31. Уровневая организация компьютерных систем. Элементы уровня организации. Уровневый архитектурный стиль. Примеры. Явление разделения на уровни (disaggregation) и их смешения. Документирование инструментальных цепочек.

Мы в основном выделяли 2 основных уровня – hardware и software + уровень абстракции, явно их разграничивающий, но это не полная классификация, в реальных системах этих уровней очень много. Уровневая организация компьютерных систем – естественный способ организации вычислительных систем.

Примеры:

1. Lava Flow (организация курильщика, антипаттерн) – стартует проект с амбициозной командой, пару лет пишут код, выпустили продукт, но рано или поздно команда разбредается или поднимается по иерархии, в общем меняется руководитель. Он офигевает с того, как все криво написано, и начинает проводить ревизию написанного, новая команда формирует «новый слой лавы», в основании остается уже написанное, в итоге получается структура пирога.

2. Прошлый пример неудачный из-за кучи уровней абстракции и сложности поддержки, однако если все написано грамотно, то формируется уровневый архитектурный стиль

   Система разделяется на группы модулей (слои) с основной идеей, что вышележащий слой может обращаться к нижележащему слою, но не наоборот.

   EG: Стандартная библиотека -> фреймворк (может вызвать СБ) -> пишем либу для прикладной задачи и тд. Если написано грамотно – то хорошая поддержка и разделение между разрабами (без конфликтов)

3. OSI модель

   Application Layer – бизнес-требования, прикладное понимание того, что происходит в сетевой составляющей (уже никого не волнует как идет сигнал и тд)
   Presentation Layer – шифрование трафика между узлами
   Session Layer – установление сессии для разового прокладывания и фиксации единой сетевой цепочки для всего последующего соединения
   Transport Layer – отвечает за сбор и формирование пакета данных для отправки и доставки их одной посылкой на принимающий узел, организовать переотправку данных в случае сбоев, фиксация что данные испорчены
   Network Layer – из-за невозможности сделать прямое соединение между 2 компами, у нас есть множество отдельных линков, которыми связаны узлы сети между ними. Отвечает за то, чтобы пакет прошел нужное кол-во этих линков и попал куда надо.
   Data Link Layer – система явно идентифицирует источник и приемник передаваемых данных (чтобы если к одному проводнику подключено несколько разных проводников, система понимала, куда конкретно мы передаем инфу)
   Physical Layer – модуляция физ. сигнала, движение электрических токов, представляющие собой передаваемые данные

4. Уровни организации вычислительного процесса (с точки зрения ее построения и функционирования):

   - Кремниевый чип
   - Исполняемые файлы
   - Программы (исходный код)
   - Акторные модели (архитектурная спецификация системы)
   - Application level (задает детальное описание юзкейса)

   Если система реализована корректно, то есть описание на каждом из этих уровней, они все работают параллельно и дополняют друг друга, так как не могут работать отдельно.

   На этих уровнях могут быть:
   - Уровневая организация проекта.
   - Frameworks, библиотеки, API.
   - Языки программирования.
   - Операционные системы.
   - Системы команд (ISA), ПЛИС.
   - Виртуализированные ресурсы.

   Элементы уровня организации:
   Отдельный уровень представляет собой совокупность из 4х элементов:
   - Язык (модель вычислений)
   - Набор типов данных, с которыми мы можем оперировать
   - Наборы экземпляров (операции, которые предоставляет сам язык, чтобы мы могли их использовать)
   - Паттерны (связующее звено, собирает все это в кучу как инструмент для решения определенного класса задач)

   Явление разделения на уровни (disaggregation):

   Раньше все компании стремились выстраивать «вертикальную» архитектуру. Они производили свои чипы, свои платы, свои компьютеры, свои ОСи, под эти оси писали свои прикладные задачи и продавали это как единый «конструктор».

   Плюсы: можно эффективно подстраивать один уровень под соседний, не надо учитывать ненужную обратную совместимость, можно легко поменять архитектуру и перестроить систему под нужные задачи за счет контроля всей иерархии.
   Было много процессорных архитектур, осей, ЯПов, компиляторов и тд. Компьютеры были эффективнее с точки зрения кол-ва операций для обеспечения непосредственно какой-то функции для пользователя.

   Потом начался процесс дисагрегации – за счет рыночных механизмов и экономии на масштабах уровни начали растягиваться вширь.
   Результат: появилось несколько независимо существующих уровней, разделенных по индустриям и специальностям.

   Отдельная индустрия по производству процессоров (ограниченное кол-во архитектур), отдельный уровень операционных систем (небольшое число

основных – Windows, Linux, MacOS), стандартизированное прикладное ПО, универсальные продажи, разделенные по магазинам.

Объяснение почему стало именно так – **закон развития иерархических систем Седова**: чтобы обеспечить многообразие на вышележащих уровнях развития, нам надо ограничить разнообразие на нижележащих уровнях. В нашем случае – если нужно разнообразие прикладного ПО, надо, чтобы кол-во осей было ограниченным. Можно писать кроссплатформенные приложения, приносящие большую прибыль.

Параллельно с явным разделением на уровни идет процесс интеграции и смешения уровней в сложных случаях. Чтобы код считался эффективным, программисту необходимо учитывать, что происходит на нижних уровнях, иногда даже приходится вставлять куски низкоуровневого кода в высокоуровневую программу. Основная цель – зафиксировать общесистемное свойство всего компьютера (энергопотребление, реальное время, производительность и т.д., например, выполнение операций за фиксированное время в системах управления за счет ухода от дополнительных уровней абстракции и выстраивания сквозной системы за счет смешения уровней).

Документирование инструментальных цепочек.

В большинстве случаев простых систем документировать вообще не надо, компоненты взаимодействуют горизонтально на одном уровне абстракции.

Если все-таки нужно (e.g. во встроенных системах, высоконагруженной разработке), то используется граф актуализации вычислительного процесса – позволяет описать цепочку преобразований представлений вычислительного процесса как ориентированный ациклический граф, вершины — модели, рёбра — трансляции/интерпретации. Задает цепочка преобразований моделей выч. процесса через: стадии жизненного цикла (Design → Dev → Use) или вычислительные
Назначение: анализ инструментальных цепочек.

Плюс: позволяет совместить как дизайн при проектировании системы, так и рантайм, когда она уже может функционировать, а также показывает, что получается на разных стадиях.

32. Особенности реализации структурного программирования в фон Неймановских процессорах. Работа с памятью, регистрами. Реализация процедур. Реентерабельность. Рекурсия. Реализация условного оператора и циклов.

Структурное программирование:

Основная идея: абстрагироваться от устройства процессора. его инструкций и условного / безусловного JMP и объединить имеющийся код в более крупные блоки и оперировать уже этими блоками с точки зрения нужной логики.

Имеет в совей основе 4 основных составляющих: последовательное выполнение кода, условный оператор (ветки – структурные блоки, и сам по себе if тоже заворачивается в структурный блок), циклы (также как и условный оператор: сам цикл – структурный блок, еще один структурный блок внутри цикла) и процедурные блоки (можем вызвать процедуру, перейти в выполнение ее структурного блока и вернуться обратно)

Распределение регистров:
Нужно замапить на физические регистры те переменные, с которыми мы работаем. Всегда лучше минимизировать кол-во переменных и обращений к памяти.

Проблемы:

- кол-во регистров конечно (ARM 15/31, x86 8/16, MIPS 32/32),
- не все регистры одинаковы (особенно в CISC).
- Проблема компилятора или программиста низкого уровня. (программист всегда должен держать в голове, какие регистры для чего используются, какие сохранить в память и прочитать оттуда и тд)
- Код не ограничивается математическими выражениями. Основная проблема – как синхронизировать 2 рядом стоящие функции, чтобы они не испортили данные друг другу, для вызова функции надо взять переменные и положить их в память, т.к. неизвестно как функция работает с регистрами
- Согласование использования регистров между вызовами процедур.

Процедуры:

1. У каждой подпрограммы своя память, свои данные и своя работа с регистрами, и они не должны конфликтовать между собой.

2. Особенности:

- Память выделена статически.
- Реализация:
- Через inline.
- Через goto.
- Через call, return.

3. Зачем: переиспользование машинного кода; оптимизация работы кеша инструкций (код процедуры).
4. Нет реентерабельности (значения являются глобальными переменными, если попадем в одну функцию несколько раз, то данные будут поломаны и потеряны).
5. Проблемы: кеширование, сброс регистров, переходы и конвейер.

Реентерабельность и рекурсивный вызов:

Нужна, когда нам нужно попадать в одну функцию несколько раз.

1. Реализация через call, return на аппаратном уровне (через стек возврата, куда сохраняем точки).
2. Статическое выделение памяти для каждого входа (для каждого входа выделяются свои уникальные переменные. Минус: нужно отдельно определить место, которое будет его обеспечивать в памяти)

   ИЛИ

Автоматическая память, стек: push, pop (рекурсия).
3. Зачем: переиспользование машинного кода в run-time (нет возможности контролировать всю кодовую базу, которая используется, поэтому за счет автоматической памяти можно легче определять интерфейсы взаимодействия между функциями и переиспользовать код); рекурсивные алгоритмы.
4. Проблемы: автоматическая память, утечки данных, перезапись адреса возврата.

Однако, когда выстраиваем работу с автоматической памятью, мы быстро автоматизируем работу с ней, а так как большинство высокоуровневых языков основаны на вызове подпрограмм и выходы из них, такая оптимизация дает много преимуществ, т.к. наверху стека лежат наиболее часто используемые данные, статическую память сложнее разложить.

Условный оператор

Позволяет проверить состояние регистра и вычислить к-л структурный блок. С точки зрения процессора: JMP, который пропускает какую-то часть инструкций и выполняет другую часть, адреса переходов считаются на уровне транслятора.

При традиционном подходе: встроенная конструкция языка, реализуемая инструментарием (компилятор, интерпретатор). Отсюда минус: нельзя поменять реализацию или написать свою если вдруг надо. Более интересный подход реализован в SmallTalk, но это уже за рамки вопроса.

Циклы

Писал выше, но еще раз: есть структурный блок с циклом, еще один структурный блок, описывающий поведение внутри цикла и который может прокрутиться сколько-то итераций на одном месте

33. Уровни параллелизма. Параллелизм уровня бит. Низкоуровневый параллелизм. Параллелизм уровня инструкций. Параллелизм уровня задач. Примеры и особенности.

Рассматривая параллелизм с точки зрения того, как эффективнее (с большей производительностью) использовать доступные физические ресурсы для организации вычислительного процесса, при этом оставаясь в рамках архитектуры фон Неймана, выделяют 3 уровня параллелизма:

Параллелизм уровня бит — это форма параллелизма, которая использует возможность процессора выполнять операции с битами параллельно, чтобы ускорить вычислительные задачи.

Достигается за счет “ширины” комбинационных схем. В контексте фон Неймановского процессора это ширина машинного слова – фрагмент данных фиксированного размера, обрабатываемый как единое целое с помощью набора команд или аппаратного обеспечения процессора. Чем больше размер слова процессора (например, 32-битный, 64-битный), тем больше битов он может обрабатывать за один такт. Операции, такие как сложение, вычитание, логические операции (И, ИЛИ, XOR), могут выполняться параллельно над всеми битами слова. Для эффективного использования данного уровня параллелизма

необходимо перейти на уровень ASIC или использовать такие вычислительные платформы как ПЛИС.

Имеет серьезные ограничения, например, не имеет смысла наращивать простые типы данных, int64 хватает для подавляющего большинства задач, и расширение до int128 приведет к тому, что чаще всего старшие биты просто будут занимать место, и не факт, что их будет достаточно даже в случаях, когда они реально нужны. Использовать же широкое машинное слово для составных данных также затруднительно в случае процессоров общего назначения из-за их многообразия (составные данные зависят от прикладной задачи) и необходимости их поддержки на уровне системы команд, без которой это теряет смысл.

Примеры: для манипуляции битами (битовые сдвиги, маскирование, и побитовые логические операции, часто используемые в системном программировании и оптимизациях)

Параллелизм уровня инструкций включает в себя:

- Конвейеризированное исполнение команд (разбиение выполнения инструкции на последовательность этапов). Данный способ использует возможность параллельной и независимой работы комбинационных схем, разделённых регистрами, сдвиг же конвейера реализуется по глобальному тактовому сигналу. Дает повышение производительности и уровня утилизации вычислительных ресурсов.

При реализации могут возникать сложности из-за конфликтов между стадиями конвейера, такие как:

- Структурные конфликты (конфликт ресурсов, аппаратура не поддерживает все возможные комбинации одновременно выполняемых команд)
- Конфликты по данным (зависимость команды от результатов предыдущей проявляется при совмещении команд в конвейере)

- Конфликты по управлению (возникает во время исполнения команд условного и безусловного перехода, вызова процедур и т.п. Наибольшая сложность – условный переход, т.к. заранее неизвестно, как измениться счетчик команд)

Также при работе с числами с плавающей точкой вычисления занимают много тактов, конвейер замирает в ожидании выполнения этой инструкции. Для избегания этого применяют реализации:
- Суперскалярный процессор (Подход, позволяющий одновременно исполнять несколько скалярных (операнд – число) и/или векторных (операнд – массив) операций называется суперскалярным. Загрузку АЛУ можно оптимизировать за счёт параллельного исполнения операций если есть доступные блоки обработки и команды не зависят друг от друга по данным. Для этого необходимо в процессе работы “на лету” находить такие ситуации, для чего формируются специальные узлы процессора.

Важной особенностью суперскалярных процессоров является то, что с точки зрения системы команд они идентичны простым процессорам и вся диспетчеризация происходит без ведома пользователя.

- VLIW (very long instruction word)

Lелает попытку повторить элементы архитектуры RISC и переложить сложность суперскалярного процессора на компилятор, а именно – принять решение о том, какие операции должны быть параллельны, а какие нет. Для этого формируются очень длинные машинные команды, в которые кодируются операции для всех устройств обработки (АЛУ), входящие в состав процессора.

Ограничения:
- Низкая плотность исходного кода (часть инструкций остается пустыми)
- Ширина команды VLIW процессора накладывает ограничение на микроархитектуру процессора, что затрудняет независимую разработку процессора и компилятора
- Нужно большое количество оптимизаций

На сегодня VLIW процессоры широко применяются для решения специализированных вычислительных задач, к примеру: обработка мультимедиа информации, расчёты и т.п., в то время как их применение для задач общего назначения, как правило, уступает решениям на базе суперскалярных процессоров.

Низкоуровневый параллелизм.

Глобально то же самое, что и прошлый пункт. Основная идея – программист не видит, как эксплуатируется параллелизм, можем полностью спрятать особенности низкоуровневого программирования, чтобы он о них не думал.

- Конвейеризация. Разбиение выполнения инструкции на последовательность этапов.
- Отделение ЦПУ от процессоров ввода/вывода. К примеру: DMA.
- Множественные функциональные узлы в ЦПУ. Независимые функциональные блоки для арифметических и булевых операций, выполняемых одновременно.
    - Суперскалярный процессор
    - Very Long Instruction Word

Параллелизм уровня задач.

Подразумевает параллельное выполнение нескольких потоков команд, которые определены разработчиком. Центральное место занимает вопрос о том, как происходит переключение между потоками команд, влияние этого на разработку ПО и синхронизацию потоков исполнения.

Варианты:
- Кооперативная многозадачность (следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам). Реализуется в рамках оси, на уровне ВМ / компилятора, в ручном режиме. Обеспечивает отсутствие необходимости защищать все разделяемые структуры
данных объектами типа критических секций и мьютексов и контроль за доступом к ресурсам процессора со стороны разработчика.

Пример использования: для оптимизации систем, требующих частого переключения между задачами или большого количества потоков команд. Реализуется на уровне виртуальной машины и/или приложения/сервиса. Пример: nginx и apache, node.js (Event-loop).

- Вытесняющая многозадачность (операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода,

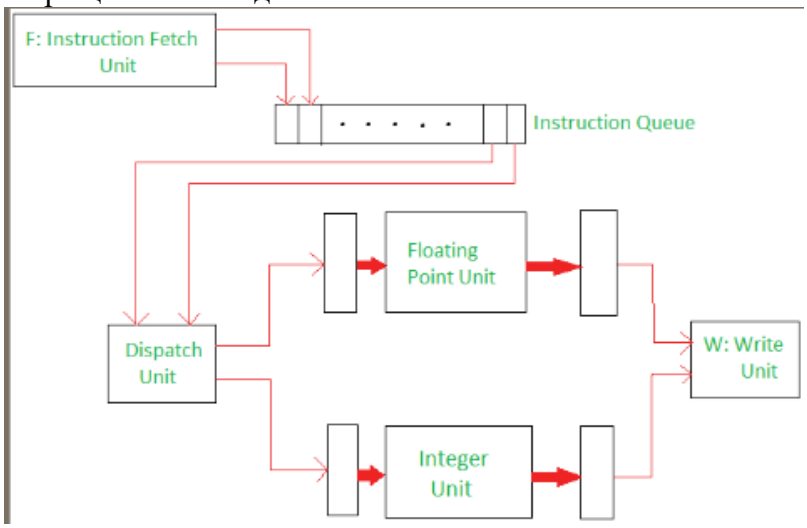
возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой.)

Основным преимуществом вытесняющей многозадачности является относительная защищённость процессов и ОС друг от друга с точки зрения захвата ресурсов процессора. Также, вытесняющая многозадачность позволяет реализовать системы, работающие более интерактивно, чем в случае кооперативной

34. Параллелизм уровня инструкций. Суперскалярные процессора. Особенности и принципы работы. Сравнение с VLIW и практика использования. Достоинства и недостатки. Барьеры памяти.

АЛУ может работать с числами с плавающей точкой. Если мы хотим произвести операции над такими числами, это занимает много тактов, и весь конвейер замирает в ожидании выполнения этой инструкции, т.к. это долго. Чтобы избавиться от этого, есть 2 подхода, 1 из них – суперскалярный процессор. Скалярная – величина, которая может быть представлена числом (целочисленным или с плавающей точкой).

Упрощенно выглядит вот так:



Стадия Instruction Fetch – достать из памяти инструкцию, положить в память, подготовиться к ее исполнению

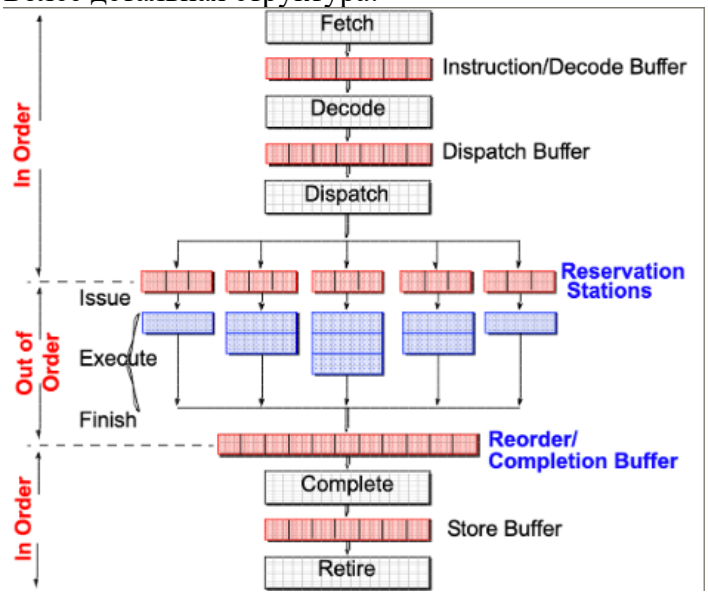
2 узла – один для всех операций с плавающей точкой (работает за 3 такта), другой для целочисленных значений (работает за 1 такт) *допустим

Основная идея – загружать инструкции во Float Point Unit, и пока он ковыряется, параллельно можно загрузить Integer Unit, он выполнит свою задачу и сохранит результат.

Плюсы: выравниваем инструкции по времени, т.к. с большей вероятностью всегда есть инструкция либо там, либо там при параллельной работе.

Проблемы: диспетчеризация проводится в рантайме, т.к. не можем предугадать, какая инструкция следующая.

Более детальная структура:



1. Инструкции извлекаются в очередь команд по порядку.
2. Декодируются, после чего мы знаем, какие вычислительные узлы процессора понадобятся для операции
3. Операция Dispatch (прям отдельный юнит на уровне процессора) – единый поток инструкций раскидывается по «исполнительным стадиям» в зависимости от того, куда они нужны, и единая очередь разбивается на несколько очередей к разным станциям обслуживания для выполнения, работающих параллельно и независимо
4. Нужно обеспечить, чтобы инструкции могли выполняться не только на основании последовательности в исходном потоке, но и чтоб была возможность привязать события подсистемы кэширования для выравнивания внешних погрешностей доступа к памяти.
5. Reorder Buffer – помещает результаты вычисления всех задач и обратно корректно соединяет весь порядок инструкций без нарушения последовательности

Недостатки:

- Конфликты по данным оказывают значительное влияние на производительность и сложность процессора.
- Высокое энергопотребление (куча перестановок, большая длина буферов, много спекулятивных вычислений, наличие логики в рантайме, все это жрет энергию)
- Проблемы детерминированности работы многоядерных процессоров (за счет перестановки инструкций случайным образом из разных потоков инструкций могут получаться самые разные, порой веселые, результаты)

Достоинства:

- Рост производительности. Сглаживание длительности выполнения инструкций.
- Повышение уровня загрузки ресурсов.
- Совместимость с существующим машинным кодом (может использовать не убитый внутри кода параллелизм, пусть даже он написан 40 лет назад, за счет чего быстрее результат).
- Компилятор может устранить значительное количество конфликтов за счёт сортировки инструкций.

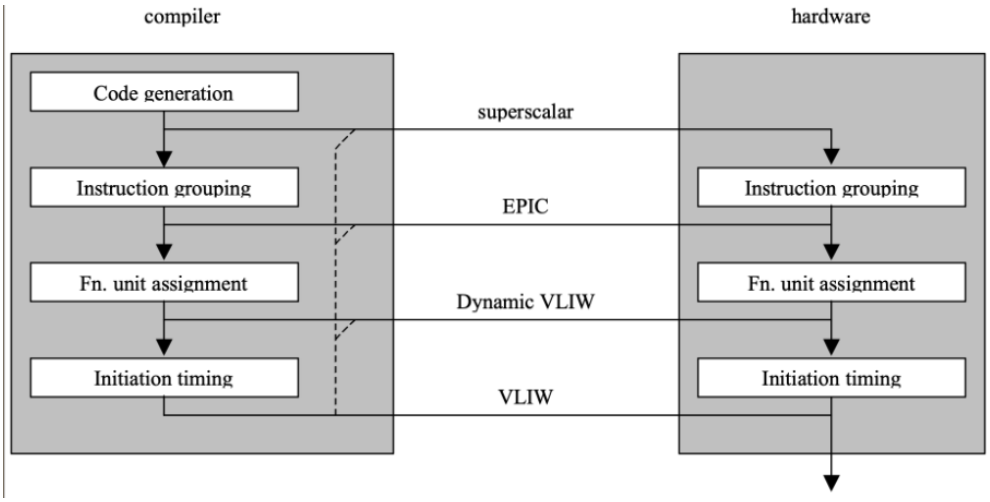
Барьеры памяти:

Барьер памяти – вид барьерной инструкции, которая приказывает компилятору (при генерации инструкций) и центральному процессору (при исполнении инструкций) устанавливать строгую последовательность между обращениями к памяти до и после барьера, их нельзя никогда менять местами.

Все обращения к памяти перед барьером будут гарантированно выполнены до первого обращения к памяти после барьера.

Много барьеров памяти снижает производительность, но при этом они четко показывают, какие операции сортируются до барьера, какие после, что защищает от ситуации, когда данные перемешиваются.

Сравнение с VLIW:



Слева compile time (то, что делает инструментарий и трансляторы), справа hardware (то, что делает аппаратура).
Суперскаляр – верхняя стрелка. Компилятор генерирует код, и дальше уже на уровне аппаратуры суперскаляр группирует инструкции в рамках диспатча, назначает функциональные блоки, которые будут их выполнять, определяет тайминги выполнения операций и получает результат.
VLIW – нижняя стрелка, все на уровне компилятора, сначала генерируем код, группируем инструкции в зависимости от формата широкой инструкции, определяем, в какой аппаратный блок будет попадать конкретная инструкция и планируем тайминги.

VLIW ориентированы на то, что поток инструкций имеет строгую структуру, на уровне потока мы знаем, какие инструкции будут выполняться параллельно и где переход от одной подпрограммы к другой.

В суперскаляре единый поток инструкций, процессор динамически генерирует нужное разбиение.

Практика использования:

На просто современных ПК суперскаляр будет работать гораздо эффективнее, потому что неизвестно, какой поток инструкций в реальной жизни, а значит не можем его полностью оптимизировать.

Во VLIW бывают приколы, когда компилятор не смог сделать инлайн или не понял, как лучше оптимизировать, в результате код работает очень медленно, хотя ожидалось, что наоборот.
Однако VLIW хороши в случаях, когда есть встроенная система, просто, например считающая какие-то объемные вычисления. Тогда можно все заинлайнить в одну большую plain функцию, где оптимизирующий компилятор уже выдаст максимум из возможностей железа лучше, чем суперскаляр.

35. Параллелизм уровня инструкций. VLIW процессора. Особенности и принципы работы. Сравнение с суперскалярными и практика использования. Достоинства и недостатки. Барьеры памяти.

Основная идея близка к RISC процессору – пересмотреть систему команд, сделать так, чтобы процессору проще было все вычислять и сэкономим энергию, упростим процессор и повысим производительность.

VLIW реализует идею заниматься параллелизмом не в рантайме, а в компайл тайме. Процессор в каждую инструкцию кодирует 4 маленьких подинструкции, где каждая инструкция направлена на конкретный внутрипроцессорный узел.

Особенности:

- Объединим несколько инструкций в одну.

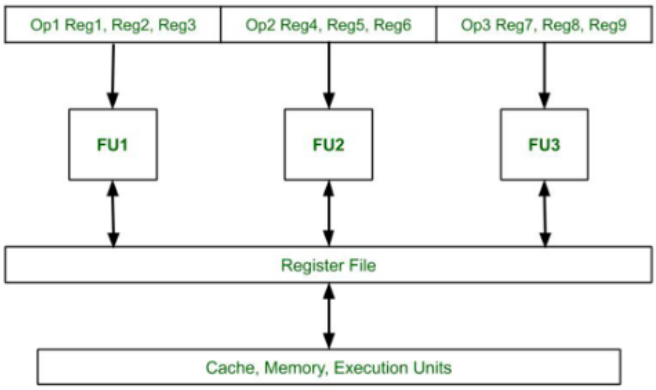
- Уберём из процессора механизмы перестановки инструкций.

- Компилятор имеет полный доступ к инструкциям, процессор о коде столько знать не может ⟶ может "выжать" весь возможный параллелизм.

Плюс за счет того, что у компилятора по сравнению с процессором бесконечная память – он может брать сколько угодно инструкций и оптимизировать их сколько угодно долго.

Общая схема работы:

CYCLE	INT ALU	INT ALU	FLOAT ALU	FLOAT ALU
1	t1 = t1+1	clr = clr+0010	y1 = x1*1010	y2 = x2*1100
2	r = r+0001		p = q*1000	
3	nop			
4	z1 = y1+0010	z2 = y2+0101		



Есть потоки инструкций для разных блоков, на уровне компилятора говорим, что нужно выполнить допустим первые 4 операции одновременно, во второй – 2 операции и тд. Исходный код получается короче, процессор выполняет его быстрее, не нужно все считывать последовательно и динамически диспетчеризировать.

Достоинства

- Упрощение процессора, снижение энергопотребления.
- Упрощение декодера. Рост частоты.
- Компилятор имеет больше информации о коде, он лучше знает, что параллельно!

Вроде кажется, что все классно, но на практике не сработало, т.к. создали очень специальный инструмент, глубоко специализированный под конкретные задачи и требующий сложного инструментария. Фактически люди не очень умеют этим пользоваться и не смогли заставить нормально работать и написать компиляторы, эффективно генерирующие машинный код, т.к. значительная часть параллелизма кода все-таки определяется в рантайме. Например, нужно, чтобы весь код был инлайном без JMP, т.к. тогда не сможем оптимизировать разбиение по инструкциям.

В результате следующие

Недостатки:

- Сложность компилятора.
- Высокая нагрузка на каналы данных и регистровые файлы.
- Конфликты конвейера приводят к простою всех узлов.
- Низкая плотность кода.
- Ширина команды — ограничение микроархитектуры, ломается абстракция

Барьеры памяти:

Концепция аналогична той, что в суперскалярах, но во VLIW они регулируются компилятором, который отвечает за выявление и разрешение зависимостей, что требует

анализа на этапе компиляции. При перечислении порядка подпрограмм (например подпрограмма 1 вызывает подпрограмму 2), либо делать инлайн, тогда компилятор сможет их перемешать, либо делать выбор процедуры, тогда перестановки быть не может, в то время как, если этот барьер функционально там не нужен, суперскаляр бы перемешал их уже в рантайме.

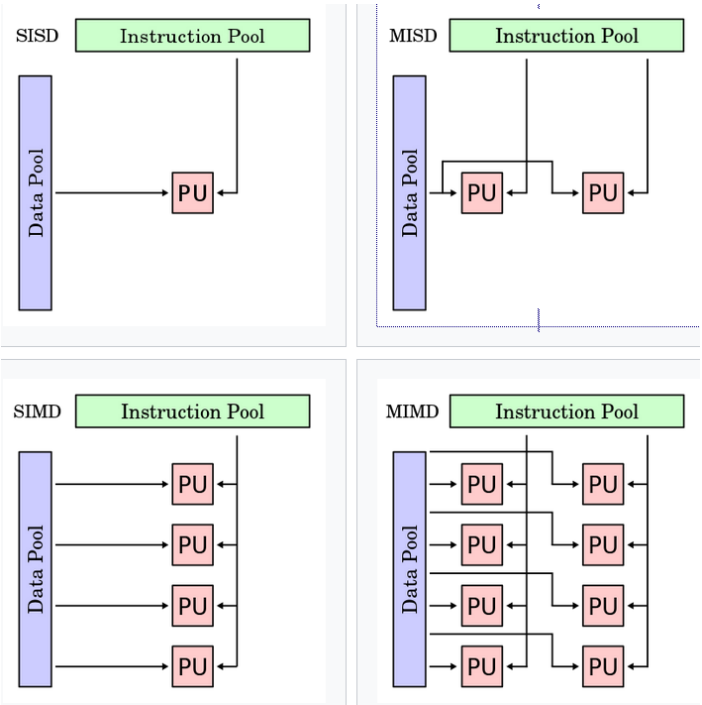
Сравнение с суперскалярами и практика использования:

Было уже выше, но еще раз ключевая разница в двух словах:

Суперскаляр – аппаратное планирование: используют аппаратное планирование для выполнения инструкций параллельно. Это включает в себя динамическое определение независимости инструкций и их правильного порядка выполнения.

VLIW – компиляторное планирование: процессоры полагаются на компилятор для планирования и параллельного выполнения инструкций. Компилятор должен определить независимость инструкций и сформировать длинные машинные слова, которые будут выполнены параллельно.

36. Классификация Флинна. Выделяемые классы и примеры машин этих классов. SIMT архитектура.



В фон Неймановском процессоре 2 потока: инструкций и данных, которые будут обрабатываться. Может быть несколько разных случаев их комбинаций.

Таксономия Флинна как раз задает классификация параллельных процессоров на основе:

- количества потоков инструкций.
- количества потоков данных.

Достоинства: простая и понятная.

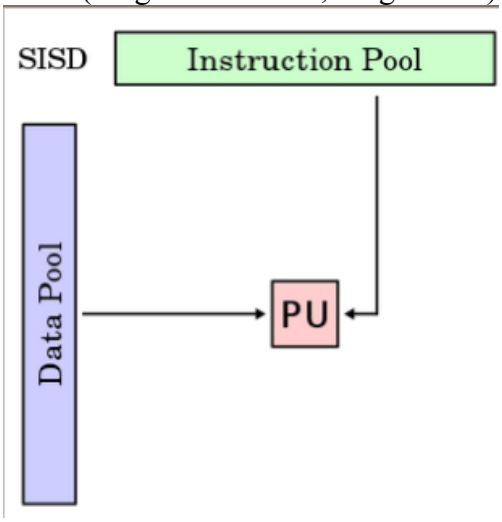
Недостатки:

- Не применима к не фон Неймановским архитектурам.

- Перегруженность класса MIMD.

Выделяемые классы:

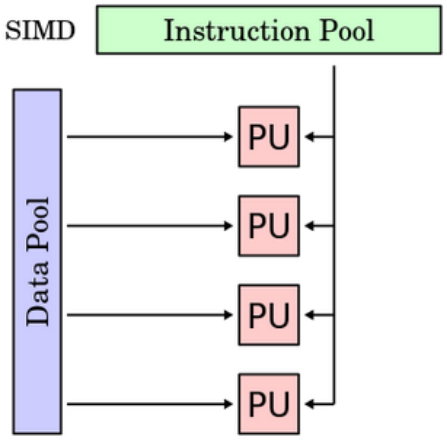
1. SISD (Single Instruction, Single Data)



- Простой классический фон Неймановский последовательный процессор.
- Абсолютное большинство рассмотренных ранее процессоров.
- Включая процессоры с низкоуровневым параллелизмом.

2. SIMD (Single Instruction, Multiple Data)

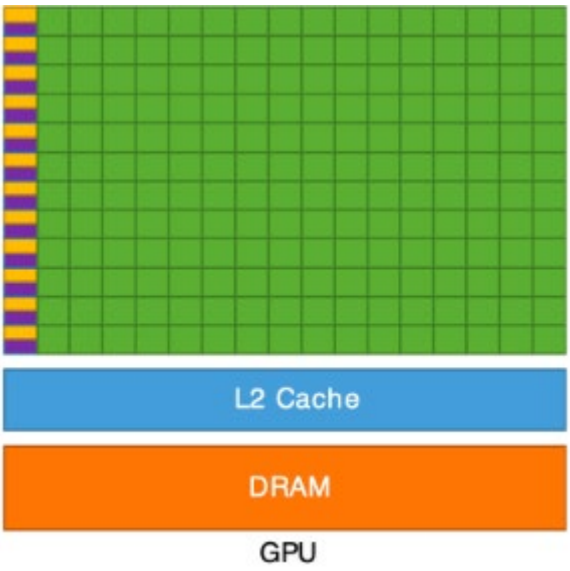
   Есть одно действие, которое мы должны применить много раз к разным данным, можем реализовать это с аппаратной оптимизацией, взяв каждый набор данных и отдать в отдельный процессор, отправляя одни и те же инструкции каждому процессорному ядру.



- Одновременное выполнение несколькими процессорами одной инструкции (Lockstep execution), все делают одно и то же, без вариантов.
- Назначение: однообразная обработка множества наборов данных.

- Пример: GPU, поиск блока для BitCoin, мат. Моделирование, в общем везде, где нужны матричные преобразования и подсчет одной и той же хэш суммы для разных наборов данных

Устройство GPU с кучей одинаковых ядер, обрабатывающих данные шаг за шагом.



- Ключевое ограничение: операции ветвления

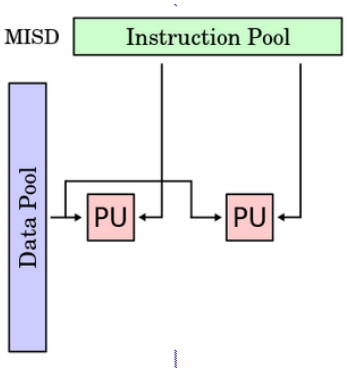
2.1. SIMT архитектура (Single Instruction, Multiple Threads, расширение SIMD)

Появляется возможность в зависимости от определенных данных выводить отдельные ядра из Lockstep execution в параллельную работу, за счет чего достигается реализация ветвления, но падает эффективность и возрастает сложность системы.

- Расширение SIMD, позволяющее работать с оператором ветвления.
- Широко применяется в современных GPU и GPGPU (CUDA).
- Включает механизмы синхронизации потоков между собой для достижения lockstep execution.

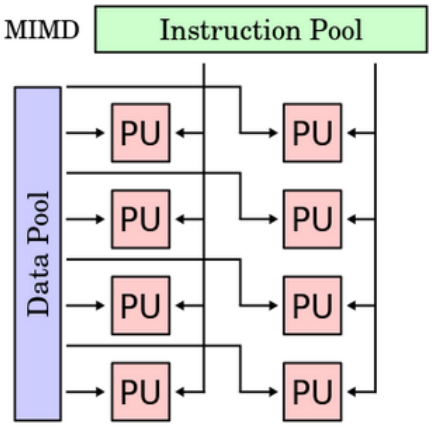
3. MISD (Multiple Instruction, Single Data)

Крайне редко встречается на практике, в основном в системах с очень высоким уровнем надежности, когда нужно параллельно посчитать разными алгоритмами одни и те же входные данные и убедиться, что именно алгоритмы работают корректно.
Пример: резервированные системы управления полетом



- Один поток данных обрабатывается разными наборами инструкций.
- Обычно выделяется для полноты классификации.
- На практике — повышение надёжности работы систем. Защита от ошибок проектирования и алгоритмов. Параллельная разработка и эксплуатация решений задачи.

4. MIMD (Multiple Instruction, Multiple Data)



Instruction пулы и потоки данных для каждого процесса уникальны, могут пересекаться или быть полностью независимыми. Примеры: параллельные и многоядерные процессора (любые современные, e.g. Intel Core i9).

- Множество процессоров автономно выполняют различные инструкции над различными данными.
- Самый разнообразный класс процессоров по классификации Флинна.
- Не выделяются подвиды, поэтому рассматривают в классификации Дункана

37. Классификация Дункана. Цели и задачи, классификация первого уровня. Синхронные архитектуры (векторные, SIMD, ассоциативные массивы, систолические). Принципы их работы и примеры.

Цель: уточнить класс MIMD из таксономии Флинна, использующийся в современных процессорах, сделать его более детализированным и разобранным

Задачи:

- Абстрагироваться от низкоуровневого параллелизма (конвейер, суперскаляр, VLIW) – поднимается на уровень выше и берет в расчет то как программируется система с точки зрения разработчика
- Переиспользовать элементы таксономии Флинна.
- Актуализировать и включить новинки(на тот момент).
- Навести порядок в MIMD.

Первый уровень классификации:

- Синхронные архитектуры

Параллельные архитектуры с единым механизмом управления (глобальные часы, центральный блок управления и т.п.), который определяет, как система будет двигаться вперед, действовать и развиваться. Различаются механизмами синхронизации.

- MIMD архитектуры

Параллельные архитектуры, которые могут исполнять независимые потоки инструкций. Потоки инструкций исполняются автономно, требуется динамическая синхронизация. Различаются структурой организации памяти.

*Рассматриваем именно с точки зрения аппаратной организации, как вычислительно построен внутри себя и как он работает с памятью

- MIMD парадигма (MIMD paradigm architectures)

Архитектура MIMD процессора определяется моделью вычислений (MoC). MoC определяет характер потоков данных, принципы взаимодействия и синхронизации. Различаются MoC.

*Больше про то, как развивается вычислительный процесс, какая в нем логика

Назад к вопросу 38 (MIMD архитектуры)

Назад к вопросу 39 (MIMD парадигмы)

Синхронные архитектуры – класс процессорных архитектур, обладающих единым/централизованным/синхронным управлением, определяющим поведение всего процессора в целом.

Варианты:

1. Векторные архитектуры

   Помогает оптимизировать операции с массивами данных, чтоб не проводить операции для каждой допустим пары значений массивов и работать не отдельно со

скалярными величинами, а сразу с векторными при помощи специальных регистров.

Скалярная величина – величина, которая может быть представлена числом (целочисленным или с плавающей точкой).
Вектор – величина, представленная последовательностью скалярных величин.
Векторные операции — операции, выполняемые на последовательностях данных.
Выигрываем на количестве кода и количестве использованных инструкций, оптимизация, меньше обращений к памяти.

Принцип работы:
Есть векторные регистры (в них хранится не 1 величина, а сразу много), дальше идет vector pipeline, в который эти данные загружаются и могут исполняться такт за тактом.

- Сокращение количества обрабатываемых инструкций.
- Конвейерное исполнение векторных операций.
- Множество функциональных узлов (суперскалярность), исполняемые задачи можно раскидывать тоже суперскалярно.

Синхронные, т.к. есть специальные операции, применяемые к потоку инструкций и позволяющие программисту эксплуатировать параллелизм внутри процессора, нужно четко знать, что задача может решаться векторными инструкциями и применять их => не низкоуровневый параллелизм.

2. SIMD архитектуры (процессорные массивы)

   Пример – графический процессор, рассматривали в таксономии Флинна.

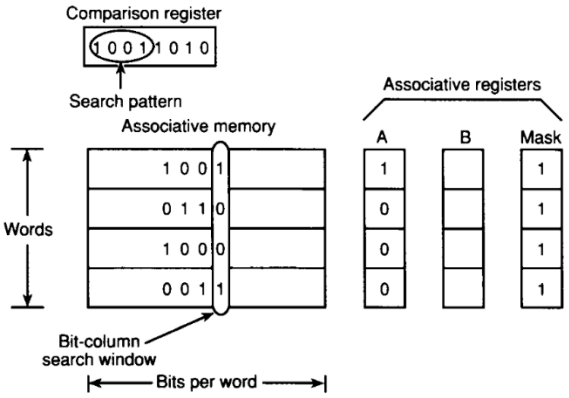
   Есть один control unit, поставляющий инструкции в каждый процессор, имеющийся в системе, в режиме Lockstep Execution.

   Инструкции исполняются не конвейерно, а параллельно, что требует изменения аппаратуры имеющегося вычислителя, за счет независимого доступа к памяти со стороны процессора как раз можно реализовать такую архитектуру.

3. Ассоциативные массивы

   Идея: можно построить массив памяти не просто пассивным хранилищем данных, а сделать его умным устройством, которое будет выполнять операции с приходящими данными при записи в него.

   Пример: реализация таблиц поиска через массив. Есть массив памяти, в него загружаются данные, по которым хотим искать, потом загружаем элемент, по которому хотим найти, сканируем одновременно все имеющиеся линии данных, в которых хранятся теги как в ассоциативной памяти, и ищем совпадения.



Позволяет быстро искать информацию в массивах данных без необходимости обходить его целиком. Используется в сетевых устройствах и иногда в реализациях нейровычислителей.

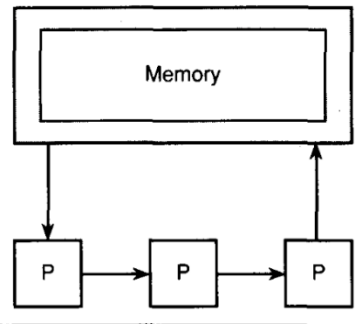
Синхронные, т.к. есть интерфейс доступа в память, в нее происходит чтение / запись и для каждой ячейки параллельно выполняются к-л операции, что дает синхронизацию.

4. Систолические архитектуры

   Используются в нейропроцессорах (Google TPU – тензорный процессор, заточенный под перемножение матриц)

   Когда есть много процессорных ядер, надо сделать так, чтобы они быстро считали большие задачи. Используются конвейерные мультипроцессоры:

   - данные ритмично поступают из памяти,
   - проходят через сеть процессоров,
   - и возвращаются в память.



   Каждый этап – не вырезанный кусок процессора, а полноценный процессор, выполняющий к-л действия.

   Синхронизация — через глобальные часы, такт вычислений.

   Процессоры объединены локальными связями, и каждый цикл процессоры выполняют короткие неизменные операции, по единому сигналу происходит продвижение данных по системе.

38. Классификация Дункана. Цели и задачи, классификация первого уровня. MIMD архитектуры (распределённая и разделяемая память). Примеры и принципы их работы.

Дункан_цели_задачи_классификация_1ур

MIMD архитектура:

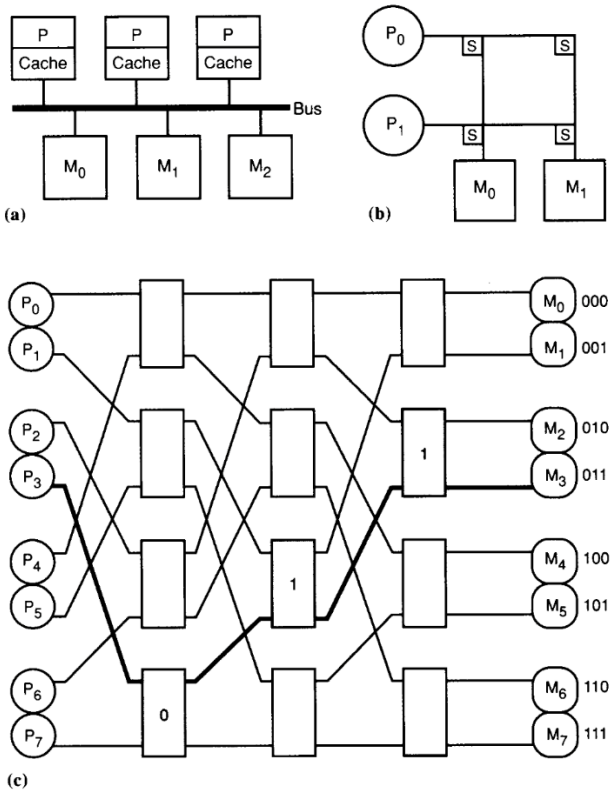
1. Разделяемая память

   Аналог в веб-приложениях: синхронизация через единую базу данных;

   Общая идея: единая память, через которую идет взаимодействие процессорных ядер, все обращаются в одно место и одинаково реагируют на окружающую действительность.

   Самый распространенный на текущий момент вид памяти, используется, например, в ПК, телефонах и тд.

   Есть набор ядер процессора, работающих против набора кэш линий, пока не дойдут до общей памяти.
   - Координация совместной работы через общую память.
   - Ключевая проблема: координация доступа, согласование кэшей.
   - Примеры аппаратных реализаций:
     - (a) шинное соединение (общая шина, через которую идет доступ к разным банкам памяти. Т.к. шина общая, то только 1 проц может обращаться к памяти, но каждый процессор работает со своим кэшом);
     - (b) перекрестная шина 2x2 (несколько банков памяти, несколько процессоров и кроссбар – решетка шин, позволяющая скоммутировать передачу от любого процессора к любой памяти, могут работать параллельно, быстро и независимо если нет пересечения по ресурсам);
     - (c) сеть на кристалле, запрос маршрутизируется от P к M.



- Широко распространены, так как привычны и прозрачны для программистов.

2. Распределенная память

   Аналог в веб-приложениях: микросервисная архитектура.

   Идея: у каждого процессора свои данные и своя память, и он сам независимо с ними работает.

   - Координация совместной работы через прямое взаимодействие процессоров (через выстраивание каналов связей, по которым один проц может оповестить другой о действии).
   - Преимущество: нет проблем с синхронизацией за счет разной памяти
   - Данные/сигналы явно передаются между процессорами (message passing).
   - Мотивация: обеспечить масштабируемость многопроцессорной архитектуры и доступ к локальным данным процессора.
   - Точки синхронизации между доступом к данным строго синхронизированы и управляемы, но это лежит на разработчике системы

   Примеры построения распределенной памяти:
   1. Кольцевая топология + [хорды]. Мало процессоров, передача данных не доминирует, плохо масштабируемо. Каждый процессор при необходимости отправить данные обозначает адрес пакета и передает по кольцу, рано или поздно сообщение или дойдет до адресата, или вернется отправителю
   2. Ячеистая / сетка. Соединены соседи + [диагональные] + [кольцевые]. Топологическое соответствие матричным алгоритмам, просто реализуется, т.к. естественное расположение ядер на чипе ~~(чипи чапа чапа)~~
   3. Дерево. Алгоритмы поиска и сортировки, обработка изображений, потоковые и редукционные машины.

4. Гиперкуб. Коммуникационный диаметр log2N

3 и 4 – более сложные реализации для 2, надо учитывать расположение в пространстве, но кратчайший путь между парами вершин меньше.

5. Реконфигурируемые. Фиксируем сеть конфигурацией (конфигурации трассы), а не маршрутом.

39. Классификация Дункана. Цели и задачи, классификация первого уровня. MIMD парадигмы (MIMD/SIMD, потоки данных, редукционные, wavefront). Примеры и принципы их работы.

Дункан_цели_задачи_классификация_1ур

Смещаем акцент восприятия с «как система должна быть реализована аппаратно» на «как она должна быть запрограммирована».

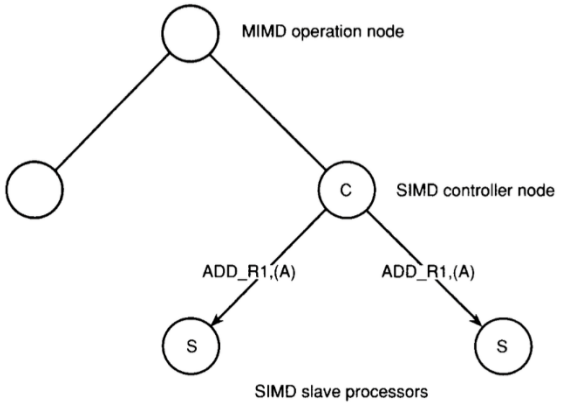
Имеется ввиду парадигма программирования или точнее — модель вычислений (MoC).

Варианты:

1. Гибридная MIMD/SIMD архитектура

   Тут я честно сам мало что понял из лекции, он базарил-базарил о чем-то отвлеченном, а потом заговорил про какой-то джаз, поэтому будет копипаст с презы + GPT. И этот человек требует от нас структурированного ответа без лишних отступлений и затягивания на экзамене, пздц.

   Диспетчер распределяет задачи по блокам, они выполняют задачи, когда есть результат диспетчер опять собирает все вместе и продолжается работа.



   - Гибридные архитектуры, где управление MIMD (малые задачи) осуществляется в стиле SIMD (крупные задачи).

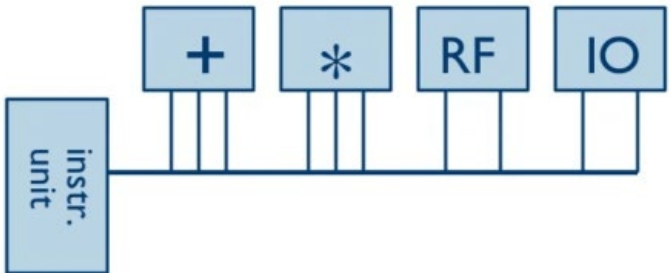
Высокоуровневое планирование (MIMD): на уровне системы задачи могут быть разделены между различными процессорами или ядрами. Например, один процессор может обрабатывать ввод данных, другой — выполнять численные расчеты, а третий — обрабатывать вывод данных.

Низкоуровневое параллелизм (SIMD): внутри каждого процессора или ядра, определенные части задачи, которые можно выполнить параллельно, передаются на SIMD-блоки. Например, если задача включает в себя умножение матриц, то SIMD-блоки могут выполнять операции умножения элементов матрицы параллельно.

- При представлении в виде дерева:
  - Ветви первого уровня (Процессоры или Ядра): Эти ветви представляют отдельные процессоры или ядра в системе. В MIMD архитектуре каждое ядро может выполнять свою собственную программу.
  - Ветви второго уровня (SIMD блоки внутри ядер): Эти ветви представляют SIMD блоки, встроенные в каждое ядро. SIMD блоки выполняют одну и ту же инструкцию над множеством данных параллельно.
- Аналогия: fork/join: запускается поток, форкаем несколько подпотоков, они решают свои задачи, в определенной точке времени они сливаются в один и продолжают работать

Пример: во всяких Intel Xeon, которые используют MIMD-архитектуру на уровне ядер, а каждое ядро может иметь SIMD-инструкции для параллельной обработки данных.

Плюс пример с Transport Triggered Architecture: при построении системы не управлять инструкциями как таковыми, а управлять пересылкой данных между вычислительными узлами

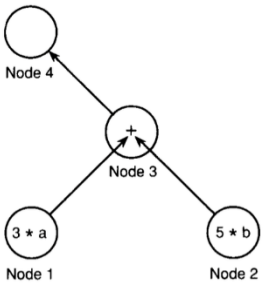


- Запуск вычислений инициируется пересылкой данных, управляемой диспетчером. При получении достаточного объема данных начинает выполнение этой операции
- Непосредственное выполнение инструкций — отсутствует.
- Нужно, когда надо обеспечить высокий уровень параллелизма и большое кол-во пересылок данных

2. Потоковая архитектура

Построена на принципе из примера, где толпа людей считает большие математические задачи.

Идея: есть граф вычислений, раскладываем его в процессорную матрицу и распространяя данные позволяет производить нужные вычисления.



Возможность выполнения и исполнения инструкций определяется исключительно наличием входных аргументов для инструкций. Порядок выполнения инструкций непредсказуем.

- Алгоритм представляется в виде графа вычислений.
- Отсутствует память (т.к. каждый блок независим, между ними идет адаптированная под имеющиеся вычисления коммуникационная среда, получается конвейерное исполнение) и канал доступа как "бутылочное горлышко".
- Проблема — передача данных между узлами.
- Аналогия: микросервисные архитектуры

Устройство в виде кольца:

- Распределённая память. Топология кольцо.
- По кольцу посылаются токены (метка + значение).
- Получатели пересылают токен дальше и сохраняют значение, если являются адресатом.
- Если задача разблокируется (все необходимые токены получены) — запускается работа.
- Результат работы отправляется токеном по кольцу.

Нечто похожее реализовано в Java Stream.

3. Редукционная архитектура

Работает наоборот относительно потоковой. Инструкции разрешаются к выполнению, когда их результаты требуются в качестве операндов для уже разрешенных инструкций.

Смотрим на формулу, видим неизвестные операнды, проваливаемся вниз по графу на определение этих операндов и получаем подстановку, дальше можем раскрывать граф независимо несколькими параллельными путями. Спустившись по графу в самый низ, дойдем до узлов с сохраненными в памяти значениями и начнем собирать граф назад.

- Разрабатывались для параллелизма и функционального программирования.
- Выполнение: выборка инструкций, замена инструкции на её результат в графе вычислений.
- Проблемы: ссылочная прозрачность, разворачивание
- Может работать в единой памяти и кол-во устройств, сканирующих эту память, может быть большим, а значит еще и с учетом кэширования будет оптимально работать

Пример использования: Haskell для реализации ленивых вычислений. Выражения в Haskell оцениваются только тогда, когда это необходимо (ленивая оценка), и результаты вычислений могут быть редуцированы по мере необходимости только когда нужен результат.

4. Wavefront

Каждый блок считает свои данные и продвигает результат дальше, как только получил его. У блока есть входной и выходной буфер, которые динамически синхронизируются.

Главный плюс: не надо формировать четкое статическое расписание для большой параллельной системы, все делается динамически.

В случае, если разные узлы имеют разную скорость выполнения, динамика Wavefront будет это компенсировать, в то время как в систолической архитектуре (которая чем-то похожа, только там единый тактовый сигнал) если тормозит один процессора – тормозит вся система. Плюс лучше масштабирование.

Пример использования: для решения СЛАУ и операций с матрицами *(ну раз напоминает систолическую, то явно юзкейсы те же)*

40. CGRA процессора. Особенности и место в индустрии. Пространственные (spatial) и временные (temporal) вычисления. Примеры и принципы их работы.

CGRA – Coarse-Grained Reconfigurable Architecture (крупно-зернистая реконфигурируемая архитектура)

Основывается на выводах из классификаций Флинна и Дункана:

- Огромное разнообразие подходов к построению процессора.
- Противоречие между пространственными и временными вычислениями (Spatial & Temporal Computation).
    - Spatial: если нужно посчитать 2 раза одно и то же, то увеличиваем вычислитель, поставив в параллель 2 вычислительных блока. Применяется, когда нужна скорость
    - Temporal: не надо ставить 2 блока, будем считать чуть дольше, но все в одном. Подходит для универсальности или снижения энергопотребления, т.к. нет большого кол-ва простаивающего оборудования
- Противоречие между универсальностью и эффективностью.

Выделяется ниша систем:

- Domain-Specific Flexibility ("Предметно-специально гибких") – не универсальны прям совсем, а адаптируемы под конкретный спектр задач.
- Сочетающих пространство и время.
- Управляемых конфигурацией и данными (Configuration / Data-driven execution).

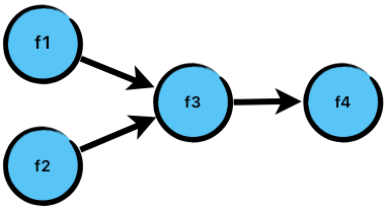
В этой нише и формируется CGRA архитектура, где мы не занимаемся разработкой цифровой схемы, а выстраиваем коммуникацию между блоками, чтобы за счет комбинации пространственных и временных вычислений эффективно решать конкретную задачу.

По сути, основными составляющими являются:

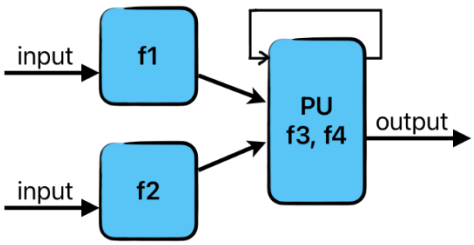
- Крупнозернистые блоки (такие как целиком АЛУ), которые могут выполнять арифметические и логические операции. Каждый блок может быть перенастроен для выполнения различных функций.
- Сеть, соединяющая вычислительные блоки, которая может быть программно перенастроена для изменения маршрутов данных между блоками. Это обеспечивает гибкость в организации вычислений и маршрутизации данных.

Принцип работы:

Из такого графа вычислений:



Можем получить такую конструкцию:



Где блоки f1 f2 для вычислений функций, которые могут быть посчитаны параллельно, а последовательные вычисления можно объединить в одной вычислительной структуре, посчитав их во временной распределенности, а не в пространственной.

CGRA архитектура определяет (снизу-вверх):

1. вычислительные блоки доступны и с какими интерфейсами (данные/управление);
2. коммуникацию между вычислительными блоками;
3. механика управления процессором;
4. механика планирования вычислительного процесса;
5. программы и программирование.

Крайне разнообразны, не имеют проработанной теории.

Являются актуальным направлением в развитии процессоров.