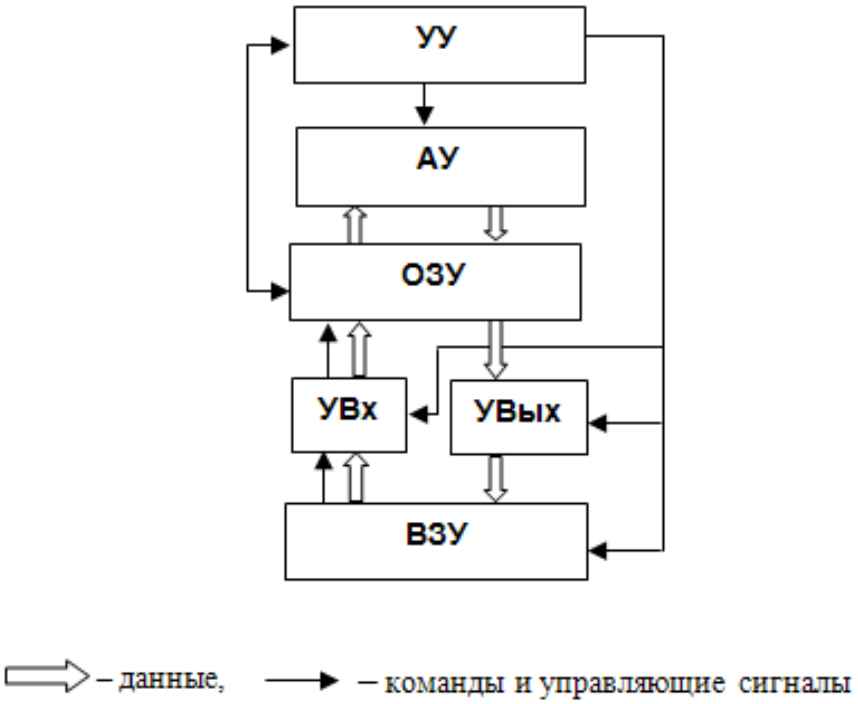


Рис. 2 - Функциональная структура машины EDVAC

В июне 1946 года они изложили свои принципы построения вычислительных машин в ставшей классической статье «**Предварительное рассмотрение логической конструкции электронно-вычислительного устройства»**. С тех пор прошло более полувека, но выдвинутые в ней положения сохраняют свою актуальность и сегодня. В статье убедительно обосновывается использование двоичной системы для представления чисел, а ведь ранее все вычислительные машины хранили обрабатываемые числа в десятичном виде. Авторы продемонстрировали преимущества двоичной системы для технической реализации, удобство и простоту выполнения в ней арифметических и логических операций. **В дальнейшем ЭВМ стали обрабатывать и нечисловые виды информации — текстовую, графическую, звуковую и другие, но двоичное кодирование данных по-прежнему составляет информационную основу любого современного компьютера.**

По сути, Нейману удалось обобщить научные разработки и открытия многих других ученых и сформулировать на их основе принципиально новое.

Принципы фон Неймана

Использование двоичной системы счисления в вычислительных машинах. Преимущество перед десятичной системой счисления заключается в том, что устройства можно делать достаточно простыми, арифметические и логические операции в двоичной системе счисления также выполняются достаточно просто.

Программное управление ЭВМ. Работа ЭВМ контролируется программой, состоящей из набора команд. Команды выполняются последовательно друг за другом. Созданием машины с хранимой в памяти программой было положено начало тому, что мы сегодня называем программированием.

Память компьютера используется не только для хранения данных, но и программ. При этом и команды программы и данные кодируются в двоичной системе счисления, т.е. их способ записи одинаков. Поэтому в определенных ситуациях над командами можно выполнять те же действия, что и над данными.

Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованы. В любой момент можно обратиться к любой ячейке памяти по ее адресу. Этот принцип открыл возможность использовать переменные в программировании.

Возможность условного перехода в процессе выполнения программы. Несмотря на то, что команды выполняются последовательно, в программах можно реализовать возможность перехода к любому участку кода. Самым главным следствием этих принципов можно назвать то, что теперь программа уже не была постоянной частью машины (как например, у калькулятора). Программу стало возможно легко изменить. А вот аппаратура, конечно же, остается неизменной, и очень простой.

Для сравнения, программа компьютера ENIAC (где не было хранимой в памяти программы) определялась специальными перемычками на панели. Чтобы перепрограммировать машину (установить перемычки по-другому) мог потребоваться далеко не один день. И хотя программы для современных компьютеров могут писаться годы, однако они работают на миллионах компьютеров после несколько минутной установки на жесткий диск.

Как работает машина фон Неймана?

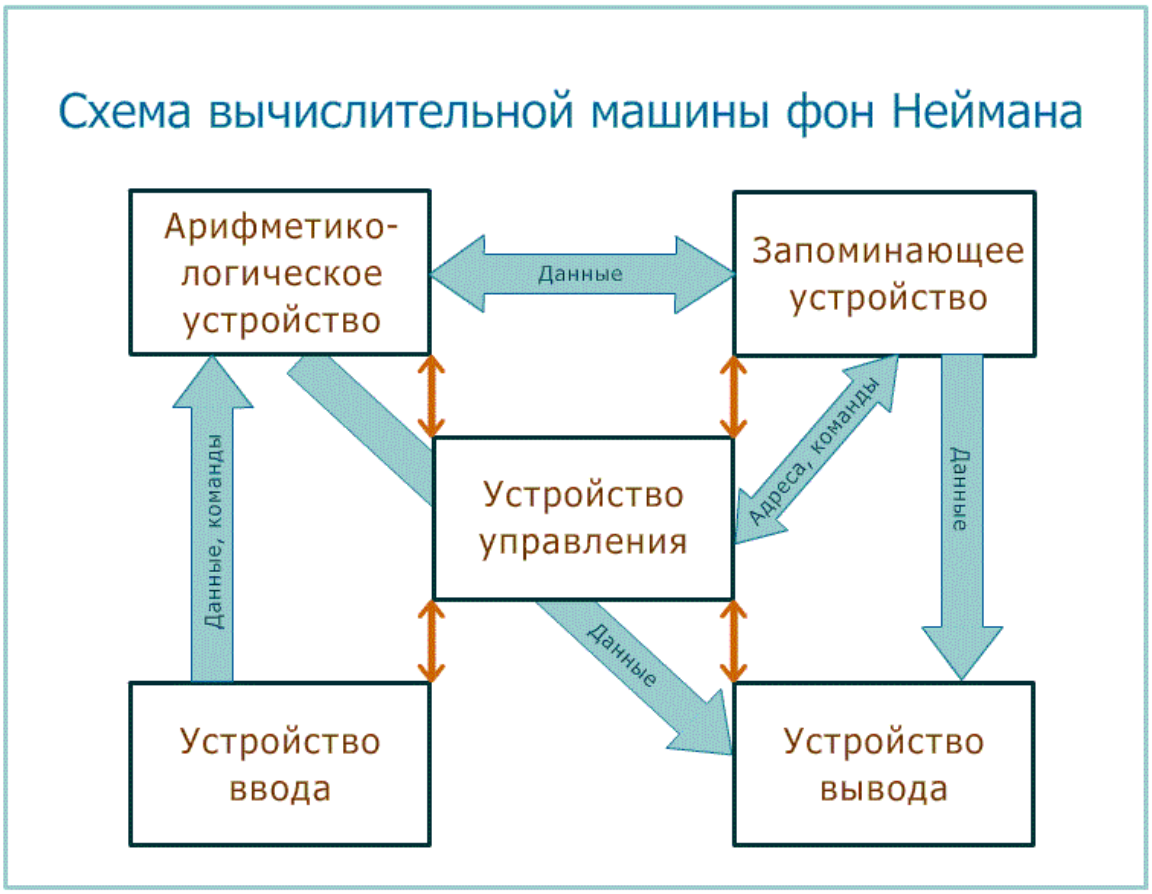


Рис. 2 – Flow-диаграмма работы рассматриваемой архитектуры

Машина фон Неймана состоит из **запоминающего устройства** (памяти) - ЗУ, **арифметико-логического устройства** - АЛУ, **устройства управления** – УУ, а также **устройств ввода и вывода**.

Программы и данные вводятся в память из устройства ввода через арифметико-логическое устройство. Все команды программы записываются в соседние ячейки памяти, а данные для обработки могут содержаться в произвольных ячейках. У любой программы последняя команда должна быть командой завершения работы.

Команда состоит из указания, какую операцию следует выполнить (из возможных операций на данном «железе») и адресов ячеек памяти, где хранятся данные, над которыми следует выполнить указанную операцию, а также адреса ячейки, куда следует записать результат (если его требуется сохранить в ЗУ).

Арифметико-логическое устройство выполняет указанные командами операции над указанными данными.

Из **арифметико-логического устройства** (АЛУ) результаты выводятся в память или устройство вывода. Принципиальное различие между ЗУ и устройством вывода заключается в том, что в ЗУ данные хранятся в виде, удобном для обработки компьютером, а на устройства вывода (принтер, монитор и др.) поступают так, как удобно человеку.

УУ управляет всеми частями компьютера. От управляющего устройства на другие устройства поступают сигналы «что делать», а от других устройств УУ получает информацию об их состоянии.

Управляющее устройство содержит специальный регистр (ячейку), который называется «счетчик команд». После загрузки программы и данных в память в счетчик команд записывается адрес первой команды программы. УУ считывает из памяти содержимое ячейки памяти, адрес которой находится в счетчике команд, и помещает его в специальное устройство — «Регистр команд». УУ определяет операцию команды, «отмечает» в памяти данные, адреса которых указаны в команде, и контролирует выполнение команды. Операцию выполняет АЛУ или аппаратные средства компьютера.

Архитектура компьютера: Гарвардский подход

В 1930-х годах правительство США поручило **Гарвардскому и Принстонскому университетам разработать архитектуру ЭВМ** для военно-морской артиллерии.

В конце 1930-х годов в **Гарвардском университете Говардом Эйкеном** была разработана архитектура компьютера Марк I, в дальнейшем называемая по имени этого университета. Оригинальная идея была продемонстрирована Эйкеном компании IBM в октябре 1937 года. Однако победила более простая в реализации разработка Принстонского университета (более известная как архитектура фон Неймана, названная так по имени авторитетного учёного-консультанта и разработчика, первым предоставившего отчёт об архитектуре, к которой пришли в ходе плодотворных дискуссий в команде создателей; авторами же идей, заложенных в этой архитектуре, являлись Джон Преспер Экерт и Джон Уильям Мокли).

Гарвардская архитектура использовалась советским учёным А. И. Китовым в ВЦ-1 МО СССР.

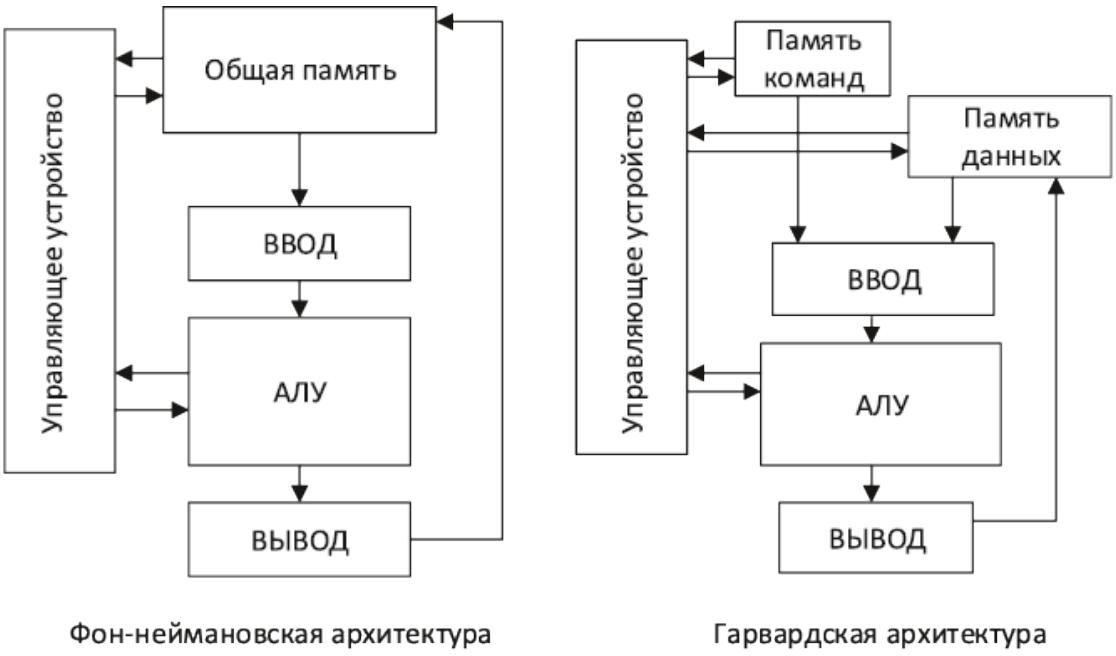


Рис. 3 – Структурные схемы рассматриваемых архитектур

В архитектуре фон Неймана процессор в каждый момент времени может либо читать инструкцию, либо читать/записывать единицу данных из/в памяти. Оба действия одновременно происходить не могут, поскольку инструкции и данные используют один и тот же поток (шину).

В компьютере с использованием гарвардской архитектуры процессор может считывать очередную команду и оперировать памятью данных одновременно и без использования кэш-памяти. Таким образом, компьютер с гарвардской архитектурой при определенной сложности схемы быстрее, чем компьютер с архитектурой фон Неймана, поскольку потоки команд и данных расположены на раздельных физически не связанных между собой аппаратных каналах.

Идея, реализованная Эйкеном, заключалась в физическом разделении линий передачи команд и данных. В первом компьютере Эйкена «Марк I» для хранения инструкций использовалась перфорированная лента, а для работы с данными — электромеханические регистры. Это позволяло одновременно пересылать и обрабатывать команды и данные, благодаря чему значительно повышалось общее быстродействие компьютера.

В гарвардской архитектуре **характеристики устройств памяти для инструкций и памяти для данных не обязательно должны быть одинаковыми.** В частности, ширина слова, тактирование, технология реализации и структура адресов памяти могут различаться. В некоторых системах инструкции могут храниться в памяти только для чтения, в то время как для сохранения данных обычно требуется память с возможностью чтения и записи. В некоторых системах требуется значительно больше памяти для инструкций, чем памяти для данных, поскольку данные обычно могут подгружаться с внешней или более медленной памяти. Такая потребность увеличивает битность (ширину) шины адреса памяти инструкций по сравнению с шиной адреса памяти данных.

Исходя из физического разделения **шин команд и данных**, разрядности этих шин могут различаться и физически не могут пересекаться.

Соответствующая схема реализации доступа к памяти имеет один очевидный недостаток — высокую стоимость. При разделении каналов передачи команд и данных на кристалле процессора последний должен иметь почти в два раза больше выводов (т.к. шины адреса и данных составляют основную часть выводов микропроцессора).

Способом решения этой проблемы стала идея использовать общую шину данных и шину адреса для всех внешних данных, а внутри процессора использовать шину данных, шину команд и две шины адреса. **Такую концепцию стали называть модифицированной Гарвардской архитектурой.**

Такой подход применяется в современных сигнальных процессорах. Еще дальше по пути уменьшения стоимости пошли при создании **однокристалльных ЭВМ — микроконтроллеров.**

В них одна шина команд и данных применяется и внутри кристалла.

Разделение шин в модифицированной Гарвардской структуре осуществляется при помощи раздельных управляющих сигналов: чтения, записи или выбора области памяти.

Расширенная гарвардская архитектура

Часто требуется выбрать три составляющие: два операнда и инструкцию (в алгоритмах цифровой обработки сигналов - это наиболее распространенная задача в БПФ и КИХ, БИХ фильтрах). Для этого существует кэш-память.

В ней может храниться инструкция — следовательно, обе шины остаются свободными и появляется возможность передать два операнда одновременно. Использование кэш-памяти вместе с разделёнными шинами получило название «**Super Harvard Architecture» («SHARC**») — расширенная Гарвардская архитектура.

Примером могут служить процессоры «Analog Devices»: ADSP-21xx — модифицированная Гарвардская Архитектура, ADSP-21xxx(SHARC) — расширенная Гарвардская Архитектура.

Гибридные модификации с архитектурой фон-Неймана

Существуют гибридные модификации архитектур, сочетающие достоинства как Гарвардской, так и фон-Неймановской архитектур. Современные CISC-процессоры обладают раздельной кэш-памятью 1-го уровня для инструкций и данных, что позволяет им за один такт получать одновременно как команду, так и данные для её выполнения, то есть процессорное ядро, формально, является гарвардским, но с программной точки зрения выглядит как фон-Неймановское, что упрощает написание программ.

Обычно в данных процессорах одна шина используется и для передачи команд, и для передачи данных, что упрощает конструкцию системы.

Современные варианты таких процессоров могут иногда содержать встроенные контроллеры сразу нескольких разнотипных шин для работы с различными типами памяти — например, DDR RAM и Flash. Тем не менее, и в этом случае шины, как правило, используются и для передачи команд, и для передачи данных без разделения, что делает данные процессоры еще более близкими к фон-Неймановской архитектуре при сохранении плюсов Гарвардской архитектуры.

Первым компьютером, в котором была использована идея гарвардской архитектуры, был Марк I. Гарвардская архитектура используется в ПЛК и микроконтроллерах, таких, как Atmel AVR, Intel 4004, Intel 8051 и т.д.

Архитектура потока данных

Dataflow architecture (с англ. — «Архитектура потока данных») — компьютерная архитектура, которая прямо контрастирует с традиционной архитектурой фон Неймана или порядком выполнения.

Архитектуры потока данных в концепции не имеют счётчика команд: выполнимость и выполнение инструкций определяются исключительно на основе наличия входных аргументов, поэтому порядок выполнения таковых непредсказуем, то есть недетерминирован.

Хотя ни одно коммерчески успешное компьютерное оборудование общего назначения не использовало архитектуру потоков данных, оно было успешно реализовано в специализированных аппаратных средствах, таких как цифровая обработка сигналов, сетевая маршрутизация, графическая обработка, телеметрия и, в последнее время, в хранилищах данных. Это также очень актуально во многих архитектурах программного обеспечения сегодня, включая конструкции ядра базы данных и платформы параллельных вычислений. Архитектуры синхронного потока данных настраиваются в соответствии с рабочей нагрузкой в реальном времени. Архитектуры потока данных, которые по своей природе являются детерминированными, позволяют программистам управлять сложными задачами, такими как балансировка нагрузки процессора, синхронизация и доступ к общим ресурсам.

В архитектуре потока данных **вся программная система** рассматривается как последовательность преобразований в последовательных частях или наборе входных данных, где данные и операции не зависят друг от друга. При таком подходе данные поступают в систему и затем проходят через модули по одному, пока они не будут назначены какому-либо конечному месту назначения (выходу или хранилищу данных).

Соединения между компонентами или модулями могут быть реализованы как поток ввода-вывода, буферы ввода-вывода, конвейерные или другие типы соединений. Данные могут передаваться в топологии графа с циклами, в линейной структуре без циклов или в структуре древовидного типа.

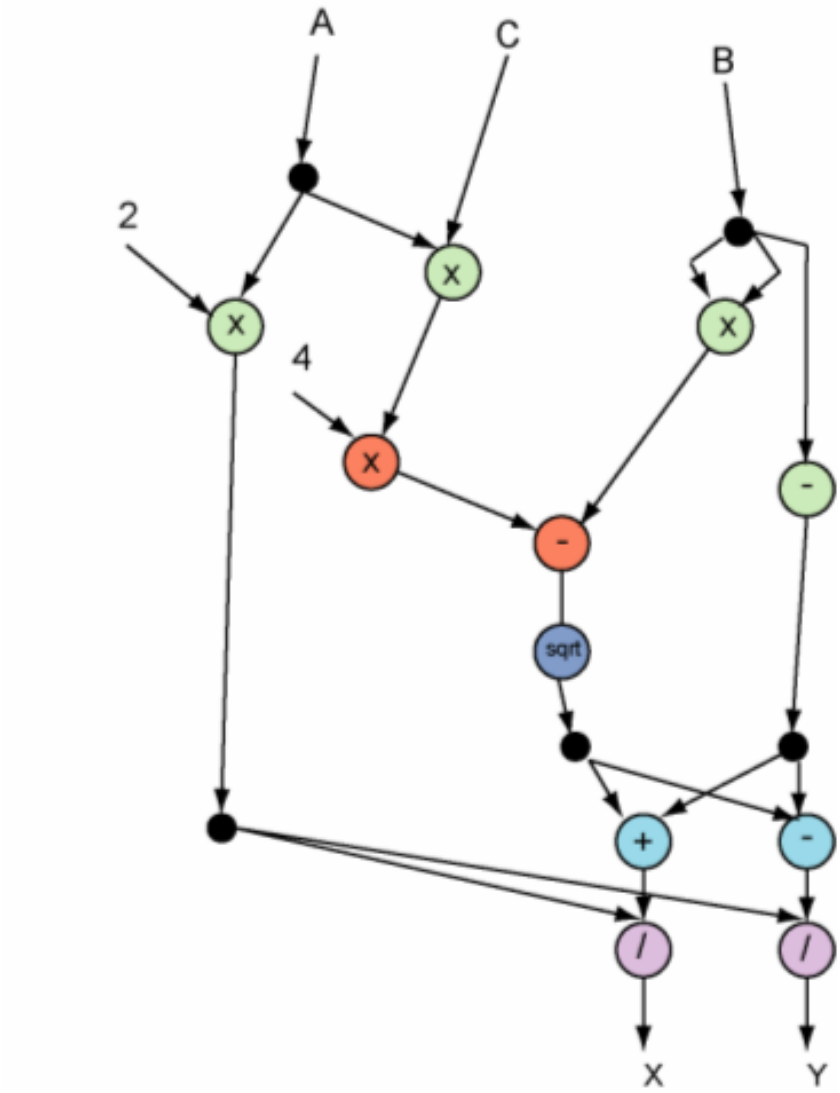


Рис. 4 - Граф потоков данных нахождения корней квадратного уравнения

Transport triggered architecture (TTA-процессоры)

Transport triggered architecture (TTA) — вариант архитектуры микропроцессоров, в которой программы непосредственно управляют внутренними соединениями (шинами) между блоками процессора (например, АЛУ, Регистровый файл). Вычисления являются побочным эффектом передачи данных между блоками: запись данных на входной порт (triggering port) функционального устройства приводит к началу их обработки данным устройством. Благодаря модульной структуре, TTA-архитектура подходит для проектирования проблемно-ориентированных процессоров (ASIP), при этом TTA-процессоры получаются универсальнее и дешевле чем аппаратные ускорители для фиксированных функций.

Обычно TTA-процессор имеет несколько транспортных шин и множество **функциональных устройств (ФУ),** подключенных к этим шинам. Обилие ФУ позволяет достичь параллелизма на уровне инструкций. Параллелизм статически определяется программистом. В этом отношении, а также из-за большой длины машинной инструкции, TTA-архитектуры напоминают архитектуры very long instruction word (VLIW) . Инструкция для TTA состоит из нескольких слотов, по слоту на каждую шину. Каждый слот определяет, как данные будут передаваться по данной шине. Столь полный контроль позволяет производить некоторые оптимизации, невозможные для классических архитектур. Например, возможна явная пересылка данных между разными ФУ без сохранения промежуточных данных в регистровом файле.

Каждое функциональное устройство выполняет одну или более операцию. Возможна реализация как простейших арифметических операций целочисленное сложение, так и сложных произвольных операций, специфичных для целевого приложения. Операнды передаются в ФУ через порты ФУ. Результат операции передается через выходной порт ФУ. В каждом ФУ может быть реализован независимый вычислительный конвейер. Доступ к памяти и взаимодействие с внешними устройствами обрабатывается специальными ФУ.

ФУ для доступа к памяти часто называют load/store unit.

Управляющее устройство контролирует процесс исполнения программ.

У него имеется доступ к памяти инструкций для получения следующих машинных команд. Также реализует команды перехода jump. Обычно управляющее устройство конвейеризовано и выделены стадии: загрузки, декодирования, исполнения инструкций.

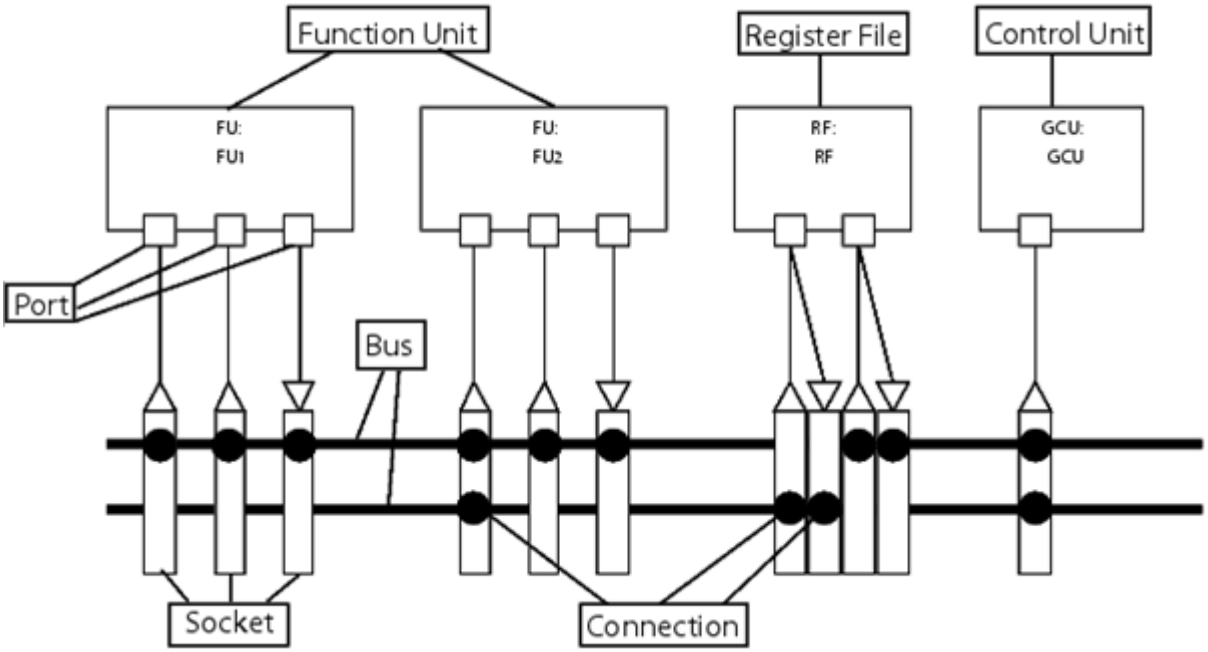


Рис. 5 – TTA-архитектура

Пример операции сложения для гипотетического TTA-процессора:

r1 -> ALU.operand1
r2 -> ALU.add.trigger
ALU.result -> r3

Один из основных принципов TTA — упростить аппаратное обеспечение, усложнив программное.

Реализации:

- Dr. Dobb's "One-Der" 32-разрядный TTA с исходниками на Verilog, ассемблером и компилятором языка Forth.
- MAXQ Dallas Semiconductor. Является OISC "one instruction set computer", то есть имеет лишь одну инструкцию MOVE.
- TCE project. Используется компилятор LLVM.
- "MOVE project"
- Процессор Able New England Digital, использовался в системе синтеза музыки Synclavier.

Архитектура набора команд: CISC MISC NISC URISC RISC ZISC EDGE (TRIPS) VLIW (EPIC)

Архитектура набора команд (англ. instruction set architecture, ISA) — часть архитектуры компьютера, определяющая программируемую часть ядра микропроцессора. На этом уровне определяются реализованные в микропроцессоре конкретного типа:

- архитектура памяти,
- взаимодействие с внешними устройствами ввода/ вывода,
- режимы адресации,
- машинные команды,
- различные типы внутренних данных (например, с плавающей запятой, целочисленные типы и т. д.),
- обработчики прерываний и исключительных состояний.

В упрощенной трактовке время выполнения программы (Твыч) можно определить через число команд в программе (Nком) среднее количество тактов процессора, приходящихся на одну команду (CPI), и длительность тактового периода:

$$Твыч = Nком * CPI * \tau_{пр}$$

Каждая из составляющих выражения зависит от одних аспектов архитектуры системы команд и, в свою очередь, влияет на другие, что свидетельствует о необходимости чрезвычайно ответственного подхода к выбору **архитектуры системы команд (АСК)**

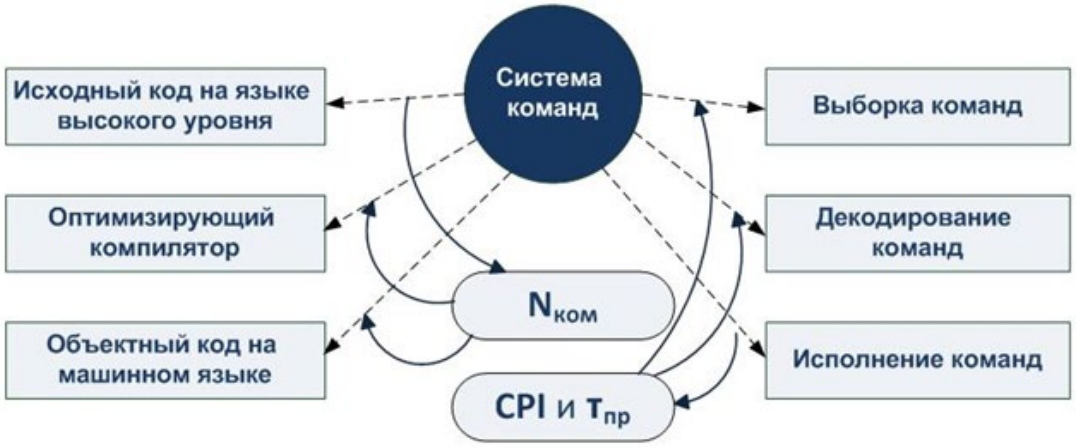


Рис. 6 - Взаимосвязь между системой команд и факторами, определяющими эффективность вычислений.

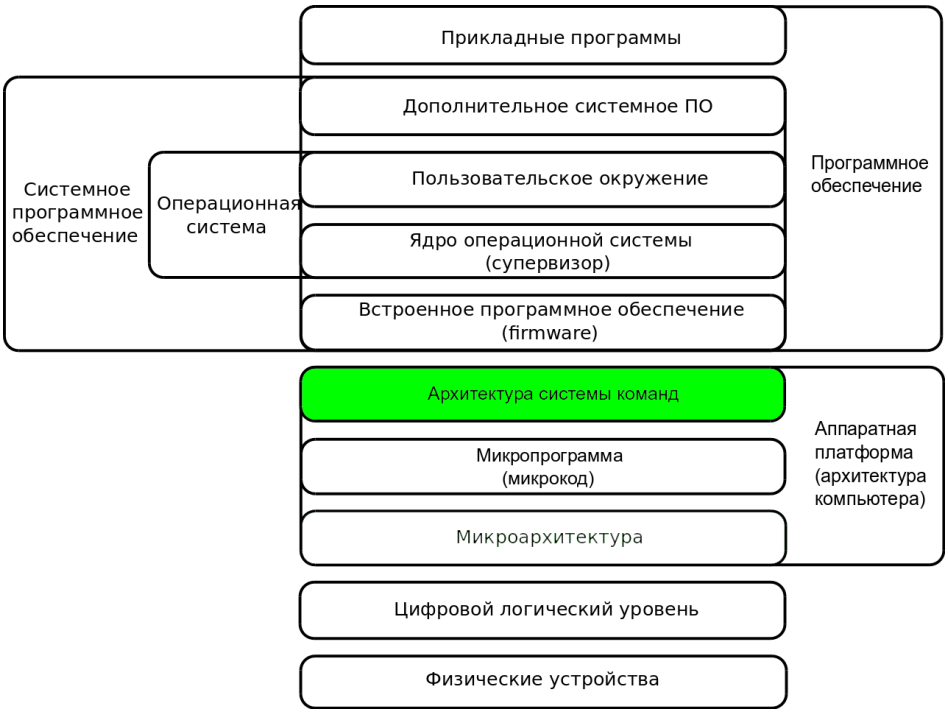


Рис. 7 - Схема, иллюстрирующая место уровней микроархитектуры, архитектуры набора команд и микрокода в многоуровневой структуре компьютера.

Архитектура (системы команд, АСК) описывает модель, топологию и реализацию ISA на микросхеме микропроцессора. На этом уровне определяется:

- конструкция и взаимосвязь основных блоков ЦП,
- структура ядер, исполнительных устройств, АЛУ, а также их взаимодействия,
- блоков предсказания переходов,
- организация конвейеров,
- организация кэш-памяти,
- взаимодействие с внешними устройствами.

В рамках одного семейства микропроцессоров микроархитектура со временем расширяется путём добавления новых усовершенствований и оптимизации существующих команд с целью повышения производительности, энергосбережения и функциональных возможностей микропроцессора. При этом сохраняется совместимость с предыдущей версией ISA.

Общая характеристика архитектуры системы команд вычислительной машины складывается из ответов на следующие вопросы:

- Какого вида данные будут представлены в вычислительной машине и в какой форме?
- Где эти данные могут храниться помимо основной памяти?
- Каким образом будет осуществляться доступ к данным?
- Какие операции могут быть выполнены над данными?
- Сколько операндов может присутствовать в команде?
- Как будет определяться адрес очередной команды?
- Каким образом будут закодированы команды?

Предметом данной главы является обзор наиболее распространенных архитектур системы команд.

CISC-архитектура (Complex Instruction Set Computer) относится к процессорам с полным набором команд. Она имеет нефиксированную длину команд, отличается кодированием арифметических действий в единой команде и малым количеством регистров, большинство из которых выполняет только выделенную функцию.

CISC реализована во множестве типов микропроцессоров, которые выполняют большое количество разноформатных команд (порядка 200-300), применяя более десяти различных способов адресации. Командная система может включать несколько сотен команд различного уровня сложности или формата (от 1 до 15 байт).

Процессору с **архитектурой CISC** приходится иметь дело с более сложными инструкциями неодинаковой длины. Выполнение одиночной CISC-инструкции может происходить быстрее, однако обрабатывать несколько таких инструкций параллельно сложнее.

Облегчение отладки программ на ассемблере влечет за собой загромождение узлами микропроцессорного блока. Для повышения быстродействия следует увеличить тактовую частоту и степень интеграции, что вызывает необходимость совершенствования технологии и, как следствие, более дорогого производства.

CISC (англ. Complex Instruction Set Computing) — наиболее распространённая архитектура современных настольных, серверных и мобильных процессоров построена по архитектуре Intel x86 (или x86-64 в случае 64-разрядных процессоров). Формально, все x86-процессоры являлись CISC-процессорами, однако процессоры, начиная с Intel486DX, являются **CISC-процессорами с RISC-ядром**. Они непосредственно перед исполнением преобразуют CISC-инструкции процессоров x86 в более простой набор внутренних инструкций RISC.

В микропроцессор встраивается аппаратный транслятор, превращающий команды x86 в команды внутреннего RISC-процессора. При этом одна команда x86 может порождать несколько RISC-команд (в случае процессоров типа P6-до 4-х RISC команд в большинстве случаев). Исполнение команд происходит на **суперскалярном конвейере** одновременно по несколько штук.

Это потребовалось для увеличения скорости обработки CISC-команд, так как известно, что любой CISC-процессор уступает RISC-процессорам по количеству выполняемых операций в секунду. В итоге, такой подход и позволил поднять производительность CPU.

Достоинства архитектуры CISC:

Компактность наборов инструкций уменьшает размер программ и уменьшает количество обращений к памяти.

Наборы инструкций включают поддержку конструкций высокоуровневого программирования.

Недостатки архитектуры CISC:

- Нерегулярность потока команд.
- Высокая стоимость аппаратной части.
- Сложности с распараллеливанием вычислений.

RISC (англ. reduced instruction set computer) — компьютер с набором коротких (простых, быстрых) команд) — архитектура процессора, в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. Первые RISC-процессоры даже не имели инструкций умножения и деления.

Это также облегчает повышение тактовой частоты и делает более эффективной **суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками).**

В середине 1970-х разные исследователи (в частности, из IBM) показали, что большинство комбинаций инструкций и ортогональных методов адресации не использовалось в большинстве программ, порождаемых компиляторами того времени. Также было обнаружено, что в некоторых архитектурах с **микрокодной реализацией** сложные операции зачастую были медленнее последовательности более простых операций, выполняющих те же действия. Это было вызвано, в частности, тем, что многие архитектуры разрабатывались в спешке и хорошо оптимизировался **микрокод только тех инструкций, которые использовались чаще.**

Поскольку многие реальные программы тратят большинство своего времени на выполнение простых операций, многие исследователи решили сфокусироваться на том, чтобы сделать эти операции максимально быстрыми. Производительность процессора ограничена временем, которое процессор тратит на выполнение наиболее медленных шагов в процессе обработки любой инструкции; уменьшение длительности таких шагов даёт общее повышение производительности, а также зачастую ускоряет выполнение инструкций за счёт более эффективной конвейеризации.[4] Фокусирование на простых инструкциях и ведёт к архитектуре RISC, цель которой — сделать инструкции настолько простыми, чтобы они легко конвейеризировались и тратили не более одного такта на каждом шаге конвейера на высоких частотах.

Позднее было отмечено, что наиболее значимая характеристика RISC в разделении инструкций для обработки данных и обращения к памяти — обращение к памяти идёт только через инструкции **load и store (загрузка и хранение)**, а все прочие инструкции ограничены внутренними регистрами.

Это упростило архитектуру процессоров: позволило инструкциям иметь фиксированную длину, упростило конвейеры и изолировало логику, имеющую дело с задержками при доступе к памяти, только в двух инструкциях. В итоге RISC-архитектуры стали называть также архитектурами load/store.

Характерные особенности RISC-процессоров

Фиксированная длина машинных инструкций (например, 32 бита) и простой формат команды.

Специализированные команды для операций с памятью — чтения или записи. Операции вида Read-Modify-Write («прочитать-изменить-записать») отсутствуют. Любые операции «изменить» выполняются только над содержимым регистров (т. н. архитектура load-and-store).

Большое количество регистров общего назначения (32 и более).

Отсутствие поддержки операций вида «изменить» над укороченными типами данных — байт, 16-разрядное слово. Так, например, система команд DEC Alpha содержала только операции над 64-разрядными словами, и требовала разработки и последующего вызова процедур для выполнения операций над байтами, 16- и 32-разрядными словами.

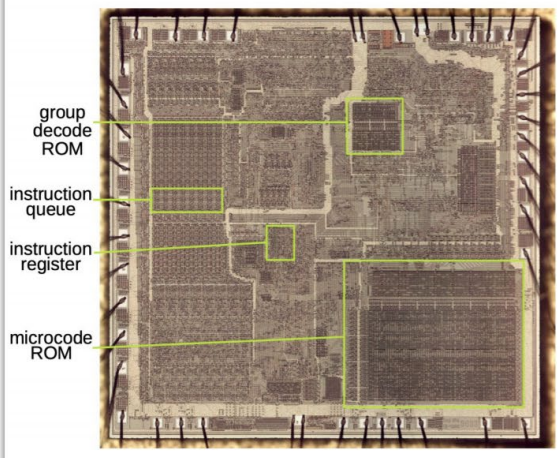
Отсутствие микропрограмм внутри самого процессора. То, что в CISC-процессоре исполняется микропрограммами, в RISC-процессоре исполняется как обыкновенный (хотя и помещённый в специальное хранилище) машинный код, не отличающийся принципиально от кода ядра ОС и приложений. Так, например, обработка отказов страниц в **DEC Alpha (64-разрядный микропроцессор класса RISC, первоначально разработанный и произведённый компанией DEC, которая использовала его в собственной линейке рабочих станций и серверов)** и интерпретация таблиц страниц содержалась в так называемом **PALcode (Privileged Architecture Library)**, помещённом в **ПЗУ.**

Заменой PALCode можно было превратить процессор Alpha из 64-разрядного в 32-разрядный, а также изменить порядок байтов в слове и формат входов таблиц страниц виртуальной памяти.

Современные RISC-архитектуры: состояние и технологии (презентационный материал, 36 слайдов)

Андрей Белеванцев
ИСП РАН им. В.П. Иванникова
8 сентября 2020 г.

RISC внутри CISC



- Фотография процессора 8086 под микроскопом (~29,000 транзисторов)
  - http://www.righto.com/2020/08/latches-inside-reverse-engineering.html
- Микрокод 8086
  - https://patents.google.com/patent/US4363091
  - https://www.reenigne.org/blog/8086-microcode-disassembled/
  - 1-8 микроинструкций на инструкцию
  - Все, кроме самых простых, реализованы в микрокоде
- Pentium и далее
  - Out-of-order, register renaming
  - Возможности обновления

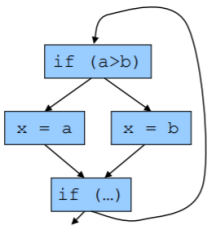
7

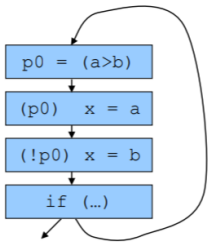
ISP RAS

ARM – Advanced (Acorn) RISC Machine

Преобразование в условную форму

- Команды условного перехода заменяются командами условного выполнения
- Количество ветвлений сокращается, это позволяет:
  - Избежать сбоя конвейера при неправильном предсказании перехода
  - Выполнять конвейеризацию циклов





13

URISC (от англ. Ultimate RISC, также OISC — англ. one instruction set computer) — **теоретическая архитектура процессора, набор команд в которой поддерживает только одну-единственную инструкцию, при этом обеспечивает полноту по Тьюрингу; предельный случай RISC.**

Полнота по Тьюрингу — характеристика исполнителя (множества вычисляющих элементов) в теории вычислимости, означающая возможность реализовать на нём любую вычислимую функцию. Другими словами, для каждой вычислимой функции существует вычисляющий её элемент (например, машина Тьюринга) или программа для исполнителя, а все функции, вычисляемые множеством вычислителей, являются вычислимыми функциями (возможно, при некотором кодировании входных и выходных данных).

Свойство названо по имени Алана Тьюринга, разработавшего абстрактный вычислитель — машину Тьюринга, и давшего определение множества функций, вычислимых посредством машин Тьюринга.

Полнота по Тьюрингу — характеристика исполнителя (множества вычисляющих элементов) в теории вычислимости, означающая возможность реализовать на нём любую вычислимую функцию. Другими словами, для каждой вычислимой функции существует вычисляющий её элемент (например, машина Тьюринга) или программа для исполнителя, а все функции, вычисляемые множеством вычислителей, являются вычислимыми функциями (возможно, при некотором кодировании входных и выходных данных).

Свойство названо по имени Алана Тьюринга, разработавшего абстрактный вычислитель — машину Тьюринга, и давшего определение множества функций, вычислимых посредством машин Тьюринга.

Самый популярный вариант единственной инструкции — «вычесть и пропустить следующую инструкцию, если вычитаемое было больше уменьшаемого» (RSBB — англ. reverse-subtract and skip if borrow); близкий вариант — «вычесть и перейти, если результат не положительный» (SUBLEQ — англ. subtract and branch unless positive).

Также возможен вариант, при котором доступна только одна инструкция — пересылка (MOV), а для выполнения операций используется АЛУ, размещённое в памяти.

Ещё один вариант единственной инструкции — BBJ (**bit-bit jump, BitBitJump**), которая содержит три адреса, копирует один бит из первого по второму адресу памяти и передаёт управление на третий адрес. Поскольку последовательность инструкций может приготовить адрес, на который перейдёт управление (самомодифицирующийся код), BBJ-процессор способен выполнять любые вычисления, которые может выполнить обычный компьютер.

Существуют и другие варианты реализации **URISC.**

С одной стороны писать на Assembler'e под RISC/URISC процессоры не очень-то удобно. Если в лоб сравнивать код, **написанный под CISC и RISC процессоры, очевидно преимущество первого.**

Так выглядит код одной и той же операции для x86 и ARM.

x86 (CISC)

```
MOV AX, 15; AH = 00, AL = 0Fh
AAA; AH = 01, AL = 05
RET
```

ARM (RISC)

```
MOV R3, #10
AND R2, R0, #0xF
CMP R2, R3
IT LT
BLT elsebranch
ADD R2. #6
ADD R1. #1
elsebranch:
END
```

Но так было раньше. Положение ассемблера слегка изменилось. Сейчас за программистов многое делают компиляторы, поэтому никаких сложностей с написанием кода под RISC-процессоры нет. **Зато есть преимущества**.

Представьте, что вы проектируете процессор. Расположение блоков на x86 выглядело бы так (рис.8, цветной вид):

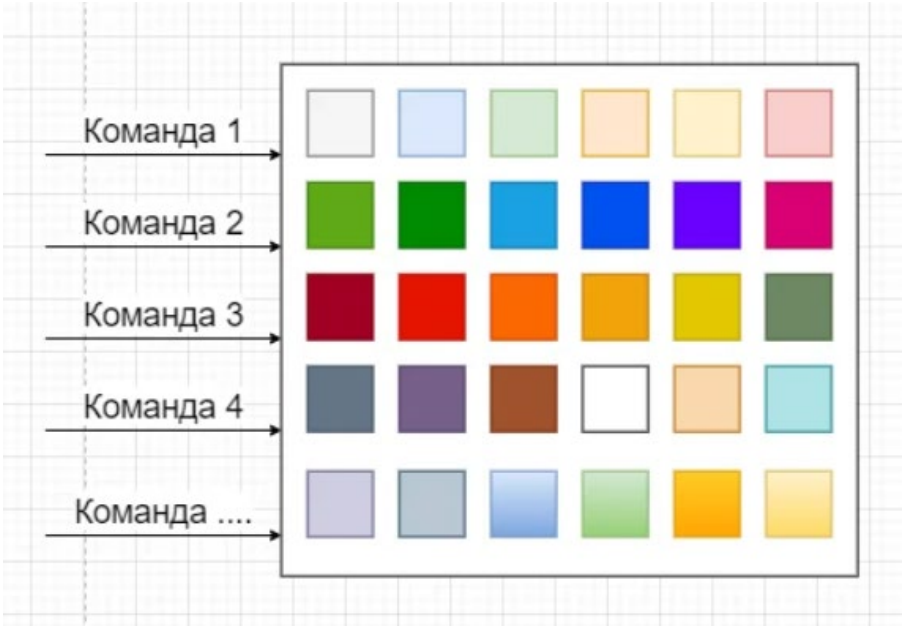


Рис. 8 - Расположение блоков команд на x86-архитектуре

Каждый цветной квадрат - это отдельные команды. Их много, и они разные. Как вы поняли, здесь мы уже говорим про **микроархитектуру, которая вытекает из набора команд**. А вот ARM-процессор скорее выглядит так.

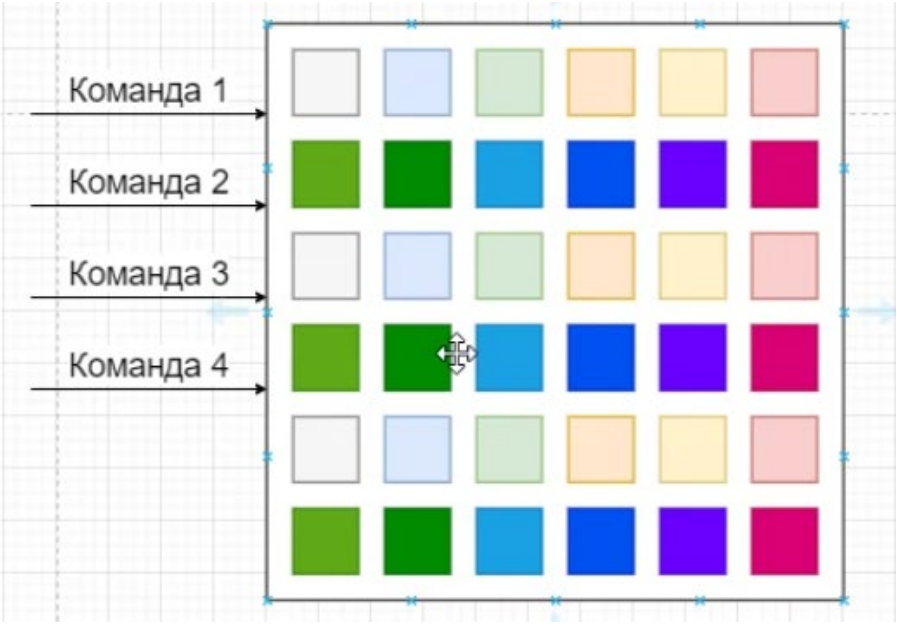


Рис. 9 - Расположение блоков команд на ARM-архитектуре

Ему не нужны блоки, созданные для функций, написанных 30-50 лет назад. По сути, тут блоки только для самых востребованных команд. Зато таких блоков много. А это значит, что можно одновременно выполнять больше базовых команд. А раритетные не занимают место.

Еще один бонус сокращенного набора RISC/URISC: меньше места на чипе занимает блок по декодированию команд. Да, для этого тоже нужно место. Архитектура RISC/ URISC проще и удобнее:

- более простая работа с памятью,
- более богатая регистровая архитектура,
- легче делать 32/64/128 разряды,
- легче оптимизировать,
- меньше энергопотребление,
- проще масштабировать и делать отладку.

Для примера вот два процессора одного поколения. ARM1 и Intel 386. При схожей производительности ARM вдвое меньше по площади. А транзисторов на нем в 10 раз меньше: 25 тысяч против 275 тысяч. Энергопотребление тоже отличается на порядок: 0.1 Ватт против 2 Ватт у Intel. Поэтому наши смартфоны, которые работают на ARM процессорах с архитектурой RISC, не требуют активного охлаждения при весьма большой эффективности работы.

ZISC (англ. Zero Instruction Set Computer — компьютер с нулевым набором команд) — архитектура процессора, основанная на таких технологиях, как сопоставление с образцом. Архитектура характеризуется отсутствием микрокоманд в привычном для микропроцессоров понимании. Концепция основана на идеях, позаимствованных из нейросетей. Для ZISC характерна аппаратная параллельная обработка данных, подобно тому, как это происходит в настоящих нейросетях.

Эта концепция была разработана Гаем Палле (Guy Paillet), вдохновлённым во время совместной работы с командой Карло Руббиа по параллельной обработке, и с Леоном Купером в начале 1990-е над RCE (Restricted Coulomb Energy — модель нейросети, опубликованная Cooper в 1982). RCE было разработано и опубликовано в книге Bruce Batchelor (Cardiff University UK) «Practical Approach to Pattern Classification».

ZISC — это статически запланированная горизонтальная нано—кодированная архитектура (SSHNA). Термин "статически запланированный" означает, что планирование операций и обработка опасностей выполняются компилятором. Термин "горизонтальное нано—кодирование" означает, что NISC не имеет никакого предопределенного набора команд или микрокода. Компилятор генерирует нано-коды, которые непосредственно управляют функциональными блоками, регистрами и мультиплексорами данного канала передачи данных. Предоставление компилятору низкоуровневого управления позволяет лучше использовать ресурсы datapath, что в конечном итоге приводит к повышению производительности. Преимущества технологии NISC заключаются в следующем:

- Более простой контроллер: нет аппаратного планировщика, нет декодера команд.
- Более высокая производительность: более гибкая архитектура, лучшее использование ресурсов
- Легче проектировать: нет необходимости в разработке наборов инструкций.

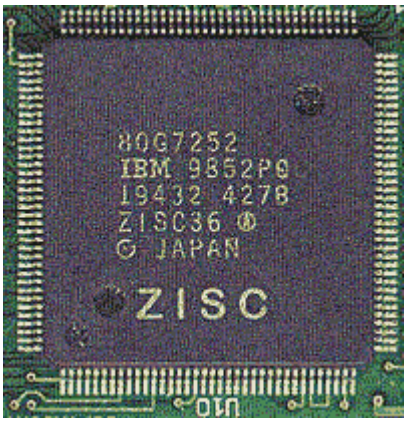
Набор команд и контроллер процессоров являются наиболее трудоемкими и трудоемкими частями для проектирования. Устраняя эти два фактора, проектирование пользовательских элементов обработки становится значительно проще.

Кроме того, datapath процессоров NISC может даже генерироваться автоматически для данного приложения. Поэтому производительность конструктора значительно повышается.

Поскольку ZISC datapaths очень эффективны и могут генерироваться автоматически, технология NISC сопоставима с подходами синтеза высокого уровня (HLS) или C к синтезу HDL. На самом деле одним из преимуществ этого стиля архитектуры является его способность соединять эти две технологии (индивидуальный дизайн процессора и HLS).

По данным TechCrunch, программные эмуляции этих типов чипов в настоящее время используются для распознавания изображений многими крупными технологическими компаниями, такими как Facebook и Google.

При применении к другим различным задачам обнаружения паттернов, таким как работа с текстом, результаты, как говорят, получаются в микросекундах даже с чипами, выпущенными в далеком 2007 году.



Дзюнко Йошида из EE Times сравнила чип NeuroMem с "машиной", машиной, способной предсказывать преступления по сканированию лиц людей, по персоне интереса (телесериал), описывая его как " сердце больших данных", и предвещая реальную эскалацию развития в эпоху массового сбора данных.

TRIPS - это микропроцессорная архитектура, разработанная командой Техасского университета в Остине совместно с IBM, Intelи Sun Microsystems. TRIPS использует архитектуру набора инструкций, предназначенную для легкого разбиения на большие группы инструкций (графиков), которые могут выполняться на независимых элементах обработки. Конструкция собирает связанные данные в графики, пытаясь избежать дорогостоящих операций чтения и записи данных и сохраняя данные в высокоскоростной памяти рядом с обрабатывающими элементами. Прототип процессора TRIPS содержит 16 таких элементов.

Предыстория: компьютерные программы состоят из серии инструкций, хранящихся в памяти. Процессор запускает программу, извлекая эти инструкции из памяти, исследуя их и выполняя действия, которые требует инструкция.

В ранних машинах скорость основной памяти обычно находилась в том же порядке времени, что и основная операция в процессоре. Например, инструкция, которая добавляет два числа, может занять три или четыре цикла инструкций, в то время как извлечение чисел из памяти может занять один или два цикла.

В этих машинах не было никакого «штрафа» за то, что данные находились в основной памяти, и архитектуры наборов команд как правило, они предназначены для обеспечения прямого доступа, например, команда **add** может взять значение из одного места в памяти, добавить его к значению из другого, а затем сохранить результат в третьем месте.

Появление все более быстрых микропроцессоров и дешевого, но более медленного динамического ОЗУ резко изменило это уравнение. В современных машинах извлечение значения из основной памяти может занять эквивалент тысячи циклов. Одним из ключевых достижений в концепции **RISC** стало включение большего количества процессорных регистров чем более ранние конструкции, как правило, несколько десятков, а не два или три. Инструкции, которые раньше предоставлялись ячейкам памяти, были устранены, заменены теми, которые работали только на регистрах. Загрузка этих данных в регистр была явной, необходимо было выполнить отдельное действие загрузки, и результаты явно сохранялись обратно. Можно было бы улучшить производительность, устранив как можно больше этих инструкций памяти. Эта техника быстро достигла своих пределов, и начиная с 1990-х годов современные процессоры добавили все большее количество кэша процессора чтобы увеличить локальное хранилище, хотя кэш работает медленнее регистров.

С конца 1990-х годов прирост производительности был достигнут в основном за счет использования дополнительных "функциональных блоков", которые позволяют некоторым инструкциям выполняться параллельно. Например, две инструкции сложения, работающие с разными данными, могут выполняться одновременно, что эффективно удваивает скорость работы программы. Современные процессоры обычно имеют десятки таких блоков, некоторые для целочисленной математики и булевой логики, некоторые для математики с плавающей запятой, некоторые для длинных слов данных и другие для работы с памятью и другими домашними делами. Однако большинство программ не работают с независимыми данными, а вместо этого используют выходы одного вычисления в качестве входных данных для другого. Это ограничивает набор инструкций, которые могут выполняться параллельно с некоторым фактором, основанным на том, сколько инструкций процессор способен изучить на лету. **Уровень параллелизма обучения быстро поднялся к середине 2010-х годов.**

Одной из попыток выйти за этот предел является концепция очень длинного слова инструкции (VLIW). VLIW передает задачу поиска параллелизма команд компилятору, удаляя его из самого процессора. Теоретически это позволяет исследовать всю программу на наличие независимых инструкций, которые затем могут быть отправлены в процессор в том порядке, который позволит максимально использовать функциональные блоки. Однако на практике это оказалось непросто, и процессоры VLIW не стали широко популярными.

Даже в случае с VLIW, другая проблема выросла, чтобы стать проблемой. Во всех традиционных конструкциях данные и инструкции обрабатываются различными частями процессора. Когда скорость обработки была низкой, это не вызывало проблем, но по мере увеличения производительности время передачи данных с одной стороны чипа (регистры) на другую (функциональные блоки) растет и составляет значительную часть общего времени обработки. Для дальнейшего повышения производительности регистры должны быть распределены ближе к их функциональным единицам.

TRIPS - это процессор, основанный на концепции явного выполнения графа данных (EDGE). EDGE пытается обойти некоторые узкие места производительности, которые стали доминировать в современных системах.

EDGE основан на том, что процессор способен лучше понимать поток команд, посылаемых ему, не рассматривая его как линейный поток отдельных инструкций, а скорее блоки инструкций, связанных с одной задачей с использованием изолированных данных. EDGE пытается выполнить все эти инструкции в виде блока, распределяя их внутри вместе с любыми данными, которые им необходимо обработать. Компиляторы изучают код и находят блоки кода, которые определенным образом обмениваются информацией.

Затем они собираются в скомпилированные "гипер-блоки" и подаются в центральный процессор. Поскольку компилятор гарантирует, что эти блоки имеют определенные взаимозависимости между собой, процессор может изолировать код в одном функциональном блоке с собственной локальной памятью.

Рассмотрим простой пример, который добавляет два числа из памяти, а затем добавляет этот результат к другому значению в памяти. В этом случае традиционный процессор должен был бы заметить зависимость и запланировать выполнение инструкций одну за другой, сохраняя промежуточные результаты в регистрах. В пограничном процессоре взаимозависимости между данными в коде будут замечены компилятором, который скомпилирует эти инструкции в единый блок. Этот блок будет затем накормлен вместе со всеми данные, которые он должен был заполнить, в единый функциональный блок и свой собственный частный набор регистров. Это гарантирует, что не требуется никакой дополнительной выборки памяти, а также сохранение регистров физически близко к функциональному блоку, который нуждается в этих значениях.

Код, не зависящий от этих промежуточных данных, будет скомпилирован в отдельные гипер-блоки. Конечно, вполне возможно, что вся программа будет использовать одни и те же данные, поэтому компиляторы также ищут случаи, когда данные передаются другому коду, а затем эффективно отбрасываются исходным блоком, что является общим шаблоном доступа. В этом случае компилятор все равно создаст два отдельных гипер-блока, но явно закодирует передачу данных, а не просто оставит их храниться в какой-то общей ячейке памяти. При этом процессор может "видеть" эти коммуникационные события и планировать их выполнение в надлежащем порядке. Блоки, которые имеют значительные взаимозависимости, перестраиваются компилятором для распространения коммуникаций, чтобы избежать узких мест при транспортировке.

Результатом этого изменения является значительное увеличение изолированности отдельных функциональных единиц. **Процессоры EDGE ограничены в параллелизме возможностями компилятора, а не систем на кристалле.** В то время как современные процессоры достигают плато при четырех-широтном параллелизме, EDGE-конструкции могут масштабироваться гораздо шире. Они также могут масштабироваться "глубже", передавая блоки от одной единицы к другой в цепочке, которая запланирована для уменьшения конкуренции из-за общих ценностей.

Техасский университет в Остине реализует концепцию EDGE- как процессор **TRIPS**, в системе Тера-ОП, которая обозначается как надежная, интеллектуально-адаптивная система обработки данных. Процессор TRIPS строится путем повторения одного базового функционального блока столько раз, сколько необходимо. Использование в конструкции TRIPS гиперблоков, загружаемых в массовом порядке, позволяет значительно увеличить спекулятивное исполнение

В то время как традиционный дизайн архитектуры может иметь несколько сотен инструкций для изучения возможного планирования в функциональных блоках, дизайн TRIPS имеет тысячи, сотни инструкций на гиперблок и сотни гиперблоков, которые рассматриваются. Это приводит к значительному улучшению использования функционального блока; масштабируя его производительность до типичной четырехэтапной суперскалярной конструкции, **TRIPS может обрабатывать примерно в три раза больше инструкций за цикл.**

В традиционных конструкциях существует множество различных типов единиц измерения, целых чисел, плавающей запятой и т. д., что позволяет больше параллелизма, чем в противном случае позволили бы планировщики. Однако для того, чтобы все блоки оставались активными, поток команд должен включать все эти различные типы команд. Поскольку на практике это часто не так, традиционные процессоры часто имеют много бездействующих функциональных блоков. В поездках отдельные блоки имеют общее назначение, позволяя любой инструкции работать на любом ядре. Это не только позволяет избежать необходимости тщательно сбалансировать количество различных типов ядер, но также означает, что конструкция TRIPS может быть построена с любым количеством ядер, необходимых для достижения определенных требований к производительности. Одноядерный процессор TRIPS с упрощенным (или исключенным) планировщиком будет запускать набор гипер-блоков точно так же, как один с сотнями ядер, только медленнее.

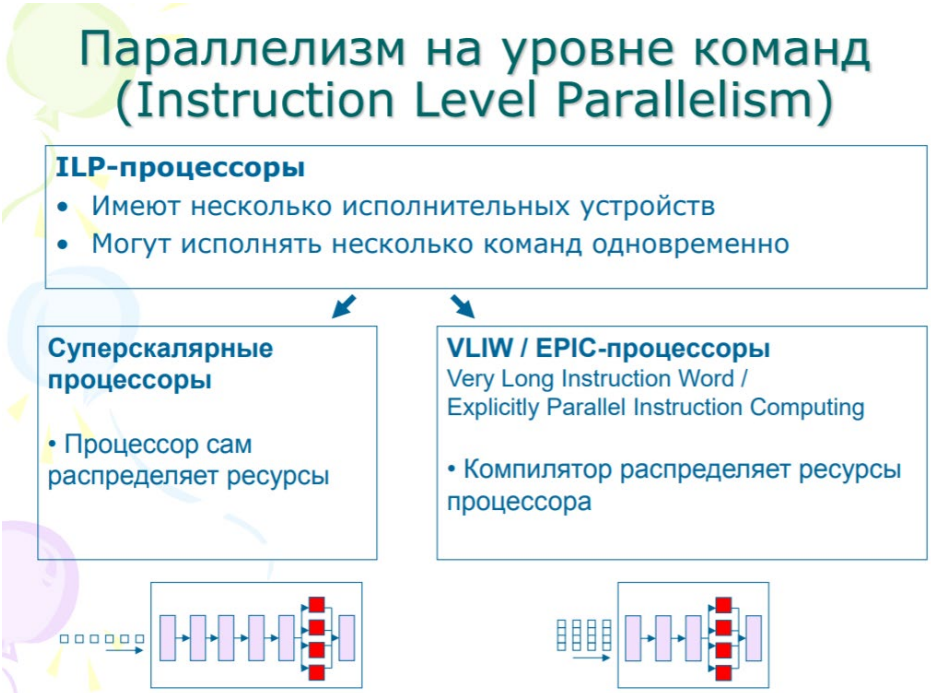
Еще лучше то, что производительность не зависит от типов подаваемых данных, а это означает, что процессор TRIPS будет выполнять гораздо более широкий спектр задач с одинаковой производительностью.

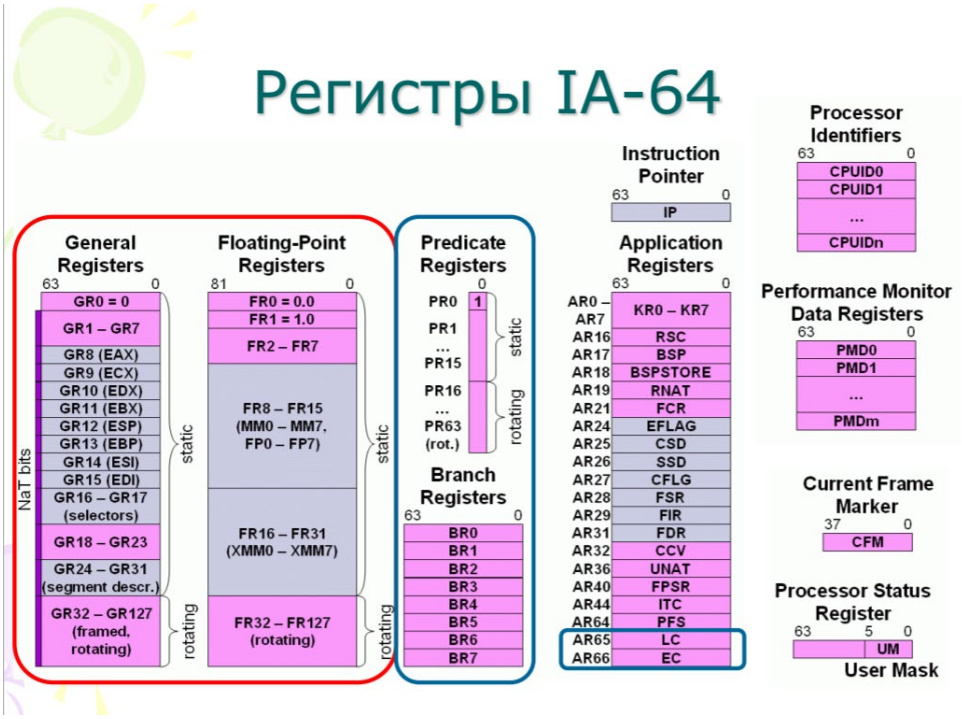
В процессоре TRIPS каждый функциональный блок будет добавлять к производительности каждой задачи, потому что каждая задача может выполняться на каждом блоке. Конструкторы называют его "полиморфным процессором". TRIPS настолько гибок в этом отношении, что разработчики предположили, что он даже заменит некоторые пользовательские высокоскоростные конструкции, такие как DSPs. Как и TRIPS, DSP (цифровые сигнальные процессоры) получают дополнительную производительность, ограничивая взаимозависимости данных, но в отличие от TRIPS они делают это, позволяя только очень ограниченному рабочему процессу работать на них.

Они были бы такими же быстрыми, как пользовательский DSP на этих рабочих нагрузках, но одинаково способными запускать другие рабочие нагрузки в то же время. Как отметили архитекторы, маловероятно, что процессор TRIPS может быть использован для замены сильно настроенных конструкций, таких как графические процессоры в современных видеокартах но они могут заменить или превзойти многие чипы с более низкой производительностью, такие как те, которые используются для обработки мультимедиа. Сокращение размера привилегированного регистра также приводит к неочевидным выгодам. Добавление новых схем к современным процессорам привело к тому, что их общий размер остался примерно таким же, даже когда они перешли на меньшие размеры процесса. В результате относительное расстояние до файла регистра увеличилось, и это ограничивает возможную скорость цикла из-за задержек связи. В EDGE данные обычно более локальны или изолированы в четко определенных межъядерных каналах, что исключает большие задержки "кросс-чипа". Это означает, что отдельные ядра могут работать на более высоких скоростях, ограниченных временем передачи сигналов гораздо более коротких путей передачи данных.

Сочетание этих двух эффектов изменения конструкции значительно повышает производительность системы.

Архитектура VLIW/EPIC: IA-64 и другие реализации
(презентационный материал, 73 слайда)





EPIC (англ. explicitly parallel instruction computing — «вычисление с явным параллелизмом машинных команд») — класс микропроцессорных архитектур с явным параллелизмом команд. Термин введён в 1997 году альянсом HP и Intel для разрабатываемой архитектуры Intel Itanium.

EPIC позволяет микропроцессору выполнять инструкции параллельно, опираясь на информацию от компилятора, а не выявляя возможность параллельной работы инструкций при помощи специальных схем во время исполнения. В теории, это могло упростить масштабирование вычислительной мощности процессора без увеличения тактовой частоты.

Цели разработки EPIC:

- удаление из процессора планировщика инструкций;
- увеличение количества инструкций, которые процессор способен выполнять одновременно (англ. instruction level parallelism — параллелизм инструкций).

Планировщик инструкций — устройство со сложной логикой, входящее в состав процессора и предназначенное для определения порядка выполнения инструкций. Удаление планировщика инструкций позволило освободить место внутри процессора для других устройств (например, для АЛУ). Функции планировщика инструкций были возложены на компилятор.

Увеличение степени параллелизма инструкций достигается использованием возможностей компилятора по поиску независимых команд.

Архитектуры VLIW в своей изначальной форме имели несколько недостатков, препятствующих их массовому внедрению: наборы инструкций VLIW не являлись совместимыми между различными поколениями процессоров (программа, скомпилированная для процессора, содержащего больше исполнительных устройств (например, больше АЛУ), не могла выполняться процессором, содержащим меньшее количество устройств); задержки загрузки данных из иерархии памяти (кэшей, DRAM) не являлись полностью предсказуемыми (из-за этого усложнялась реализация статического планирования статического планирования инструкций загрузки и использования данных).

Архитектура EPIC имеет следующие особенности для устранения недостатков VLIW:

Каждая группа из нескольких инструкций называется бандлом (bundle). Каждый бандл может иметь **стоповый бит**, обозначающий, что следующая группа зависит от результатов работы данной. Такой бит позволяет создавать будущие поколения архитектуры с возможностью параллельного запуска большего числа бандлов. Информация о зависимостях вычисляется компилятором, и поэтому аппаратуре не придётся проводить дополнительную проверку независимости операндов.

Для предподкачки данных используется инструкция программной подкачки (**software prefetch**). **Предподкачка увеличивает вероятность того, что к моменту исполнения команды загрузки данные уже будут в кэше**. Также в этой инструкции могут быть дополнительные указания для выбора различных уровней кэша для данных.

Инструкция спекулятивной загрузки используется для загрузки данных до того, как станет известно, будут ли они использованы (**bypassing control dependencies**), или будут они изменены перед использованием (**bypassing data dependencies**).

Инструкции проверки загрузки (**check load instruction**) помогают инструкциям спекулятивной загрузки при помощи проверок, зависела ли инструкция загрузки от последующей записи. В случае наличия подобной зависимости спекулятивная загрузка должна быть повторена.

Архитектура EPIC также включает в себя несколько концепций (grab-bag) для увеличения ILP (параллелизма инструкций):

Предсказание ветвлений используется, чтобы снизить частоту переходов и для увеличения спекулятивности исполнения[en] инструкций. В последнем случае условное ветвление преобразуется в заполнение предикатных регистров, затем выполняются обе ветви. Результат той ветви, которая не должна была выполняться, отменяется по значению предикатного регистра.

Отложенные исключительные ситуации, использующие бит Not a thing в регистрах общего назначения. Они позволяют продолжать спекулятивное исполнение даже после исключительных ситуаций.

Таксономия Флинна: SISD SIMD MISD MIMD

По-видимому, самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М.Флинном. Классификация базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

В 1984 году Ванг и Бриггс сделали некоторые дополнения к классификации Флинна, конкретизировав классы SISD, SIMD, MIMD.

Всё разнообразие архитектур ЭВМ в этой таксономии Флинна сводится к четырём классам:

ОКОД — Вычислительная система с одиночным потоком команд и одиночным потоком данных (**SISD**, single instruction stream over a single data stream).

ОКМД — Вычислительная система с одиночным потоком команд и множественным потоком данных (**SIMD**, single instruction, multiple data).

МКОД — Вычислительная система со множественным потоком команд и одиночным потоком данных (**MISD**, multiple instruction, single data).

МКМД — Вычислительная система со множественным потоком команд и множественным потоком данных (**MIMD**, multiple instruction, multiple data).

	Одиночный поток команд (single instruction)	**Множество потоков команд** (multiple instruction)
Одиночный поток данных (single data)	SISD (ОКОД)	MISD (МКОД)
Множество потоков данных (multiple data)	SIMD (ОКМД)	MIMD (МКМД)

Рис. 10 - Классификация по Флинну

Поскольку в таксономии в качестве основного критерия используют параллелизм, таксономию Флинна чаще всего упоминают в технической литературе при классификации параллельных вычислительных систем. В большинстве случаев параллельные вычислительные системы относятся к SIMD или MIMD-классам. При этом SISD-машина параллельной не считается, а существование ЭВМ с архитектурой MISD под большим вопросом, потому что даже главный пример (систолический массив) нельзя строго рассматривать как подходящий под определение данного типа.

Чтобы лучше понять, что такое классификация Флинна, что отличает SIMD и MIMD, а также SISD-тип от рассматриваемой в данной статье MISD-архитектуры, ниже будет представлена некоторая характеристика этих категорий. Стоит также отметить, что наиболее популярные классы (**SIMD и MIMD**) делятся по способу работы с памятью с точки зрения программирования. Таким образом, выделяют системы с общей памятью (англ. shared memory, SM) и с распределённой (англ. distributed memory, DM).

Оговорка «с точки зрения программиста» обусловлена тем, что есть вычислительные системы, где память физически распределена по узлам системам, тем не менее, для всех процессоров системы она вся видна как общее единое глобальное адресное пространство.

Так как в таксономии в качестве основного критерия используется параллелизм, то таксономия Флинна наиболее часто упоминается в технической литературе при классификации параллельных вычислительных систем. **MISD** — редко используемая архитектура, в основном с целью защиты от сбоев (например, для горячего резервирования вычислительных машин в лётных системах типа «Space Shuttle» или «Энергия — Буран», в SCADA, критичных к сбоям, и т. п.). Поскольку **SISD**-машина параллельной машиной не является, а MISD не является типичной параллельной архитектурой, все параллельные вычислительные системы попадают в класс либо SIMD, либо в MIMD.

С развитием технологий классы SIMD и MIMD стали охватывать слишком большой круг машин, кардинально отличных друг от друга. В связи с этим в технической литературе используется дополнительный критерий — способ работы с памятью с «точки зрения программиста». По этому критерию системы делятся на «системы с общей памятью» (англ. shared memory, SM) и «системы с распределенной памятью» (англ. distributed memory, DM).

Соответственно, каждый класс — SIMD и MIMD — делится на подклассы: SM-SIMD/DM-SIMD и SM-MIMD/DM-MIMD.

Следует обратить особое внимание на уточнение «с точки зрения программиста». Дело в том, что существуют вычислительные системы, где память физически распределена по узлам системы, но для всех процессоров системы она вся видна как общее единое глобальное адресное пространство. Подробнее об этом ниже:

Архитектура SISD — это традиционный компьютер фон-Неймановской архитектуры с одним процессором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. В данном классе не используется параллелизм ни данных, ни инструкций, и, следовательно, **SISD-машина не является параллельной**. К этому классу также принято относить конвейерные, суперскалярные и VLIW-процессоры. На рис.11 и так далее «БО» – блок обработки:

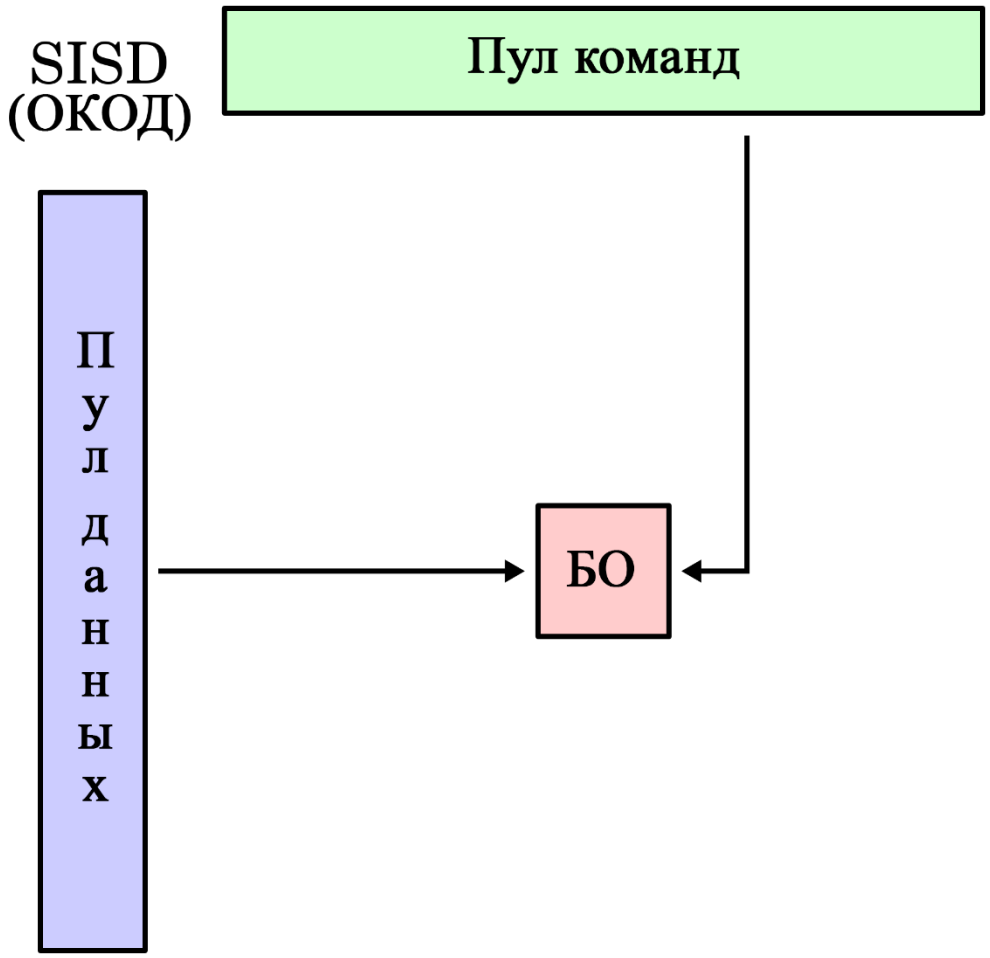


Рис. 11 - SISD

Типичными представителями SIMD являются векторные процессоры, обычные современные процессоры, когда работают в режиме выполнения команд векторных расширений, а также особый подвид с большим количеством процессоров — матричные процессоры. В SIMD-машинах один процессор загружает одну инструкцию, набор данных к ним и выполняет операцию, описанную в этой инструкции, над всем набором данных одновременно.

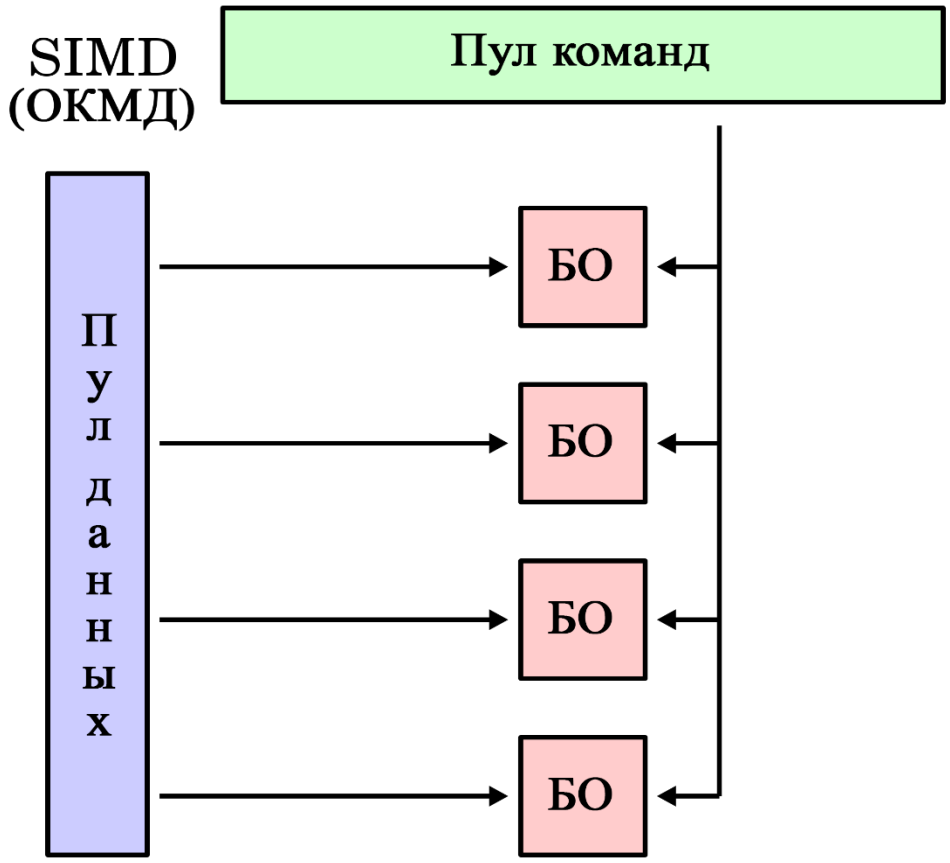


Рис. 12 - SIMD

Внутренняя классификация:

SM-SIMD (shared memory SIMD) — подкласс SIMD с общей памятью с точки зрения программиста. Сюда относятся векторные процессоры.

DM-SIMD (distributed memory SIMD) — подкласс SIMD с распределённой памятью с точки зрения программиста. Сюда относятся матричные процессоры — особый подвид с большим количеством процессоров.

MISD — по сути, гипотетический класс, поскольку реальных систем-представителей данного типа пока не существует. Некоторые исследователи относят к нему конвейерные ЭВМ.

MIMD (англ. Multiple Instruction stream, Multiple Data stream, множественный поток команд, множественный поток данных, МКМД) — ещё один распространённый тип параллельных ЭВМ. Включает в себя многопроцессорные системы, где процессоры обрабатывают множественные потоки данных. Сюда, как правило, относят традиционные мультипроцессорные машины, многоядерные и многопоточные процессоры, а также компьютерные кластеры.

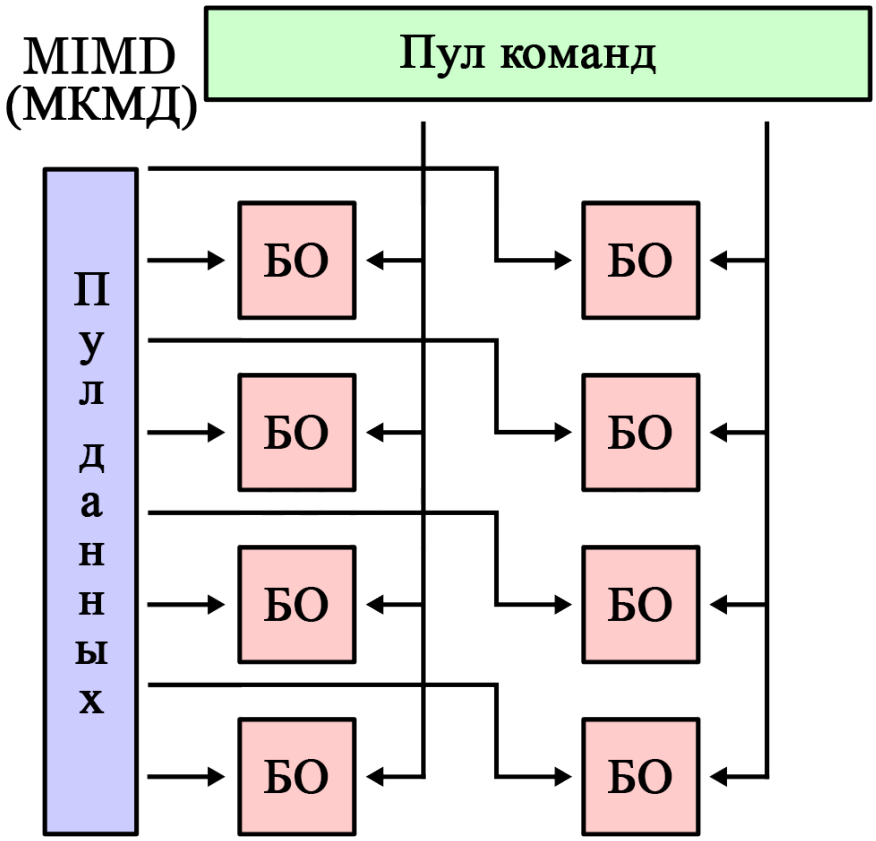


Рис. 13 – MIMD

Внутренняя классификация:

SM-SIMD (shared memory SIMD)

К этому подклассу относятся векторные процессоры.

В научных вычислениях большая часть операций связана с применением какой-то одной операции к большому массиву данных.

Причем эту операцию можно осуществлять над каждым элементом данных независимо друг от друга, то есть присутствовал параллелизм данных, для использования которого и были созданы векторные процессоры.

Векторные процессоры получили распространение в начале 70-х годов, в первую очередь в суперкомпьютерах тех времен (CDC STAR-100, Cray-1). С середины 70-х до конца 80-х все суперкомпьютеры были векторными машинами, и под суперкомпьютером в те годы подразумевалась векторная машина. Векторные суперкомпьютеры до сих пор находят применение в промышленности и научных вычислениях, и они до сих пор входят в перечень продукции почти всех ведущих производителей суперкомпьютеров: NEC, Fujitsu, Hitachi, Cray. Развитие миниатюризации в вычислительной технике позволило добавить векторный способ обработки данных в современные микропроцессоры, где они представлены набором специальных команд-расширений ассемблера. Выполняя их, процессор переходит в векторный режим и превращается на это время в SM-SIMD-машину.

DM-SIMD (distributed memory SIMD)

К этому подклассу относятся так называемые «матричные процессоры». Они представляют собой массив процессоров, которые контролируются одним управляющим процессором, выполняя по его команде одну операцию над своей собственной порцией данных, хранящихся в локальной памяти. Так как обмена данными между процессорами нет, не требуется никакой синхронизации, что позволяет достигать огромных скоростей вычислений и с легкостью расширять систему, просто увеличивая количество процессоров. Для понимания работы матричного процессора достаточно представить себе утренние телевизионные уроки по аэробике, где актёр в студии задает движения, а миллионы телезрителей повторяют их в такт одновременно по всей стране.

Так как матричные процессоры можно использовать только на ограниченном круге задач, долгое время они существовали только в виде экспериментальных, узкоспециализированных машин. Кроме того, для их производства требовалось создавать специализированные процессоры.

Первой попыткой (довольно неудачной) построить матричный процессор был компьютер ILLIAC IV в начале 70-х годов, второй знаменитой попыткой были машины CM-1 и CM-2 компании Thinking Machines и суперкомпьютеры компании MasPar в начале 80-х.

Развитие миниатюризации в вычислительной технике позволило вернуться к идее матричных процессоров и возродить её в графических картах (GPGPU), которые используются для высокопроизводительных вычислений.

К классу **MISD** ряд исследователей относит конвейерные ЭВМ, однако это не нашло окончательного признания. Также, возможно считать MISD системами, системы с «горячим» резервированием. На рис. 14 указан «PU» - блок обработки.

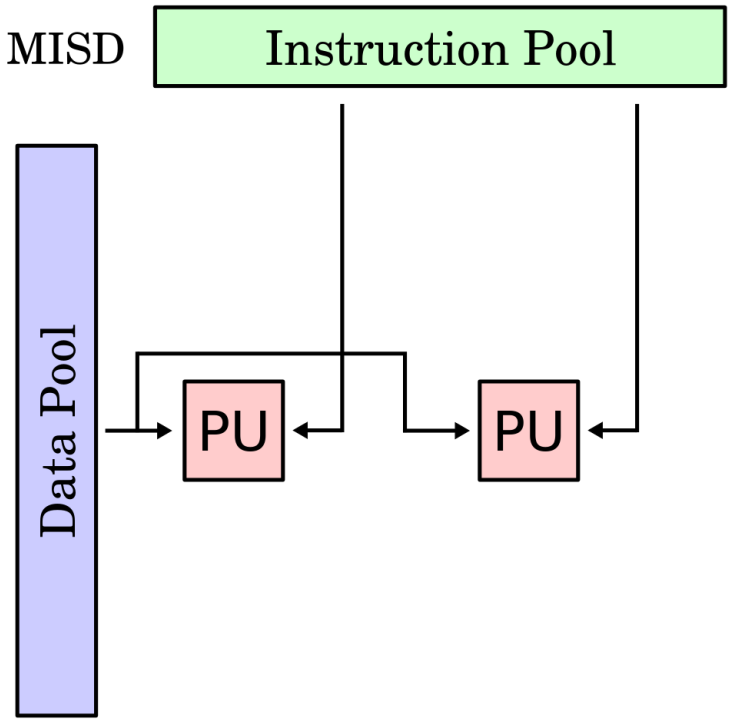


Рис. 14 – MISD

Внутренняя классификация:

SM-MIMD (shared memory MIMD) — подкласс MIMD с общей памятью с точки зрения программиста. К нему относятся мультипроцессорные машины и многоядерные процессоры с общей памятью. Мультипроцессоры легко программировать, поддержка SMP (симметричной мультипроцессорности) давно присутствует во всех основных операционных системах. Однако у таких ЭВМ невысокая масштабируемость: увеличение процессоров в системе чревато высокой нагрузкой на общую шину.

Существует так называемый **DSM-MIMD** (distributed shared memory MIMD) подкласс.

То есть программисту память видна как одно общее адресное пространство, но физически она может быть распределена по узлам системы. В этом подклассе у каждого процессора есть своя локальная память, а к другим участкам памяти процессор обращается через высокоскоростное соединение. Поскольку доступ к разным участкам общей памяти неодинаков, такие системы называются NUMA (англ. Non-Uniform Memory Access). Возникает проблема: надо, чтобы процессор видел в памяти изменения, сделанные другими процессорами. Для её решения возникли ccNUMA (через согласование кэша) и nccNUMA (без согласования кэша). NUMA-системы имеют более высокую масштабируемость, по сравнению с мультипроцессорами, что позволяет создавать массово-параллельные вычислительные системы, где число процессоров достигает нескольких тысяч.

DM-MIMD (distributed memory MIMD) — подкласс MIMD с распределённой памятью с точки зрения программиста. Сюда относятся многопроцессорные ЭВМ с распределённой памятью и компьютерные кластеры типа Beowulf как Network of Workstations. Локальная память отдельно взятого процессора не видна другим. У каждого процессора своя задача. Если же ему необходимы данные из памяти другого процессора, он обменивается с ним сообщениями. У таких машин высокая масштабируемость, как у NUMA.

SPMD (англ. Single Program Multiple Data, одиночная программа, множество данных, ОПМД) — система, где на всех процессорах MIMD-машины выполняется только одна программа, и на каждом процессоре она обрабатывает разные блоки данных.

MPMD (англ. Multiple Programs, Multiple Data, множество программ, множество данных, МПМД) — система, где

а) на одном процессоре MIMD-машины работает мастер-программа, а на других — подчинённая программа, чьей работой руководит мастер-программа (принцип master / slave или master / worker);

б) на разных узлах MIMD-машины работают разные программы, которые по-разному обрабатывают один и тот же массив данных (принцип coupled analysis), большей частью они работают независимо друг от друга, но время от времени обмениваются данными для перехода к следующему шагу.

Практикум: Р-алгоритм крупноблочного распараллеливания сложных задач.

Структуры **параллельных алгоритмов** (р-программ) определяются графами информационных и управляющих связей, вершинам которых сопоставлены операторы ветвей, а ребрам информационные и управляющие связи между операторами. Анализ прямых и итерационных методов вычислительной математики показывает, что в их основе лежат, как правило, операции над матрицами и векторами данных.

Проиллюстрируем методику крупноблочного распараллеливания на примере умножения матриц больших размеров. Требуется построить параллельный алгоритм, вычисляющий произведение двух прямоугольных матриц:

H[1:N;1:L],**A**[1:L;1:M]= **C**[1 : N; 1 : M], или, что, то же самое,

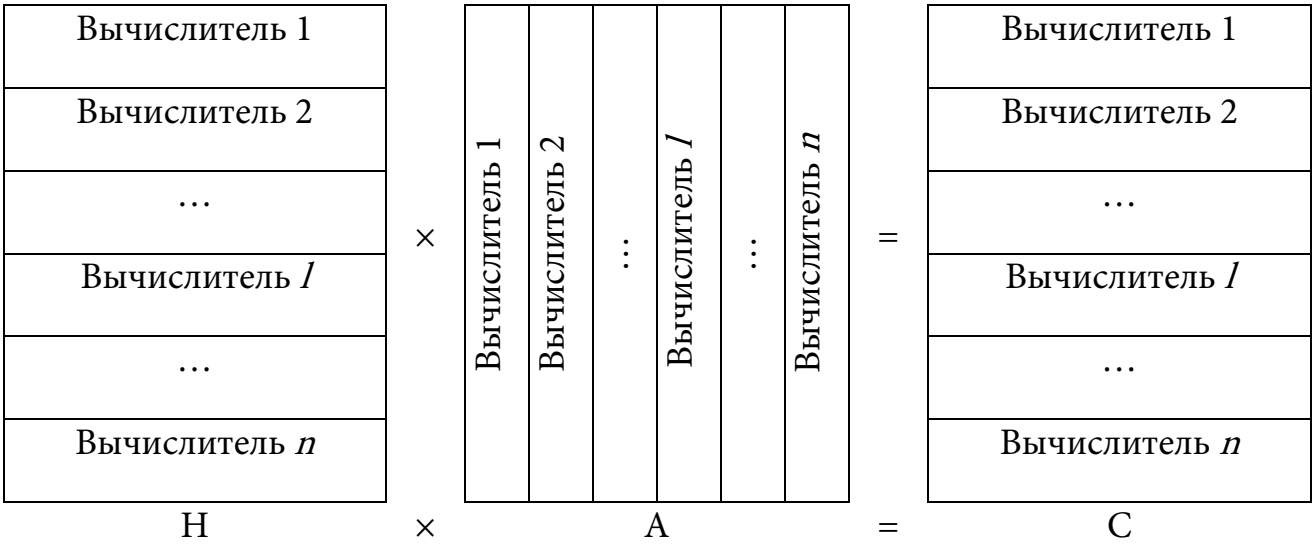
$$\begin{Vmatrix} b_{11} & b_{12} & \cdots & b_{1h} & \cdots & b_{1K} \\ b_{21} & b_{22} & \cdots & b_{2h} & \cdots & b_{2K} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ b_{i1} & b_{i2} & \cdots & b_{ih} & \cdots & b_{iK} \\ \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & \cdots & b_{Nh} & \cdots & b_{NK} \end{Vmatrix} \times \begin{Vmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ a_{h1} & a_{h2} & \cdots & a_{hj} & \cdots & a_{hM} \\ \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ a_{K1} & a_{K2} & \cdots & a_{Kj} & \cdots & a_{KM} \end{Vmatrix}$$

$$= \begin{Vmatrix} c_{11} & c_{12} & \cdots & c_{1j} & \cdots & c_{1M} \\ c_{21} & c_{22} & \cdots & c_{2j} & \cdots & c_{2M} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{iM} \\ \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ c_{N1} & c_{N2} & \cdots & c_{Nj} & \cdots & c_{NM} \end{Vmatrix}$$

где элементы матрицы-произведения **C**[1 : N; 1 : M] вычисляются по формуле:

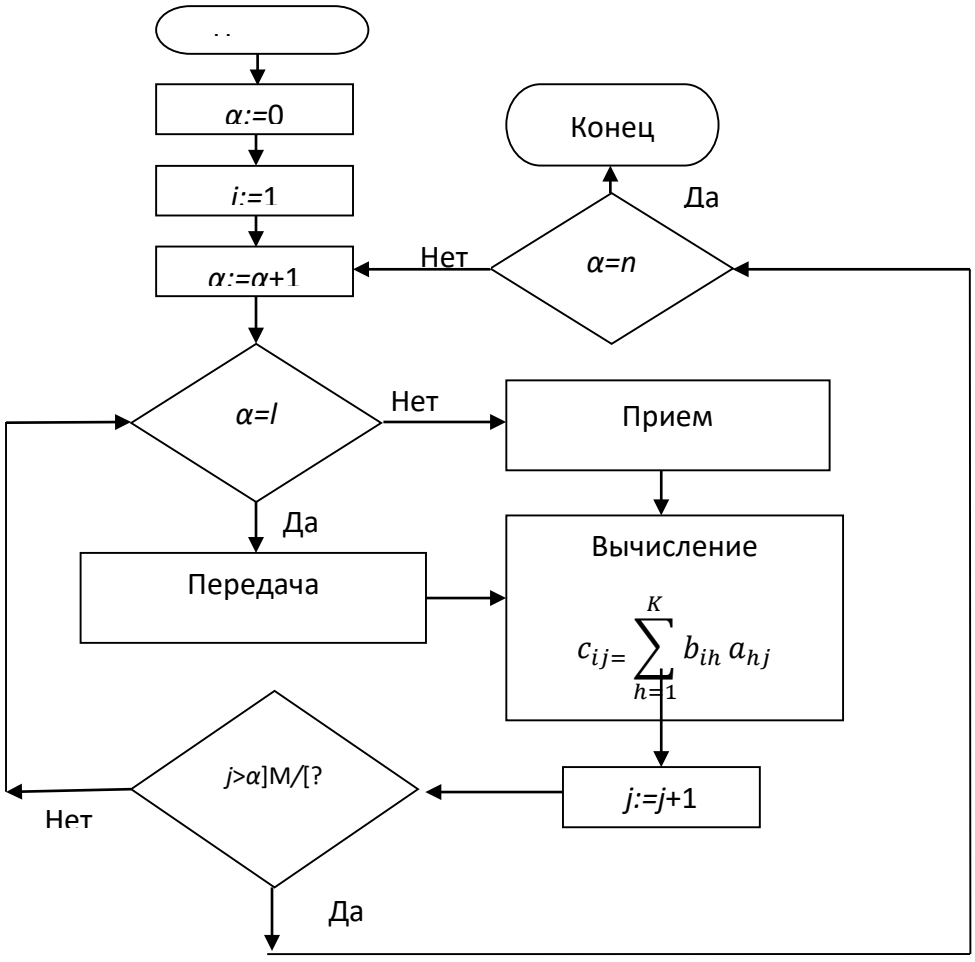
$$c_{ij=} \sum_{h=1}^{K} b_{ih}\, a_{hj}$$

Размеры матриц H и A N×L и L×M достаточно большие => имеют место неравенства N>>n, L>>n, M>>n, где n – это вычислители.



На этом рисунке показано **распределение данных по вычислителям НС.**

Рассмотрим блок-схему р-алгоритма:



α – номер передающего вычислителя, $\{1,2,...,\alpha-1,\ \alpha+1,...,n\}$ – номера принимающих вычислителей.

Показатель накладных расходов $\varepsilon = \frac{t}{T}$, где t – время, которое расходуется на организацию и собственно реализацию обменов информации, Т – время на счет, выполнения арифметических, логических и других операций. Оценим ε для алгоритма умножения матриц. При обмене передаётся строка, состоящая из L элементов матрицы А, после этого каждая машина выполнит $\left\lceil \frac{N}{n} \right\rceil * (L-1)$ операций сложения и $\left\lceil \frac{N}{n} \right\rceil * L$ операций умножения. Так как у нас дана матрица большого размера, т.е. $L>>n$, поэтому можно считать, что на каждый переданный элемент А приходится $\rho = \left\lceil \frac{N}{n} \right\rceil$ сложений и умножений. Пусть t_n – время пересылки одного слова (элемента матрицы); t_y и t_c – время выполнения операций умножения и сложения. Тогда эффективность параллельного алгоритма умножения матриц большого размера можно характеризовать показателями:

$$\varepsilon = \frac{t_n}{\rho(t_y + t_c)} = \frac{e}{\rho}, \qquad e = \frac{t_n}{t_y + t_c}$$

Очевидно, что максимум накладных расходов будет при $\rho = 1$, или, что то же самое, равенство $\varepsilon = e$ достигается при $n = N$. Таким образом, максимум коэффициента ε накладных расходов определяется формулой:

$$\varepsilon = \frac{t_n}{t_y + t_c}, \qquad t_n = \frac{l}{v}$$

Подставляя свои значения **(генерируем в рандомайзере)**

Пример: l=64, v=5 Гигабод, t_c=0,5нс, t_y=1 нс, получаем:

$t_n = l/v = 64 / (0,5)^9 = 12,8$нс,

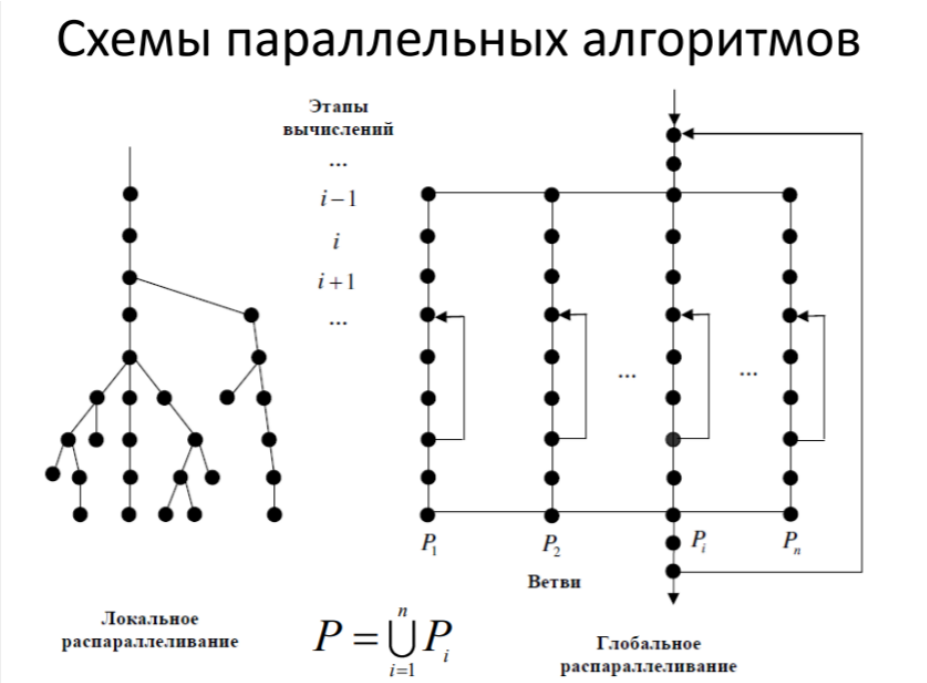
тогда

$\varepsilon = 12,8 / (1 + 0,5) = 8,53$

Ответ: максимум коэффициента накладных расходов ε при реализации р-алгоритма на вычислительной системе составляет 8,53.

В ходе работы были сделаны выводы, что методика крупноблочного распараллеливания сложных задач является основой для понимания параллельных вычислительных технологий и их совершенствования.

Параллельное программирование. Параллельный алгоритм умножения матриц. Показатели эффективности параллельных алгоритмов (презентационный материал, 29 слайдов)



§ РАЗДЕЛ 2: СИСТЕМА НА КРИСТАЛЛЕ (SoC)

Введение в SoC

Термин «система на кристалле» приобрел большую популярность. Изделия этого класса часто анонсируются и воспринимаются как новое направление в развитии электронной техники, призванное вытеснить **«классические» сверхбольшие интегральные схемы (СБИС)**. В данном лекционном материале расскажем о том, что представляют собой реальные СнК (системы на кристалле), и сравнив их достоинства и недостатки.

Система на кристалле (СнК), однокристальная система (англ. System-on-a-Chip, SoC — электронная схема, выполняющая функции целого устройства (например, компьютера) и размещённая на одной интегральной схеме.

Анализируя данные в технической литературе и описания различных изделий, называемых авторами «системами на кристалле», можно сформулировать следующее определение: система на кристалле — это СБИС, интегрирующая на кристалле различные функциональные блоки, которые образуют законченное изделие для автономного применения в электронной аппаратуре.

Структура типовой СнК представлена на рисунке 15:

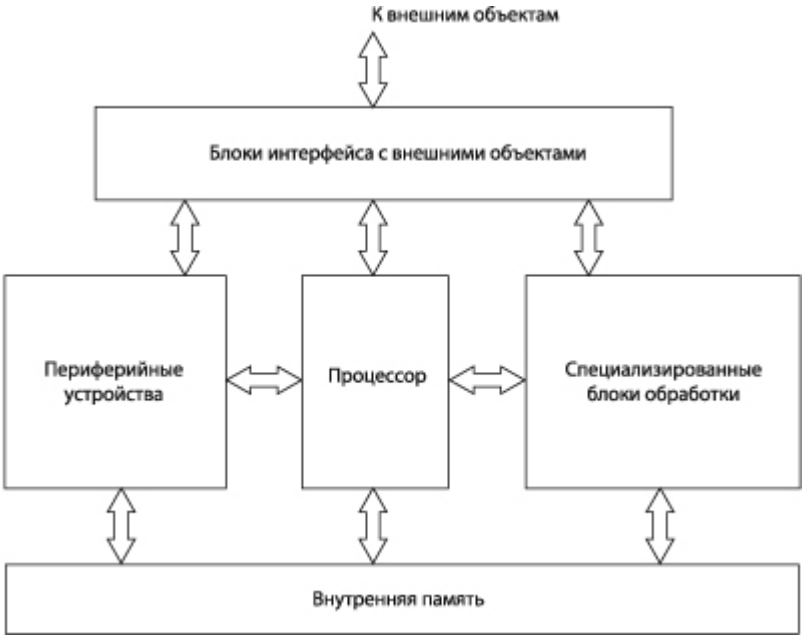


Рис. 15 – Структурная схема СнК в общем виде.

Система на кристалле может включать как цифровые, так и аналоговые блоки. Основным цифровым блоком обычно является процессор, выполняющий программную обработку цифровых данных. Специализированные блоки обработки обеспечивают аппаратное выполнение функций, специфических для данной системы.

Это могут быть, например, блоки цифровой обработки сигналов (DSP), аналоговые схемы, преобразователи потоков данных и др. устройства. Различные типы модулей памяти (SRAM, DRAM, ROM, EEPROM, Flash) могут входить в состав СнК или подключаться к ней как внешние блоки. Таймеры, АЦП и ЦАП, широтно-импульсные модуляторы и другие цифровые устройства могут интегрироваться в состав СнК в качестве периферийных устройств.

Интерфейс с внешними устройствами обеспечивается с помощью параллельных и последовательных портов, различных шинных и коммуникационных контроллеров и других интерфейсных блоков, в т.ч. аналоговых (усилителей, преобразователей). Состав блоков, интегрируемых в конкретной СнК, варьируется в зависимости от ее функционального назначения. Организация связей между блоками системы также может быть различной: возможно использование различных стандартизованных шин (типа AMBA) или специализированных локальных интерфейсов.

Как видно из рисунка 15, структуру СнК составляют в основном те же функциональные блоки, которые входят в состав сложно-функциональных СБИС класса микроконтроллеров и микропроцессоров. Фактически современные СнК отличаются от микроконтроллеров только наличием специализированных блоков обработки данных. В

Выпуск микроконтроллеров (называвшихся прежде однокристальными микроЭВМ) начался в 1981 г. Таким образом, можно считать, что СнК без специализированных блоков обработки производятся и применяются уже более 40 лет.

Особенности проектирования систем на кристалле

В большинстве случаев СнК **представляет собой цифровую СБИС, которая может также содержать ряд аналоговых блоков**. Поэтому для проектирования СнК используются те же **методы и средства, что и для СБИС**.

Эти средства реализованы в виде систем автоматизированного проектирования (САПР), поставляемых компаниями Cadance, Synopsis, Mentor Graphics и др. В качестве элементной базы эти САПР используют библиотеки функциональных элементов, в состав которых входят как простые логические вентили и триггеры, так и макроэлементы, выполняющие более сложные функции: регистры, счетчики, сумматоры, умножители, арифметико-логические устройства и т. д.

При разработке микроконтроллеров в 90-х гг. прошлого века широкое распространение получила концепция создания микроконтроллерных семейств, имеющих одинаковое процессорное ядро и различающихся набором периферийных устройств и объемом внутренней памяти. Для реализации этой концепции при проектировании СБИС микроконтроллеров кроме функциональных библиотек стали использоваться сложно-функциональные блоки (СФ-блоки) — процессоры, таймеры, АЦП, различные интерфейсные блоки (UART, SPI, CAN, Ethernet и т.д). Эти СФ-блоки формировали верхний уровень функциональных библиотек, используемых разработчиками и производителями микроконтроллеров. Они были достаточно жестко ориентированы на конкретную технологию компании-производителя, являясь внутрифирменной материальной ценностью.

Повышение сложности проектируемых СБИС, жесткие требования к срокам их проектирования (сокращение времени выхода изделия на рынок) поставили перед разработчиками новые проблемы. В сложившихся условиях самостоятельное проектирование разработчиком СнК всех **СФ-блоков (готовые блоки для проектирования микросхем)**, входящих в ее состав, не всегда целесообразно. Поэтому в последние годы широкое распространение получила практика разработки отдельных СФ-блоков для их последующего представления на рынок средств проектирования СнК.

СФ-блоки, предназначенные для использования в разнообразных проектах, стали называть **IP (Intellectual Property)** модулями, тем самым подчеркивается, что эта продукция является предметом интеллектуальной собственности.

СФ-блоки, используемые при проектировании СнК, имеют две основные формы представления:

– в виде топологических фрагментов, которые могут быть непосредственно реализованы в физической структуре кристалла — аппаратно реализованные (hard) СФ-блоки;

– в виде моделей на языке описания аппаратуры (Verilog, VHDL), которые средствами САПР могут быть преобразованы в топологические фрагменты для реализации на кристалле — синтезируемые (soft) СФ-блоки.

Таким образом, разработчик может либо непосредственно «вмонтировать» в структуру проектируемой СБИС топологически готовый СФ-блок, либо использовать имеющуюся модель СФ-блока и выполнить его схемотехническое и топологическое проектирование в составе реализуемой СБИС СнК.

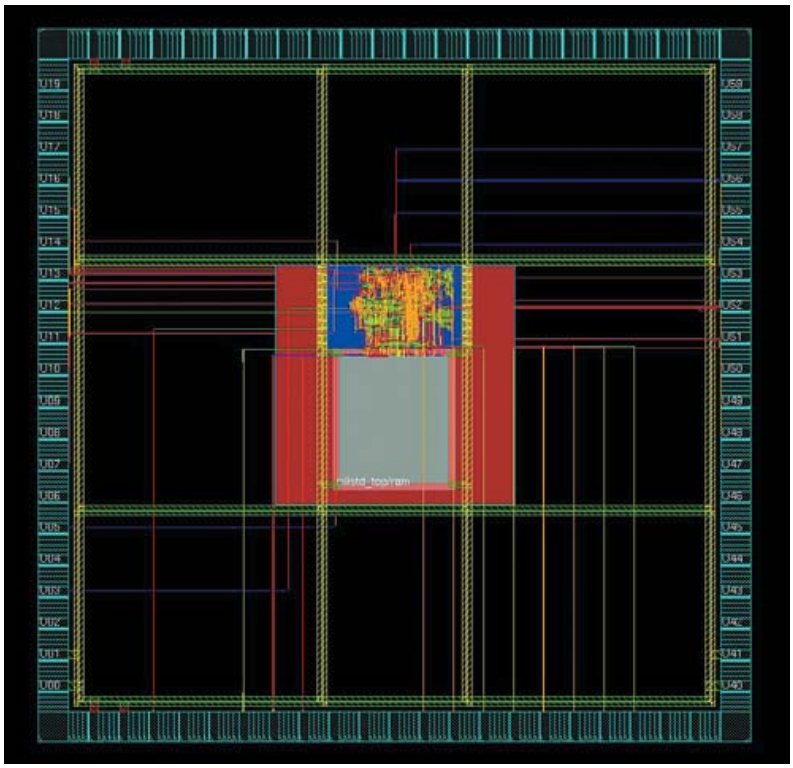


Рис. 16 – СФ (IP)-блок в литографическом срезе.

В процессе проектирования SoC разработчик имеет возможность выбора следующих решений:

– самостоятельная разработка необходимых СФ-блоков;

– покупка СФ-блоков у ведущих разработчиков и производителей микросхем;

– поиск и применение СФ-блоков, предоставляемых в открытом доступе (www.opensource и др. источники).

Каждый из этих вариантов имеет свои достоинства и недостатки. Как уже отмечалось, самостоятельная разработка всех СФ-блоков может привести к увеличению сроков проектирования и задержке выпуска конечного изделия. **Покупка СФ-блоков сопряжена с определенными финансовыми затратами, повышающими стоимость разработки**. Применение СФ-блоков, имеющихся в свободном доступе, возможно только после их тщательной верификации, что требует обычно значительных временных затрат. При выполнении каждого проекта разработчик должен провести оценку поставленных требований и имеющихся ресурсов, чтобы выбрать оптимальный вариант реализации СнК.

Таким образом, основная особенность проектирования СнК — возможность использования достаточно широкой номенклатуры синтезируемых СФ-блоков, имеющихся на рынке и в свободном доступе, которые могут быть реализованы на базе различных функциональных библиотек и технологий и интегрированы в кристалл средствами современных САПР.

Возможности реализации систем на кристалле

Современная микроэлектронная технология обеспечивает следующие варианты реализации СнК:

– в виде заказной СБИС (ASIC);

– на базе ПЛИС высокой интеграции (FPGA).

Оба варианта реализации имеют свои достоинства и недостатки, которые целесообразно оценить в сравнении с традиционным способом монтажа систем на печатной плате из отдельных микросхем — системами на плате.

При реализации СнК в виде ASIC используется традиционный маршрут проектирования ASIC с использованием аппаратно реализованных СФ-блоков, интегрированных в структуру СБИС, и синтезируемых СФ-блоков, которые изготовитель транслирует в физическую структуру с помощью собственных библиотек функциональных элементов. Используя средства САПР, набор необходимых СФ-блоков и современные технологии, можно реализовать в виде ASIC большинство электронных устройств, монтируемых в настоящее время на печатных платах.

Таким образом, имеется возможность замены систем на плате системами на кристалле. **Возникает альтернатива** — разработка системы на плате или реализация функционально аналогичной СнК в виде ASIC.

Преимущества систем на плате:

– использование хорошо проверенных серийных компонентов;

– более простой процесс тестирования и отладки;

– возможность замены неисправных компонентов;

– низкая стоимость создания опытных образцов и малых серий.

Преимущества систем на кристалле:

– возможность получения более высоких технических показателей (производительность, энергопотребление, массогабаритные характеристики);

– более низкая стоимость при крупносерийном выпуске.

Следует отметить, что реализация СнК в виде специализированной ASIC требует значительных финансовых затрат. Изготовление опытной партии специализированных СБИС (несколько тысяч образцов) по технологии **0,13 — 0,18 нм** стоит несколько сотен тысяч долларов, а по технологии **0,09 нм** — свыше трех миллионов долларов (**данные 2019 г**.).

При этом имеющийся опыт разработки СнК показывает, что только в 25% проектов первоначально полученные опытные образцы соответствуют заданным требованиям. В большинстве случаев для получения необходимого результата требуется несколько итераций, что значительно увеличивает стоимость проекта. Можно надеяться, что развитие средств САПР позволит снизить риски при выполнении таких проектов. Однако в настоящее время реализация СнК в виде ASIC является приемлемой только для ограниченного числа высокобюджетных проектов. Во всех случаях, когда можно достичь заданных характеристик, реализуя системы на плате, этот вариант является более предпочтительным ввиду названных преимуществ.

Альтернативой может быть реализация СнК на базе высокоинтегрированных FPGA, содержащих миллионы эквивалентных логических вентилей.

Преимущества реализации СнК на базе FPGA:

– малые затраты на разработку и создание опытных образцов;

– возможность многократной коррекции проекта;

– использование хорошо проверенных серийных изделий;

– более простой процесс тестирования и отладки (возможность реализации и отладки «по частям»).

Таким образом, СнК на базе FPGA имеют практически те же достоинства, что и системы на плате, но отличаются лучшими техническими характеристиками — более низким энергопотреблением, меньшими габаритами и массой. При этом по таким параметрам как производительность и энергопотребление СнК на базе FPGA уступают СнК, реализованным в виде ASIC.

Исходя из сказанного, можно сделать вывод, что СнК на базе FPGA будут конкурировать и постепенно вытеснять системы на плате. При этом вместо микропроцессоров и микроконтроллеров в этих СнК будут использоваться различные варианты процессорных СФ-блоков.

УСЛУГИ КОНТРАКТНОЙ РАЗРАБОТКИ ПРИ ПРОЕКТИРОВАНИИ НА БАЗЕ СНК

При всех преимуществах СнК не стоит забывать, что процесс разработки продукта на такой элементной базе весьма трудоёмкий и требует наличия не только грамотных специалистов, но и большой ответственности.

Проекты с использованием СнК не могут быть выполнены «на коленке» одним-двумя разработчиками, т.к. необходимо пройти трудоёмкие этапы:

- выбрать архитектуру
- подобрать элементную базу с учётом её стоимости, доступности и совместимости
- разработать принципиальную схему с большим числом связей
- трассировать печатную плату с высокой плотностью монтажа
- верифицировать схему и трассировку ПП
- разместить производство опытных образцов в надёжной фирме
- осуществить первичный «подъём» (bring-up) платы
- разработать тестовое ПО и при необходимости дополнительное оборудование, а также монитор ПО
- подготовить полный пакет поддержки аппаратуры (BSP), который включает в себя: первичный загрузчик с подпрограммами тестирования периферии и памяти, операционную систему с драйверами интерфейсов и устройств, установленных на плате, системные утилиты, автоматические скрипты и пакеты сборки ПО

- разработать прикладное и пользовательское ПО, а также интерфейсы пользователя
- провести интеграционное тестирование продукта
- подготовить конструкторскую документацию и инструкции по установке, прошивке, тестированию и программированию устройства
- выпустить установочную партию изделий и подготовить продукт к серийному выпуску (отдельный трудоёмкий этап работы)

Не все фирмы могут обеспечить эффективное выполнение перечисленных этапов. Тем не менее, разработка изделия требует решения всего комплекса задач с участием менеджеров, маркетологов, системотехников, программистов, дизайнеров, конструкторов, инженеров и других экспертов. В этой ситуации использование услуг контрактной разработки имеет большое значение, т.к. компания, занимающаяся применением СнК, выполнит разработку изделия быстрее и качественнее, чем собственная команда инженеров, не специализирующихся в данной области.

В таком проекте контрактный разработчик принимает на себя комплекс обязательств по исполнению всех этапов проекта (электроника, схемы, платы, корпуса, интерфейсы, программное обеспечение), а также изготовление конечного устройства. Разработчик обеспечивает своевременное исполнение этапов и информирует заказчика о ходе их выполнения. В конечном счёте, именно он несёт полную ответственность за работоспособность конечного устройства, что избавляет заказчика от рисков, связанных с потерей времени и средств при невыполнении проекта собственной группой разработчиков.

Эффективность контрактной разработки объясняется привлечением высококвалифицированных экспертов с узкой специализацией, прозрачностью бюджета и сроков разработки, применением решений, опробованных и отработанных в других проектах и отраслях промышленности.

Во многих случаях производителю выгоднее передать разработку изделия контрактному разработчику, а свои усилия сосредоточить на исследовании рынка и продвижении продукции. Поэтому грамотное использование услуг контрактной разработки способно обеспечить новый качественный уровень изделий и сократить их время выхода на рынок.

Разновидности однокристальных систем

Большое число производителей и «неограниченный» выбор кристаллов СнК вносят некоторую неразбериху при выборе элементной базы для реализации того или иного устройства.

Существует ряд признаков, по которым можно классифицировать системы на кристалле:

- по процессорному ядру: ARM, MIPS, PowerPC, x86 и др.
- по производительности ядра и частоте системной шины
- по набору интерфейсов
- по стоимости кристалла и его минимальной обвязки
- по позиционированию кристалла производителем

Кроме того, инженеру-разработчику необходимо учитывать доступность СнК, планируемый цикл производства кристалла, назначение и эксплуатационные характеристики будущего изделия, технологии печатных плат и пайки, полноту документации и технической поддержки и т.п. В такой ситуации даже грамотный специалист может принять не оптимальное решение.

Предлагаемая классификация сформирована на основе выполненных проектов (с учётом доступности микросхем) и призвана помочь российским разработчикам готовых изделий.

Классификация СнК по применению:

Стандартные решения могут быть реализованы кристаллами СнК из следующих классификационных групп:

- для бюджетных применений
- для устройств удалённого управления
- для терминальных устройств
- многоядерные, предназначенные для обработки большого количества данных
- для специализированных вычислителей на основе ПЛИС

Для бюджетных применений

СнК этой группы могут быть использованы в бортовой электронике, системах управления и контроля доступа, системах оповещения, промышленных контроллерах, устройствах вывода звуковой информации.

Кристаллы могут применяться в проектах, не требующих большого объёма программного кода или использования операционных систем.

Кристалл из этой группы (см. табл. 1) характеризуется низкой стоимостью (до 10 долл. США), несложной схемотехникой конечного продукта (не требует применения внешней памяти, имеет невысокие частоты обмена по шинам), возможностью реализации устройства на двухслойных печатных платах (ПП), простотой монтажа, отладки и тестирования.

Данные СнК произошли от микроконтроллеров и унаследовали их периферию (GPIO, UART, I2C, SPI, АЦП/ЦАП, ШИМ), но получили более производительное ядро ARM7 и специфичные для СнК интерфейсы (USB, ЖКИ, Ethernet). СнК для устройств удалённого управления.

СнК для устройств удалённого управления

Кристаллы этой группы (см. табл. 2) целесообразно применять в изделиях, реализующих удалённое управление – с использованием Ethernet или беспроводных интерфейсов – в устройствах сбора данных, серверах контроллерного оборудования, сетевом оборудовании (точки доступа, шлюзы, маршрутизаторы). Высокопроизводительное ядро позволяет использовать операционные системы с поддержкой файловых систем, стека протокола TCP/IP, FTP-сервера и web-сервера. Стоимость кристаллов составляет 10…20 долл.; схемотехника – средней сложности (требуется подключение микросхем памя ти, реализация физического уровня интерфейсов), возможна реализация устройства на ПП с 4 – 6 слоями.

Таблица 1. СнК для бюджетных применений

Производитель	NXP	Atmel	CirrusLogic
Класс	LPC21xx, LPC22xx, LPC24xx	AT91SAM7x	EPM7309, EPM7311, EPM7312
Ядро	ARM7TDMI	ARM7TDMI	ARM7TDMI
Частота, МГц	60...72	55	74
Интерфейсы	ЖКИ, SD/MMC, USB Host Device, OTG, Ethernet 10/100	USB, Ethernet 10/100	ЖКИ, контроллер клавиатуры и тактильного дисплея, цифровой звуковой интерфейс, порт кодека мультимедиа
Периферия	ADC, DAC, UART, SPI/SSP, I2S/I²C, CAN	UART, SPI, SSC, TWI, CAN	IrDA, UART, SSI

Таблица 2. СнК для устройств удалённого управления

Производитель	RDC	Atmel	CirrusLogic
Типономинал	R8610	AT91RM9200, AT91SAM9x	EPM9301, EPM9302
Ядро	Ядро x86	ARM920, ARM926	ARM920
Частота, МГц	150	180...240	166...200
Интерфейсы	2 × Ethernet MAC, USB 2.0, UARTs, LPC, PCI	Ethernet MAC, USB 2.0, UARTs, SPI, SSP, TWI, MCI	Ethernet MAC 10/100, USB 2.0, IrDA, АЦП, SPI, I2S

СнК для терминальных устройств

Кристаллы этой группы (см. табл. 3) подходят для электронных устройств со встроенными ЖК-матрицами большого разрешения. Конечным продуктом могут быть планшетные и панельные компьютеры, измерительные приборы, бортовые компьютеры с экранами высокого разрешения, медицинские мониторы и терминалы, информационные киоски и панели. Микросхемы этой группы стоят довольно дорого (20…30 долл.), но оправдывают своё применение за счёт высокой степени интеграции современных интерфейсов. Поэтому разрабатываемая схемотехника имеет среднюю сложность: печатную плату можно выполнить в 6 – 8 слоев, и в большинстве случаев потребуется монтаж корпусов типа BGA.

Двухъядерные СнК для обработки данных

Микросхемы этой группы (см. табл. 4) идеально подойдут для применения в устройствах, где требуется параллельная обработка данных или сбор информации с её одновременным выводом. Устройства, в которых могут потребоваться такие возможности, бывают абсолютно разными: от мультимедиа до измерительной техники. К примеру, в измерительных приборах часто необходимо выполнять свёртку с использованием ядра ЦПОС и одновременный вывод информации на ЖКИ, а также обработку сигналов клавиатуры. С такой задачей успешно справится процессор **OMAP5912,** оснащённый ядрами DSP и ARM с общей системной шиной. Микросхемы этой группы подойдут для различных мобильных устройств, т.к. обладают низким энергопотреблением. Стоимость и другие характеристики СнК этой группы схожи с характеристиками предыдущей группы, однако существенным отличием является работа с двухъядерной архитектурой при написании и отладке встраиваемого программного обеспечения.

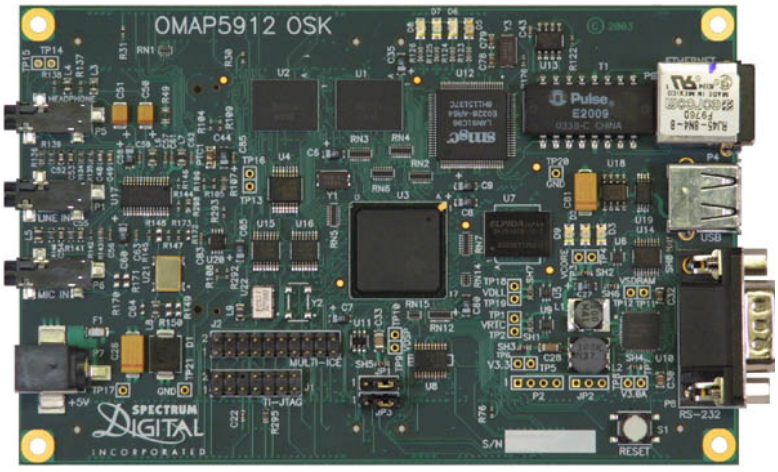


Рис. 17 – OMAP5912 - высокоинтегрированная программно-аппаратная платформа, разработанная для последнего поколения мобильных и встраиваемых приложений.

Таблица 3. СнК для терминальных устройств

Производитель	Freescale	Atmel	CirrusLogic
Типономинал	MC9328MXL	AT91SAM9261, AT91SAM9263	EPM9312(без акс.), EPM9307, EPM9315
Ядро	ARM9	ARM9	ARM9
Частота, МГц	200...350	200	200
Интерфейсы	USB Host Device	Ethernet 10/100, USB Host Device	Ethernet 10/100, USB Host Device
Периферия	ЖКИ 16/18 бит, кодер/декодер, MPEG-4, H.263, 2D-ускоритель	ЖКИ 2000×2000 пикселей, 2D-ускоритель	ЖКИ 1024×768 пикселов, 2D-ускоритель

Таблица 4. СнК для обработки данных с двухъядерной архитектурой

Производитель	Texas Instruments	Analog Devices
Типономинал	OMAP5912	BlackFin BF561
Ядро	ARM926 + TMS320C55x	Dual BlackFin
Частота, МГц	192	600
Интерфейсы	USB 1.1 Host, Client, OTG, UART, SPI, IrDA, I²C, контроллер SDRAM	UART, IrDA, SPI, SPORT, контроллер SDRAM
Периферия	Интерфейс камеры, интерфейс матричной клавиатуры, HDQ/1-Wire, интерфейс, MMC/SD, контроллер ЖКИ	

СнК для специализированных вычислителей на основе ПЛИС

Безусловно, среди разработчиков популярны гибкие решения на базе ПЛИС (см. табл. 5), но их применение оправдано при реализации алгоритмов параллельной обработки данных, скоростных алгоритмов обработки потоков, совокупности уникальных или специфических интерфейсов, интеграции различных ядер и алгоритмов цифровой обработки сигналов в одном устройстве. Применение ПЛИС усложняет процесс разработки и повышает стоимость конечного продукта. Большим преимуществом ПЛИС является реконфигурируемость, что позволяет даже небольшой проектной фирме иметь 1 – 2 платформы и строить на их основе разнообразные изделия.

Таблица 5. Гибкие СнК на базе ПЛИС (CSoC и SoPC)

Производитель	Altera	Xilinx
Семейство	Stratix, Cyclone	Spartan, Virtex
Ядро	Nios, Nios2, ARM, 8051 core, PIC	MicroBlaze, PicoBlaze, ARM, ядро I8051, PIC, интегрированный PowerPC
IP-ядра для цифровой обработки сигналов	DSP	Фильтрация, модуляция/демодуляция, шифрование/дешифрование, корреляция, генерирование сигналов, синхронизация
IP-ядра для работы с видеоданными и изображениями	Обработка изображений и потокового видео	Передача видео по Ethernet (100/1000 Мбит/с), 2D- и медианный фильтры, кодер/декодер JPEG, JPEG2000, MPEG

Но в общем виде классификация SoC (СнК) сосредоточена по следующим типам:

- SoC построенные вокруг микроконтроллера (рассматривали ранее)
- SoC, построенные вокруг микропроцессора, встречаются в мобильных телефонах и смартфонах;
- Специализированные прикладные интегральные схемы SoC, предназначенные для конкретных применений, которые не вписываются в вышеперечисленные две категории, и:
- **Programmable SoCs (PSoC);**
- **Multiprocessor System-on-Chip, (MPSoC);**

А многопроцессорные системы на кристалле (MPSoC) представляют собой систему на кристалле (СнК), который включает в себя несколько микропроцессоров. Таким образом, это многоядерная система на чипе.

MPSoCs обычно предназначены для встроенных приложений. Он используется платформами, которые содержат несколько, обычно гетерогенных, обрабатывающих элементов со специфическими функциональными возможностями, отражающими потребности ожидаемой области применения, иерархии памяти и компонентов ввода-вывода. Все эти компоненты соединены друг с другом межконтинентальным соединением на кристалле, таким как шины и **сети на кристалле (NoCs)**.

Эти архитектуры удовлетворяют потребности в производительности мультимедийных приложений, телекоммуникационных архитектур, сетевой безопасности и других областей применения, ограничивая при этом энергопотребление за счет использования специализированные элементы обработки и архитектуры.

Многопроцессорная система на кристалле по определению должна иметь несколько процессорных ядер. MPSoC часто также содержат несколько логически различных процессорных модулей. Кроме того, MPSoCs обычно содержат:

- Блоки памяти, часто использующие оперативную память scratchpad RAM и прямой доступ к памяти.
- источники синхронизации для генерации тактовых сигналов для управления выполнением функций SoC.
- кварцевые генераторы и циклы фазовой автоподстройки частоты являются популярными генераторами тактовых импульсов.
- периферийные устройства, включая счетчики и генераторы сброса питания.
- внешние интерфейсы, как правило, для протоколов связи.

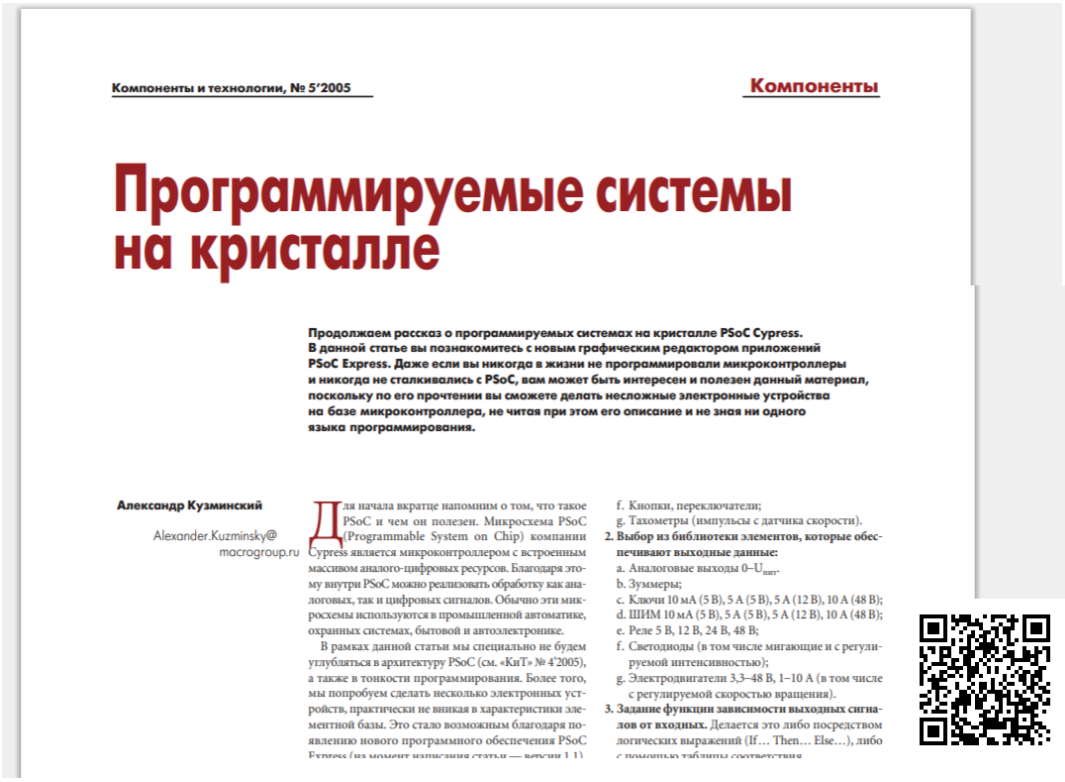
Они часто основаны на отраслевых стандартах, таких как USB, FireWire, Ethernet, USART, SPI, HDMI, I2Cи т. д.

- каждый интерфейс обычно относится к одному заданному ядру или логическому блоку на MPSoC;
- сеть на чипе (NoC) для связи и обмена данными между процессорами и функциональными блоками MPSoC;

MPSoC используются, когда микроконтроллеры или системы на кристалле должны иметь возможности многопроцессорной обработки. Это могут быть смартфоны, встроенные системы, процессоры цифровых сигналов и другие различные приложения.

PSoC (англ. Programmable System-on-Chip, программируемая система на чипе) — программируемая система, которая вмещает функциональные составляющие целого устройства на одном чипе. В отличие от обычных микроконтроллеров, помимо процессорного ядра, PSoC имеет матрицу цифровых и аналоговых блоков. Разрабатывается компанией Cypress Semiconductor.

Благодаря конфигурированным аналоговым и цифровым блокам, становится возможным создание внутри микросхемы PSoC таких функций, как АЦП, ЦАП, компаратора, ФНЧ, аудиовыхода и т.п.. Эти конфигурации доступны в библиотеках среды **PSoC Creator** (**см. статью по интер-отклик материалу ниже**), который можно скачать на официальном сайте разработчика.



Компоненты и технологии, № 5'2005 Компоненты

Программируемые системы на кристалле

Продолжаем рассказ о программируемых системах на кристалле PSoC Cypress. В данной статье вы познакомитесь с новым графическим редактором приложений PSoC Express. Даже если вы никогда в жизни не программировали микроконтроллеры и никогда не сталкивались с PSoC, вам может быть интересен и полезен данный материал, поскольку по его прочтении вы сможете делать несложные электронные устройства на базе микроконтроллера, не читая при этом его описание и не зная ни одного языка программирования.

Александр Кузминский
Alexander.Kuzminsky@macrogroup.ru

Для начала вкратце напомним о том, что такое PSoC и чем он полезен. Микросхема PSoC (Programmable System on Chip) компании Cypress является микроконтроллером с встроенным массивом аналого-цифровых ресурсов. Благодаря этому внутри PSoC можно реализовать обработку как аналоговых, так и цифровых сигналов. Обычно эти микросхемы используются в промышленной автоматике, охранных системах, бытовой и автоэлектронике.

В рамках данной статьи мы специально не будем углубляться в архитектуру PSoC (см. «КиТ» № 4'2005), а также в тонкости программирования. Более того, мы попробуем сделать несколько электронных устройств, практически не вникая в характеристики элементной базы. Это стало возможным благодаря появлению нового программного обеспечения PSoC Express (на момент написания статьи — версии 1.1)

f. Кнопки, переключатели;
g. Тахометры (импульсы с датчика скорости).

2. **Выбор из библиотеки элементов, которые обеспечивают выходные данные:**
a. Аналоговые выходы 0–$U_{пит}$.
b. Зуммеры;
c. Ключи 10 мА (5 В), 5 А (5 В), 5 А (12 В), 10 А (48 В);
d. ШИМ 10 мА (5 В), 5 А (5 В), 5 А (12 В), 10 А (48 В);
e. Реле 5 В, 12 В, 24 В, 48 В;
f. Светодиоды (в том числе мигающие и с регулируемой интенсивностью);
g. Электродвигатели 3,3–48 В, 1–10 А (в том числе с регулируемой скоростью вращения).

3. **Задание функции зависимости выходных сигналов от входных.** Делается это либо посредством логических выражений (If… Then… Else…), либо с помощью таблицы соответствия

Межмодульная связь в SoC

SoC содержат множество исполнительных блоков. Эти устройства должны часто отправлять данные и инструкции туда и обратно. Из-за этого все, кроме самых тривиальных SoC, требуют подсистем связи. Первоначально, как и в случае с другими микрокомпьютерными технологиями, использовались архитектуры шины данных, но в последнее время все большую популярность приобрели проекты, основанные на разреженных сетях связи, известных как сети на кристалле **(NoC),** и, по прогнозам, в ближайшем будущем они превзойдут архитектуры шин при разработке SoC.

Связь по шине. Исторически сложилось так, что общая глобальная компьютерная шина обычно соединяла различные компоненты, также называемые «блоками» SoC.

Очень распространенной шиной для связи SoC является бесплатный стандарт архитектуры шины микроконтроллеров ARM (**AMBA**).

Контроллеры прямого доступа к памяти направляют данные напрямую между внешними интерфейсами и памятью SoC, минуя ЦП или блок управления, тем самым увеличивая пропускную способность SoC.

Это похоже на некоторые драйверы периферийных устройств на компонентных многочиповых модульных архитектурах ПК. Компьютерные шины имеют ограниченную масштабируемость, поддерживая до десятков ядер (**многоядерность**) на одном кристалле. Задержка проводов не масштабируется из-за продолжающейся миниатюризации, производительность системы не масштабируется с количеством подключенных ядер, рабочая частота SoC должна уменьшаться с каждым дополнительным подключенным ядром, чтобы питание было устойчивым, а длинные провода потребляют большое количество электроэнергии. Эти проблемы препятствуют поддержке многоядерных систем на кристалле.

AMBA - это внутренняя спецификация межмодульных соединений с открытым стандартом для подключения и управления функциональными блоками в проектах системы на кристалле (SoC). Это облегчает разработку многопроцессорных конструкций с большим количеством контроллеров и компонентов с шинной архитектурой. С момента своего создания AMBA, несмотря на название, вышла далеко за рамки микроконтроллеров.

Сегодня AMBA широко используется в ряде частей ASIC и SoC, включая процессоры приложений, используемые в современных портативных мобильных устройствах, таких как смартфоны.

AMBA является зарегистрированным товарным знаком компании ARM Ltd. AMBA была представлена ARM в 1996 году. Первыми шинами AMBA были Advanced System Bus (ASB) и Advanced Peripheral Bus (APB). Во второй версии AMBA 2 в 1999 году ARM добавила высокопроизводительную шину AMBA (AHB), которая представляет собой протокол с одним фронтом тактовой частоты.

В 2003 году ARM представила третье поколение AMBA 3, включая Advanced eXtensible Interface (AXI) для достижения еще более высокой производительности межсоединений и Advanced Trace Bus (ATB) как часть встроенного решения CoreSight для отладки и трассировки.

В 2010 году были представлены спецификации AMBA 4, начиная с AMBA 4 AXI4, а затем в 2011 году, расширив общесистемную согласованность с помощью AMBA 4 AXI Coherency Extensions (ACE). В 2013 году была представлена спецификация AMBA 5 Coherent Hub Interface (CHI) с переработанным высокоскоростным транспортным уровнем и функциями, предназначенными для уменьшения перегрузки. Эти протоколы сегодня являются стандартом де-факто для архитектур со встроенными процессорами, поскольку они хорошо документированы и могут использоваться без лицензионных отчислений.

Важным аспектом SoC является не только то, какие компоненты или блоки в нем находятся, но и то, как они связаны между собой. AMBA - это решение для взаимодействия блоков друг с другом.

Целью спецификации AMBA является необходимость:

облегчить с первого раза разработку встраиваемых микроконтроллеров с одним или несколькими процессорами, графическими процессорами или сигнальными процессорами, быть технологически независимыми, чтобы позволить повторное использование IP-ядер, периферийных и системных макро-ячеек в различных процессах IC;

поощрять модульную конструкцию системы для повышения независимости процессора и разработку многоразовых периферийных и системных IP-библиотек свести к минимуму кремниевую инфраструктуру;

поддерживая высокую производительность и низкое энергопотребление внутри-кристальной коммуникации.

Характеристики протокола AMBA Спецификация AMBA определяет внутрикристальный коммуникационный стандарт для разработки высокопроизводительных встроенных микроконтроллеров. Он поддерживается ARM Limited с широким участием представителей разных отраслей.

Спецификация **AMBA 5** определяет следующие шины / интерфейсы:

- Спецификация протоколов AXI5, AXI5-Lite и ACE5
- Усовершенствованная высокопроизводительная шина (AHB5, AHB-Lite) Когерентный интерфейс концентратора (CHI)
- Распределенный интерфейс перевода (DTI)
- Универсальная шина Flash (GFB)

Семейство синтезируемых ядер интеллектуальной собственности (IP MBA Products лицензируется ARM Limited, которые реализуют цифровую шину в SoC для эффективного перемещения и хранения данных с использованием спецификаций протокола AMBA. Семейство AMBA включает в себя сетевое соединение AMBA (CoreLink NIC-400), Cache Coherent Interconnect (CoreLink CCI-500), контроллеры памяти SDRAM (CoreLink DMC-400), контроллеры DMA (CoreLink DMA-230, DMA-330), кэш 2-го уровня. контроллеры (L2C-310) и др.

Требования AMBA 4 определяют следующие шины/взаимосвязи:

AXI Coherency Extensions (ACE) - широко применяемую в последних процессорах ARM Cortex-A, включая Cortex-A7 and Cortex-A15

AXI Coherency Extensions Lite (ACE-Lite)

Advanced Extensible Interface 4 (AXI4)

Advanced Extensible Interface 4 Lite (AXI4-Lite)

Advanced Extensible Interface 4 Stream (AXI4-Stream v1.0)

Advanced Trace Bus (ATB v1.1)

Advanced Peripheral Bus (APB4 v2.0)

Требования AMBA 3 определяют четыре шины/взаимосвязи:

Advanced Extensible Interface (AXI3 or AXI v1.0) - широко применяемую в процессорах ARM Cortex-A, включая Cortex-A9

Advanced High-performance Bus Lite (AHB-Lite v1.0)

Advanced Peripheral Bus (APB3 v1.0)

Advanced Trace Bus (ATB v1.0)

Требования AMBA 2 определяют три шины/взаимосвязи:

Advanced High-performance Bus (AHB) - широко применяемую в разработках, основанных на ARM7, ARM9 и ARM Cortex-M

Advanced System Bus (ASB)

Advanced Peripheral Bus (APB2 или APB)

Требования AMBA (Первой разновидности) определяют две шины/взаимосвязи:

Advanced System Bus (ASB)

Advanced Peripheral Bus (APB)

Особенности задержек и уровней voltage на шине требования не диктуют.

AXI Coherency Extensions (ACE и ACE-Lite)

ACE, определённая как часть требований AMBA 4, расширяет AXI дополнительным средством оповещения передач широкой **когерентности**.

Свойство когерентности позволяет множеству процессоров разделять память и включает технологии ARM-овской обработки big.LITTLE. Протокол ACE-Lite включает однонаправленную или иначе IO когерентность, например сетевая взаимосвязь, которая может читать из кеша, полностью когерентного процессора ACE.

Advanced eXtensible Interface (AXI)

AXI, третье поколение взаимосвязи AMBA, определённое в требованиях AMBA 3, нацелено на разработку высокопроизводительных, высокочастотных средств и включает возможности, которые делают её пригодной для высокоскоростных субмикронных межсоединений:

раздельные фазы адреса/управления и данных

поддержка передач невыровненных данных, применяя стробы байта

передачи на основе пакетов, с выдачей лишь начального адреса

выдачу множества внешних адресов с неупорядоченными ответами

лёгкость добавления регистровых стадий для обеспечения близких задержек.

Advanced High-performance Bus (AHB)

AHB - это протокол, представленный в Advanced Microcontroller Bus Architecture 2 разновидности, обнародованный компанией ARM Ltd.

В дополнение к предыдущему исполнению, он имеет следующие возможности:

множество разрядностей шины (64/128/256/512/1024 бит).

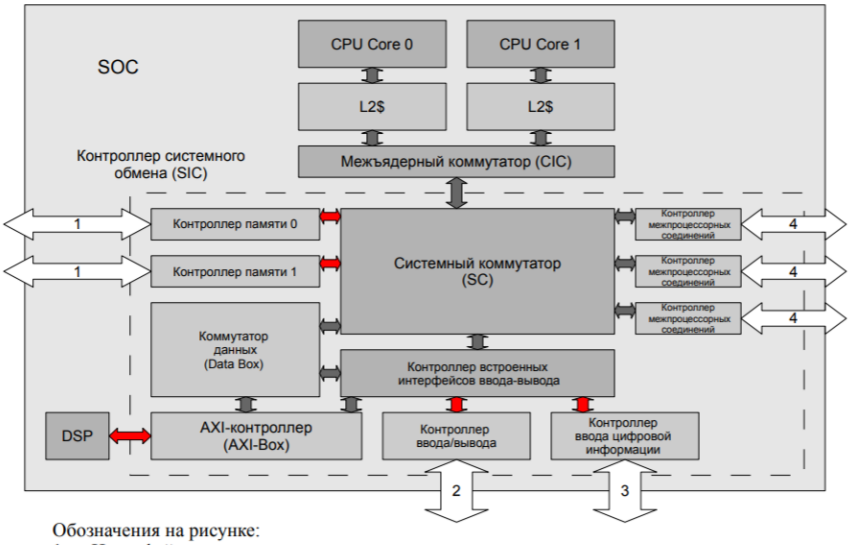
Простая передача по AHB содержит фазу адреса и подпоследовательность фазы данных (без состояний ожидания: лишь два цикла шины). Доступ к целевому устройству управляется через MUX (без Z-уровня), таким образом признаётся доступ к шине одного хозяина одновременно.

AHB-Lite - это подвид AHB, формально определённый стандартом AMBA 3. Этот подвид упрощает разработку шины с одним хозяином.

Разработка контроллера ввода/вывода с интерфейсом AXI для микропроцессоров семейства «Эльбрус» (презентационный материал, 18 слайдов)

Московский физико-технический институт

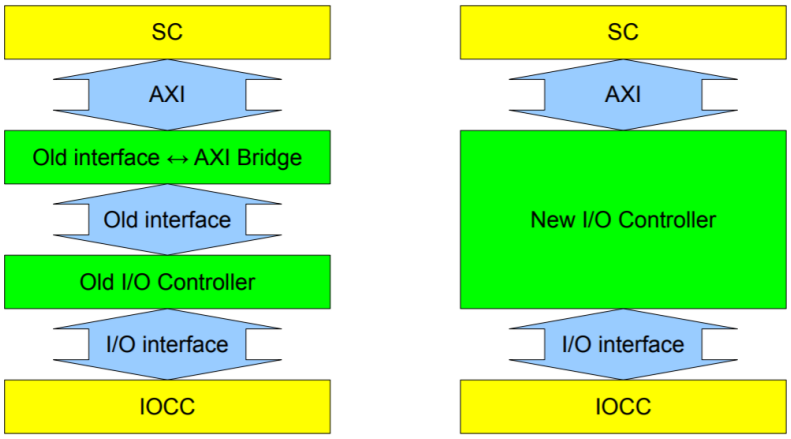
Система на кристалле СБИС МП



Обозначения на рисунке:
1 — Интерфейс с памятью
2 — Канал ввода-вывода
3 — Канал ADC
4 — Межпроцессорные линки

⟷ - интерфейсы, подходящие для стандартизации

Методы реализации:



I: Разработка моста между существующим контроллером и Системным коммутатором

II: Разработка нового контроллера

SoC design flow: Принципы проектирования

Система на кристалле состоит как из аппаратного обеспечения, описанного ранее, так и из программного обеспечения, управляющего микроконтроллером, микропроцессором или ядрами процессора цифровых сигналов, периферийными устройствами и интерфейсами. Процесс проектирования SoC направлен на одновременную разработку этого оборудования и программного обеспечения, что также известно, как совместное архитектурное проектирование. Процесс проектирования также должен учитывать цели оптимизационного характера и настоящие ограничения.

Большинство SoC разрабатываются на основе предварительно проверенных спецификаций IP-ядра аппаратных компонентов для аппаратных элементов и исполнительных блоков, вместе взятых «блоков», описанных выше, вместе с программными драйверами устройств, которые могут управлять их работой. Особое значение имеют стеки протоколов, управляющие стандартными интерфейсами, такими как USB. Аппаратные блоки собираются с использованием средств автоматизированного проектирования, в частности средств автоматизации электронного проектирования; программные модули интегрируются с использованием интегрированной программной среды разработки.

Компоненты SoC также часто разрабатываются на языках программирования высокого уровня, таких как C ++, MATLAB или SystemC, и **преобразуются в проекты RTL с помощью инструментов синтеза высокого уровня (HLS), таких как C в HDL.**

Продукты HLS, называемые «алгоритмическим синтезом», позволяют разработчикам использовать C ++ для моделирования и синтеза уровней системы, схем, программного обеспечения и проверки на одном языке высокого уровня, широко известном компьютерным инженерам, независимо от временных масштабов, которые обычно указываются в HDL. Другие компоненты могут оставаться программными и компилироваться, и встраиваться в программные процессоры, включенные в SoC как модули в HDL как IP-ядра.

После определения архитектуры SoC любые новые элементы оборудования записываются на абстрактном языке описания оборудования, называемом **уровнем регистровых передачи (RTL),** который определяет поведение схемы, или синтезируются в RTL из языка высокого уровня посредством синтеза высокого уровня. Эти элементы соединены вместе на языке описания оборудования для создания полного дизайна SoC.

Логика, указанная на рис. 18 для соединения этих компонентов и преобразования между, возможно, разными интерфейсами, предоставляемыми разными поставщиками, называется **связующей логикой разработки дизайна проектирования SoC**.

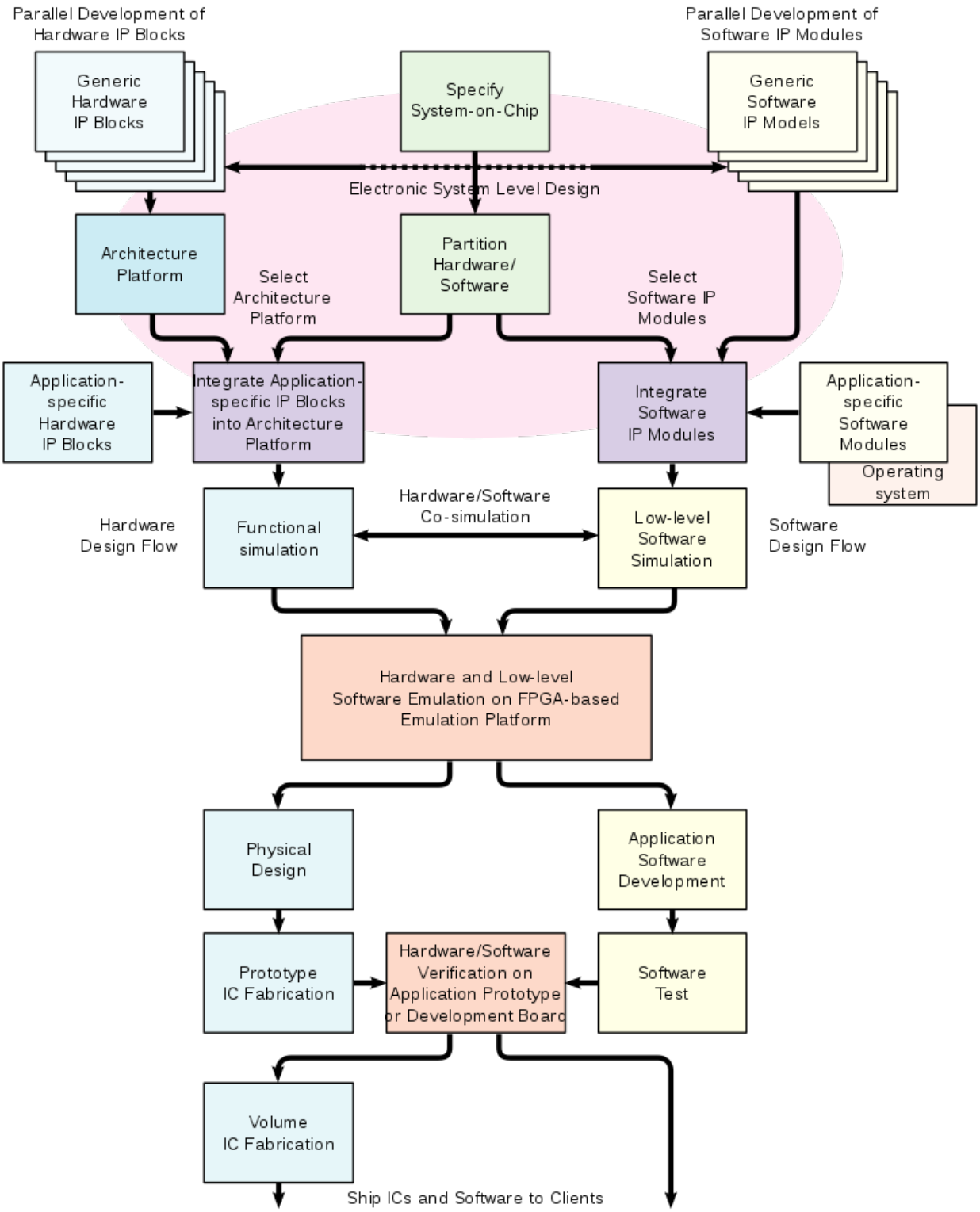


Рис. 18 – Поток проектирования SoC

В основе методологии проектирования SoC лежит принцип повторного использования блоков (reuse). Т.е. сложные функциональные блоки, разрабатываемые в рамках одного проекта или специально, затем используются в других проектах. По аналогии с системой на плате, где в качестве компонент выступают готовые микросхемы, система на кристалле конструируется из повторно используемых блоков.

В настоящее время для обозначения reuse-блока наиболее часто используется **термин IP-блок (Intellectual Property**), т.е. блок, представляющий собой объект интеллектуальной собственности. Одновременно с ним используются и другие термины:

- СФ-блок (сложный функциональный блок) - используется в основном в пределах РФ;
- macro;
- core – обычно применяется для обозначения блоков типа CPU (Central Processor Unit) или DSP (Digital Signal Processor);
- VC (Virtual Component) введен VSIA (Virtual Socket Interface Alliance) – некоммерческой международной организацией, основной задачей которой является разработка нормативной документации по проблемам проектирования IP-блоков и SoC на их основе (www.vsi.org);
- блок (Block) – функционально-завершенная часть SoC или ASIC, не обязательно reuse;
- субблок (Subblock) – часть блока.

IP-блоки могут быть двух типов: «мягкие» (soft) описанные на RTL-уровне и «жесткие» (hard) – на топологическом уровне. Иногда выделяют firm IP, в состав которых входят разные типы представлений от RTL до списка цепей с планировкой субблоков.

Одна из проблем проектирования SoC – создание IP-блоков. Практический опыт показывает, что стоимость reuse-блока в среднем в 10 раз превышает стоимость аналогичного разово используемого блока, а для процессоров эта величина на порядок выше. По этой причине reuse-блоки решают, как правило, общие логически формализуемые задачи, например, MPEG2 кодер, CPU, DSP, USB интерфейс, PCI интерфейс и т.п.

Фактически весь процесс разработки SoC делится на четыре этапа:

- Разработка архитектуры SoC на системном уровне;
- Выбор имеющихся IP-блоков из базы данных (внутри фирмы, других фирм или поставщиков IP-блоков);
- Проектирование оставшихся блоков;
- Интеграция всех блоков на кристалле.

Другая принципиальная особенность SoC это наличие программируемых блоков – процессоров. Поэтому SoC это не просто интегральная схема (ИС), а комплекс, в состав которого входят как аппаратная часть – чип, так программная часть – встраиваемое ПО. Предполагается, что маршрут проектирования SoC должен содержать операции по совместной верификации и отладке программной и аппаратной частей.

В настоящее время не существует четкого детерминированного определения СБИС типа «система на кристалле», работа в этом направлении все еще ведется. **Тем не менее, исходя из вышеперечисленного, можно сделать вывод, что SoC должна изготавливаться по технологии не ниже 28 нм (на 2020 г.).**

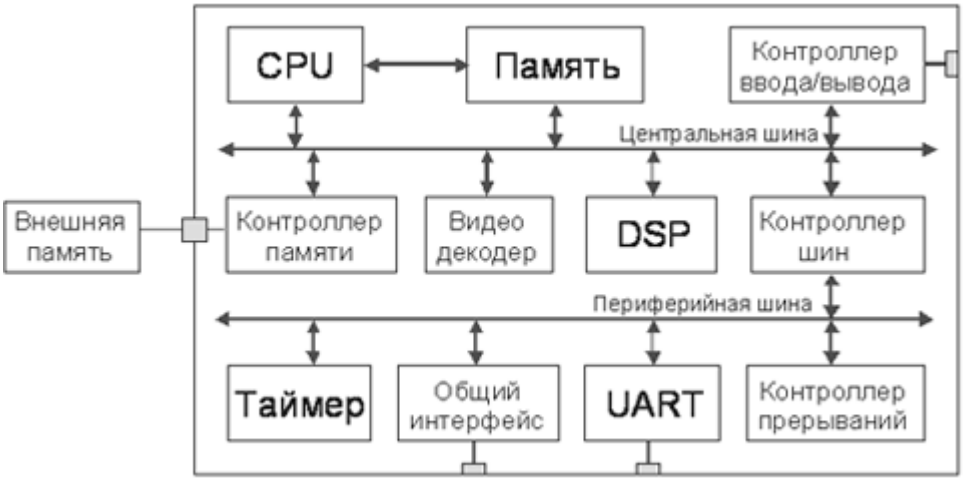


Рис.19 - Пример типичной структуры SoC

На рис.19 представлен пример эквивалентной структуры SoC в общей форме. Как видно из рисунка, в состав SoC входят следующие компоненты:

Микропроцессор (или микропроцессоры) и подсистема памяти (статической и/или динамической). Тип процессора может варьироваться от простейшего 8-разрядного до высокоскоростного 64-разрядного RISC-процессора.

- Шины центральная (высокоскоростная) и периферийная, чтобы обеспечивать обмен данными между блоками.
- Контроллер внешней памяти для расширения памяти, например, DRAM, SRAM или Flash.
- Контроллер ввода/вывода информации: PCI, Ethernet, USB и т.п.
- Видео-декодер, например: AVI, ASF.
- Таймер и контроллер прерываний.

- Общий интерфейс ввода/вывода, например, чтобы вывести на светодиодный индикатор информацию о наличии питания.
- Интерфейс UART (universal asynchronous receiver/transmitter).

Итак, можно выделить **следующие принципиальные особенности системы на кристалле:**

SoC конструируется из сложных функциональных блоков, которые могут быть как разработаны «с нуля», так и получены от поставщика IP-блоков. Поэтому в маршруте проектирования должен быть этап архитектурного проектирования, в задачи которого входит разработка общей архитектуры SoC. При выборе IP-блоков также учитывается стоимость готового блока и оцениваются затраты на разработку собственного блока. SoC имеет в своем составе встраиваемое ПО, поэтому в маршруте должны быть этапы совместной верификации программного и аппаратного обеспечения (АО) – программно-аппаратной верификации (HW/SW co-verification); Наблюдается устойчивый рост доли смешанных цифро-аналоговых систем в общем объеме SoC, поэтому в маршруте проектирования должны быть включены этапы по совместной разработке и верификации цифровой и аналоговой частей SoC.

Поток проектирования SoC

Традиционный маршрут проектирования ASIC представлен на рис. 20. Процесс проектирования начинается с разработки спецификации на проектируемую ASIC. Для сложных СБИС таких, как устройства графической обработки информации, в состав спецификации включается алгоритм обработки информации, который затем используется разработчиками для написания RTL-кода.

После функциональной верификации происходит синтез СБИС на вентильном уровне в виде списка цепей. Здесь же выполняется верификация временных требований. Как только временные требования удовлетворены, список цепей передается на физический синтез: размещение элементов и трассировка цепей. В конце создается и тестируется физический прототип СБИС, на основе которого впоследствии выполняется системная интеграция и тестируется ПО.

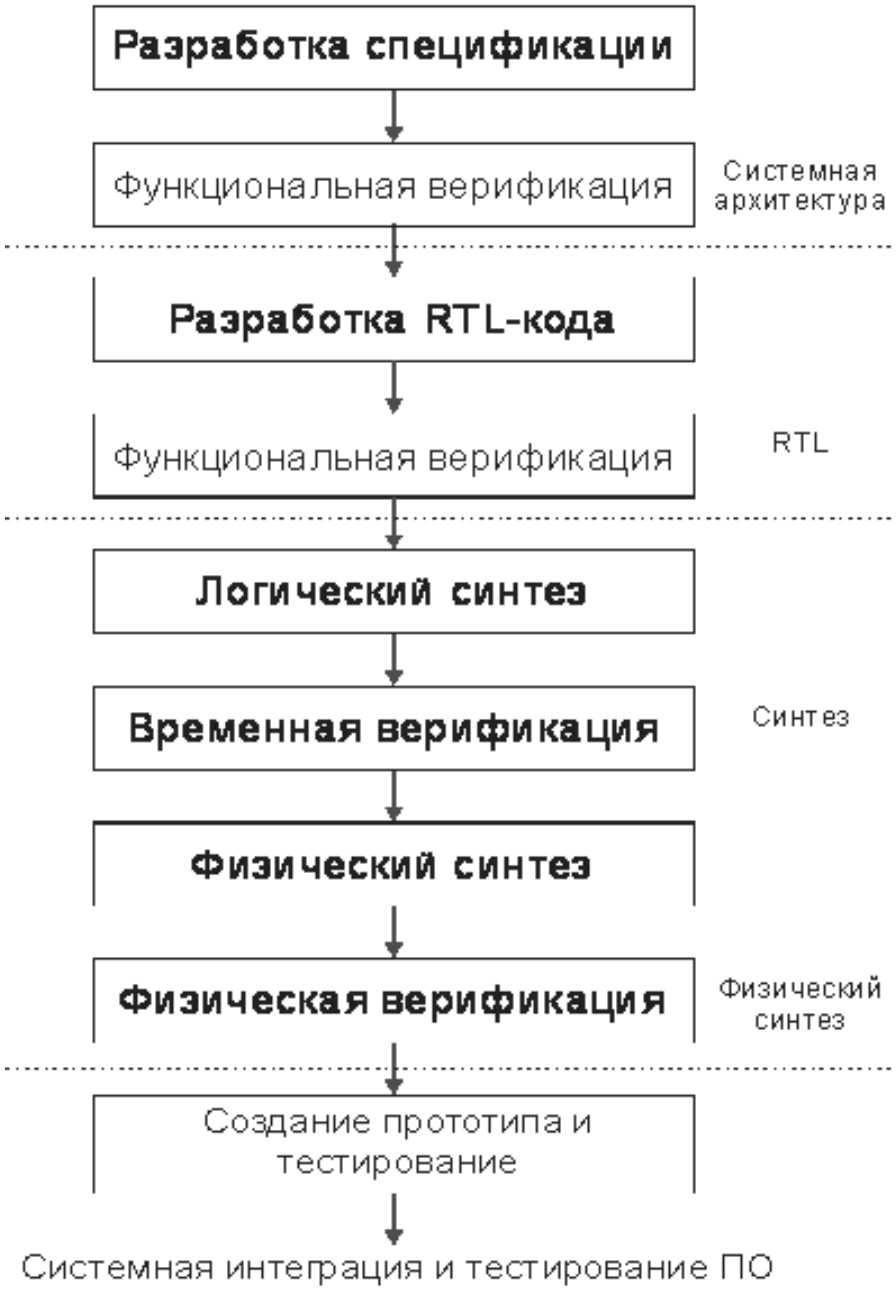


Рис. 20 - Маршрут проектирования ASIC

Используя такой маршрут можно проектировать схемы сложностью не более 100 тыс. вентилей по технологии не ниже 0.5 мкм. Это связано с тем, что проект здесь продвигается поэтапно от одной фазы к другой и никогда не происходит возврат на предыдущие фазы. Например, RTL дизайнер не может прийти к системному разработчику и сказать, что его алгоритм нереализуем, или команда логического синтеза не может попросить изменить RTL-код, чтобы добиться необходимых временных параметров.

Для схем, изготавливаемых по DSM-технологии, такой маршрут вообще не будет работать, поскольку особенности физической реализации должны учитываться уже на логическом уровне проектирования.

Многие фирмы за рубежом переходят от традиционной нисходящей модели маршрута проектирования к новой спиралевидной модели (**рис. 21**).

Здесь проектирование выполняется одновременно по 4-м направлениям: разработка ПО, разработка RTL-кода, логический синтез, физический синтез. При этом в процессе работы группы разработчиков обмениваются результатами проектирования.

Рис. 21 - Спиралевидная модель процесса проектирования

Новая модель характеризуется следующими полезными свойствами:

- параллельная верификация и логический синтез блоков;
- планировка, размещение и трассировка включены в процесс синтеза;
- разрешен возврат на предыдущие фазы проектирование и корректировка результатов.

Системное проектирование SoC

Начальный этап проектирования заключается в рекурсивной разработке, верификации и уточнении набора спецификаций до такой степени детализации, чтобы на их основе можно было начать создавать RTL-код. Быстрая разработка четких, полных и последовательных спецификаций – сложная задача. В хорошей методологии проектирования это трудоемкий, ответственный и протяженный этап проектирования. Если четко знать, что надо построить, то ошибки при дальнейшей реализации могут быть быстро обнаружены и устранены. В противном случае можно не заметить ошибку на протяжении всего цикла проектирования вплоть до изготовления чипа.

Спецификации описывают поведение системы. Точнее, они описывают, как управлять системой так, чтобы получить от нее нужное поведение. В этом смысле, понятие спецификации в значительной мере связано с понятием интерфейса. Функциональная спецификация описывает интерфейс системы или блока так, как его видит внешний пользователь. Она содержат информацию о контактах, шинах, регистрах и о том, как с ними обращаться. Архитектурная спецификация описывает взаимодействия между частями блока и поведение на системном уровне.

Спецификации должны разрабатываться, как на аппаратную, так и на программную части проекта. Спецификации должны включать в себя следующую информацию.

Для аппаратной части:

- выполняемые функции;
- внешний интерфейс к другим блокам (контакты, шины, протоколы);
- интерфейс с ПО (регистры);
- временные параметры;
- быстродействие;
- особенности физического уровня (площадь кристалла, потребляемая мощность).

Для программной части:

- выполняемые функции;
- временные параметры;
- быстродействие;
- интерфейс к аппаратной части;
- структура, ядро.

Традиционно спецификации пишутся на естественных языках таких, как русский, английский, что вносит в них неопределенности, двусмысленности и ошибки. Чтобы избавиться от этих проблем, многие компании начинают использовать исполняемые спецификации. На высоких уровнях для написания исполняемых спецификаций используют языки C, C++ или вариации C++, например, SystemC (www.systemc.org). Для описания аппаратуры обычно пользуются **VHDL или Verilog**.

Разработка исполняемых моделей позволяет дизайнерам верифицировать основные выполняемые функции и интерфейсы на ранних стадиях проектирования, задолго до детализации проекта.

Процесс проектирования SoC на системном уровне представлен на **рисунке 22**.

Процесс разработки начинается с идентификации целей и задач, выполняемых SoC. На начальном этапе следует определить основные эксплуатационно-технические свойства: требуемое быстродействие, допустимую потребляемую мощность, время, необходимое для разработки, и т.п.

На основании этих свойств создается системная спецификация, которая может выступать частью технического задания на разработку системы. Как правило, этот этап выполняется без применения специализированных программных средств САПР.

Рис.22 - Маршрут системного проектирования SoC

Затем создается высокоуровневая поведенческая модель всей разрабатываемой системы.

Поведенческая модель системы, как правило, строится в виде блок-схемы. Для верификации разработанной поведенческой модели создается тестовое окружение (**Testbench**) системы, которое включает в себя генераторы входных сигналов, тестовые последовательности и блоки отображения выходной информации. Тестовое окружение должно максимально полно верифицировать функционирование системы. Впоследствии, на основе этого тестового окружения будут разрабатываться тестовые последовательности для верификации проекта на нижних уровнях проектирования и для тестирования опытных образцов СБИС.

Верификация поведенческой модели осуществляется путем компьютерного моделирования с использованием специальных программных средств. Если в процессе верификации обнаруживаются какие-либо отклонения от требований системной спецификации, то модель корректируется и моделирование повторяется.

Кроме верификации на данном шаге может быть выполнен выбор оптимальных параметров алгоритма системы. Например, разработчик может найти компромисс между вычислительной сложностью и точностью.

Как было сказано выше, система на кристалле включает в себя одно или несколько программируемых процессорных ядер. Поэтому на следующем этапе разработчик должен принять решение о том, какие части поведенческой модели будут в последствии реализованы на аппаратном уровне, а какие – на программном в виде встроенного в СБИС программного обеспечения. Кроме этого необходимо определить, каким образом будут взаимодействовать программная и аппаратная части, т.е. следует разработать интерфейс между АО и ПО.

Здесь же определяется общая архитектура SoC: тип процессора, тип памяти и ее объем, аппаратные блоки, интерфейс АО-ПО, тип используемой шины, описание программной части и т.п.

В итоге формируется набор спецификаций на разработку программного обеспечения и на разработку каждого аппаратно реализуемого блока.

В методологии проектирования ASIC программно-аппаратная верификация выполняется только, после изготовления опытного образца. Возникающие при этом ошибки функционирования, как правило, можно исправить внесением соответствующих изменений в ПО, не переделывая сам кристалл.

Такой метод не подходит к процессу проектировании SoC потому, что из-за большой сложности схемы исправить ошибки и погрешности АО путем корректировки ПО очень трудно, а зачастую просто невозможно. Поэтому в маршрут проектирования на разных уровнях вводится операция программно-аппаратной верификации. Наиболее ответственными в этом смысле являются этапы функционального и логического проектирования.

Программно-аппаратная верификация системного уровня на сегодняшний день не является обязательной операцией. Тем не менее, многие разработчики включают ее в маршрут проектирования SoC. В качестве аппаратной части здесь выступают исполняемые спецификации аппаратно реализуемых блоков, в качестве программной – прототип ПО. Таким образом можно убедиться, что аппаратная часть, разрабатываемая в соответствии с имеющимися спецификациями, будет корректно функционировать под управлением встраиваемого ПО в режиме реального времени.

Программные средства САПР для системного уровня

До недавнего времени задачи системного уровня – разработка спецификаций – в большинстве случаев решались без использования специальных программных средств. При переходе к методологии проектирования SoC стала очевидной необходимость автоматизации процесса системного проектирования.

Во-первых, поскольку SoC – высокоинтегрированная СБИС, то в ее состав входит большое количество сложных блоков: процессоры, жесткая логика, память, схемы контроля, аналоговые и цифроаналоговые компоненты и т.п. Проверить работоспособность такой системы, используя только аналитические методы расчета, невозможно. Поэтому, как было сказано выше, возникает необходимость в моделировании всей системы на поведенческом уровне, а для этого требуется специальное прикладное ПО.

Во-вторых, исполняемая спецификация должна быть представлена в определенном формате на языках C, C++, SystemC, Verilog или VHDL. Получить такое описание невозможно без использования соответствующих программных средств.

Кроме этого при переходе к спиралевидной модели проектирования в процессе работы будут возникать постоянные «откаты» от нижних уровней проектирования к верхним. Часто системный уровень сливается с функциональным уровнем проектирования (разработка RTL-кода), образуя системно-функциональный уровень проектирования. В этом случае удобно пользоваться едиными программно-техническими средствами.

В настоящее время в мире существует огромное количество программных средств, которые могут быть использованы для автоматизации системного проектирования. **Эти средства можно классифицировать на пять групп.**

Первая группа – средства разработки и отладки прикладного программного обеспечения. Достаточно популярным и удобным для представления спецификаций оказался язык программирования С и его модификация С++. Поэтому разработчики системного уровня активно используют средства разработки программных приложений.

Основные достоинства - это низкая стоимость, простота в освоении и использовании. Главный недостаток связан с отсутствием специализированных библиотек системного уровня, поэтому поведенческую модель разработчику приходится создавать практически «с нуля». Иногда также используют специальные средства для анализа и автоматизации разработки ПО, например, Rational Rose и язык UML (www.uml.ru).

Вторая группа – средства математического моделирования. Наиболее типичным представителем здесь выступает распространенный программный пакет MATLAB/Simulink (www.mathworks.com). Основными достоинствами здесь, как и в первой группе, являются низкая стоимость и простота в использовании, кроме этого есть возможность выполнять математическое моделирование и наличие библиотек моделей. К недостаткам можно отнести недостаточный объем специализированных библиотек, т.е. большинство моделей приходится создавать вручную, и использование собственного формата данных (М-файлы, MEX-файлы) в качестве базового. Последнее замечание носит, по всей видимости, временный характер, так как фирма The MathWorks, планирует выпустить полноценный транслятор из формата М-файлов в формат С/С++.

В третью группу можно объединить средства моделирования общего назначения, например, MLDesigner (www.mldesigner.com) или SES/Workbench (www.hyperformix.com). Отличительная особенность этих средств в том, что они не привязаны к какому-то конкретному объекту проектирования. С их помощью можно моделировать, например, как архитектуру СБИС, так и систему спутниковой связи или навигации. Достоинства: сравнительно низкая цена, широкий спектр областей применения. Основной недостаток –отсутствие связи с другими уровнями проектирования: функциональным и логическим.

Четвертая группа самая многочисленная. Сюда относятся программные средства, каждое из которых предназначено для решения какого-либо определенного круга проектных задач системного уровня.

При этом общий спектр решаемых задач велик: от разработки программно-аппаратной архитектуры до интеграции процессорных ядер и разработки встраиваемого программного обеспечения. В качестве примера производителей данного ПО можно привести названия таких фирм, как Coware (www.coware.com), Mentor Graphics (www.mentor.com), Elanix (www.elanix.com), Summit Design (www.sd.com) и др. Использование такого специализированного ПО представляется достаточно привлекательным с точки зрения экономии средств.

Пятая группа это мощные интегрированные программные пакеты, при помощи которых разработчик способен выполнять весь цикл системного и функционального проектирования, а также весь цикл проектирования вплоть до физической реализации. На сегодняшний день в эту группу входят пакеты ПО только двух фирм:

- Synopsys (www.synopsys.com): CoCentric System Studio, Design Ware, VCS, VCSi, Scirocco, SystemC HDL Co-Sim, CoCentric SystemC Compiler;
- Cadence Design Systems (www.cadence.com): Incisive-SPW, Incisive unified simulator, Incisive-XLD, Incisive-AMS, NC-SystemC, NC-Verilog, NC-VHDL.

Главным недостатком обоих пакетов является их большая стоимость, что весьма существенно в условиях российского рынка.

При окончательном выборе программных средств и разработке на их основе маршрута проектирования необходимо дополнительно учитывать целый ряд факторов, например, специфику разрабатываемых устройств, общий объем работ (число одновременно выполняемых проектов), имеющееся на предприятии ПО для функционального и логического уровней проектирования и т.п.

Технологии производства SoC: ASIC

ASIC (аббревиатура от англ. application-specific integrated circuit, «интегральная схема специального назначения») — интегральная схема, специализированная для решения конкретной задачи. В отличие от обычных интегральных схем для общего назначения, специализированные интегральные схемы применяются в конкретном устройстве и выполняют строго ограниченные функции, характерные только для данного устройства; вследствие этого выполнение функций происходит быстрее и, в конечном счёте, дешевле. Примером ASIC может являться микросхема, разработанная исключительно для управления мобильным телефоном, микросхемы аппаратного кодирования/декодирования аудио- и видеосигналов (сигнальные процессоры).

Микросхема ASIC имеет узкий круг применения, обусловленный жёстко предопределённым набором её функций.

Современные ASIC часто содержат 64-битный процессор, иногда в количестве нескольких ядер, блоки памяти (как ПЗУ, так и ОЗУ) и другие крупные блоки. Такие ASIC часто называют однокристальной системой.

При разработке цифровых ASIC для описания их функциональности используют языки **описания аппаратных устройств (HDL**), такие как **Verilog и VHDL.**

Ранние ASIC использовали технологию **вентильной матрицы**. К 1967 году Ferranti и Interdesign производили первые биполярные вентильные матрицы. В 1967 году Fairchild Semiconductor представила семейство биполярных диодно-транзисторных логических схем (DTL) и транзисторно-транзисторных логических схем (TTL) семейства Micromatrix. Комплементарная технология металл-оксид-полупроводник (CMOS) открыла дверь к широкой коммерциализации матриц затворов.

Первые вентильные матрицы КМОП были разработаны Робертом Липпом в 1974 году для International Microcircuits, Inc. (IMI). Стандартная технология ячеек металл-оксид-полупроводник (MOS) была представлена Fairchild и Motorola под торговыми названиями Micromosaic и Polycell в 1970-х годах. Позднее эта технология была успешно коммерциализирована компаниями VLSI Technology (основана в 1979 г.) и LSI Logic (1981 г.).

Успешное коммерческое применение схемы вентильной матрицы было найдено в недорогих 8-битных персональных компьютерах ZX81 и ZX Spectrum , представленных в 1981 и 1982 годах. Они использовались Sinclair Research (Великобритания) в основном как недорогие решения ввода-вывода. нацелен на обработку компьютерной графики.

Настройка производилась путем изменения металлической маски межблочного соединения. Массивы ворот имели сложность до нескольких тысяч ворот; это теперь называется среднеуровневой интеграцией. Более поздние версии стали более универсальными, с различными базовыми матрицами, адаптированными как для металлических, так и для поликремниевых слоев. Некоторые базовые матрицы также включают элементы оперативной памяти (RAM).

В середине 1980-х разработчик выбирал производителя ASIC и реализовывал свой дизайн, используя инструменты проектирования, доступные от производителя. Хотя сторонние инструменты проектирования были доступны, не было эффективной связи от сторонних инструментов проектирования с топологией и фактическими характеристиками производительности полупроводниковых процессов различных производителей ASIC. Большинство дизайнеров использовали заводские инструменты для завершения реализации своих проектов. Решением этой проблемы, которое также привело к созданию устройства с гораздо большей плотностью, стало внедрение стандартных ячеек.

Каждый производитель ASIC может создавать функциональные блоки с известными электрическими характеристиками, такими как задержка распространения, емкость и индуктивность, которые также могут быть представлены в сторонних инструментах. Конструкция со стандартной ячейкой - это использование этих функциональных блоков для достижения очень высокой плотности затвора и хороших электрических характеристик.

Дизайн со **стандартными ячейками** занимает промежуточное положение между **вентильным массивом** и **частично настраиваемым дизайном** и **полностью настраиваемым дизайном** с точки зрения его единовременных затрат на проектирование и периодических затрат на компоненты, а также производительности и скорости разработки (включая время выхода на рынок).

К концу 1990-х годов стали доступны инструменты логического синтез. Интегральные схемы (ИС) со стандартными ячейками проектируются на следующих концептуальных этапах, называемых **потоком проектирования электроники**, хотя на практике эти этапы существенно перекрываются:

Разработка требований: группа инженеров-проектировщиков начинает с неформального понимания требуемых функций для новой ASIC, обычно получаемого на основе анализа требований. Дизайн на уровне передачи регистров (RTL): группа разработчиков создает описание ASIC для достижения этих целей, используя язык описания оборудования.

Этот процесс похож на написание компьютерной программы на языке высокого уровня.

Функциональная проверка: пригодность для использования подтверждается функциональной проверкой. Это может включать такие методы, как логическое моделирование с помощью испытательных стендов, формальная проверка, эмуляция или создание и оценка эквивалентной чистой модели программного обеспечения, как в Simics. Каждый метод проверки имеет преимущества и недостатки, и чаще всего для проверки ASIC используются несколько методов вместе. В отличие от большинства ПЛИС, ASIC не могут быть перепрограммированы после изготовления, и поэтому не совсем правильные конструкции ASIC намного дороже, что увеличивает потребность в полном тестировании.

Логический синтез: логический синтез преобразует дизайн RTL в большую коллекцию, называемую конструкциями нижнего уровня, называемыми стандартными ячейками. Эти конструкции взяты из библиотеки стандартных ячеек, состоящей из предварительно охарактеризованных наборов логических вентилей, выполняющих определенные функции. Стандартные ячейки обычно специфичны для планируемого производителя ASIC. Полученный набор стандартных ячеек и необходимых электрических соединений между ними называется списком соединений на уровне ворот.

Размещение: затем список соединений на уровне ворот обрабатывается с помощью инструмента размещения, который помещает стандартные ячейки в область кристалла интегральной схемы, представляющую окончательную ASIC. Инструмент размещения пытается найти оптимальное размещение стандартных ячеек с учетом множества заданных ограничений.

Маршрутизация: инструмент маршрутизации электроники принимает физическое размещение стандартных ячеек и использует список соединений для создания электрических соединений между ними. Поскольку пространство поиска велико, в результате этого процесса будет получено «достаточное», а не «оптимальное» решение. На выходе получается файл, который можно использовать для создания набора фотошаблонов, позволяющих предприятию по производству полупроводников, обычно называемому «фабрикой» или «литейным цехом», производить физические интегральные схемы.

Размещение и маршрутизация тесно взаимосвязаны и в электронном дизайне все вместе называются местом и маршрутом.

Подпись: Учитывая окончательную схему, извлечение схемы вычисляет паразитные сопротивления и емкости.

В случае цифровой схемы - это затем будет дополнительно преобразовано в информацию о задержке, по которой можно оценить характеристики схемы, обычно с помощью статического временного анализа. Этот и другие заключительные тесты, такие как проверка правил проектирования и анализ мощности, которые вместе называются подписанием, предназначены для обеспечения правильной работы устройства во всех экстремальных условиях процесса, напряжения и температуры. Когда это тестирование завершено, информация о фотомаске передается для изготовления чипа.

Эти шаги, реализованные с уровнем квалификации, обычным в отрасли, почти всегда приводят к созданию конечного устройства, которое правильно реализует исходную конструкцию, если только в процессе физического изготовления позже не появятся недостатки. Этапы проектирования, также называемые потоком проектирования, также являются общими для стандартного дизайна продукта. Существенная разница заключается в том, что при проектировании стандартных ячеек используются библиотеки ячеек производителя, которые потенциально использовались в сотнях других реализаций дизайна и поэтому представляют гораздо меньший риск, чем полностью индивидуальный дизайн. Стандартные ячейки обеспечивают экономичную плотность конструкции и могут также эффективно интегрировать IP-ядра и статическую оперативную память (SRAM), в отличие от массивов вентилей.

Воротная решетка и полу-индивидуальный дизайн

Конструкция матрицы затворов - это метод производства, в котором предварительно определены диффузные слои, каждый из которых состоит из транзисторов и других активных устройств, а электронные пластины, содержащие такие устройства, «хранятся на складе» или не соединяются до стадии металлизации в процессе изготовления.

Процесс физического проектирования определяет взаимосвязи этих уровней для конечного устройства. Для большинства производителей ASIC он состоит из двух-девяти металлических слоев, каждый из которых проходит перпендикулярно тому, что находится под ним. Единовременные инженерные расходы намного ниже, чем у полностью нестандартных конструкций, поскольку фотолитографические маски требуются только для металлических слоев. Производственные циклы намного короче, поскольку металлизация - сравнительно быстрый процесс; тем самым ускоряя выход на рынок.

ASIC с вентильным массивом всегда являются компромиссом между быстрым дизайном и производительностью, поскольку отображение заданного дизайна на то, что производитель считает стандартной пластиной, никогда не дает 100% -ного использования схемы. Часто трудности с маршрутизацией межмодульного соединения требуют миграции на массивное устройство большего размера с последующим увеличением стоимости отдельных частей.

Эти трудности часто являются результатом программного обеспечения EDA для компоновки, используемого для разработки межсоединения. Чистая логическая конструкция вентильной матрицы сегодня редко реализуется разработчиками схем, поскольку она почти полностью заменена программируемыми устройствами. Наиболее известными из таких устройств являются программируемые вентильные матрицы (FPGA), которые могут быть запрограммированы пользователем и, таким образом, предлагают минимальные затраты на инструменты, единовременное проектирование, лишь незначительно увеличенную стоимость детали и сопоставимую производительность.

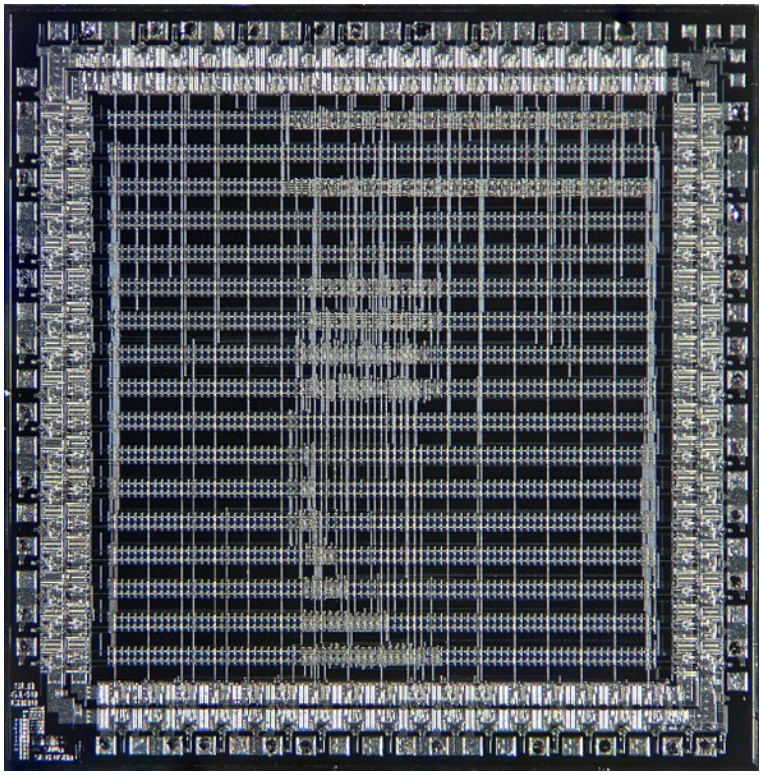


Рис.23 - Фотография микросхемы вентильной матрицы ASIC с предварительно заданными логическими ячейками и пользовательскими соединениями. В этой конкретной конструкции используется менее 20% доступных логических вентилей.

Чистая логическая конструкция вентильной матрицы сегодня редко реализуется разработчиками схем, поскольку она почти полностью заменена программируемыми устройствами. Наиболее известными из таких устройств являются программируемые вентильные матрицы (FPGA), которые могут быть запрограммированы пользователем и, таким образом, предлагают минимальные затраты на инструменты, единовременное проектирование, лишь незначительно увеличенную стоимость детали и сопоставимую производительность.

Сегодня вентильные массивы превращаются в структурированные ASIC, которые состоят из большого IP-ядра, такого как ЦП, блоков цифрового сигнального процессора, периферийных устройств, стандартных интерфейсов, интегрированной памяти, SRAM и блока реконфигурируемой незафиксированной логики.

Этот сдвиг во многом объясняется тем, что устройства ASIC способны интегрировать большие блоки системных функций, а системы на кристалле (**SoC**) требуют связующей логики, подсистем связи (таких как сети на кристалле), периферийных устройств и других компонентов, а не только функциональных блоков и базовых компонентов. взаимосвязь. В их частом использовании в полевых условиях термины «вентильный массив» и «полу-пользовательский» являются синонимами при обращении к ASIC. Инженеры-технологи чаще используют термин **«полу-заказной»,** тогда как «вентильная матрица» чаще используется разработчиками логики (или на уровне вентилей).

Полностью индивидуальный дизайн

Напротив, полностью настраиваемый дизайн ASIC определяет все фотолитографические слои устройства. Полностью индивидуальный дизайн используется как для дизайна ASIC, так и для стандартного дизайна продукта. Преимущества полностью настраиваемой конструкции включают уменьшение площади (и, следовательно, повторяющиеся затраты на компоненты), повышение производительности, а также возможность интеграции аналоговых компонентов и других предварительно разработанных - и, таким образом, полностью проверенных - компонентов, таких как ядра микропроцессоров, которые образуют система на микросхеме.

К недостаткам полностью индивидуального проектирования можно отнести увеличенное время производства и проектирования, повышенные единовременные затраты на проектирование, большую сложность систем автоматизированного проектирования (САПР) и автоматизации

электронного проектирования, а также гораздо более высокие требования к квалификации со стороны команда дизайнеров.

Однако для цифровых проектов библиотеки ячеек со "стандартными ячейками" вместе с современными системами САПР могут предложить значительные преимущества в производительности / стоимости с низким риском. Инструменты автоматического макета просты и быстры в использовании, а также предлагают возможность «вручную настраивать» или вручную оптимизировать любой аспект дизайна, ограничивающий производительность. Это разработано с использованием основных логических вентилей, схем или макета специально для проекта.

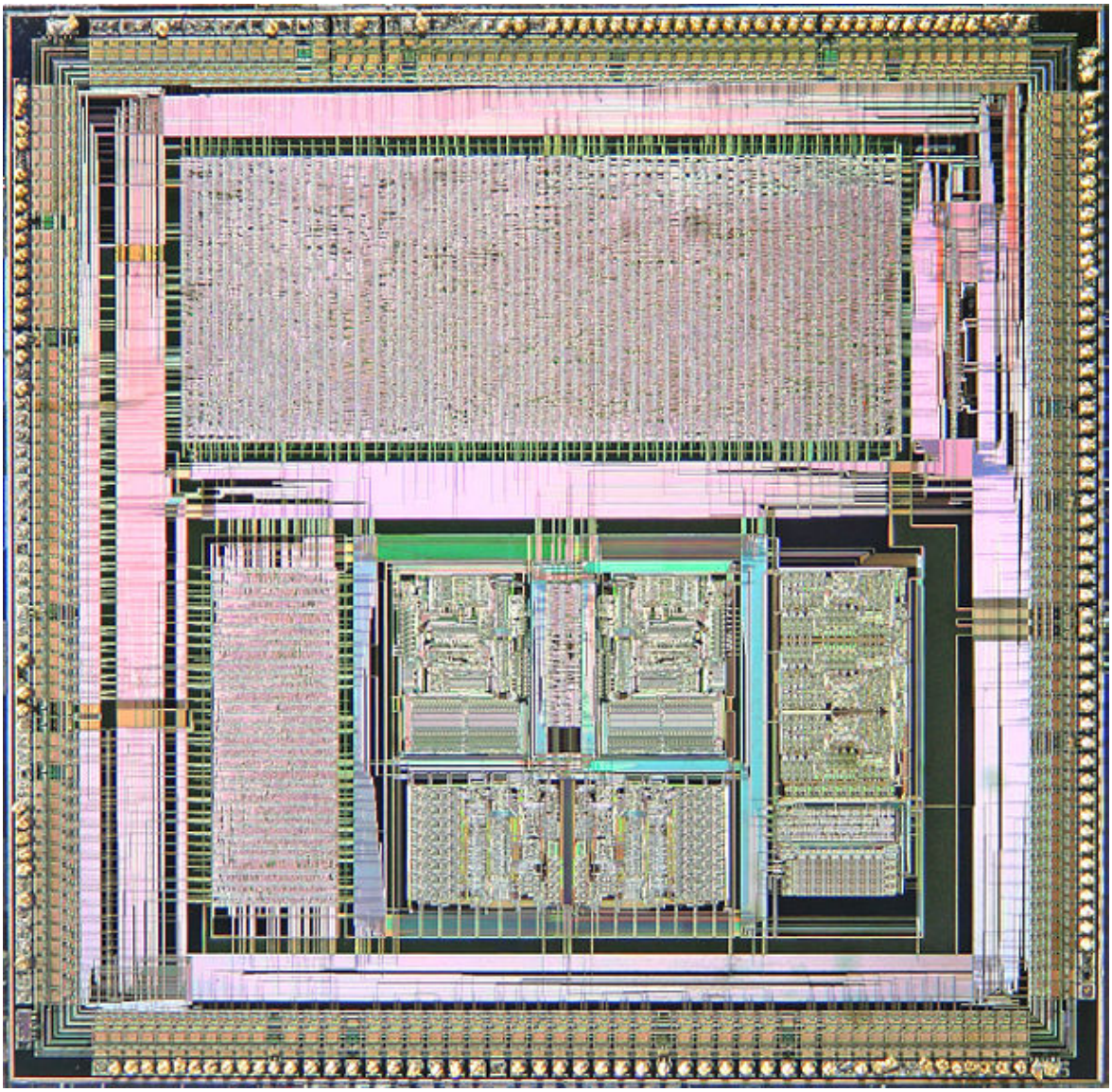


Рис. 24 - Фотография под микроскопом пользовательской ASIC (набор микросхем 486), показывающая дизайн на основе затвора вверху и нестандартную схему внизу.

Структурированный дизайн ASIC (также называемый платформенным дизайном ASIC») - относительно новая тенденция в полупроводниковой промышленности, что привело к некоторым вариациям в ее определении.

Однако основная предпосылка структурированной ASIC заключается в том, что как время производственного цикла, так и время цикла проектирования сокращаются по сравнению с ASIC на основе ячеек, благодаря наличию заранее определенных металлических слоев (что сокращает время производства) и предварительной характеристики того, что находится на кремнии (что сокращает время цикла проектирования).

Определение из основ встроенных систем гласит, что: в структуре «структурированной ASIC» уровни логической маски устройства заранее определены поставщиком ASIC (или в некоторых случаях третьей стороной). Различия в дизайне и настройка достигаются за счет создания пользовательских металлических слоев, которые создают пользовательские связи между предопределенными логическими элементами нижнего уровня.

Технология «структурированной ASIC» рассматривается как преодоление разрыва между программируемыми логическими схемами вентильных матриц и ASIC «со стандартной ячейкой». Поскольку только небольшое количество слоев микросхем должно производиться на заказ, конструкции «структурированных ASIC» имеют гораздо меньшие единовременные затраты (NRE), чем микросхемы «со стандартной ячейкой» или «полностью настраиваемые», для которых требуется полный набор масок. производиться для любого дизайна. Это фактически то же определение, что и массив вентилей.

Что отличает структурированную ASIC от матрицы затворов, так это то, что в матрице затворов заранее определенные металлические слои служат для ускорения производственного цикла. В структурированной ASIC заранее заданная металлизация используется в первую очередь для снижения стоимости наборов масок, а также для значительного сокращения времени цикла проектирования. Например, в проекте на основе ячеек или вентильной матрицы пользователь часто должен сам проектировать структуры питания, тактовой частоты и тестирования. Напротив, они предопределены в большинстве структурированных ASIC и поэтому могут сэкономить время и деньги разработчика по сравнению с проектами на основе вентильных матриц. Точно так же инструменты проектирования, используемые для структурированных ASIC, могут быть значительно дешевле и проще (быстрее) в использовании, чем инструменты на основе ячеек, потому что им не нужно выполнять все функции, которые выполняют инструменты на основе ячеек.

Технологии производства SoC: FPGA

Программируемая пользователем вентильная матрица (ППВМ, англ. field-programmable gate array, FPGA) — полупроводниковое устройство, которое может быть сконфигурировано производителем или разработчиком после изготовления; наиболее сложная по организации разновидность программируемых логических интегральных схем.

Программируются путём изменения логики работы принципиальной схемы, например, с помощью исходного кода на языке описания аппаратуры (например Verilog). Могут быть модифицированы практически в любой момент в процессе их использования.

Состоят из конфигурируемых логических блоков, подобных переключателям с множеством входов и одним выходом (**логические вентили, gates**). В цифровых схемах такие переключатели реализуют базовые двоичные операции AND, NAND, OR, NOR и XOR. Принципиальное отличие ППВМ состоит в том, что и функции блоков, и конфигурация соединений между ними могут меняться с помощью специальных сигналов, посылаемых схеме.

! В специализированных интегральных схемах (ASIC) используются логические матрицы, аналогичные ППВМ по строению, однако они конфигурируются один раз в процессе производства, в то время как ППВМ могут постоянно перепрограммироваться и менять топологию соединений в процессе использования. Однако такая гибкость требует существенного увеличения количества транзисторов микросхемы.

В ранних ПЛИС программированием можно было изменять только связи между вентилями; в 1985 году сооснователи **Xilinx** Росс Фримен и Бернард Вандершмит разработали первую коммерчески успешную ППВМ — **XC2064**, имеющую программируемые вентили и программируемые соединения между ними (**в 2005 году за это изобретение Фримен был занесён в Национальный зал славы изобретателей США**).

Сама концепция программируемых вентильных матриц, логических вентилей и логических блоков запатентована Дэвидом Пейджем и Луверном Петерсоном в 1985 году.

В 1990-х годах произошёл резкий скачок интереса к ППВМ, возросла их сложность и объёмы производства: если в первые годы они использовались в основном в области телекоммуникаций и сетей связи, то к концу десятилетия они нашли применение в потребительских товарах, в автомобильной промышленности и других отраслях.

В 1997 году Адриан Томпсон объединил генетические алгоритмы и технологию ППВМ для создания устройства, способного отличать звуковые тоны частотой 1 КГц и 10 КГц. Генетические алгоритмы позволили с помощью вентильной матрицы размером 64×64 на микросхеме фирмы Xilinx создать конфигурацию, необходимую для решения поставленной задачи.

В те же годы начали широко применяться для прототипирования специализированных интегральных и процессоров общего назначения. В начале 2000-х годов ППВМ начали использоваться для ускорения специфических операций в серверных узлах как в индустрии высокопроизводительных вычислений, так и в машинах баз данных (Netezza).

Во второй половине 2010-х годов отмечен всплеск интереса к технологии в связи с эффективностью применения для глубокого обучения (прежде всего за счёт возможности реализации арифметики с пониженной точностью и безрегистровых вычислений — аналога тензорного процессора Google), а у инфраструктурных облачных провайдеров появилась возможность приобрести ППВМ по подписке из публичного облака (Amazon F1, Baidu, Tencent, Huawei).

К 2018 году объём мирового рынка ППВМ составил около $5,7 млрд, крупнейшие производители — Xilinx (51 % рынка), Intel (36 %, за счёт активов Altera), Microchip (17 %), Lattice Semiconductor (9 %)

В ППВМ (FPGA) имеется три типа программируемых элементов:

- нескоммутированные программируемые логические блоки (ПЛБ);
- блоки ввода-вывода (БВВ);
- внутренние связи.

ПЛБ являются функциональными элементами для построения логики пользователя. БВВ обеспечивают связь между контактами корпуса и внутренними сигнальными линиями.

Программируемые ресурсы внутренних связей обеспечивают управление путями соединения входов и выходов ПЛБ и блоков ввода-вывода (БВВ) на соответствующие сети.

Все каналы трассировки имеют одинаковую ширину (одинаковое количество проводников). Большинство блоков ввода-вывода (БВВ) вписываются либо в одну строку (по высоте), либо в один столбец (по ширине) массива вентилей. Логический блок (ПЛБ) классической ППВМ состоит из таблицы истинности (англ. lookup table, LUT) на несколько входов и один триггер (в ранних реализациях использовалось 4 входа, впоследствии — большее число входов, что позволило задействовать меньшее число логических блоков для типичных приложений.

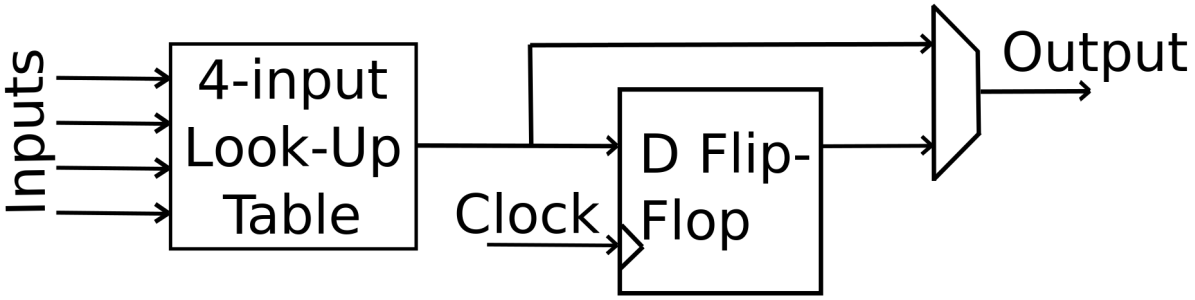


Рис. 25 - Типичный логический блок

Логический блок (ПЛБ) имеет таблицу истинности на четыре входа и вход синхронизации (clock). Выход блока только один — регистровая или нерегистровая выходная таблица истинности. Поскольку сигналы синхронизации в коммерческих ППВМ (а часто и другие сигналы, распараллеливающиеся на большое количество входов — high-fanout signals) трассируются особым образом специальными трассировочными цепями, управление этими сигналами делается отдельно.

Для приведённого примера архитектуры расположение контактов логического блока показано ниже (рис. 26):

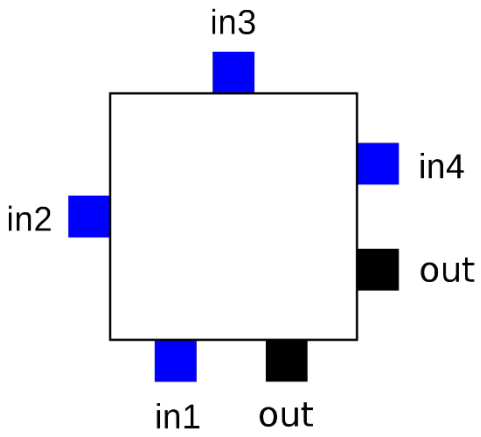


Рис. 26 - Расположение контактов логического блока

Входы расположены на отдельных сторонах логического блока; выходной контакт может трассироваться в двух каналах: либо справа от блока, либо снизу. Выходные контакты каждого логического блока могут соединяться с трассировочными сегментами в смежных каналах. Аналогично, контактная площадка блока ввода-вывода (pad) может соединяться с трассировочным элементом в любом смежном канале. Например, верхняя контактная площадка чипа может соединяться с любым из W проводников (где W — ширина канала) в горизонтальном канале, расположенном непосредственно под ним.

Как правило, трассировка ППВМ несегментирована, то есть каждый сегмент проводника соединяет только один логический блок с переключательным блоком. Из-за огибания программируемых переключателей в переключательном блоке трассировка получается более длинной. Для увеличения скорости внутрисистемных соединений, в некоторых архитектурах ППВМ между логическими блоками используются более длинные трассировочные соединения.

В месте пересечения вертикальных и горизонтальных каналов создаются переключательные блоки. При такой архитектуре для каждого проводника, входящего в переключательный блок, существуют три программируемых переключателя, которые позволяют ему подключаться к трём другим проводникам в смежных сегментах канала. Модель или топология выключателей, используемая в этой архитектуре, является планарной или доменной топологией переключательных блоков. В этой топологии проводник трассы номер 1 подключается только к проводнику трассы номер 1 в смежных каналах, проводник трассы номер 2 подключается только к проводникам трассы номер 2 и так далее.

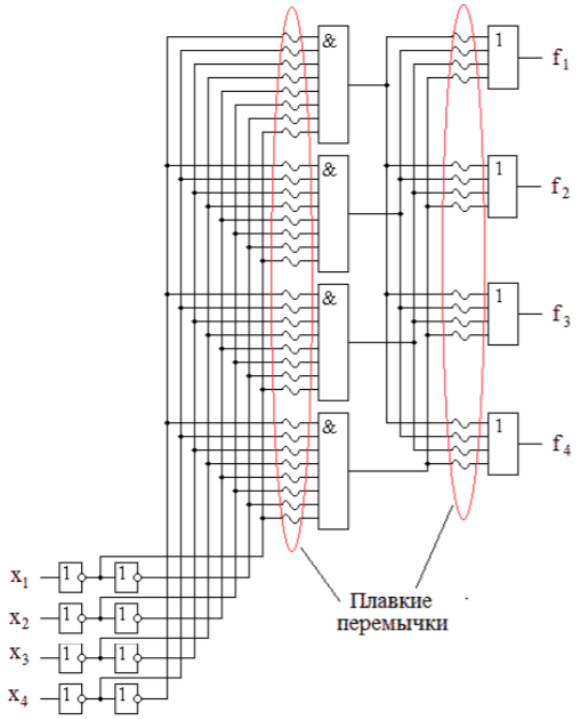
Современные семейства ППВМ расширяют перечисленные выше возможности и обладают встроенными функциями высокого уровня, благодаря наличию которых удаётся уменьшить площадь кристалла и ускорить выполнение типовых подзадач в сравнении с реализацией на основе примитивов. Примерами таких функций являются мультиплексоры, блоки цифровой обработки сигналов, встроенные процессоры, быстрая логика ввода-вывода и встроенная память.

ППВМ также широко применяются для систем проверки пригодности, в том числе в докремниевой и послекремниевой проверке пригодности, а также при разработке программ для **встраиваемых систем и SoC**. Это позволяет компаниям-производителям интегральных схем проверять работоспособность своих устройств до изготовления их на заводе, сокращая время выхода изделия на рынок.

ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ (FPGA)

(презентационный материал, 33 слайда)

ФГАОУ ВО Национальный Исследовательский Томский Политехнический университет



Программируемые логические матрицы — наиболее традиционный тип ПЛИС, имеющий программируемые матрицы «И» и «ИЛИ». В зарубежной литературе соответствующими этому классу аббревиатурами являются FPLA (Field Programmable Logic Array) и FPLS (Field Programmable Logic Sequensers). Примерами таких ПЛИС могут служить отечественные схемы К556РТ1, РТ2, РТ21.

Недостатком структуры ПЛМ является слабое использование ресурсов программируемой матрицы ИЛИ. Поэтому была предложена более простая, но тем не менее, более эффективная архитектура программируемой матричной логики (ПМЛ). В английской терминологии - Programmable Array Logic (PAL).

Рис.6 Пример ПЛМ

7

§ РАЗДЕЛ 3: Network on a chip (NoC)

Сравнение NoC и шинной архитектуры

Мы уже смирились с тем, что рост тактовой частоты процессоров остановился и производители пошли по пути распараллеливания вычислений. Однако и число ядер типичного процессора общего назначения, быстро одолев отметки 2 и 4, остановилось в районе 8. Некоторые даже собрались хоронить закон Мура.

У такого застоя есть объективная причина. Если разница между 2, 4 или 8 ядрами скорее количественная, то уже 16-ядерный процессор сталкивается с принципиальными ограничениями традиционной архитектуры. Дело в том, что на протяжении последних нескольких десятилетий основой коммуникации между отдельными IP-блоками чипа служила шина. Пока блоков было немного, она справлялась, но, когда начали «плодиться» ядра, эта архитектура исчерпала себя. Шина представляет собой общую среду передачи данных, к которой подключено несколько блоков процессора. В каждый момент времени один блок может передавать данные, а все остальные — получать. Если нескольким блокам нужно передавать одновременно — возникает коллизия, а значит и задержка. При числе ядер больше восьми задержки становятся неприемлемо большими, практически полностью перечёркивая преимущества параллельной работы нескольких ядер.

Число ядер можно увеличить ещё немного, разделив шину на несколько сегментов, объединённых мостами, однако это скорее «костыль», который плохо масштабируется и не решает основную проблему. Настоящее решение, которое позволит объединять сотни блоков на одном чипе — это хорошо известная всем сеть с коммутацией пакетов, или **Network on Chip.**

Переход от шины к сети вполне закономерен. Именно так развивались телекоммуникационые сети: радиоэфир — типичная «шина», телефонные сети — коммутация каналов с помощью матричных коммутаторов, интернет — коммутация пакетов. Так же развивалась и компьютерная периферия — современная шина PCI Express на самом деле вовсе не шина, а сеть с топологией типа звезда.

Так же развиваются и процессоры — сначала прямые соединения между блоками, затем шины и матричные коммутаторы и, наконец, сети.

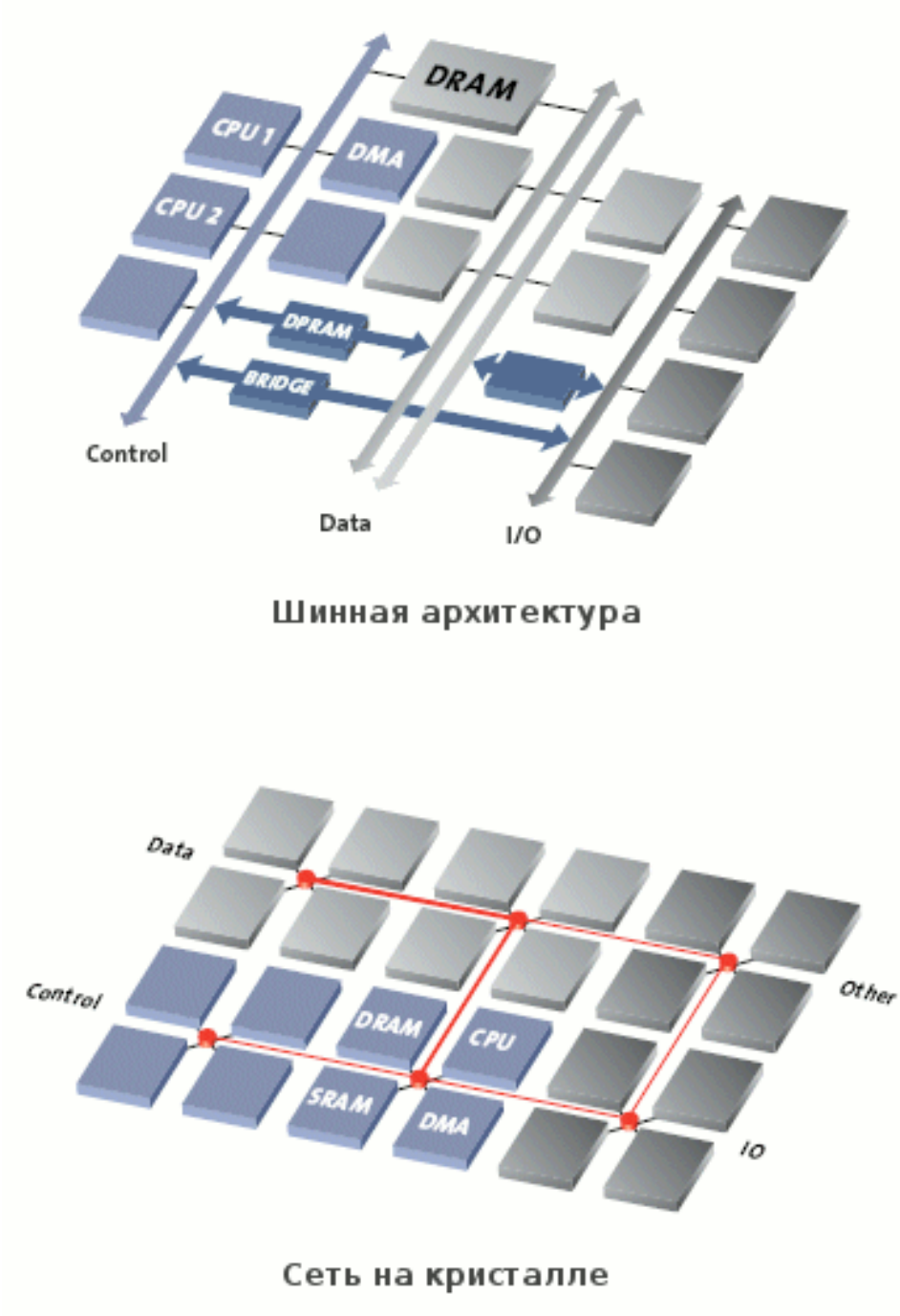


Рис. 27 - Сравнение NoC и шинной архитектуры

В архитектуре NoC каждое ядро или блок процессора соединён с маршрутизатором, через который происходит его общение с другими блоками. Сами маршрутизаторы объединены в сеть, по которой пакеты данных путешествуют от одного блока к другому, так же как пакеты в обычной компьютерной сети.

Это значительно упрощает топологию микросхемы и снимает ограничения по масштабированию — в отличие от шины, множество блоков способно общаться одновременно, не мешая друг другу. Компьютерное моделирование и опытные образцы многоядерных процессоров показывают, что при большом количестве ядер такая архитектура превосходит традиционную по многим показателям.

Естественно, напрямую перенести логику и протоколы работы интернета внутрь чипа было бы неразумно и неэффективно. Здесь совсем другие технологические ограничения и задачи:

Очень жёсткие требования к задержкам и энергопотреблению. Коммутаторы должны работать с наносекундными задержками и быть очень экономичными. Расходы энергии на передачу данных между блоками составляют значительную часть общего потребления современных чипов.

Простота и минимализм. Коммутаторы на чипе должны занимать мало места, а значит не могут иметь сложную логику и большой размер буфера.

Параллельное, а не последовательное соединение. На физическом уровне внутри чипа выгоднее предавать биты не последовательно по одному проводнику, а по 32 или 64 параллельным каналам.

Исследованиями NoC занимаются ведущие компании и университеты мира. Так, еще в 2007 году Intel разработала экспериментальный процессор с 80-ю ядрами и производительностью 1 терафлопс при энергопотреблении всего в 62 ватта. В 2010 был представлен 48-ядерный «Облачный компьютер на кристалле» (Single chip cloud computer).

В апреле этого года была опубликована работа группы ученых MIT, которые создали прототип 16-ядерного процессора, в котором были применены специфические для NoC-систем оптимизации — виртуальный обход (virtual bypassing) и сигналы с малой амплитудой (low-swing signaling). Эти технологии позволили приблизиться к теоретическим пределам пропускной способности и задержек и заметно снизить энергопотребление.

Как они работают? Обычный маршрутизатор сохраняет полученный пакет в буфер, анализирует его заголовок и решает, куда его отправить дальше. Virtual bypassing позволяет передать пакет практически без задержек, за счёт того, что заголовок посылается заранее, и коммутатор успевает сделать нужные переключения цепей к тому моменту, как придёт тело пакета.

Таким образом, пакет идёт без остановок, минуя буфер. Low-swing signaling — это уменьшение разницы между напряжениями 0 и 1 в проводнике, за счёт чего удалось дополнительно сократить энергопотребление. В сумме эти усовершенствования поднимают пропускную способность и экономичность более чем в полтора раза.

Кроме улучшения таких характеристик, как энергопотребление и скорость, архитектура NoC даёт ещё одно важное преимущество. Она легко позволяет объединять не только однородные ядра, но и вообще любые блоки на одном чипе. Как и в компьютерных сетях, физический и транспортный уровни работают одинаково для любых типов данных и протоколов.

Можно без особых проблем поставить на место одного или нескольких из универсальных вычислительных ядер любой другой IP-блок, например, графическое ядро, специализированный сигнальный процессор или контроллер какого-либо устройства. И, также, как и в сетях, можно реализовать поддержку Quality of Service на уровне чипа, что может быть полезно для систем реального времени и виртуализации.

NoC для объединения ядер процессоров пока ещё имеют экспериментальный статус, однако для объединения разнородных блоков в системах на кристалле NoC разрабатываются и применяются довольно давно. Решения таких компаний, как Sonics или Arteris используются в микросхемах Samsung, Qualcomm и даже Intel. Возможно, уже скоро сетевая архитектура начнёт вытеснять шины и из “святая святых” микроэлектроники — многоядерных центральных процессоров. И тогда число ядер снова начнёт стремительно расти. Так что закон Мура хоронить ещё рано.

Научная статья
«A comparison of Network-on-Chip and Busses»
by Arteris (www.arteris.com)

Wireless network on chip (WiNoC)

При проектировании систем на кристалле перед разработчиками часто стоит проблема задержек и рассинхронизации сигналов. Различные методы организации сетевых структур на кристалле, теоретически позволяющие минимизировать задержки и потери, имеют недостатки и сложны в реализации. Наиболее перспективным путем решения этой проблемы представляется парадигма беспроводных сетей на кристалле, которая позволяет обойти ограничения классических сетей, а также обеспечить связь между наномасштабными компонентами микросхем и макроуровнем.

В последнее время усилия многих специалистов сосредоточились на теории создания беспроводных сетей на чипе (**Wireless Network on Chip, WiNoC**). Нужно отметить, что идея WiNoC важна также и для развития нанотехнологий. На сегодняшний день актуальна проблема обеспечения электрического контакта наноэлектронного устройства или схемы и макроскопического элемента без существенных потерь в достижимой на наноуровне плотности тока. Используемый сейчас метод литографического изготовления контактных площадок неэффективен для широкомасштабной параллельной обработки сигналов, требующей объединения множества наносистем на кристалле.

Эту проблему можно решить с помощью беспроводной связи. Для реализации бесконтактных соединений между слоями микросхемы могут использоваться эффекты емкостной связи между миниатюрными контактными площадками или индуктивной – **между спиральными катушками индуктивности.**

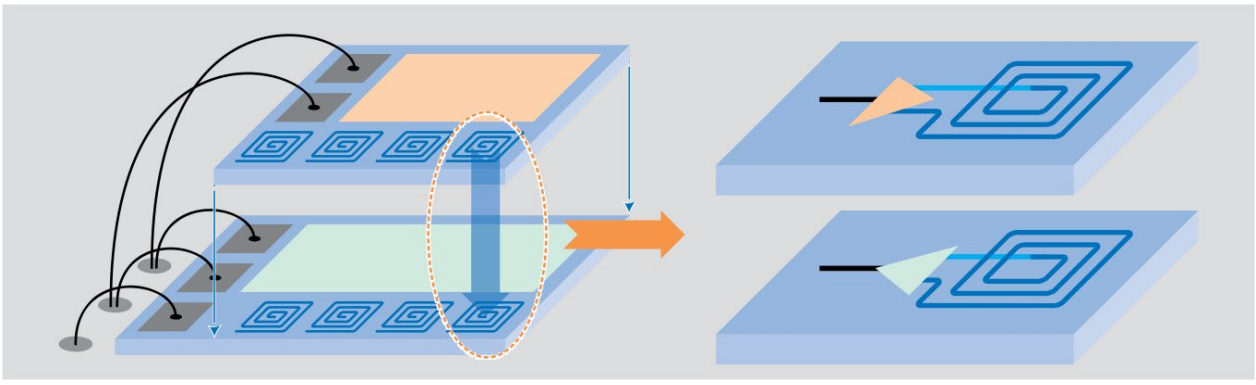


Рис. 28 – Тракт беспроводного обмена в NoC

Реализация емкостной связи при толщине слоя более 1 мкм требует мощности более 10 мВт на одиночный контакт, что неприемлемо при большом (>1000) количестве связей между чипами.

Впрочем, по сравнению с проводными соединениями величину 10 мВт можно считать неплохим показателем. Например, при скорости передачи данных 23 Гбит/с традиционный проводной контакт рассеивает мощность около 200 мВт, а контактная пара на основе металлических микросфер – 20 мВт. Тем не менее, энергетически выгодней применять индуктивную связь (**рис.28**).

Например, при технологической норме изготовления индукторов 90 нм при скорости 23 Гбит/с рассеиваемая мощность снижается до 2 мВт на контакт или – в пересчете на удельное энергопотребление – до рекордных 0,11 пикоДж/бит в режиме передачи данных и 0,03 пикоДж/бит – в режиме приема

Заключение. Увеличивающийся интерес к технологиям NoC среди разработчиков систем на кристалле способствовал появлению еще одной разновидности беспроводных сетей на кристалле (Wireless Network on Chip, WiNoC), в которых используются решения, сходные с макроскопическими.

Среди возможных подходов к реализации WiNoC в первую очередь следует отметить использование в одном кристалле и беспроводных, и традиционных проводниковых линий передачи сигналов. При этом вся система на кристалле условно делится на подсети из групп ядер, внутри которых коммуникация осуществляется проводными линиями.

A-WiNoC: Adaptive Wireless Network-on-Chip Architecture for Chip Multiprocessors, January 2014, IEEE Transactions on Parallel and Distributed Systems 26(12):1-1 DOI: 10.1109/TPDS.2014.2383384

Практическое занятие 1. Знакомство с процедурой проектирования цифровых устройств в пакете WebPACK ISE

Эта ознакомительная Практическое занятие позволяет студентам познакомиться с пакетом WebPACK ISE 10.1. Цели лабораторной работы состоят в следующем:

- создать новый проект;
- добавить файлы к проекту;
- провести анализ иерархического описания проекта;
- провести анализ выполненных назначений выводов СБИС;
- осуществить моделирование;
- осуществить полную компиляцию проекта с порождением файла для конфигурации СБИС.

Все поставленные цели выполняются пошагово с иллюстрацией промежуточных и итоговых результатов. Предполагается, что рабочим каталогом студентов при выполнении лабораторных работ является **D:\Student**.

Шаг 1. Запуск пакета и создание проекта

1. Выполните команду *Start* > *Programs* > *Design Suite 10.1* > *ISE* > *Project Navigator*.

2. В *Навигаторе* проекта выполните команду *File* > *New Project*. Откроется диалоговая панель помощника - *New Project Wizard* (рис. 3.1).

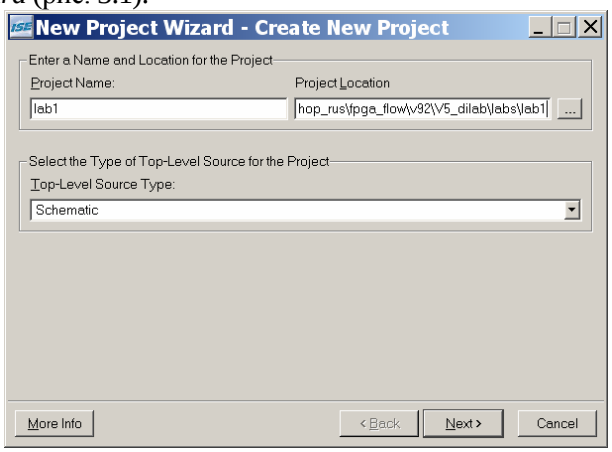


Рис. 3.1. Диалоговая панель мастера создания проекта

3. В поле *Project Name* введите имя проекта – **lab1**. Используя кнопку для выбора рабочей папки проекта введите **D:\Student\lab1**.

В поле *Top-Level Source Type* укажите способ представления проекта – *Shematic* (схемный ввод) или *HDL*.

Нажмите кнопку *Next*. Появится диалоговая панель *Device Properties* (рис. 3.2).

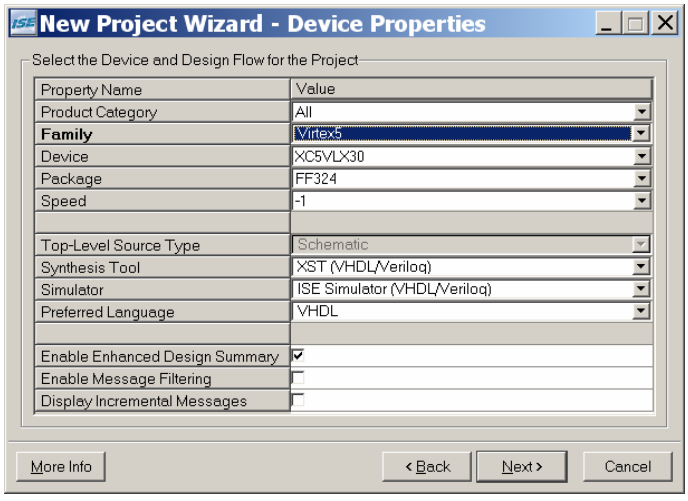


Рис. 3.2. Диалоговая панель выбора ПЛИС

4. Заполните поля указанными ниже значениями, затем нажмите кнопку *Next*:

- в поле *Device Family*: Virtex5;
- в поле *Device*: xc5vlx30;
- в поле *Package*: ff324;
- в поле *Speed Grade*: –1;
- в поле *Synthesis Tool*: XST (VHDL/Verilog);
- в поле *Simulator*: ISE Simulator (VHDL/Verilog);
- в поле *Preferred Language*: VHDL.

Появится диалоговая панель *Create New Source* (рис. 3.3). Панель может быть использована для создания файла (например, файла верхнего уровня в иерархии) для описания проектируемого модуля.

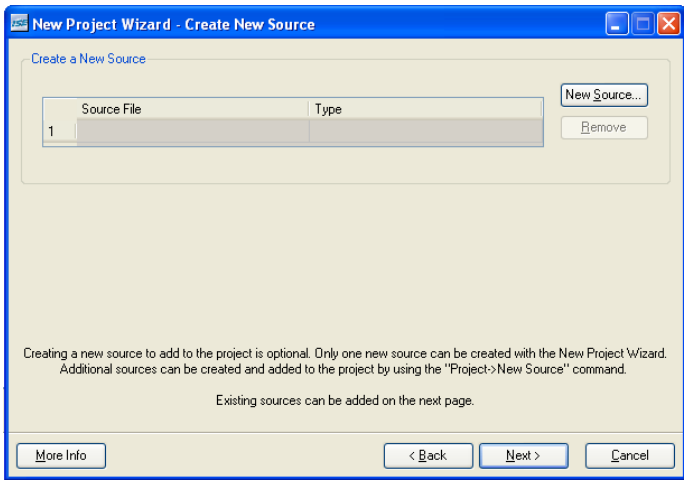


Рис. 3.3. Диалоговая панель создания нового источника проекта

В данной лабораторной работе все файлы с описанием модулей уже созданы.

5. Нажмите кнопку *Next*. Появится диалоговая панель *Add Existing Sources* (рис. 3.4). Эта панель позволяет добавить текстовые файлы и схемы в создаваемый проект.

6. Нажмите кнопку *Add Source*.

7. Укажите папку с исходными описаниями (рабочую папку проекта): **D:\Student\lab1** (папка с именем **lab1** создана заранее).

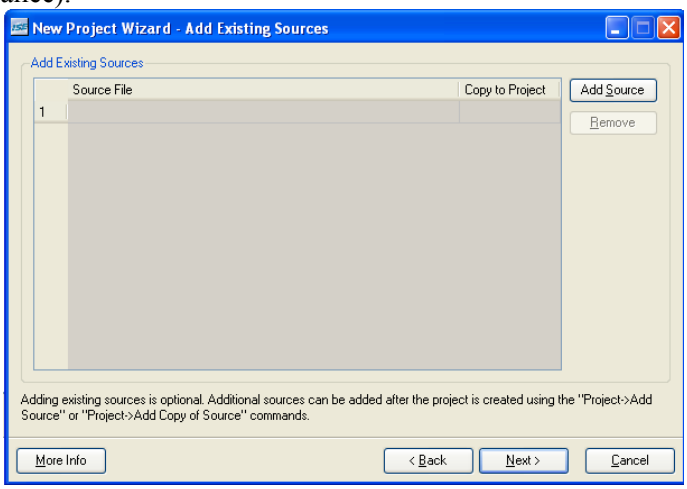


Рис. 3.4. Панель добавления новых источников в проект

8. Выберите файлы с текстовыми описаниями *.vhd (кроме файла clkx2.vhd), схемный файл lab1.sch, файл с настройками *core generator* – clkx2.xaw, файл с назначением выводов СБИС lab1.ucf.

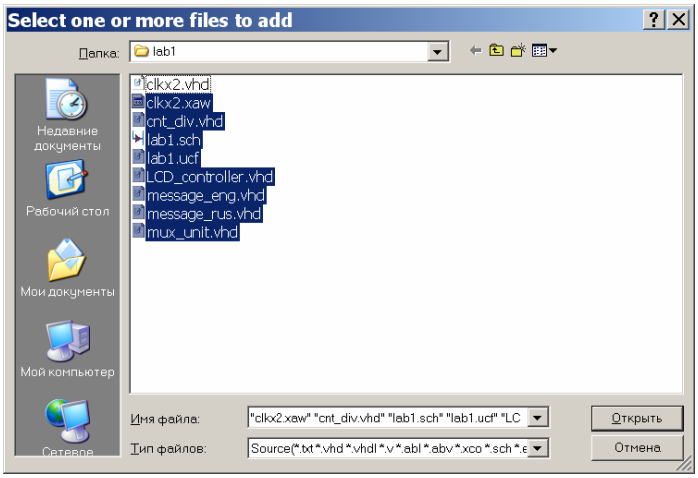


Рис. 3.5. Выбор необходимых для проекта файлов

9. Нажмите кнопку *Open* (*Открыть*). В появившемся окне (рис. 3.6) в поле *Copy to Project* оставьте все файлы неотмеченными, т.к. они уже находятся в рабочей папке проекта. Нажмите кнопку *Next*. В появившемся окне, содержащим информацию о сделанных назначениях, нажмите кнопку *Finish*.

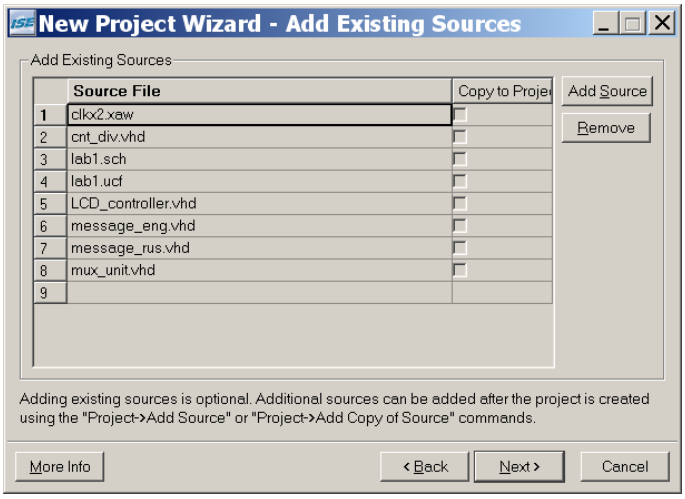


Рис. 3.6. Добавление файлов в проект

10. В появившемся окне (рис. 3.7) Вы можете указать этапы проектирования, в которых будет использован соответствующий файл. Нажмите кнопку *OK*.

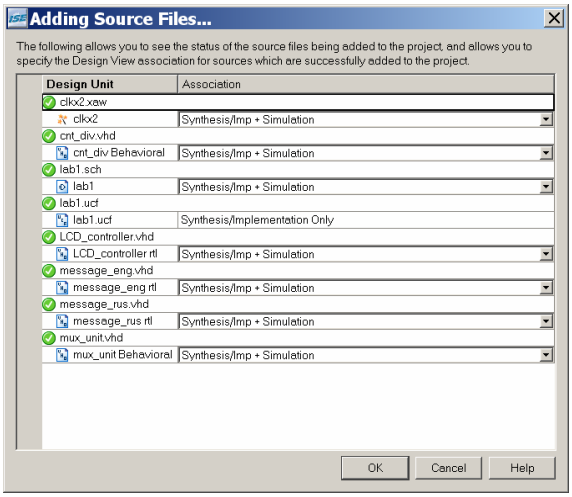


Рис. 3.7. Этапы проектирования, в которых используются добавленные в проект файлы

Проект создан. Появится окно *Навигатора* проекта (рис. 3.8).

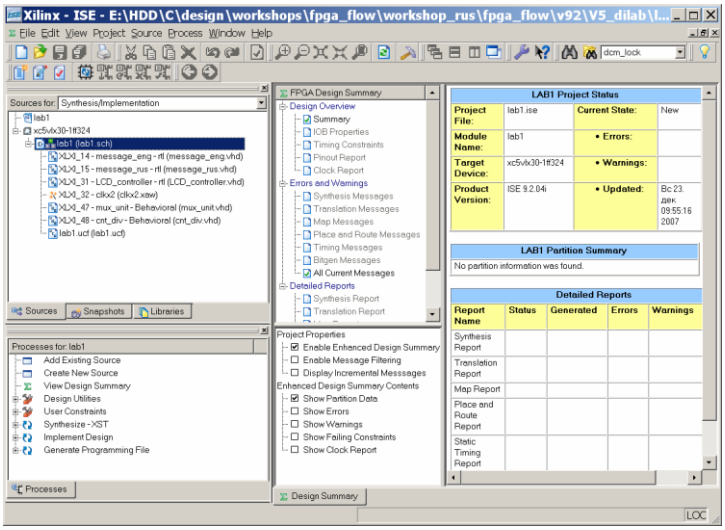


Рис. 3.8. Окно *Навигатора* проекта

Шаг 2. Анализ иерархии проекта и назначений выводов цифрового устройства

1. На странице иерархического отображения проекта закладки *Sources* (рис. 3.9) представлены:

- имя проекта – lab1;
- СБИС, выбранная для реализации проекта – xc5vlx30-1ff324;
- файл верхнего уровня иерархии – lab1.sch;
- файлы нижних уровней иерархии;
- файл с установками проекта – lab1.ucf.

Чтобы открыть схемный или текстовый файл необходимо дважды щелкнуть левой клавишей мыши по соответствующей строке.

2. Откройте схемный файл верхнего уровня иерархии описания проекта lab1.sch. Убедитесь, что открытая схема соответствует схеме, приведенной на (рис. 3.10).

3. Выбрав закладку *Sources* и затем закладку *Processes* (рис. 3.10), Вы перейдите к окну иерархического отображения проекта и окну этапов реализации проекта (рис. 3.11).

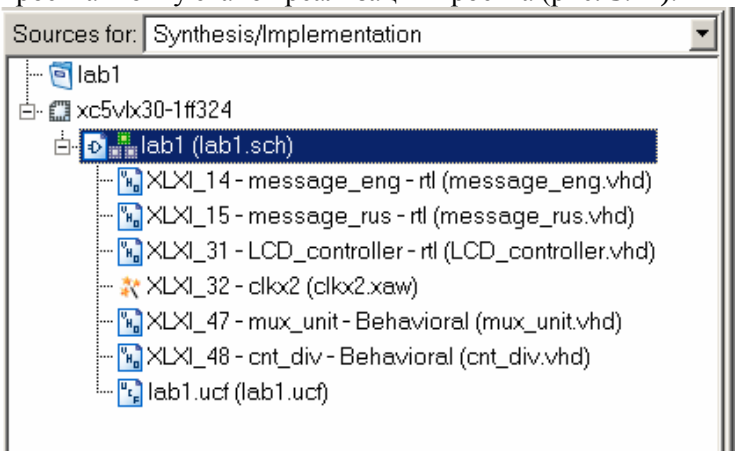


Рис. 3.9. Иерархическое отображение проекта

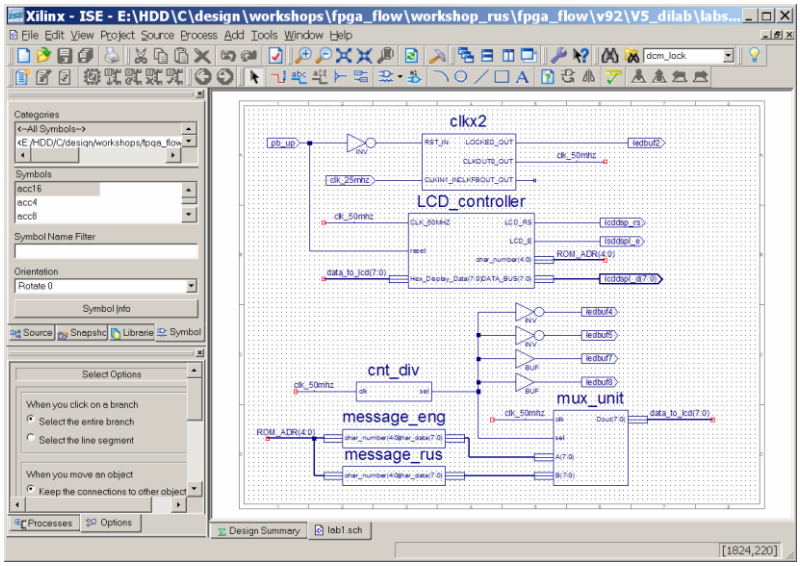


Рис. 3.10. Схемная реализация проекта

4. Выберите файл верхнего уровня в иерархии описаний lab1.sch (наведите на него курсор и один раз щелкните левой кнопкой мыши). В окне *Processes for: lab1* разверните папку *User Constraints* (пользовательские установки) и выберите строчку *Floorplan IO-Pre-Synthesis* (план выводов предварительного синтеза).

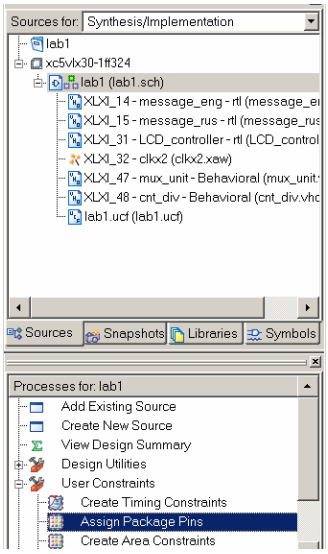


Рис. 3.11. Окна иерархического отображения проекта
и этапов реализации проекта

5. Дважды щелкните левой кнопкой мыши в строчке *Floorplan IO-Pre-Synthesis* – запустится графический редактор назначения выводов. В данной лабораторной работе все выводы уже заданы. Вам необходимо только проверить их.

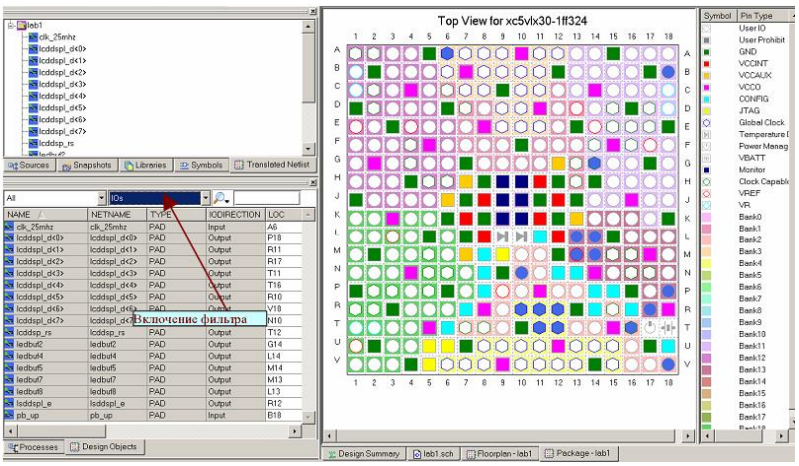


Рис. 3.12. Графический редактор назначения выводов

6. В окне редактора назначений *Design Objects* (рис. 3.12) включите фильтр для отображения только выводов проекта *IOs*.

7. Проверьте, что выводы, заданные в Вашем проекте, соответствуют назначениям, приведенным на рис. 3.12.

Шаг 3. Моделирование

В рамках данной лабораторной работы будет осуществляться поведенческое моделирование с помощью встроенной в пакет ISE системы моделирования ISE simulation. Моделирование будет реализовано для модуля проекта **mux_unit** (рис. 3.10).

1. Выберите модуль **mux_unit** в закладке *Sources* (рис. 3.13).

2. В меню **Project** выберите команду *New Source*.

3. В диалоговой панели *New source Wizard* (Мастер создания нового источника) (рис. 3.14) укажите имя создаваемого файла mux_unit_tb и тип файла *Test Bench WaveForm*. Нажмите кнопку *Next*.

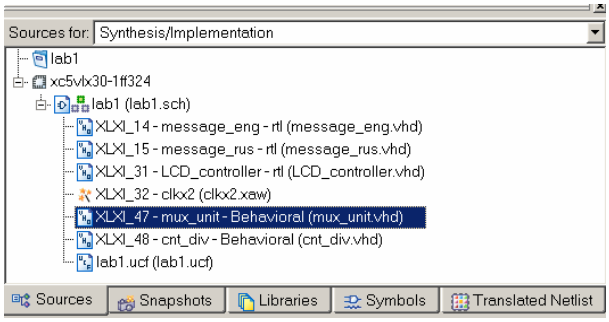


Рис. 3.13. Выбор модуля для поведенческого моделирования

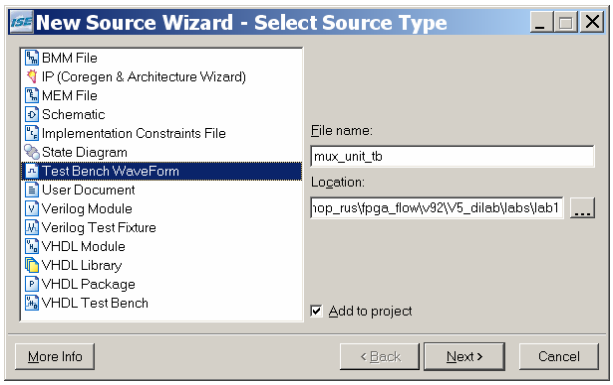


Рис. 3.14. Создание модуля для поведенческого моделирования

4. В окне *Associate Source* (рис. 3.15) укажите модуль **mux_unit** и нажмите кнопку *Next*.

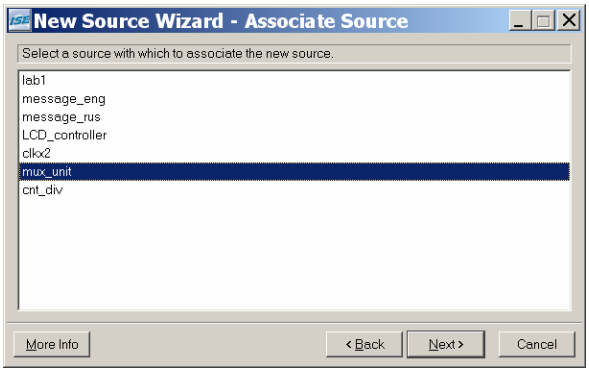


Рис. 3.15. Диалоговая панель ассоциации с модулем моделирования

5. Появится окно *Summary*. Нажмите кнопку *Finish*.

6. Запустится панель инициализации временных параметров *Initial Timing and Clock Wizard*. Установите все параметры так, как показано на рис. 3.16. Нажмите кнопку *Finish*.

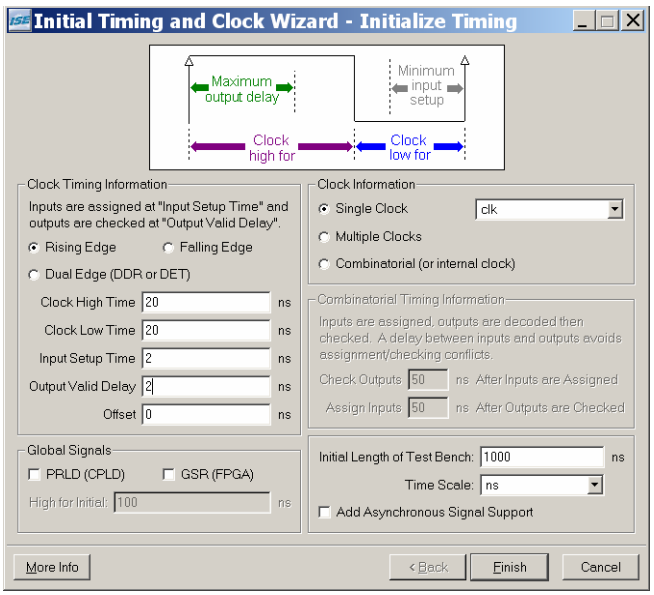


Рис. 3.16. Диалоговая панель инициализации временных параметров

7. Появится окно с редактором временных диаграмм (рис. 3.17).

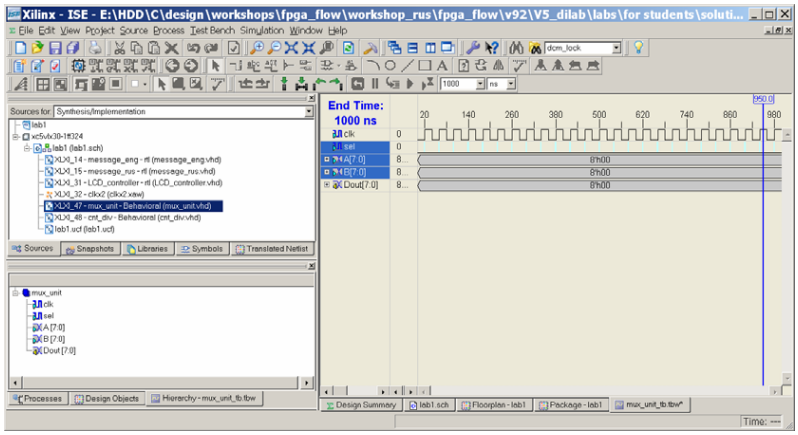


Рис. 3.17. Окно генерации тестовых сигналов

8. На временной диаграмме сигнала *Sel*, который является входным сигналом для модуля **mux_unit** (рис. 3.18), установите курсор мыши в зоне до 20 ns, нажмите правую кнопку мыши и укажите команду *Set Value*.

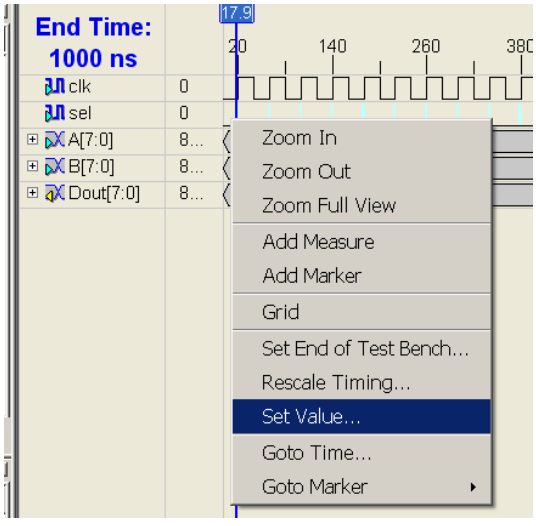


Рис. 3.18. Установка сигнала *Sel*

9. В диалоговой панели *Set Value* (рис. 3.19) нажмите кнопку *Pattern Wizard*.

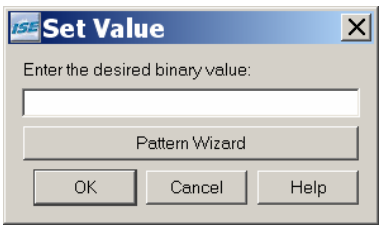


Рис. 3.19. Диалоговая панель установки значения сигнала *Sel*

10. В диалоговой панели *Pattern Wizard* установите значения параметров в соответствии с образцом, приведенном на рис. 3.20. Нажмите кнопку *OK*.

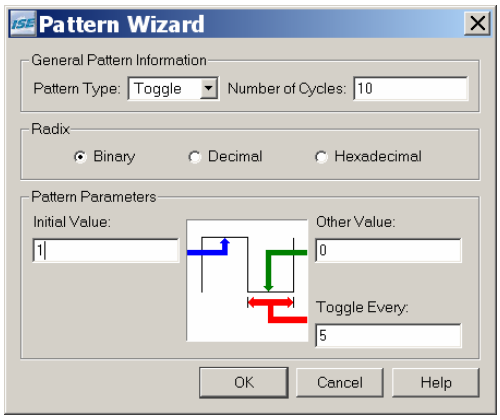


Рис. 3.20. Установка значений сигнала *Sel*

Toggle – два устойчивых состояния.

11. На временной диаграмме сигнала **A**[7:0] установите курсор мыши в зоне до 20 ns, нажмите правую кнопку мыши и укажите команду *Set Value*.

12. В диалоговой панели *Set Value* нажмите кнопку *Pattern Wizard*.

13. В диалоговой панели *Pattern Wizard* установите значения параметров соответствии с образцом, приведенном на рис. 3.21. Нажмите кнопку *OK*.

14. На временной диаграмме сигнала **B**[7:0] установите курсор мыши в зоне до 20 ns, нажмите правую кнопку мыши и укажите команду *Set Value*.

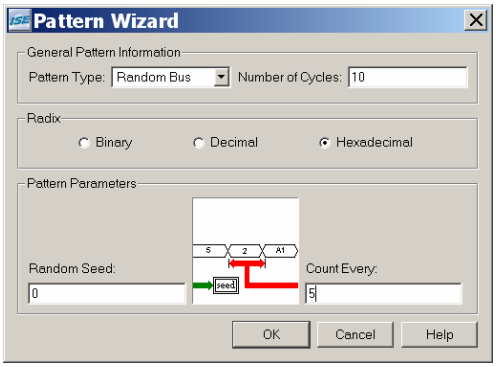


Рис. 3.21. Установка значений сигнала **A**[7:0]

15. В диалоговой панели *Set Value* нажмите кнопку *Pattern Wizard*.

16. В диалоговой панели *Pattern Wizard* установите значения параметров соответствии с образцом, приведенном на рис. 3.22. Нажмите кнопку *OK*.

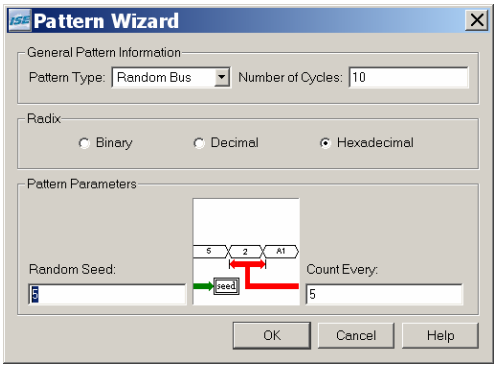


Рис. 3.22. Установка значений сигнала **B**[7:0]

17. Полученная временная диаграмм должна соответствовать временной диаграмме приведенной на рис. 3.23. Сохраните ее (нажатием на пиктограмму дискеты). В окне *Sources for:* менеджера пакета укажите *Behavioral Simulation*. В закладке *Sources* укажите модуль mux_unit_tb. Выберите закладку *Processes*. Выполните двойной щелчок левой клавишей мыши в разделе *Simulate Behavioral Model* (закладка *Processes*, раздел *Xilinx ISE Simulation*) – процедура моделирования запущена.

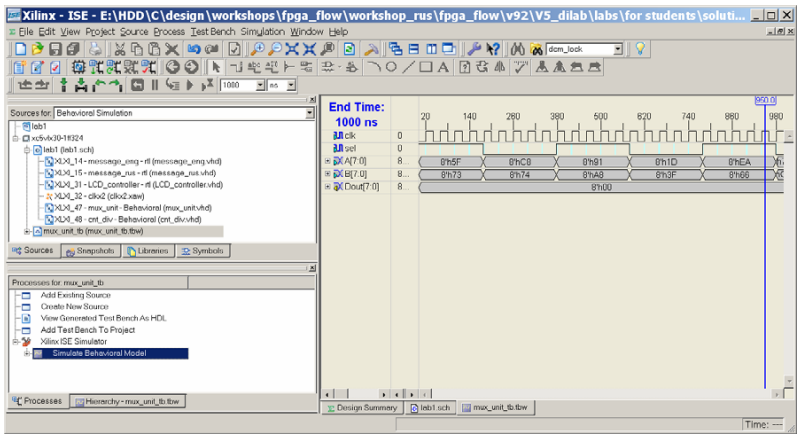


Рис. 3.23. Результаты задания тестовых сигналов

18. После окончания процедуры моделирования на экране будут отображены результаты моделирования

Шаг 4. Реализация проекта

Реализация проекта – полная компиляция проекта с получением файла для конфигурации СБИС.

1. В окне *Sources for:* менеджера проекта (рис. 3.24) укажите *Implementation* (реализация). В закладке *Sources* укажите модуль **lab1** (lab1.sch). Выберите закладку *Processes*. Выполните двойной щелчок левой клавишей мыши в поле *Generate programming file* - будет запущена полная процедура реализации проекта (включая получение битового файла для загрузки проектируемого цифрового устройства).

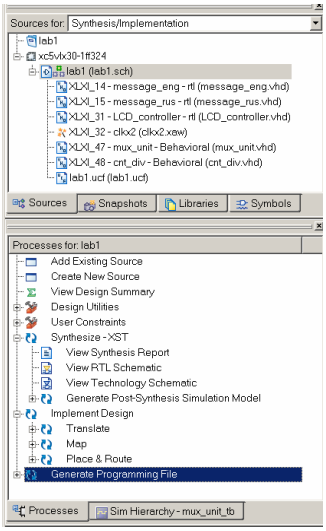


Рис. 3.24. Запуск процедуры реализации проекта

2. После окончания процесса реализации проекта в окне сообщений (закладка *Console*) появится надпись: *Process "Generate Programming File" completed successfully* (рис. 3.25). Будет создан файл lab1.bit (расширение bit справедливо для ПЛИС семейства FPGA).

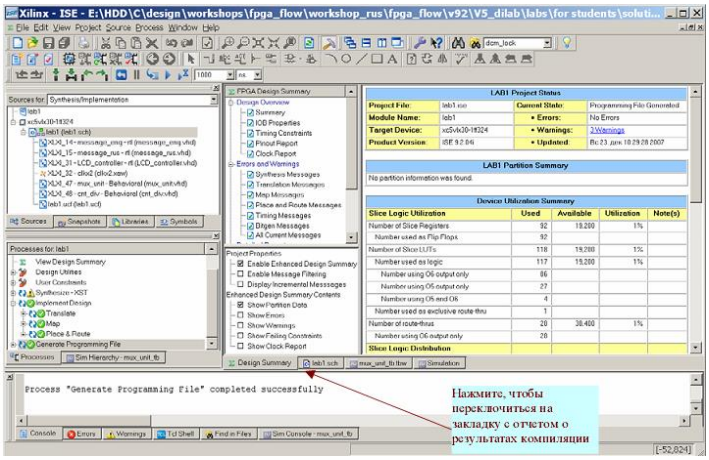


Рис. 3.25. Отчет о результатах компиляции

3. Переключитесь на закладку *Design Summary*, в ней будет отображен отчет о результатах компиляции.

Задание для самостоятельной работы: проанализировать таблицу *Device Utilization Summary* с результатами компиляции проекта, сделать вывод о затраченных ресурсах ПЛИС.

Практическое занятие 2. Создание VHDL-проектов простейших логических элементов

Задание: реализовать VHDL-проект для четырехвходового логического элемента И.
Покажем шаги реализации проекта. VHDL-проект элемента 4И имеет вид:

```
-- Заголовок проекта, (раздел подключения библиотек и пакетов).
library ieee; -- Оператор подключения библиотеки.
use ieee.std_logic_1164.all; -- Оператор подключения пакета.
-- Раздел объявлений проекта.
entity gate_and is -- Оператор объявления интерфейса компонента.
port (      -- Оператор объявления портов интерфейса.
x1: in std_logic;
x2: in std_logic;
x3: in std_logic;
x4: in std_logic;
y  : out std_logic);
end entity;
-- Раздел архитектуры проекта.
architecture gate_arch of gate_and is -- Оператор объявления архитектуры.
-- Раздел объявлений архитектуры проекта.
begin
-- Раздел выполняемых операторов архитектуры проекта.
   y <= x1 and x2 and x3 and x4;
end architecture gate_arch;
```

Шаг 1. Запуск пакета и создание проекта

1. Выполните команду *Start > Programs > Design Suite 10.1 > ISE > Project Navigator*;

2. В *Навигаторе* проекта выполните команду *File > New Project*. Откроется диалоговая панель помощника - *New Project Wizard* (рис. 3.26). Заполняем поля диалоговой панели.

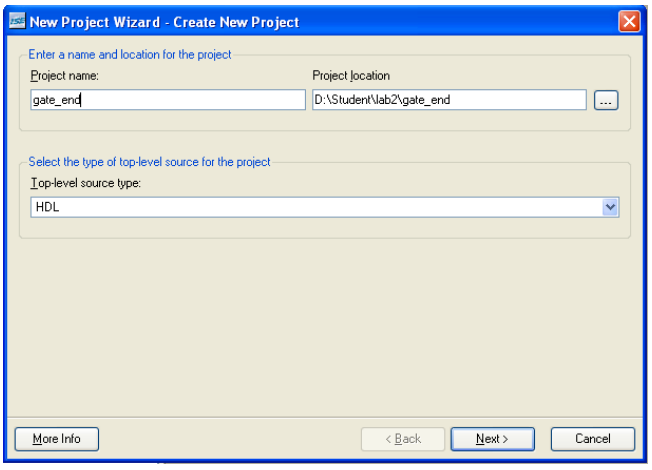


Рис. 3.26. Диалоговая панель мастера создания проекта

3. Нажмите кнопку *Next*. Появится диалоговая панель *Device Properties* (рис. 3.27). Заполните поля диалоговой панели.

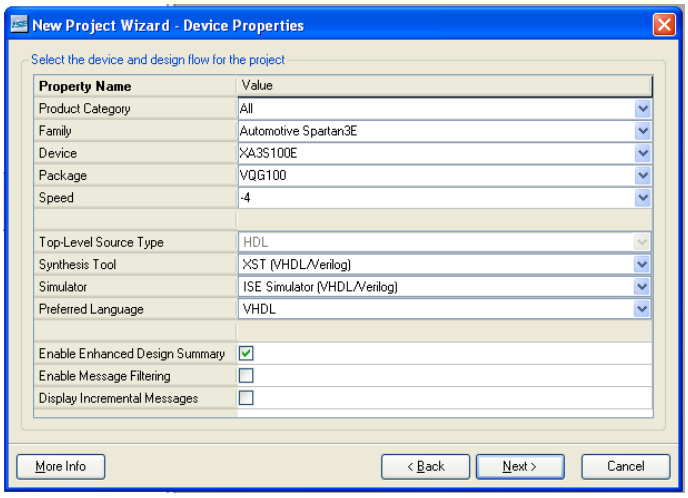


Рис. 3.27. Диалоговая панель выбора ПЛИС

4. Нажмите кнопку *Next*. Появится диалоговая панель *Create New Source*. В данном проекте создание новых источников не предполагается.

5. Нажмите кнопку *Next*. Появится диалоговая панель *Add Existing Sources*. В данном проекте добавление новых источников не предполагается.

6. Нажмите кнопку *Next*. В итоге получаем отчет о текущей спецификации проекта.

7. Нажмите кнопку *Finish*. Открывается окно *Навигатора* проекта, созданный новый проект появляется в закладке *Sources*.

Шаг 2. Созданию файла программы проекта

1. В закладке *Sources for: Навигатора* проекта указываем *Behavioral Simulation*.

2. Щелкая правой кнопкой мыши на странице *Sources*, выбираем во всплывающем меню пункт *New Sources…* (рис. 3.28).

3. В диалоговой панели выбора типа источника *Select Source Type* выбираем пункт *VHDL Module*, задаем имя файла проекта (например, 4gate) (рис. 3.29). Нажимаем кнопку *Next*.

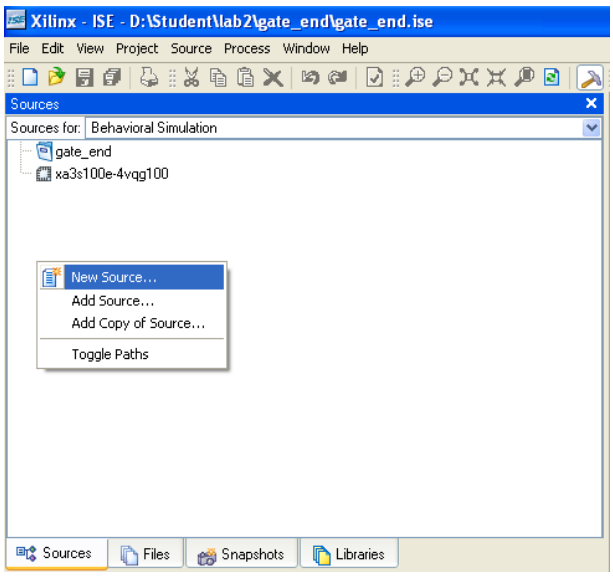


Рис. 3.28. Выбор нового модуля проекта

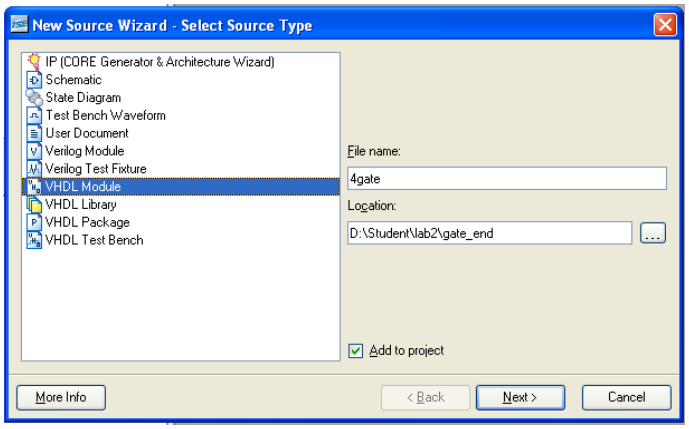


Рис. 3.29. Создание нового модуля VHDL-описания

В появившемся окне (рис. 3.30) можно задать порты ввода/вывода. На данном этапе задавать их не будем. Нажимаем кнопку *Next* и далее *Finish*.

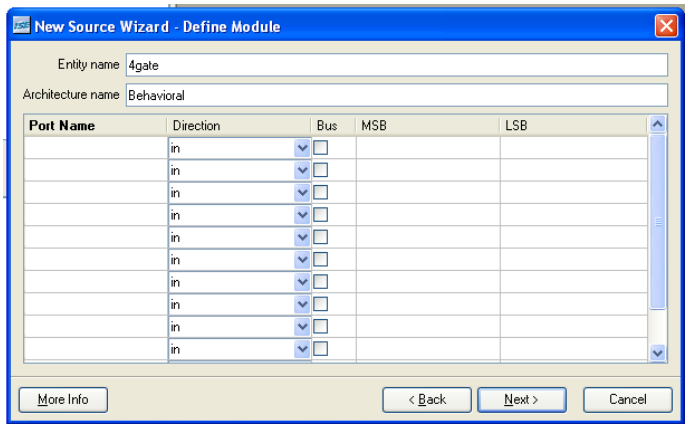


Рис. 3.30. Диалоговая панель задания портов ввода/вывода

На странице *Sources* появится добавленный файл проекта, который представляет собой основную конструкцию VHDL-проекта.

4. Вставьте вместо заготовленной конструкции VHDL-проект элемента 4И.

5. Проверим программу на ошибки. Для этого на странице *Sources* выбираем добавленный файл проекта 4gate.vhd, на странице *Processes* выбираем пункт *Xilinx ISE Simulator*, в открывшемся списке щелкаем два раза левой кнопкой мыши по пункту *Behavioral Check Syntax* (рис. 3.31). После проверки рядом с пунктом *Behavioral Check Syntax* должен появиться зеленый значок, означающий успешную компиляцию проекта. В случае наличия ошибок в программе появляется красный значок. Все предупреждения и ошибки отображаются в закладке *Console*.

Шаг 3. Моделирование

1. Щелкаем правой кнопкой мыши на странице *Sources* и выбираем пункт *New Sources*….

2. В диалоговой панели выбора типа источника *Select Source Type* выбираем пункт *Test Bench Waveform*, задаем имя файла тестовых воздействий (например, takty) (рис. 3.32).

3. После нажатия кнопки *Next* выбираем в открывшемся окне *Associate Source* модуль, для которого будет сформировано тестовое воздействие. Нажимаем кнопку *Next* и далее *Finish*.

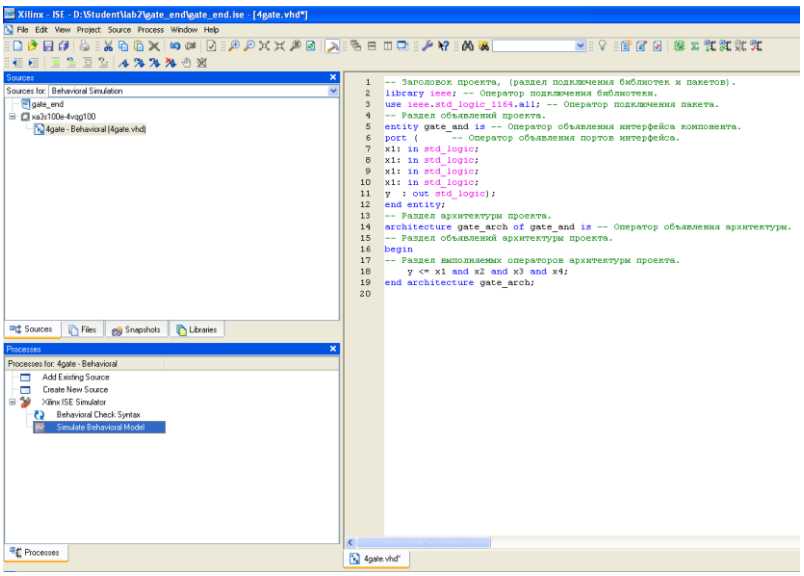


Рис. 3.31. Окно Навигатора проекта со вставленным VHDL-проектом

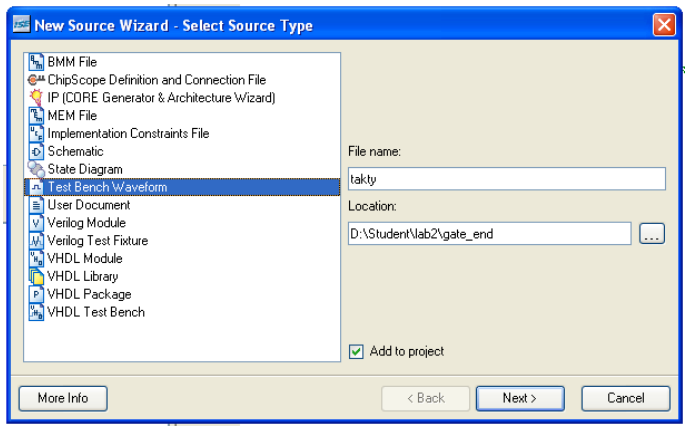


Рис. 3.32. Создание нового модуля тестового воздействия

4. Запускается диалоговая панель инициализации временных параметров *Initial Timing and Clock Wizard*, в которой осуществляется настройка тактового сигнала. В нашем проекте в качестве тактового будет выступать сигнал на порте x1 (рис. 3.33).

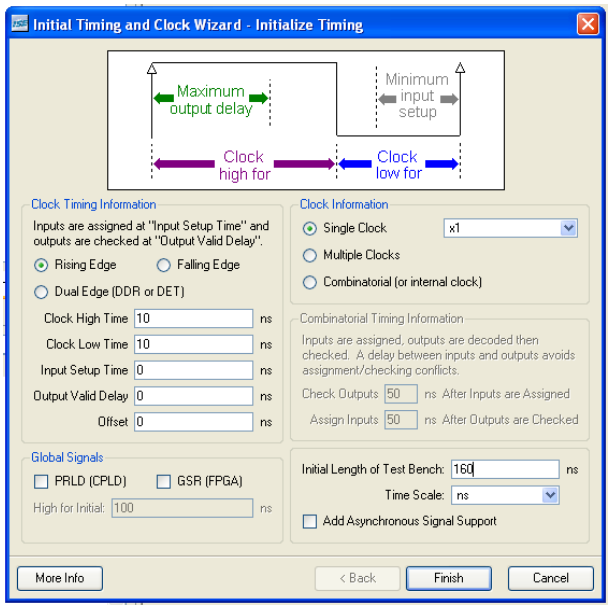


Рис. 3.33. Настройка параметров тестовых сигналов

5. После нажатия кнопки *Finish* запускается редактор тестового воздействия (рис. 3.34), в котором вручную формируем сигналы на портах x2, x3, x4.

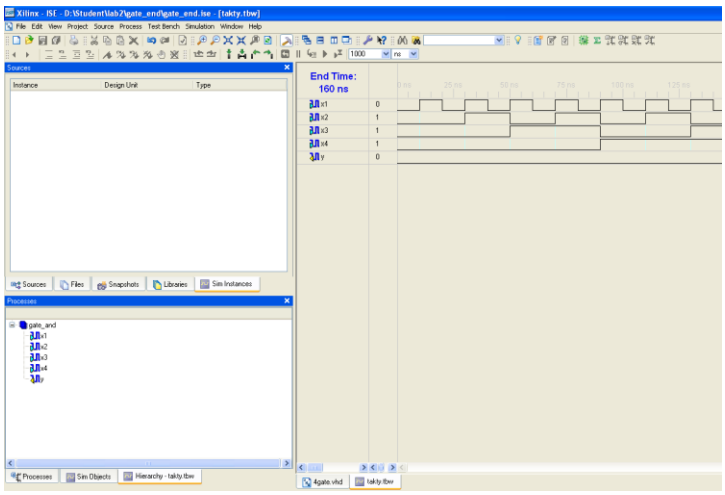


Рис. 3.34. Графическое редактирование тестовых сигналов

6. Сохраняем сформированный файл (takty.tbw). На странице *Sources* появится файл тестовых воздействий. Выбираем его одним щелчком левой кнопки мыши. На странице *Processes* появляется возможность запустить этап функционального моделирования: *Xilinx ISE Simulator – Simulate Behavioral Model*.

7. Запускаем этап функционального моделирования. В результате появляется окно с временной диаграммой работы логического элемента.

Замечание. По умолчанию время моделирования составляет 1000 нс. Если Вы хотите моделировать в течение другого времени, то на инструментальной панели нажимаем на кнопку *Restart Simulation*, выставляем необходимое время и нажимаем на кнопку *Run For Specified Time*.

Результаты моделирования показаны на рис. 3.35.

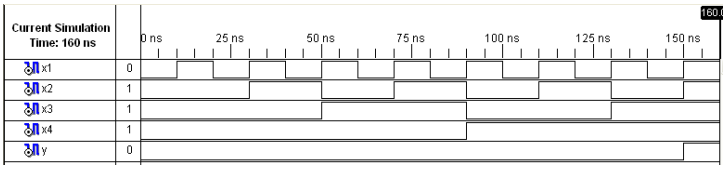


Рис. 3.35. Результаты моделирования

Задание для самостоятельной работы: для логических элементов 4И-НЕ, 3ИсключающееИЛИ привести таблицу истинности, создать VHDL-проект и провести моделирование логических элементов. Привести отчет по работе.

Практическое занятие 3. Создание VHDL-проектов мультиплексора и демультиплексора

Задание: реализовать VHDL-проект для мультиплексора 4×1 (3 бита в каждом канале).

Эта Практическое занятие иллюстрирует использование операторов **if…then** и **case**. VHDL-проект мультиплексора имеет вид:

```
-- ------------------------------------------------------------------------
--                              Проект Mux
-- ------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity mux is
port (
    EN : in std_logic; -- Сигнал разрешения выбора канала.
    IN0 : in std_logic_vector (2 downto 0);
    IN1: in std_logic_vector (2 downto 0);
    IN2: in std_logic_vector (2  downto 0);
    IN3: in std_logic_vector (2 downto 0);
    SEL : in std_logic_vector (1 downto 0); -- Выбор канала.
    OUT : out std_logic_vector (2 downto 0));
end entity;
architecture mux_arch of mux is
constant NON_ACTIVE : std_logic_vector (2 downto 0) := (others => '0');
begin
process (SEL, EN, IN0, IN1, IN2, IN3)
begin
    if (EN = '0') then
        OUT <= NON_ACTIVE;
    else
        case CONV_INTEGER(SEL) is
            when 0 => OUT <= IN0;
            when 1 => OUT <= IN1;
            when 2 => OUT <= IN2;
            when 3 => OUT <= IN3;
            when others => OUT <= NON_ACTIVE;
        end case;
    end if;
end process;
end architecture;
```

В этом проекте подключен пакет std_logic_unsigned библиотеки ieee, который содержит определения функций, позволяющих преобразовать тип std_logic_vector в integer.

Покажите в проекте операцию преобразования одного типа в другой.

Объявление

```
constant NON_ACTIVE : std_logic_vector (2 downto 0) := (others => '0');
```

использует операцию, называемую *агрегат*, которая объединяет одно или несколько значений в значение составного типа. Например, агрегат (*i* => '1', **others** => '0') задает вектор, в котором на *i*-м месте стоит 1, а остальные биты - нулевые.

Задание: реализовать VHDL-проект для 4-канального демультиплексора с сигналом разрешения выбора номера канала.

VHDL-проект демультиплексора имеет вид:

```
-- ------------------------------------------------------------------------
--                              Проект DMux
-- ------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity dmux is
  port (
  EN : in std_logic; -- Сигнал разрешения выбора канала.
  SEL : in std_logic_vector (1 downto 0); -- выбор номера канала.
  IN1 : in std_logic_vector (7 downto 0); -- входной канал.
  OUT0 : out std_logic_vector (7 downto 0); -- выходной канал 0.
  OUT1 : out std_logic_vector (7 downto 0); -- выходной канал 1.
  OUT2 : out std_logic_vector (7 downto 0)); -- выходной канал 2.
  OUT3 : out std_logic_vector (7 downto 0)); -- выходной канал 3.
end entity;
architecture dmux_arch of dmux is
  constant NON_ACTIVE : std_logic_vector (7 downto 0):= (others => '0');
  begin
  OUT0 <= IN1 when (SEL = 0) and (EN = '1') else NON_ACTIVE;
  OUT1 <= IN1 when (SEL = 1) and (EN = '1') else NON_ACTIVE;
  OUT2 <= IN1 when (SEL = 2) and (EN = '1') else NON_ACTIVE;
  OUT3 <= IN1 when (SEL = 3) and (EN = '1') else NON_ACTIVE;
end architecture;
```

Задание для самостоятельной работы: реализовать VHDL-проекты мультиплексора и демультиплексора, провести их функциональное моделирование. Подумайте, как в проекте для демультиплексора можно использовать оператор выбора **case**. В отчете представить листинги проектов с комментариями, поясняющими выполнение каждого оператора, и диаграммы временного моделирования.

Практическое занятие 4. Создание VHDL-проектов полусумматора и сумматора

Задание: реализовать VHDL-проект для одноразрядных полусумматора и полного сумматора.

VHDL-проект полусумматора имеет вид:

```
library ieee;
use ieee.std_logic_1164.all;
entity half is
port ( a,b : in std_logic ;
        sum , carry : out std_logic);
end half ;
architecture halfadder of half is
begin
  sum <= a xor b;
  carry <= a and b;
end halfadder ;
```

VHDL-проект одноразрядного полного сумматора имеет вид:

```
library ieee;
use ieee.std_logic_1164.all;
entity full is
port ( a,b,c : in std_logic ;
        sum , carry : out std_logic);
end full ;
architecture fulladder of full is
begin
  sum <= a xor b xor c ;
  carry <= (a and b) or ( b and c) or (c and a);
end fulladder;
```

Полный одноразрядный сумматор можно представить как объединение двух полусумматоров. Первый полусумматор служит для сложения двух чисел, принадлежащих одному разряду, и обеспечивает формирование промежуточной суммы s_i и переноса p_{i+1}.

Второй полусумматор складывает перенос с предыдущего разряда p_i с промежуточной суммой s_i.

Задание для самостоятельной работы: 1) реализовать VHDL-проекты одноразрядных полусумматора и полного сумматора, провести их функциональное моделирование. 2) Предложите другой вариант VHDL-кода для реализации одноразрядного сумматора, основанный на использовании оператора языка, выражающего условие (например, оператора **if**). 3) Предложите и реализуйте проект 4-разрядного сумматора с последовательным переносом. Для реализации этого проекта можно использовать модульное конструирование с помощью операторов **component** и **port map**. 4) Изучите, как выглядит проект 4-разрядного сумматора на RTL-уровне (см. раздел 2.8).

Практическое занятие 5. Создание проекта счетчика Джонсона

Задание: реализовать проект счетчика Джонсона на уровне VHDL и схемотехническом уровне.

Если коэффициент пересчета счетчика равен $M=2N$,где N-количество триггеров в схеме, то такой счётчик называется счётчиком Джонсона. По мере поступления тактовых импульсов счётчик сначала заполняется единицами, а потом от них освобождается. В схеме счётчика Джонсона используется перекрестная обратная связь. Таблица работы счетчика имеет вид

Такт	Q1	Q2	Q3	Q4
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

VHDL – проект счетчика Джонсона имеет вид:

```
Library ieee;
use IEEE.std_logic_1164.all;
entity jc2_top is
port(LEFT, RIGHT, STOP, CLK : in std_logic;
Q : inout std_logic_vector (3 downto 0) := "0000");
end jc2_top;
architecture jc2_top_arch of jc2_top is
signal DIR: std_logic := '0';
signal RUN: std_logic := '0';
begin
  process (CLK)
  begin
    if (CLK' event and CLK='1') then
    if (RIGHT='0') then
    DIR <= '0';
    elsif (LEFT='0') then
       DIR <= '1';
    end if;
    if (STOP='0') then
      RUN <= '0';
    elsif (LEFT='0' or RIGHT='0') then
      RUN <= '1';
    end if;
    if (RUN= '1') then
      if (DIR= '1') then
         Q(3 downto 1) <= Q(2 downto 0);
         Q(0) <= not Q(3);
    else
      Q(2 downto 0) <= Q(3 downto 1);
      Q(3) <= not Q(0);
    end if;
  end if;
end if;
end process;
end jc2_top_arch;
```

Замечание: оператор **S'event** возвращает значение true типа boolean, если на сигнале произошло событие, и false в противном случае.

При формировании тестового воздействия установите длительность импульсов на входе CLK равной 20 нс, а период повторения – 40 нс.

Рассмотрим этапы создания схемотехнического проекта счетчика Джонсона.

1. Для создания принципиальной схемы разрабатываемого устройства или его функциональных блоков необходимо выполнить процедуру подготовки основы нового модуля исходного описания проекта, выбрав команду *New Source* из раздела **Project** основного меню *Навигатора* проекта. В качестве типа нового модуля в открывшейся диалоговой панели, показанной на рис. 3.36, необходимо выбрать *Schematic*.

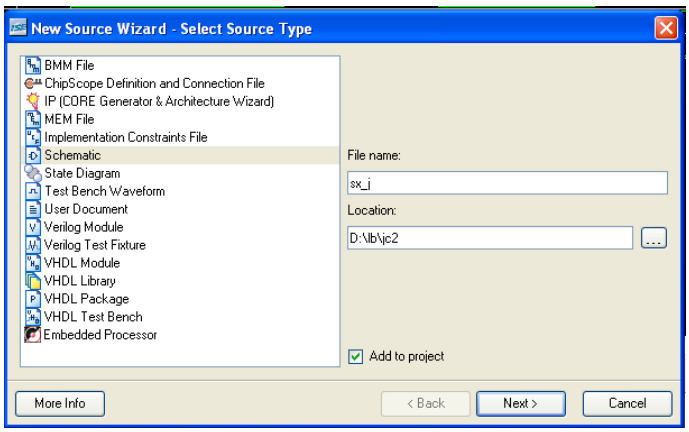


Рис. 3.36. Создание модуля схемотехнического описание проекта

2. При нажатии кнопки *Next*, на экран выводится информационная панель (рис. 3.37), в которой указываются исходные параметры создаваемой схемы. После их подтверждения нажатием кнопки *Finish* открывается окно схемотехнического редактора пакета WebPACK ISE, в строке заголовка которого отображается название новой схемы.

3. Если дважды щелкнуть левой кнопке мышки по названию созданного файла с расширением *.sch в окне *Sources for:* закладки *Sources*, то автоматически открывается *Окно Схемотехнического Редактора* принципиальных схем проекта. Здесь и создается принципиальная схема проекта для ПЛИС. Разобраться с ним не сложно, и на сегодняшний день данный способ создания принципиальной схемы проекта является одним из основных и простых для разработчиков.

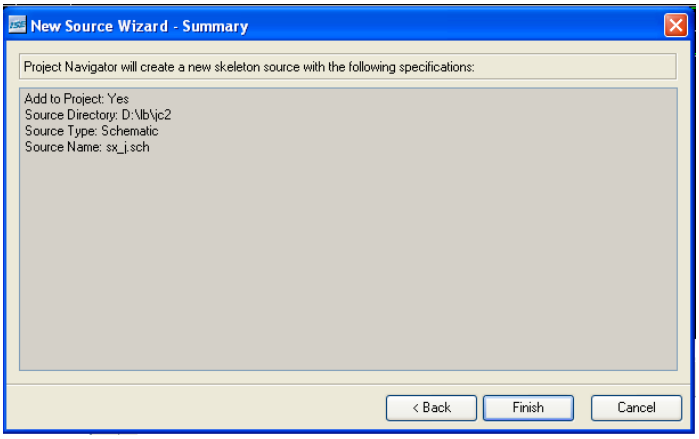


Рис. 3.37. Исходные параметры создаваемой схемы

Для создания новой схемы необходимо выбрать его элементы. Для быстрого их нахождения имеется фильтр (рис. 3.38).

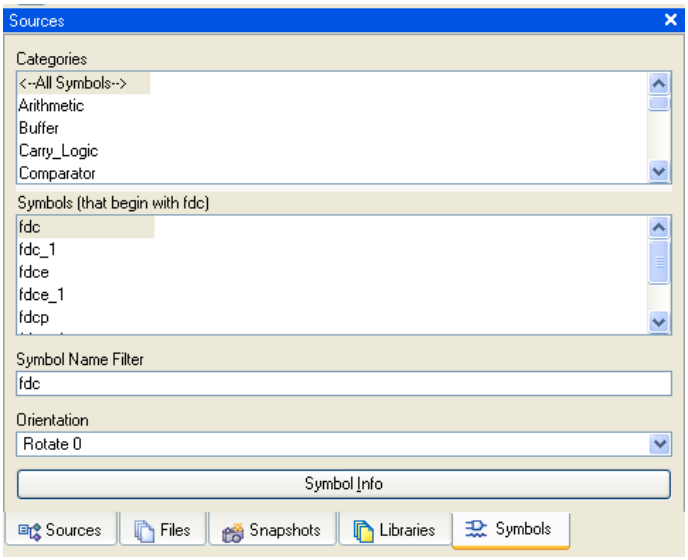


Рис. 3.38. Страница Sources с полями выбора элементов схемы

Нам понадобятся 4*D*-триггера (рис. 3.39), инвертор и маркеры ввода/вывода.

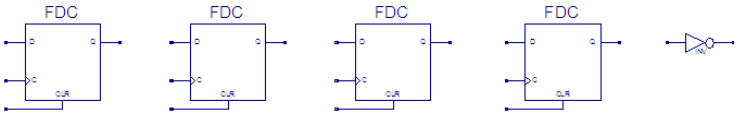


Рис. 3.39. Элементы для построения счетчика Джонсона

При прорисовке принципиальной схемы необходимо пользоваться инструментальной панелью Shematic Editor (рис. 3.40).

Рис. 3.40. Инструментальная панель схемотехнического редактора

4. Собираем схему счетчика Джонсона (рис. 3.41), используя для соединения элементов кнопку *Add Wire* инструментальной панели.

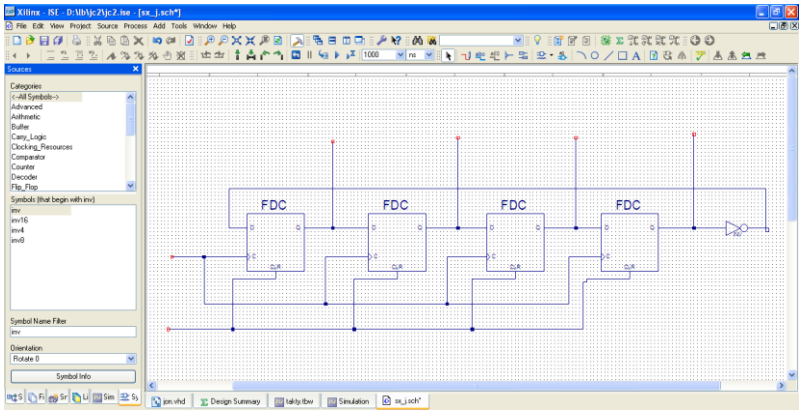


Рис. 3.41. Электрическая схема счетчика Джонсона

5. Выбираем на основной панели инструментов кнопку *Добавление маркеров ввода-вывода* (*Add I/O Marker*). С помощью мышки совмещаем курсор с выводами элементов и нажимаем левую кнопку мышки для постановки входных и выходных маркеров в наш проект (рис. 3.42).

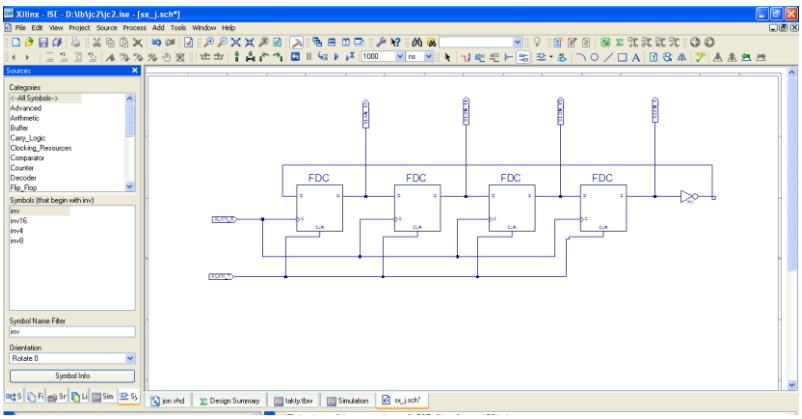


Рис. 3.42. Добавление маркеров в электрическую схему

Система автоматически дает название всем элементам принципиальной схемы, в нашем случае входные маркеры получили названия XLXN_8, XLXN_7. Эти названия можно изменить, нажав два раза левой кнопкой мыши по необходимому элементу (рис. 3.43). Измените названия выходных портов на O1, O2, O3, O4, входному порту тактовых импульсов дайте название CLK, а порту сброса – RST.

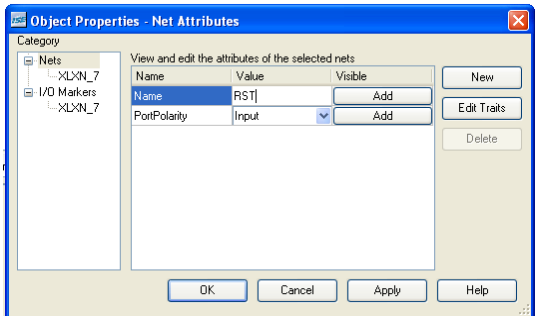


Рис. 3.43. Изменение названия портов

Далее создаем тестовые воздействия по уже известному методу и запускаем схему на моделирование.

Задание для самостоятельной работы: 1) VHDL-проект снабдить комментариями, поясняющими выполнение каждого оператора. 2) Проведите функциональное моделирование двух реализаций проекта.

Практическое занятие 6. Создание проекта суммирующего 4-разрядного счетчика

Задание: реализовать проект суммирующего 4-разрядного счетчика с последовательным переносом на уровне VHDL и схемотехническом уровне.

Счетчик должен подсчитывать количество поступивших на него тактовых импульсов *clk*, когда на его счетном входе *en* присутствует сигнал 1. Счетчик должен сбрасываться в 0 сигналом *rst*=0, не ожидая прихода очередного тактового сигнала *clk*.

VHDL-проект счетчика с последовательным переносом имеет вид:

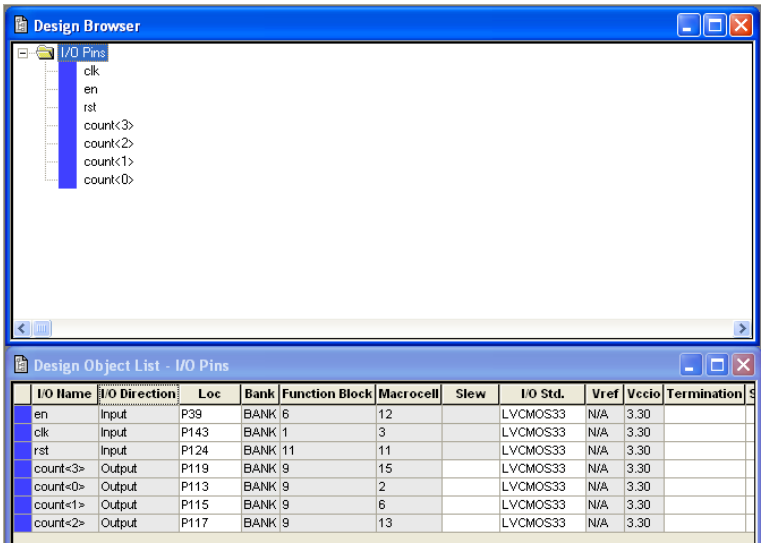
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity counter16 is
port (clk, en, rst : in std_logic;
        count : out std_logic_vector (3 downto 0) := "0000");
end counter16;
architecture behav of counter16 is
signal cnt: std_logic_vector (3 downto 0);
begin
process (clk, en, cnt, rst)
begin
    if (rst = '0') then
        cnt <= (others => '0');
    elsif (clk'event and clk = '1') then
        if (en = '1') then
            cnt <= cnt + 1;
        end if;
    end if;
end process;
count <= cnt;
end behav;
```

Программирование и визуализация проекта будет вестись на отладочной плате NI Digital Electronics FPGA Board в автономном режиме. Поэтому необходимо знать номера ножек ПЛИС, к которым будут подключены переключатели разрешения работы (порт *en*), сброса (порт *rst*) и кнопка тактовых импульсов (порт *clk*). Для решения этой задачи необходимо воспользоваться рис. 1.1 и подразделом «Параметры в UCF-файлах» раздела 1.

Рассмотрим назначение выводов ПЛИС.

1. В закладке *Sources* укажите файл выполняемого проекта. В закладке *Processes* выберите *Floorplan IO-Pre-Synthesis*. Двойным щелчком левой клавиши мыши запустите редактор PACE. В результате будет создан файл *.ucf, в котором будут храниться все заданные вами назначения.

2. Задаем номера выводов ПЛИС в соответствующих строках столбца LOC (закладка *Design Objects List*) (рис. 3.44).



I/O Name	I/O Direction	Loc	Bank	Function Block	Macrocell	Slew	I/O Std.	Vref	Vccio	Termination
en	Input	P39	BANK	6	12		LVCMOS33	N/A	3.30	
clk	Input	P143	BANK	1	3		LVCMOS33	N/A	3.30	
rst	Input	P124	BANK	11	11		LVCMOS33	N/A	3.30	
count<3>	Output	P119	BANK	9	15		LVCMOS33	N/A	3.30	
count<0>	Output	P113	BANK	9	2		LVCMOS33	N/A	3.30	
count<1>	Output	P115	BANK	9	6		LVCMOS33	N/A	3.30	
count<2>	Output	P117	BANK	9	13		LVCMOS33	N/A	3.30	

Рис. 3.44. Ассоциация портов с ножками ПЛИС

3. После назначения выводов ПЛИС необходимо для каждого вывода задать стандарт ввода\вывода – LVCMOS33. Для этого в соответствующей строке столбца I/O Std (рис. 3.44) щелкните левой клавишей мыши и выберите LVCMOS33.

4. Если все правильно, то сохраните назначения, выполненные в редакторе PACE, если нет – необходимо внести исправления.

Следующим этапом реализации проекта является полная компиляция проекта с получением файла для конфигурации ПЛИС.

1. В закладке *Sources* укажите модуль верхнего уровня в иерархии описания проекта. В закладке *Processes* выберите пункт *Generate Programming File* и дважды щелкните левой клавишей мыши – будет запущена полная процедура реализации проекта (включая получение битового файла для загрузки ПЛИС).

2. После окончания процесса реализации проекта в окне процессора сообщений (закладка *Console*) появится сообщение: *Process "Generate Programming File" completed successfully*. Будет создан файл с расширением bit.

Этап программирования ПЛИС проведите в соответствии с рекомендациями подраздела «Программирование ППЗУ» раздела 1 «Сведения о лабораторной установке».

Задание для самостоятельной работы: 1) VHDL-проект снабдить комментариями, поясняющими выполнение каждого оператора. 2) Реализовать проект на схемотехническом уровне. 3) Провести функциональное моделирование для двух реализаций проекта. 4) Удостовериться в правильности аппаратной реализации проекта.

Практическая работа 7. Графический ввод схемы устройства и функциональная симуляция с использованием САПР MAX+plusII

Цель работы: изучение САПР **MAX+plusII,** методов описания проекта в виде принципиальной схемы, трансляции и анализа с исполь-зованием функционального симулятора.

Программный продукт **MAX+plusII** фирмы Altera представляет собой интегрированную систему автоматизированного проектирования (EDA – electronic design automation) цифровых систем, которая предполагает реализацию проекта с использованием программируемых логических интегральных схем (ПЛИС), производимых этой фирмой. Система позволяет описать проект несколькими способами: посредством ввода принципиальной схемы, текста на языке AHDL (Altera Hardware Description Language), в виде временных диаграмм функционирования, как машину состояний.

Система включает 11 программных модулей - приложений (applica-tions), каждый из которых используется для выполнения определенного этапа обработки проекта. В список входят графический редактор, текстовый редактор, символьный редактор, редактор диаграмм, компилятор, монитор структуры проекта, симулятор, редактор конфигурации БИС, временной анализатор, программатор, процессор сообщений. Пользователь имеет мощную поддержку через подсистему **Help,** которая выдает информацию в виде гипертекста. Вид окна Manager системы **MAX+plusII** с открытым списком приложений представлен на рис 1.1.

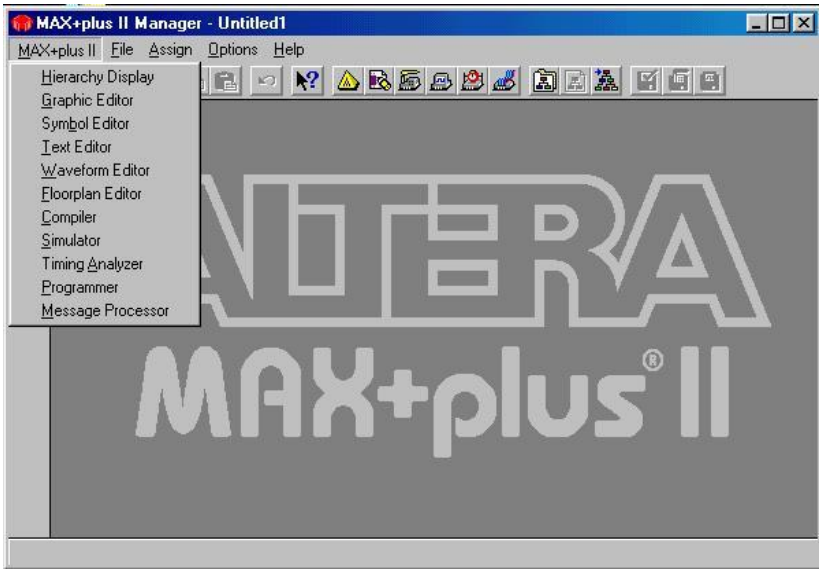


Рис. 1.1. Вид окна Manager системы **Max+plusII** с открытым списком приложений

ОПИСАНИЕ ЗАДАЧИ

Предположим, что проектируемое устройство определено булевым уравнением, а описание проекта в системе **MAX+plusII** предполагается выполнить в графической форме с использованием библиотеки примитивов. Последовательность решения такой задачи следующая:

• исходя из уравнения необходимо определить количество переменных и построить таблицу истинности;

• используя графический редактор, ввести схему устройства. Начинать рекомендуется с входных портов, количество которых определяется количеством переменных в уравнении. Далее анализируется вид членов уравнения и вызываются соответствующие примитивы из библиотеки системы MAX+plusII.. В завершение проводятся межсоединения (цепи и шины), вызывается примитив выходного порта;

• используя редактор временных диаграмм, на основе таблицы истинности формируются тестовые векторы для проверки (верификации) соответствия введенной схемы и первоначального уравнения. Для этого используется функциональная симуляция;

• выполняется трансляция проекта, вызывается симулятор, который на основе тестовых векторов формирует диаграмму выходной функции устройства;

• сравнивая диаграмму состояний функции с таблицей истинности делается заключение о правильности функционирования разработанного устройства.

При выполнении логического проектирования цифровых устройств следует иметь в виду, что для получения наиболее эффективного схемного решения во многих случаях целесообразно произвести определенные преобразования исходного алгебраического уравнения. В первую очередь следует произвести ***минимизацию*** заданной функции, используя один из методов дискретной математики (например, метод Квайна-МакКласки или метод, использующий карты Карно). Если многие импликанты в исходном выражении функции содержат общие логические переменные, целесообразно выполнить ***факторизацию*** - вынесение за скобки общих переменных. В результате получается скобочная форма представления функции, реализация которой обычно требует меньшего числа логических элементов (вентилей). В результате уменьшается число используемых элементов ПЛИС, поэтому можно реализовать на базе ПЛИС большее число требуемых функций. Другим методом получения скобочных форм является ***разложение функций по теореме Шеннона:***

$$f(x_1, x_2, \ldots x_i \ldots x_n) = x_i f_0 (x_1, x_2, \ldots x_i = 0, \ldots x_n) + x_i f_1 (x_1, x_2, \ldots x_i = 1, \ldots x_n),$$

где x_i - выделенная переменная, f_0 , f_1- логические функции, полученные из исходной функции f подстановкой значений $x_i=0$, $x_i=1$, соответственно. Данное соотношение позволяет реализовать функцию n переменных как композицию функций f_0 , f_1 имеющих $(n-1)$ переменную.

Выполнение такого рода преобразований в процессе схемотехнического проектирования цифровых устройств описано в учебном пособии [1] и ряде других монографий. Использование этих преобразований позволяет получить для заданной функции несколько эквивалентных выражений. При их схемотехнической реализации получаются различные варианты схем, выполняющие заданную функцию. Проектировщик имеет возможность провести анализ их характеристик и выбрать вариант схемы, в наибольшей степени соответствующий требованиям технического задания.

В качестве примера в данной работе будем рассматривать цифровое устройство, которое выполняет логическую функцию

$f = x_1 x_2 + \overline{x}_2 x_3$

Таблица истинности имеет следующий вид

Таблица 1.1. Таблица истинности

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Осуществим ввод принципиальной схемы устройства с использованием графического редактора системы **MAX+plusII,** выполним трансляцию проекта с использованием приложения Compiler и проверим выполнение таблицы истинности, используя приложение Simulator.

ПОСЛЕДОВАТЕЛЬНОСТЬ РЕШЕНИЯ ЗАДАЧИ

Определение имени проекта. Разрабатываемое устройство представляется в системе **MAX+plusII** как проект. Вначале работы с системой необходимо определить текущий проект, т.е. указать его имя и директорию. Выберем директорию c:\program files\maxplus2\max2work \tutorial, а в качестве имени проекта укажем graphic1. Из меню Manager выберите File|Project|Name, откроется диалоговое окно, приведенное на рис. 1.2. Имя тома и директория выбираются с помощью соответствующих меню, имя файла вводится в строке File Name. Завершается определение вводом OK. Имя проекта отобразится в титульной строке окна Manager.

Использование графического редактора. Для вызова графическо-го редактора нужно в меню Manager выбрать Max+plusII | Graphic Editor. Откроется окно графического редактора, в титульной строке окна появится сообщение (Untitled1 - Graphic Editor), говорящее о том, что текущим приложением системы **MAX+plusII** является графический редактор и открыт неименованный файл. Строка меню Manager содержит имена команд и набор инстументальных панелей, которых не было на рис. 1.1, т.е. вид окна Manager зависит от текущего приложения. Чтобы узнать назначение каждой панели, нужно навести на нее указатель мыши, информация высветится под окном в строке состояния.

Графическому файлу со схемой узла необходимо присвоить имя с расширением **.gdf** (Graphic Design File). Для этого выберите File | Save As и в строке File Name появившегося диалогового окна укажите имя graphic1.gdf, введите OK.

Для ввода графических изображений элементов будем импортировать их из библиотеки, которая в этой системе называется Primitives. Дважды щелкните мышью в центре экрана графического редактора. Откроется диалоговое окно, в котором в меню Symbol Libraries

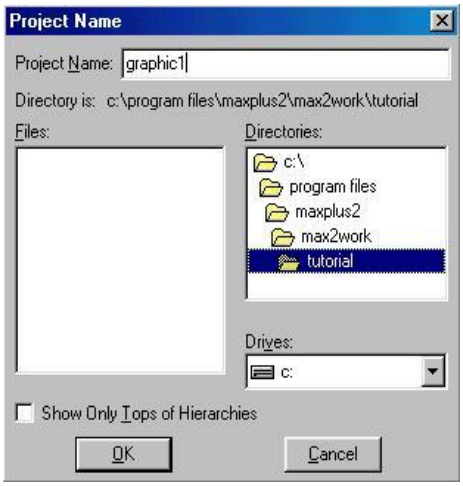


Рис. 1.2. Вид диалогового окна определения имени проекта

указанная библиотека находится по адресу c:\program files\maxplus2\ max2lib\prim. Дважды щелкните мышью по этой строке, в меню Symbol Files появится список логических элементов. Двойной щелчок по имени *and2* приводит к копированию элемента в окно графического редактора в позицию, определенную ранее курсо-

ром. Щелчок мышью по элементу производит его выбор, о чем свидетельствует окрашивание в красный цвет. После этого передвигая курсор мыши при нажатой кнопке 1 можно двигать элемент по окну редактора. Для определения положения элемента полезна сетка, которая появляется при активизации опции Option | Show guidelines.

Для реализации функции *f* нужен один элемент *or2,* два элемента *and2* и элемент *not*.Введите эти элементы (примитивы) указанным выше образом.

После того, как логические элементов введены, нужно ввести символы входных и выходных портов. Их примитивы находятся в той же библиотеке под именами *input* и *output.* Введите три входных порта и один выходной, чтобы получить вид экрана, приведенный на рис 1.3.

Далее необходимо присвоить имена всем портам. Для этого дважды щелкните мышью по слову PIN_NAME на символе входного порта, находящегося в верхнем левом углу экрана. Слово засветится, позволяя прямо набрать имя порта. Ввод Enter непосредственно после имени переводит курсор на следующий порт и так далее. Присвойте имена *x2* и *x3* оставшимся двум входным портам и имя *f* выходному порту.

Следующим шагом является ввод линий, соединяющих логические элементы между собой и с портами.

Нажмите панель "Выбор объекта" - верхнюю в вертикальном меню слева от окна редактора (на ней изображена стрелка). Если после этого подвести курсор к концу линии вывода (pinstub) порта *x1,* его указатель приобретет вид креста, а после нажатия правой клавиши мыши потянет за собой соединительную линию (node), которая кончается при отпускании клавиши. Соединение выхода и входа двух элементов выполняется в виде горизонтальных и вертикальных отрезков прямых. Любой отрезок можно выделить, щелкнув по нему мышью (он станет красным), и стереть (например клавишей Delete). Проведите все соединения, чтобы схема приобрела вид соответствующий рис. 1.4.

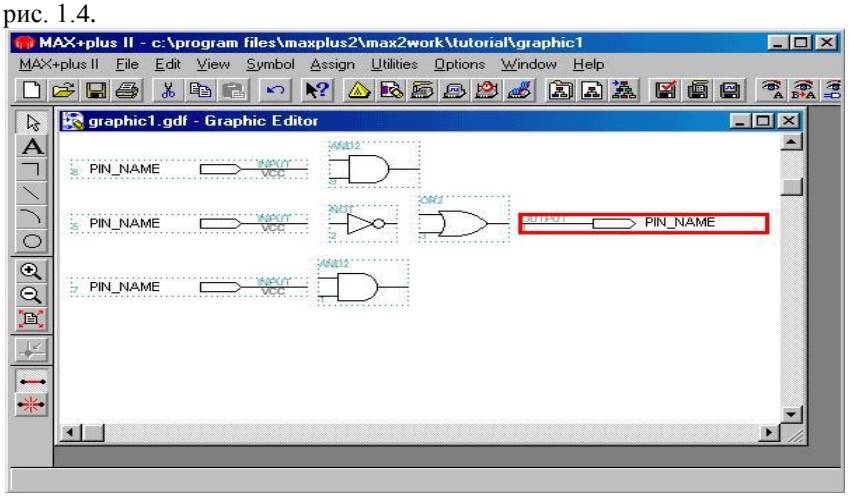


Рис. 1.3. Вид окна графического редактора с изображением примитивов

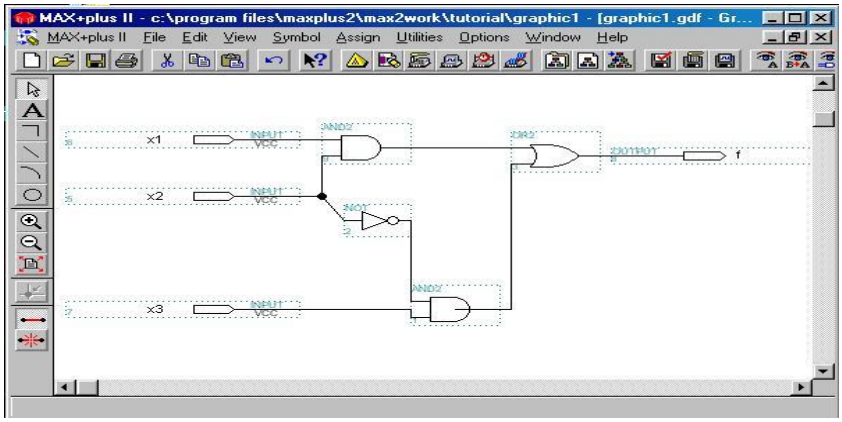


Рис. 1.4. Вид окна графического редактора со схемой устройства graphic1

Работа с компилятором. После ввода схемы система проектирования анализирует ее и генерирует булевы уравнения для всех логических функций. Этот этап обработки выполняет приложение-компилятор, который вызывается выбором Max+plusII | Compiler или щелчком по панели компилятора в меню инструментов. Перед компиляцией нужно выбрать тип микросхемы ПЛИС, на которой будет реализован проект. Наберите Assign | Device и в открывшемся окне укажите тип микросхемы ПЛИС, установленной на плате Starter Kit - EPP8282ALC84-4. Время перекомпиляции проекта сокращается, если установлена опция Smart Recompile (Processing | Smart Recompile). Если выбрать Processing | Design Doctor, специальная утилита проверит все файлы проекта на соответствие правилам реализации на выбранном типе ПЛИС. Для дальнейшей симуляции понадо-

бится SNF-файл, для его генерации нужно указать Processing | Functional SNF Extractor.После установки опций щелчок по клавише Start запускает процесс компиляции, после окончания которого высвечиваются сообщения об ошибках и предупреждения. После успешной трансляции закройте окно компилятора (клавишей X в верхнем правом углу).

Симуляция. Симуляцией обычно называют процесс функционального моделирования с использованием программно-логической модели. Перед проверкой функционирования схемы необходимо создать тестовые векторы, которые представляют значения входных сигналов. Мы будем использовать для их создания редактор диаграмм, который выбирается последовательностью команд Max+plusII | Waveform Editor. Когда окно редактора откроется, создайте файл *graphic1.scf,* последовательностью File | Save As и указанием *graphic1.scf* в строке File Name открывшегося диалогового окна.

Далее определим входные и выходные линии схемы для процесса симуляции. Для этого используем линии, занесенные в SNF-файл (Simulator Netlist File), созданный на этапе компиляции схемы. Введите Node | Enter Node from SNF.Откроется экран, в котором имеется два окна : Available Nodes & Groups и Selected Nodes & Groups.После нажатия List в первом окне появится список входных и выходных линий из SNF-файла. Нужно скопировать входные линии *x1, x2, x3* и выходную линию *f* во второе окно. Для этого нужно отметить линии поодиночке или блоком и нажать панель => между окнами. Чтобы отметить одну линию, нужно щелкнуть по ней мышью. Чтобы отметить блок, нужно протащить указатель по списку при нажатой правой клавише мыши. В завершение введите OK и вернитесь в окно редактора.

Определим параметры процесса симуляции, значения входных пере-менных для нашей схемы. Вначале вводом File | End Time откроем окно определения времени симуляции, введем значение 160ns и OK. Далее определим интервал сетки окна редактора, введя Options | Grid Size и набрав 20ns. После возврата в окно редактора (вводом OK) экран. системы выглядит так, как приведено на рис 1.5. На входных линиях значения логического "0", а выходная линия заштрихована, что указывает на неопределенность значения выходной переменной.

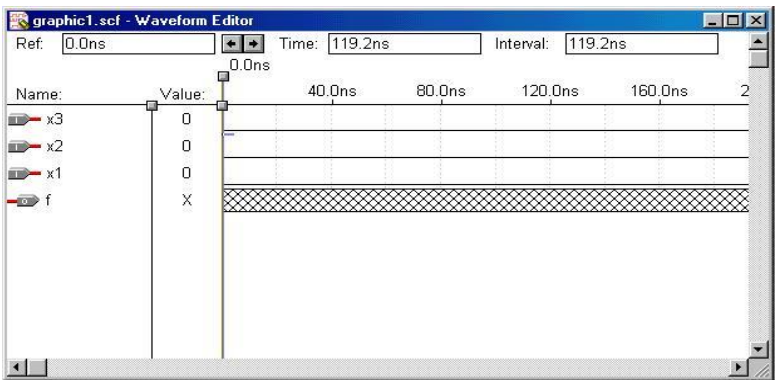


Рис. 1.5. Вид окна редактора диаграмм (определены входные и выходные линии и сетка)

Для полной симуляции функционирования нашего устройства необ-ходимо подать на входы все комбинации значений переменных. Поскольку переменных три, комбинаций $2^3 = 8$. Длительность каждой комбинации при полном времени симуляции в 160пс равна 20пс. Таким образом, переменная *x1* должна иметь значение логической "1" в интер-валы времени 20-40пс, 60-80пс, 100-120пс. Переменная x2 должна иметь значение логической "1" на интервалах 40-80пс, 120-160пс, а переменная *x3* на интервале 80-160пс.

Для редактирования временной диаграммы переменной x1 прота-щите указатель при нажатой правой кнопке мыши над линией логи-ческого "0" во втором интервале сетки. Этот интервал будет "залит" черным цветом, в окнах Ref: и Time: отобразятся значения 20пс и 40пс. Переведите указатель на панель установки значения "1" в левом верти-кальном меню инструментов и щелкните мышью. Временная диаграмма x1 в указанном интервале примет значение "1", заливка исчезнет. Анало-гичными действиями отредактируйте диаграммы переменных x1, x2, x3 так, чтобы они приняли значения, указанные в предыдущем абзаце (рис. 1.6). Сохраните созданный файл (комбинация "горячих" клавиш Ctrl+S).

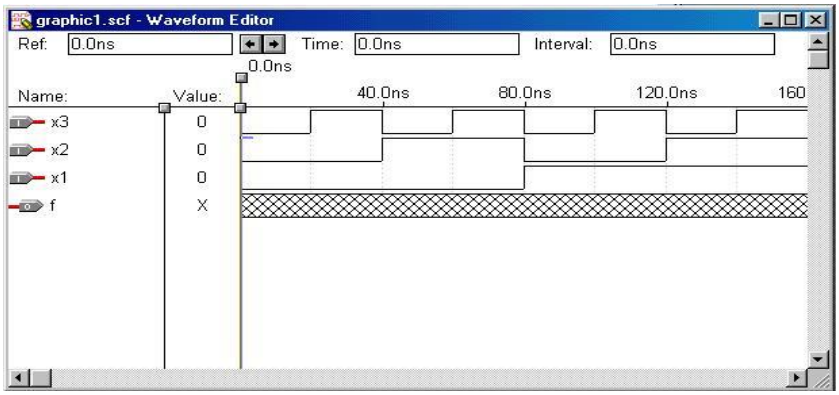


Рис. 1.6. Окно редактора диаграмм с определенными для симуляции диаграммами входных переменных

Для вызова приложения-симулятора выберите Max+plusII | Simulator или нажмите соответствующую панель на верхнем меню инструментов. В открывшемся окне симулятора в заголовке указан режим функциональной симуляции, потому что на этапе компиляции была введена опция Processing | Functional SNF Extractor. В качестве входного файла указан *graphic1.scf.* Укажите в качестве Start Time: значение 0.0пс, а в качестве End Time: значение 160.0пс и щелкните по панели Start. После сообщения об отсутствии ошибок щелкните по OK и вернитесь в окно симулятора. Результаты симуляции записаны в файл *graphic1.scf* и отображаются в окне редактора диаграмм (рис. 1.7.).

Проверьте, что значения функции *f* соответствуют таблице истинности. Закройте окно редактора диаграмм.

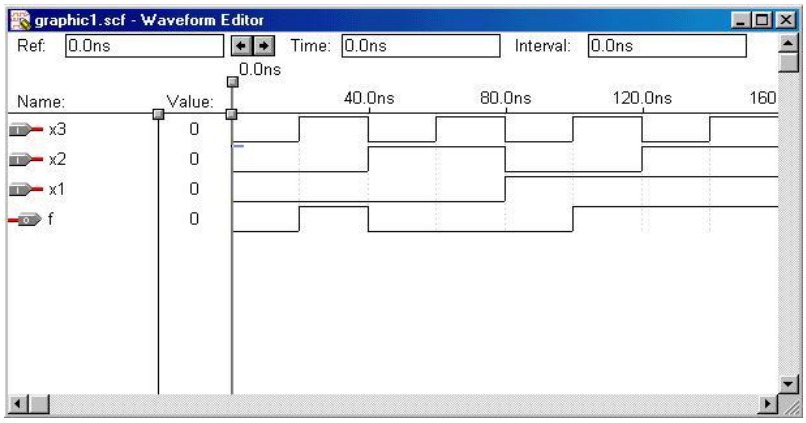


Рис. 1.7. Окно редактора диаграмм с результатами симуляции

ЗАДАНИЯ

1. Используя графический редактор **MAX+plusII** введите схемы следующих функций

$$f1 = x2\overline{x3}\,\overline{x4} + \overline{x1}x2x4 + \overline{x1}x2x3 + x1x2x3$$

$$f2 = x2\overline{x4} + \overline{x1}x2 + x2x3$$

Используя функциональную симуляцию, докажите, что f1=f2

2. Используя графический редактор **MAX+plusII** введите схему следующих функций

$$f1 = (x1 + x2 + \overline{x4})(\overline{x2} + x3 + \overline{x4})(\overline{x1} + x3 + \overline{x4})(\overline{x1} + \overline{x3} + \overline{x4})$$

$$f2 = (x2 + \overline{x4})(x3 + \overline{x4})(\overline{x1} + \overline{x4})$$

Используя функциональную симуляцию, докажите, что f1=f2

3. Используя графический редактор **MAX+plusII** введите схемы следующих функций

$$f1 = x1\overline{x3} + x2\overline{x3} + \overline{x3}\,\overline{x4} + x1x2 + x1\overline{x4}$$

$$f2 = (x1 + \overline{x3})(x1 + x2 + \overline{x4})(x2 + \overline{x3} + \overline{x4})$$

4. Напишите булевы уравнения реализации функции XOR (исключающее ИЛИ)

$$f = x1 \oplus x2$$

в виде суммы произведений входных переменных, на И-НЕ и ИЛИ-НЕ. Используя графический редактор **MAX+plusII** введите схемы разработанных вариантов, используя функциональную симуляцию докажите тождественность реализованных вариантов.

5. Выполните минимизацию логической функции

$$f = x1x2 + \overline{x2}\,\overline{x3} + x1x2x3 + x1\overline{x2}\,\overline{x3}$$

Спроектируйте логические схемы, реализующие исходную и минимизированную функцию. Докажите тождественность их функционирования с помощью функциональной симуляции. Проведите сравнительную оценку параметров полученных схем по числу используемых логических элементов и максимальной задержке переключения.

6. Выполните минимизацию логической функции

$$f = x1x3 + x1x2x3 + \overline{x1}x3 + \overline{x1}\,\overline{x2}\,\overline{x3} + x2x3$$

Спроектируйте логические схемы, реализующие исходную и минимизированную функцию. Докажите тождественность их функционирования с помощью функциональной симуляции. Проведите сравнительную оценку параметров полученных схем по числу используемых логических элементов и максимальной задержке переключения.

7. Выполните минимизацию логической функции

$$f = x1x2 + \overline{x2}\,\overline{x3} + x1x2x3 + x1\overline{x2}\,\overline{x3}.$$

Спроектируйте логические схемы, реализующие исходную и минимизированную функцию. Докажите тождественность их функционирования с помощью функциональной симуляции. Проведите сравнительную оценку параметров полученных схем по числу используемых логических элементов и максимальной задержке переключения.

8. Выполните минимизацию логической функции

$$f = \overline{x1}\,\overline{x2}(x1x3 + \overline{x2}) = (x1 + x2)(x1\overline{x2}\,\overline{x3} + \overline{x1}x2x3).$$

Спроектируйте логические схемы, реализующие исходную и минимизированную функцию. Докажите тождественность их функционирования с помощью функциональной симуляции. Проведите сравнительную оценку параметров полученных схем по числу используемых логических элементов и максимальной задержке переключения.

9. Спроектируйте на базе элементов И-НЕ и ИЛИ-НЕ логические схемы, реализующие данные логические функции:

f = x1 (x2 XOR x3),
f = (x1 x2) XOR (x1 x3).

Докажите тождественность их функционирования с помощью функциональной симуляции. Проведите сравнительную оценку параметров полученных схем по числу используемых логических элементов и максимальной задержке переключения.

10. Проведите факторизацию логической функции

$$f = x1\overline{x2} + x1\overline{x3} + x1\overline{x4}.$$

Спроектируйте на базе элементов НЕ, И-НЕ логические схемы, реализующие исходную и факторизованную функции. Проведите сравнительную оценку параметров полученных схем по числу используемых логических элементов и максимальной задержке переключения.

11. Выполните разложение исходной функции 5 переменных f по теореме Шеннона, минимизируйте полученные функции 4 переменных f0,f1:

$$f = \overline{x1}x2x3\overline{x4} + x2x3x4 + x1\overline{x2}\,\overline{x3}x5 + \overline{x1}\,\overline{x2}\,\overline{x3}x4 + x1\overline{x2}x3 + x1x5 + x1\overline{x2}x3 + x2\overline{x3x4}$$

Спроектируйте логические схемы, реализующие исходную функцию и полученное выражение. Докажите тождественность функционирования разработанных схем с помощью функциональной симуляции.

12. Используя карту Карно упростить данное выражение

$$ABCD + ABC\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}B\overline{C}D + \overline{A}\overline{B}C = F$$

Реализовать полученные выражения на логических элементах. Построить и сравнить полученные временные диаграммы. Сделать выводы.

13. Используя карту Карно упростить данное выражение

$$\overline{A}BCD + ABCD + \overline{B}CD + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}CD = F$$

Реализовать полученные выражения на логических элементах. Построить и сравнить полученные временные диаграммы. Сделать выводы.

14. Используя карту Карно упростить данное выражение

$$\overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}C + ABC\overline{D} + \overline{A}\overline{B}CD + \overline{A}\overline{B} = F$$

Реализовать полученные выражения на логических элементах. Построить и сравнить полученные временные диаграммы. Сделать выводы.

15. Используя карту Карно упростить данное выражение

$$A\overline{B}CD + \overline{A}\overline{B}CD + \overline{A}\overline{B}C + ABC\overline{D} + \overline{A}\overline{B}C\overline{D} = F$$

Реализовать полученные выражения на логических элементах. Построить и сравнить полученные временные диаграммы. Сделать выводы.

Практическое занятие 8. Ввод описания схемы на языке AHDL, использование монитора иерархии проекта САПР MAX+plusII

Цель работы: изучение описания цифровой схемы на языке AHDL, ввода с использованием тестового редактора **MAX+plusII,** использования монитора структуры проекта.

ВВЕДЕНИЕ

Система **MAX+plusII** имеет возможность ввода текстового описания цифровой схемы на языке AHDL(Altera Hardware Description Language), созданного с помощью встроенного или любого другого текстового редактора. Текстовые файлы в системе **MAX+plusII** имеют расширения **.tdf** (Text Design File). Встроенный текстовый редактор способен оказывать мощную поддержку пользователю, предоставляя шаблоны конструкций языка AHDL.

ОПИСАНИЕ ЗАДАЧИ

В данной работе опишем в виде текста на языке AHDL мультиплексор, схема которого приведена рис. 2.1.

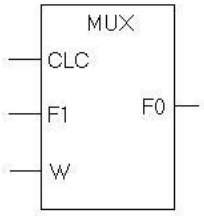


Рис. 2.1. Схема мультиплексора

Сохраним описание в файле mux1.tdf. Далее с помощью графического редактора создадим комбинированный проект верхнего уровня, подключив к одному из входов мультиплексора комбинационное устройство, описанное в виде принципиальной схемы в предыдущей работе (графический файл graphic1.gdf). Проект верхнего уровня рассмотрим с помощью монитора структуры проекта, а затем проверим работу мультиплексора методом моделирования с использованием приложения Simulator.

ПОСЛЕДОВАТЕЛЬНОСТЬ РЕШЕНИЯ ЗАДАЧИ

Создание текстового файла. Для вызова приложения - текстового редактора нужно выбрать File | New и в диалоговом окне указать Text Editor file. После ввода OK откроется окно текстового редактора с неименованным файлом. Нужно присвоить ему имя, сохранив его (File | Save As) в нужной директории, например, с именем mux1.tdf. Связать проект с текущим файлом можно, выбрав File | Project | Set Project to Current File. Указанное имя mux1появится в заголовке окна Manager. После этих шагов можно создавать собственно текст описания устройства на языке AHDL. Для этого первоначально целесообразно использовать шаблоны конструкций языка. Шаблоны доступны после выбора Templates | AHDL Template. В диалоговом окне перечислены доступные разделы, первым идет Overall Structure, помогающий формировать структуру программы. Выберите этот раздел, и после появления его содержимого в окне текстового редактора, ознакомьтесь со структурой программы. Видно, что многие разделы не обязательны. Полезен раздел Title, задающий текст заголовка для файла Report File(**.rpt**). Его шаблон можно вызвать, дважды щелкнув по разделу Templates |AHDL Template | Statement Title. После двойного щелчка между кавычками это поле "зальется", позволяя прямо вводить текст. Введите слово *Multiplexer.* Получить информацию о назначении и синтаксисе этого поля можно, выбрав Help | AHDL |Design Structure | Statement Title. После ввода заголовка соответствующую строку Title от Overall Structure нужно стереть.

Обязательным разделом является Subdesign, описывающий входные, выходные и двунаправленные порты. Получить информацию об этом разделе можно, выбрав Help | AHDL | Design Structure | Subdesign. Вызовете шаблон, дважды щелкнув по разделу Templates | AHDL Template | Subdesign, после чего укажите в качестве входных портов *clc,fi,w*, а в качестве выходного *fo*, остальное сотрите.

Булевы уравнения связывают значения сигналов на входных и выходных портах. Они вводятся в разделе Logic, который начинается ключевым словом BEGIN и завершается словом END. В этот раздел вложим раздел If Then, описывающий мультиплексор на два направления с управляющим входом *w.* Вызовите шаблон этого раздела, в качестве логического выражения после ключевого слова IF укажите просто имя входного переключающего сигнала *w.*После ключевого слова THEN введите выражение *fo=fi*, выполняющееся при истинном значении выражения ("1" на входе *w).* После ключевого слова ELSE введите выражение *fo=clc,* выполняющееся при ложном значении выражения ("0" на входе *w).* Строку ELSEIF уберите. В результате окно редактора должно содержать текст, приведенный на рис.2.2.

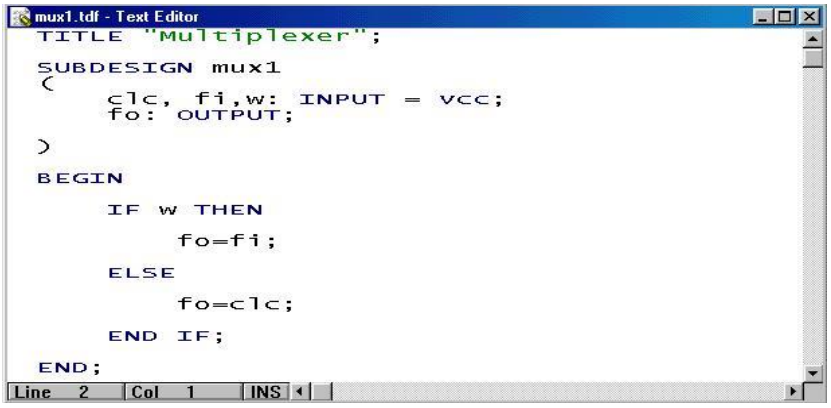


Рис. 2.2. Окно редактора текста с программой на языке AHDL

Сохраните файл (Ctrl+S) и проверьте его на синтаксические ошибки, выбрав File | Project | Save&Check (Ctrl+K). Создайте из него символьный файл *mux1.sym*, выбрав File | Create Default Symbol.

Создание графического файла верхнего уровня. Сейчас с помощью графического редактора мы создадим головной файл проекта *f_mux1.gdf*, включающий как составные части ранее созданные файлы *graphic1 и mux1*. Схема, которая будет получена в результате, приведена на рис.2.3.

Создание иерархического проекта, состоящего из различных модулей, имеет следующие преимущества:

- улучшается восприятие проекта и связей внутри него;
- после определения интерфейса между модулями реализацию отдельных модулей можно поручить разным специалистам;
- созданные и отлаженные модули можно использовать в этом проекте и последующих, как библиотечные.

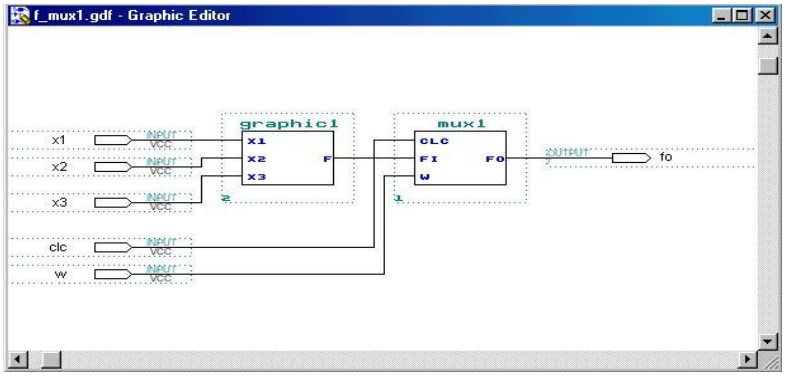


Рис. 2.3. Графический файл верхнего уровня

Последовательность действий следующая:

1. создадим новый графический файл и сохраним его с именем *f_mux1.gdf*;
2. укажем в качестве имени текущего проекта *f_mux1* (как указано ранее или комбинацией клавиш *Ctrl+Shift+J*);
3. скопируем в окно графического редактора созданный на предыдущем этапе работы символьный файл *mux1.sym.* Для этого щелкните мышью в нужном месте окна и введите Symbol | Enter Symbol (или дважды щелкните мышью), в открывшемся диалоговом окне укажите имя *mux1*.После ввода OK условное символьное изображение описанного ранее на языке AHDL мультиплексора появится в выбранном месте экрана;
4. скопируем в окно графического редактора символьный файл созданного в предыдущей работе графического файла *graphic1*.Вначале символьный файл нужно создать, открыв файл *graphic1.gdf* и выбрав File | Create Default Symbol. Далее закройте этот файл, вернитесь в головной файл проекта *f_mux1.gdf* и щелкнув дважды мышью в нужном месте экрана (перед предыдущим изображением) введите в диалоговом окне имя *graphic1*.После ввода OK условное символьное изображение описанной ранее в графической форме схемы появится в выбранном месте экрана;
5. введем, используя библиотеку примитивов, четыре входных порта и один выходной;
6. определим имена всех портов в соответствии с рис. 2.3;
7. соедините линии выводов входных портов и символьных изображений обоих устройств, а также выход устройства *mux1* с выводом выходного порта.

Сохраните получившийся графический файл, который должен содержать изображение, приведенное на рис. 2.3.

Работа с монитором структуры проекта. Структуру (иерархию) проекта можно увидеть, используя специальное приложение. Введите команду Max+plusII | Hierarhy Display или щелкните по соответствующей панели меню инструментов. Откроется окно, в котором проект представлен в виде, соответствующем рис. 2.4. Структура головного проекта изображена в виде дерева, указаны имя каждого файла и исходный тип, а также иконка. Двойной щелчок по иконке открывает исходный файл нижнего уровня с помощью соответствующего редактора. Слева от каждой ветви указаны файлы с такими же именами, но другими расширениями, которые созданы в процессе обработки исходного файла.

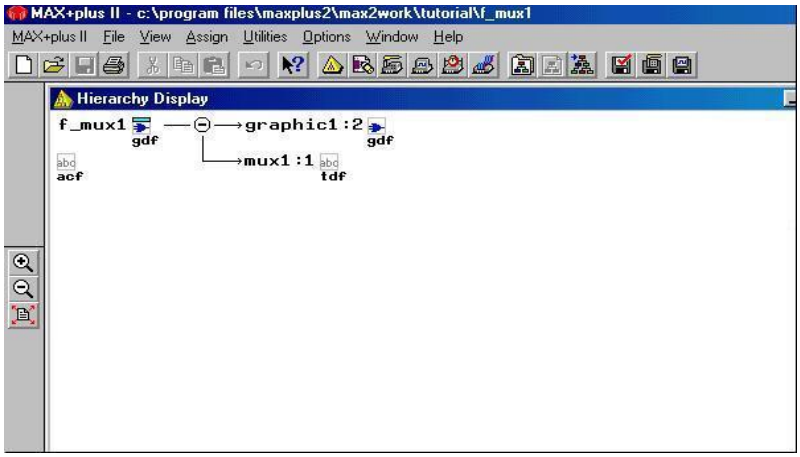


Рис. 2.4. Окно монитора структуры проекта

Симуляция. Проведем симуляцию нашего иерархического проекта. Последовательность действий следующая:

1.Создадим с помощью редактора временных диаграмм файл, где будут находиться тестовые векторы и результаты симуляции. Для этого откроем редактор временных диаграмм (Max+plusII | Waveform Editor), сохраним файл с именем *f_mux1.scf* (File | Save As);

2.Определим входные и выходные линии для симуляции. Для этого откроем графический файл *f_mux1.gdf* (File | Open), вызовем компилятор (Max+plusII | Compiler) и установив опцию функциональной симуляции (Processing | Functional SNF Extractor) запустим процесс трансляции. Далее закроем файл *f_mux1.gdf* и вернемся т.о. в редактор временных диаграмм. Откроем список доступных в SNF - файле цепей (Node | Enter Node from SNF) и скопируем имена входов и выходов в список выбранных цепей. После ввода OK в окне редактора будут видны входные линии *x1, x2, x3, clc,w* и выходная линия *fo*;

3.Определим время симуляции и интервал временной сетки. Время симуляции целесообразно установить равным 320нс (File | End Time), а интервал сетки равным 10нс (Options | Grid Size);

4.Определим значения входных векторов таким образом, чтобы временные диаграммы соответствовали приведенным на рис. 2.5. Сохраним созданный файл (комбинация "горячих" клавиш Ctrl+S);

5.Вызовем симулятор (Max+plusII | Simulator), укажем в качестве End Time значение 320.0 nc и щелкнем по панели Start. После сообщения об отсутствии ошибок указав OK вернемся в окно симулятора. Результаты симуляции записаны в файл *f_mux1.scf* и отображаются в окне редактора диаграмм (рис. 2.6.).

Проверьте, что значения функции *f* соответствуют таблице истинности. Закройте окно редактора диаграмм.

Рис. 2.5. Диаграммы входных сигналов для симуляции

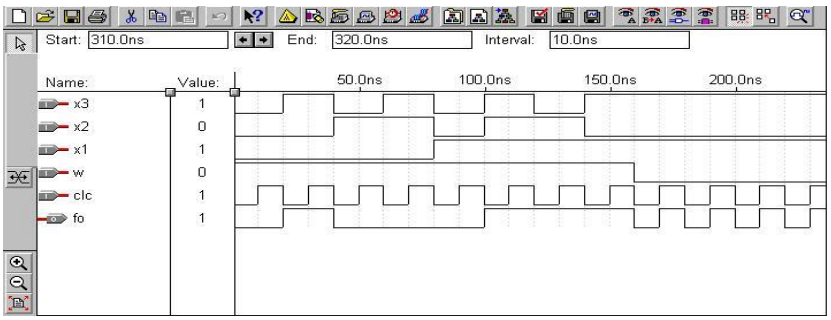


Рис. 2.6. Результаты симуляции проекта *f_mux1*

ЗАДАНИЯ

1. Разработать схему сравнения двух 4-разрядных операндов A=a3a2a1a0 и B=b3b2b1b0, формирующую флаг Z=1 при равенстве операндов A=B, флаг Y=1 при A>=B. Проверить работу схемы, используя симуляцию.

2. Разработать преобразователь чисел, представленных в коде "с из-бытком 3", в двоично-десятичный код прямого замещения (одна декада). Проверить работу схемы, используя симуляцию.

Схема контроля четности 8-разрядных чисел.

3. Разработать схему контроля четности 8-разрядных чисел. Про-верить работу схемы, используя симуляцию.

4. Разработать схему, выдающую сигнал Z=1 при значении пос-тупающего 8-разрядного операнда A=0 и формирующая номер старшего (первого слева) единичного бита при A>0. Проверить работу схемы, используя симуляцию.

5. Разработать схему сдвига 4-разрядного числа влево или вправо на 0, 1 или 2 разряда (свободные разряды заполняются 0). Проверить работу схемы, используя симуляцию.

6. Разработать схему формирования модуля (абсолютного значения) 4-разрядного двоичного числа A=a3a2a1a0 со знаковым разрядом N. Отрицательные числа, представленные в дополнительном коде, перево-дятся в прямой код. Проверить работу схемы, используя симуляцию.

7. Разработать схему преобразователя, изменяющего знак 8-разря-дного операнда A (старший бит операнда является знаковым: a7=N). Отрицательные числа представляются в дополнительном коде. Проверить работу схемы, используя симуляцию.

8. Разработать схему, определяющую число единичных битов в 8-разрядном операнде. Проверить работу схемы, используя симуляцию.

9. Разработать преобразователь чисел, представленных в двоично-десятичном коде, в код “с избытком 3” (одна декада). Проверить работу схемы, используя симуляцию.

10. Разработать схему преобразователя кода Грея в двоично-деся-тичный код (одна декада). Проверить работу схемы, используя симуляцию.

11. Разработать схему, выдающую сигнал Z=1 при значении пос-тупающего 8-разрядного операнда A=7 и формирующая номер младшего (первого справа) единичного бита при поступлении операнда A≠7. Проверить работу схемы, используя симуляцию.

12. Разработать схему сдвига 4-разрядного числа влево или вправо на 0, 1 или 2 разряда (свободные разря-ды заполняются 1). Проверить работу схемы, используя симуляцию.

13. Разработать схему преобразователя, вычитающего из операнда A (старший бит операнда является зна-ковым: a7=N) число 5. Отрицательные числа представляются в дополнительном коде. Проверить работу схемы, используя симуляцию.

14. Разработать схему преобразователя, складывающего с операндом A (старший бит операнда является знаковым: a7=N) число 3. Проверить работу схемы, используя симуляцию.

15. Разработать преобразователь чисел, представленных в двоичном коде, в код “1-2-2-5” (одна декада). Проверить работу схемы, используя симуляцию.

Практическое занятие 9. Проектирование комбинационных схем, программирование ПЛИС и анализ размещения схемы на кристалле

Цель работы: освоение проектирования комбинационных схем с привлечением платы LabKit8000, анализ размещения схемы устройства на кристалле с использованием редактора конфигурации БИС системы **MAX+plusII.**

ВВЕДЕНИЕ

Комбинационными называются цифровые устройства, логическое состояние которых зависит только от комбинации логических сигналов, поступающих на их входы в настоящий момент времени. К этому классу устройств относятся такие широко распространенные функциональные узлы цифровых систем, как преобразователи кодов, шифраторы и дешифраторы, мультиплексоры и демультиплексоры, компараторы, сумматоры, арифметико-логические устройства (АЛУ), перемножители и ряд других.

Функциональное описание комбинационного устройства задается в виде таблицы истинности, алгебраического выражения или описания на специализированном языке высокого уровня (VHDL, AHDL, Verilog HDL и другие).

В настоящее время для реализации комбинационных устройств используются два возможных способа.

1. Сборка устройства из набора отдельных логических элементов (И-НЕ, ИЛИ-НЕ и других). Этот способ применяется при разработке относительно несложных устройств, реализуемых на серийно выпускаемых микросхемах малой степени интеграции, или при проектировании комбинационных блоков в составе сложнофункциональных устройств, реализуемых в виде заказных или полузаказных БИС, которые разрабатываются с использованием библиотек логических элементов.

2. Создание устройства на базе программируемых логических интегральных схем (ПЛИС), в которых заданная логическая функция реализуется путем соответствующего программирования и соединения универсальных логических элементов и блоков, составляющих внутреннюю структуру ПЛИС.

Традиционные методы проектирования комбинационных устройств, ориентированные на первый способ их реализации, описаны в учебном пособии [1] и ряде других монографий. В данной лабораторной работе описывается методика проектирования цифровых устройств на базе ПЛИС, выпускаемых компанией Altera, с помощью системы **MAX+plusII.**

Система **MAX+plusII** предоставляет возможность анализа разме-щения схемы устройства на кристалле с использованием редактора кон-фигурации БИС. Редактор конфигурации формирует две разновидности изображения ПЛИС – Device View и LAB View. Первое изображение представляет корпус микросхемы с указанием всех выводов, их номеров и функций. Второе представляет внутреннюю структуру ПЛИС в виде совокупности блоков ячеек (LABs – logic array blocks), отдельные ячейки внутри блоков, ячейки ввода/вывода. Этот тип изображения также включает информацию о выводах, поэтому можно проследить связи между ними и внутренними ячейками микросхемы.

ОПИСАНИЕ ЗАДАЧИ

В данной работе рассмотрим дешифратор для семисегментного ин-дикатора, который формирует в зависимости от комбинации сигналов на 4 входных линиях код выборки сегментов на семи выходных линиях. Активным значением сигнала на выходной линии (при котором зажигается сегмент) является уровень "1". Обозначение дешифратора с указанием входных и выходных линий, распределение имен сегментов и индицируемые знаки приведены на рис. 3.1.

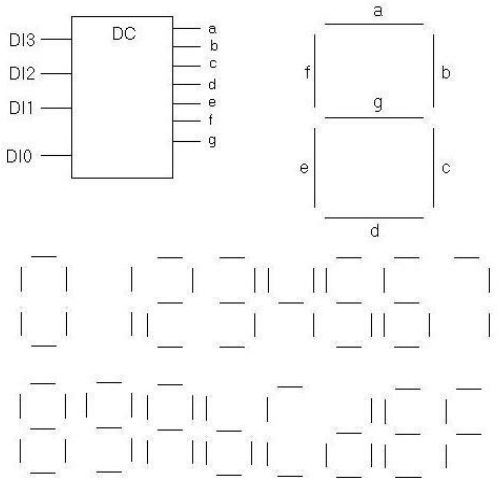


Рис. 3.1. Дешифратор 7-сегментного индикатора, распределение имен сегментов, индицируемые знаки

Описать такой дешифратор на языке AHDL можно в виде таблицы следующим образом:

```
TITLE                                                    “LabKit 8000.Decoder7”;

SUBDESIGN decode7
(
DI[3..0]            :INPUT;
a,b,c,d,e,f,g       :OUTPUT;
)

BEGIN

                                TABLE
                                                                    DI[ ]
                                => (a,b,c,d,e,f,g);
                                                                B”0000”  =>;
B”1111110”; -- H”0”
                                                                B”0001”  =>;
B”0110000”; -- H”1”
                                                                B”0010”  =>;
B”1101101”; -- H”2”
                                                                B”0011”  =>;
B”1111001”; -- H”3”
                                                                B”0100”  =>;
B”0110011”; -- H”4”
                                                                B”0101”  =>;
B”1011011”; -- H”5”
                                                                B”0110”  =>;
B”1011111”; -- H”6”
                                                                B”0111”  =>;
B”1110000”; -- H”7”
                                                                B”1000”  =>;
B”1111111”; -- H”8”
                                                                B”1001”  =>;
B”1111011”; -- H”9”
                                                                B”1010”  =>;
B”1110111”; -- H”A”
                                                                B”1011”  =>;
B”0011111”; -- H”B”
                                                                B”1100”  =>;
B”1001110”; -- H”C”
                                                                B”1101”  =>;
B”0111101”; -- H”D”
                                                                B”1110”  =>;
B”1001111”; -- H”E”
                                                                B”1111”  =>;
B”1000111”; -- H”F”

                                END TABLE;
END;
```

Справа в таблице указаны двоичные значения кода на входных линиях, а слева - двоичные значения выходного кода. Два следующих подряд знака "минус" означают начало комментария.

ПОСЛЕДОВАТЕЛЬНОСТЬ РЕШЕНИЯ ЗАДАЧИ

Создание и трансляция текстового файла. Вызовем текстовый редактор (Max+plusII | |Text Editor) и сохраним файл (File | Save As) с именем *decode7.tdf.* Свяжем проект с текущим файлом, выбрав File | Project | Set Project to Current File. Введем текст примера и сохраним файл (Ctrl+S).

Созданный проект дешифратора можно проверить с привлечением платы LabKit8000. На этой плате (Приложение 1) имеется 8-разрядный переключатель, который можно использовать для задания входных сигналов, и три 7-сегментных индикатора, один из которых может быть использован для отображения результатов. Эти устройства на плате подключены к определенным выводам ПЛИС EPF8282ALC84, поэтому в проекте необходимо указать тип ПЛИС и соответствие номеров ее выводов входным и выходным линиям дешифратора. Тип ПЛИС определяется командой Assign | Device. В открывшемся диалоговом окне в строке Device Family нужно указать семейство FLEX8000, а в строке Devices определить тип микросхемы - EPF8282ALC84-2. Для указания соответствия выводов входным и выходным линиям дешифратора предназначена команда Assign/Pin Location Chip, при вводе которой открывается диалоговое окно, приведенное на рис.3.2. После ввода имени линии в строке Node Name в строке Pin следует указать номер вывода и щелкнуть по панели Add. В списке Existing

Pin/Location/Chip Assigment появляется строка соответствия вывода и линии. После назначения всех линий вводом OK нужно вернуться в редактор.

Трансляция исходного текстового файла осуществляется обычным образом, например компилятор можно вызвать щелчком по соот-ветствующей панели меню инструментов (рис.3.2). Однако лучше ком-пилятор вызывать командой Max+plusII | Compiler и осмотрев открыв-шееся окно компилятора убедиться, что к процессу трансляции на последней стадии будет подключен ассемблер. Если его в списке нет, следует выбрать команду Processing и отключить опцию Functional SNF Extractor. После этого можно щелкнуть указателем мыши по панели Start диалогового окна компилятора.

Отладка с привлечением платы LabKit8000. Поскольку целью работы является создание дешифратора для индикатора, правильность функционирования спроектированного устройства можно проверить, наблюдая отображаемые знаки на левом 7-сегментном индикаторе платы LabKit8000. Задавать входной код можно с использованием тумблеров 1-4 8-разрядного переключателя. Нужно учитывать, что младший разряд управляется тумблером 1, т.е. визуально порядок тумблеров является обратным относительно общепринятого, когда младшим разрядом яв-ляется крайний правый.

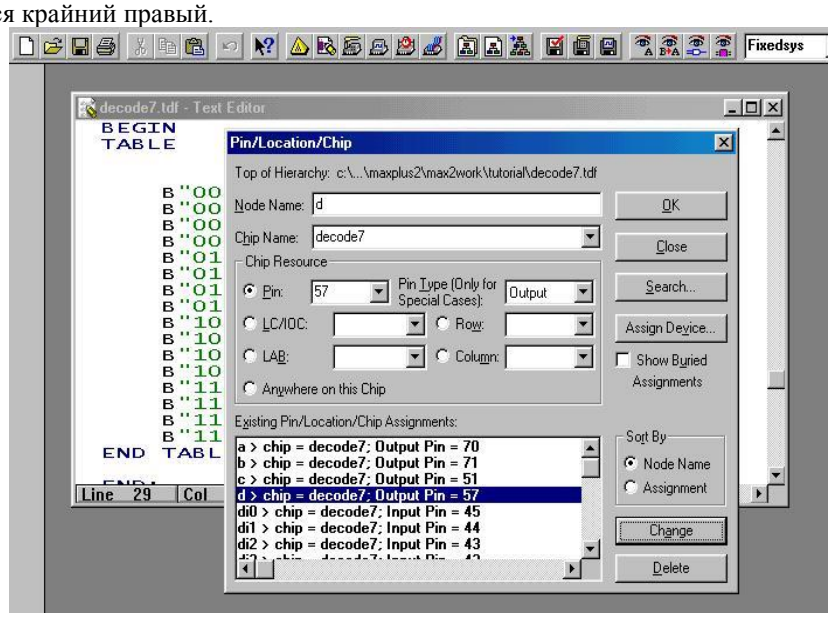


Рис. 3.2. Назначение выводам ПЛИС входных и выходных линий проекта

Для проверки функционирования разработанного дешифратора не-обходимо:

1. Подключить к разъему платы LabKit8000 кабель устройства ByteBlaster, который в свою очередь должен быть подключен к разъему LPT инструментального компьютера;
2. Подключить к разъему платы LabKit8000 кабель кабель питания +12В;
3. Включить источник питания - на плате должен загореться инди-катор (в правом верхнем углу),
4. Ввести в системе **MAX+plusII** команду Assign | Global Project Device Options и в открывшемся диалоговом окне в строке Configuration Scheme указать Passive Serial (выбрать из меню в соответствии с рис 3.3);
5. Вызвать приложение Programmer, при активном окне приложения щелкнуть по команде Options | Hardware Setup и установить Hardware_Type = ByteBlaster, ввести OK. Эта операция выполняется один раз при настройке системы на загрузку через ByteBlaster. Далее щелкнуть по панели Configure, инициируя процесс загрузки ПЛИС;
6. Переключая тумблеры 1-4 8-разрядного переключателя, наблюдать отображаемые на левом 7-сегментном индикаторе шестнад-цатеричные цифры. Проверить правильность работы дешифратора.

Анализ размещения схемы устройства на кристалле с исполь-зованием редактора конфигурации БИС. Вызвать редактор кон-фигурации можно командой Max+plusII | Floorplan Editor или щелчком по панели в меню инструментов. Откроется одно из двух вариантов изображения конфигурации ПЛИС (в данном случае типа EPF8282ALC84-2). Пусть это изображение типа Device View, приве-денное на рис.3.4. На нем видно расположение всех выводов ПЛИС и назначенные им имена входных и выходных линий проекта. Переключить тип изображения конфигурации можно командой Layout или двойным щелчком мыши по изображению.

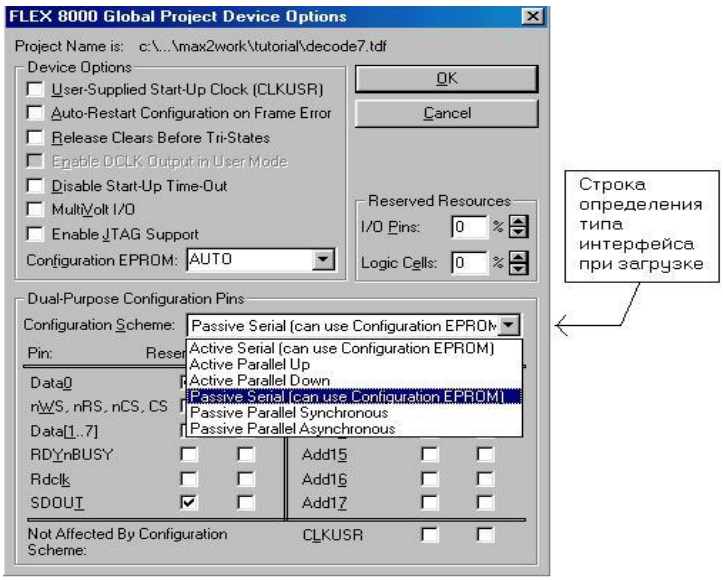


Рис. 3.3. Определение способа загрузки ПЛИС

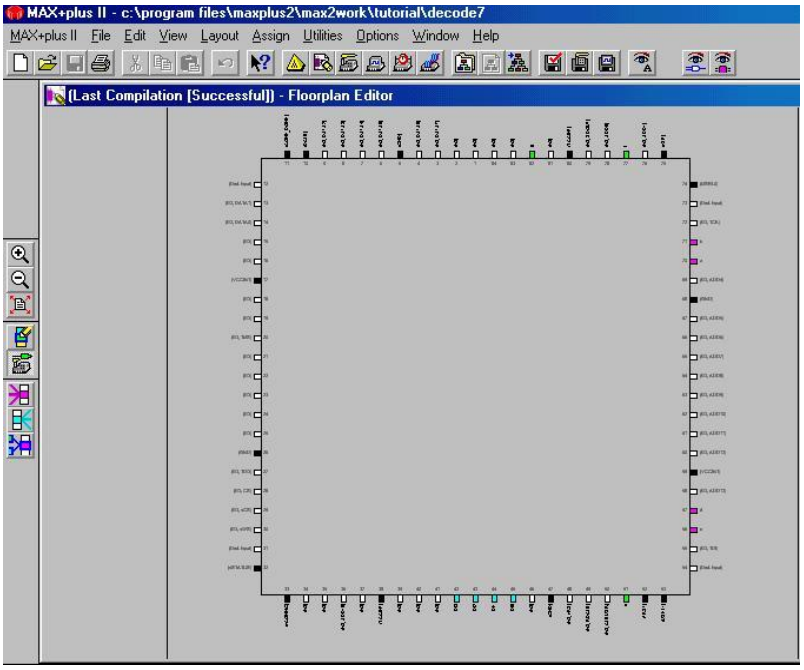


Рис. 3.4. Изображение конфигурации ПЛИС типа Device View

Изображение типа Lab View, приведено на рис. 3.5.

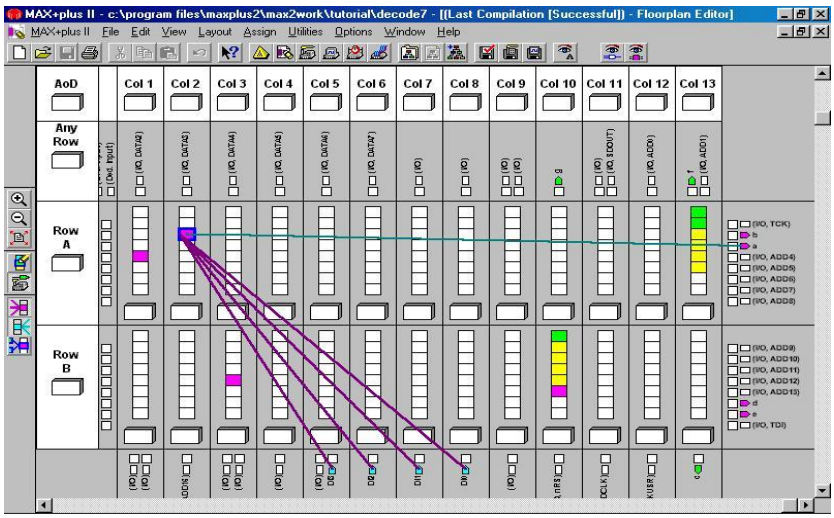


Рис. 3.5. Изображение конфигурации ПЛИС типа LAB View

Из рисунка видно, что при наведении указателя на вывод, рядом с указателем в рамке отображается имя линии проекта и номер вывода микросхемы. Редактор конфигурации позволяет получить информацию о связях отмеченных логических ячеек и выводов. Это делается следующим образом:

1. Выберем опцию **Show Node Fan-In** ("входные цепи") и/или опцию **Show Node Fan-Out** ("выходные цепи") команды Option.

2. Установим режим Lab View и отметим одну или более логических ячеек или выводов.

Редактор отобразит входные и выходные цепи отмеченных элементов. На рис.3.5 отображены связи третьей логической ячейки ряда A, столбца 2. Редактор конфигурации позволяет просматривать и изменять текущие назначения, которые хранятся в файле project`s Assignment & Configuration File(.acf). После трансляции проекта можно вручную редактировать назначения, сделанные компилятором и хранящиеся в файле project`s Fit File (.fit). Для этого существует опция **Back-Annotate Project** команды Assign, которая осуществляет копирования назначений из файла .fit в файл .acf. Процесс корректировки назначений проекта с использованием редактора конфигурации следующий:

1. Введем команду Assign | Back-Annotate Project.

2. В открывшемся диалоговом окне включим опцию *Chips, Logic Cells, Pins & Devices* раздела *Back-Annotate to ACF*. После ввода OK будет проведено копирование назначений из файла .fit в файл .acf.

3. Введем команду Layout | Current Assignments Floorplan. Редактор конфигурации отобразит текущие назначения проекта.

4. Выберем опцию **Show Node Fan-In** ("входные цепи") и/или опцию **Show Node Fan-Out** ("выходные цепи") команды Option.

5. Введем команду поиска Utilities | Find Text, в диалоговом окне включим опцию *Pin & Node Names* и выключим опцию *All*. Укажем слово для поиска DI0 и завершим OK. В окне редактора конфигурации будет отмечен назначенный этому имени вывод и указаны его связи.

6. Введем команду Option | Show Moved Nodes in Gray, которая отме-чает серым цветом новые назначения.

7. Наведем указатель мыши на отмеченный вывод линии DI0 и при нажатой правой кнопке мыши "перетащим" назначение с вывода 45 на вывод 37. Аналогично переназначим линию DI1 с вывода 44 на вывод 39, линию DI2 с вывода 43 на вывод 40, линию DI3 с вывода 42 на вывод 41. Новые назначения будут отмечены серым цветом.

8. Запустим перекомпиляцию проекта (Max+plusII | Compiler), убедившись, что отключена опция Processing | Functional SNF Extractor;

9. После успешной перекомпиляции вернемся в окно редактора конфигурации и включив опцию Layout | Last Compilation Floorplan убедимся, что новые назначения выполнены. Эти назначения определяют в качестве формирователей входных векторов на плате тумблеры 5-8 8-разрядного переключателя, причем младший разряд (DI0) задается переключателем 8, т.е. целью переназначения было установить общепринятый порядок, когда младший значащий бит находится справа;

10. Введем команду Assign | Back-Annotate Project;

11. Загрузим файл проекта в ПЛИС на плате LabKit8000 командой Max+plusII | Programmer и переключая тумблеры 5-8 8-разрядного переключателя, проверим правильность работы дешифратора, наблюдая отображаемые на левом 7-сегментном индикаторе шестнадцатеричные цифры.

ЗАДАНИЯ

1.Разработать комбинационный сумматор 4-разрядных операндов A = a3a2a1a0 и B = b3b2b1b0 с последовательным формированием переносов. Операнд A вводится переключателями S4-S1, операнд B - переключателями S8-S5, вывод результата на светодиоды L5-L1.

2. Разработать комбинационный сумматор двоично-десятичных опе-рандов A = a3a2a1a0 и B = b3b2b1b0 со схемой коррекции результата. Операнд A вводится переключателями S4-S1, операнд B - переключателями S8-S5, вывод суммы на светодиоды L4-L1, сигнала переноса в старшую тетраду - на светодиод L8.

3. Разработать комбинационный сумматор операндов A = a3a2a1a0 и B = b3b2b1b0, представленных в коде "с избытком 3", со схемой коррекции результата. Операнд A вводится переключателями S4-S1, операнд B - переключателями S8-S5, вывод суммы на светодиоды L4-L1, сигнала переноса в старшую тетраду - на светодиод L8.

4. Разработать преобразователь чисел, представленных в коде "с из-бытком 3", в 7-сегментный код с индексацией поступления нештатных комбинаций символом E (error).Операнд вводится переключателями S4-S1, вывод результата на 7-сегментный дисплей.

5. Разработать преобразователь двоично-десятичного кода (одна де-када) со знаковым разрядом в 7-сегментный код с индикацией знака "-" (ми-нус). Отрицательные числа представлены в дополнительном коде. Операнд вводится переключателями S5-S1, вывод результата "знак-число" на два 7-сегментных дисплея.

6. Разработать схему формирования модуля (абсолютного значения) 4-разрядного двоичного числа A=a3a2a1a0 со знаковым разрядом N. Отрицательные числа, представленные в дополнительном коде, переводятся в прямой код. Операнд вводится переключателями S5-S1, вывод результата на светодиоды L4-L1.

7. Разработать схему преобразователя, изменяющего знак 8-разря-дного операнда A (старший бит операнда является знаковым: a7=N). Отрицательные числа представляются в дополнительном коде. Операнд вводится переключателями S8-S1, вывод результата на светодиоды L8-L1.

8. Разработать схему, определяющую число единичных битов в 8-разрядном операнде. Операнд вводится переключателями S8-S1, вывод результата на светодиоды L3-L1.

9. Разработать схему, выдающую сигнал Z=1 при значении посту-пающего 8-разрядного операнда A=0 и формирующая номер старшего (первого слева) единичного бита при A>0. Операнд вводится пере-ключателями S8-S1, вывод сигнала Z на светодиод L8, номер старшего единичного бита на светодиоды L3-L1.

10. Разработать схему сдвига 4-разрядного числа влево или вправо на 0, 1 или 2 разряда (свободные разряды заполняются 0). Операнд вво-дится переключателями S6-S3, при сдвиге вправо число разрядов зада-ется переключателями S2-S1, при сдвиге влево - переключателями S8-S7. Индикация результата на светодиодах: при отсутствии сдвига на L6-L3, при сдвигах - соответствующее изменение позиций влево или вправо. При неправильном задании (сдвиг на 3 позиции или одновременный сдвиг влево и вправо) - индикация "1" на всех светодиодах.

11. Разработать логическое устройство, реализующее при посту-плении микрокоманды M=m2m1m0 следующий набор логических функ-ций над двумя 2-разрядными операндами A=a1a0 и B=b1b0:

инверсия A: M = 000,
инверсия B: M = 001,
конъюнкция: M = 010,
дизъюнкция: M = 011,
исключающее ИЛИ: M = 100,
равнозначность: M = 101.

Операнд A вводится переключателями S2-S1, операнд B – пере-ключателями S4-S3,микрокоманда M - переключателями S8-S6. Резуль-тат выводится на светодиоды L2-L1. При поступлении неправильного кода M = 110 или 111 зажигается сигнал ошибки - светодиод L8.

12. Разработать логическое устройство, реализующее при посту-плении микрокоманды M=m3m2m1m0 полный набор из 16 логических функций над двумя 2-разрядными операндами A=a1a0 и B=b1b0. Операнд A вводится переключателями S2-S1, операнд B – пере-ключателями S4-S3, микрокоманда M - переключателями S8-S5. Резуль-тат выводится на светодиоды L2-L1.

13. Разработать схему вычитания (A - B) двух 4-разрядных опе-рандов A = a3a2a1a0 и B = b3b2b1b0 с формированием знакового разряда N. Отрицательный результат представляется в дополнительном коде. Операнд A вводится переключателями S4-S1, операнд B – пере-ключателями S8-S5, результат выводится на светодиоды L4-L1, знаковый разряд N на светодиод L5.

14. Разработать схему умножения двух 4-разрядных операндов A = a3a2a1a0 и B=b3b2b1b0. Операнд A вводится переключателями S4-S1, операнд B - переключателями S8-S5, результат выводится на светодиоды L8-L1.