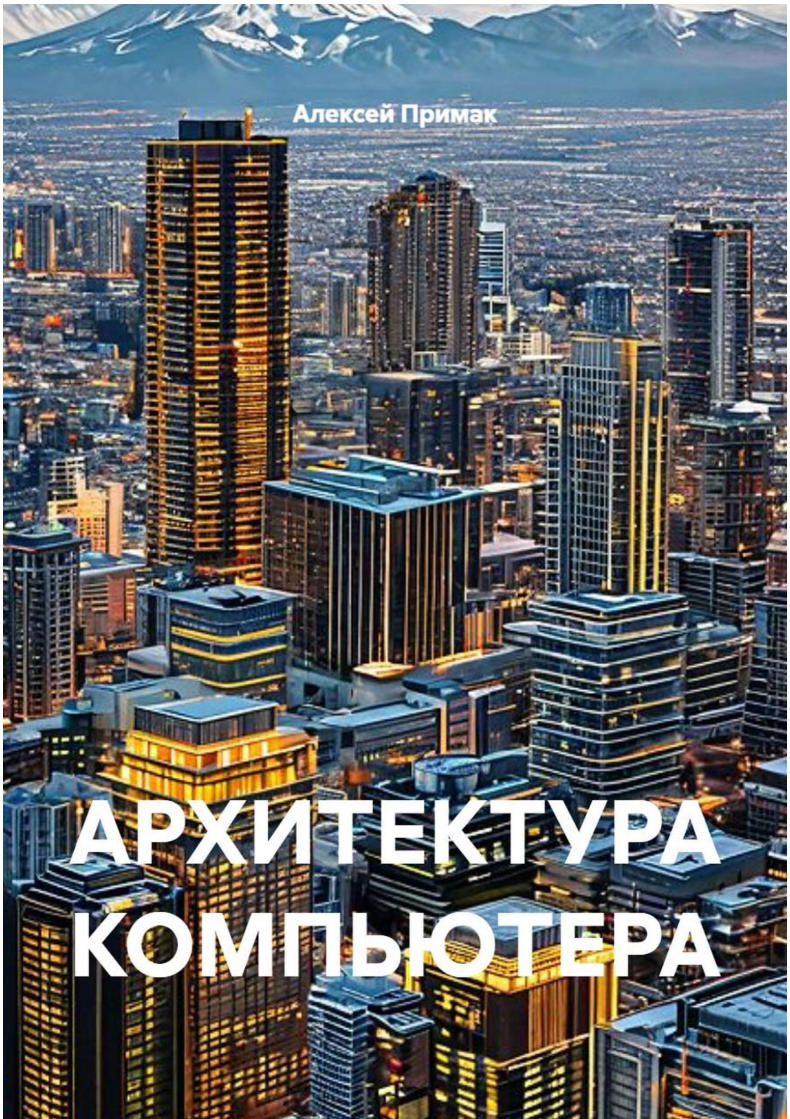
Алексей Примак
АРХИТЕКТУРА
КОМПЬЮТЕРА

Алексей Примак

АРХИТЕКТУРА КОМПЬЮТЕРА

https://litres.ru/74367928
SelfPub; 2026

Аннотация

В данной работе рассматриваются фундаментальные принципы построения и функционирования современных вычислительных систем. Основное внимание уделено концепции многоуровневой организации компьютера — от базового цифрового логического уровня до уровня операционной системы и языков высокого уровня. В работе анализируются ключевые компоненты ЭВМ: центральный процессор, структура памяти, подсистемы ввода-вывода, а также организация интерфейсов и системных шин. Рассматриваются различия между классической архитектурой фон Неймана и альтернативными решениями, включая современные параллельные и распределенные вычислительные системы.

Алексей Примак
АРХИТЕКТУРА КОМПЬЮТЕРА

ВВЕДЕНИЕ

Сущность цифрового компьютера и программы

Цифровой компьютер — это устройство, созданное для выполнения задач вместо человека, но действующее исключительно в строгих рамках заданных алгоритмов. Последовательность таких указаний, определяющая порядок решения конкретной задачи, и есть программа.

Базовые инструкции и машинный язык

Аппаратное обеспечение — электронные схемы любого компьютера — способно распознавать и исполнять лишь ограниченный набор элементарных команд. Перед запуском любая программа должна быть переведена именно в этот формат.

Как правило, базовые операции предельно просты и сводятся к четырем фундаментальным действиям:

Арифметические операции: например, сложение или вычитание числовых значений.

Логические операции: И, ИЛИ, НЕ, исключающее ИЛИ.

Перемещение данных: копирование или загрузка блока информации из одной области памяти в другую либо в регистр процессора.

Управление потоком выполнения: условные и безусловные переходы, включая проверку числа на равенство

нулю или сравнение значений.

Совокупность этих примитивных команд и формирует машинный язык — фундаментальный формат, на котором «говорит» аппаратное обеспечение.

Принципы формирования машинного языка

При проектировании новой архитектуры компьютера инженерам предстоит определить набор инструкций его машинного языка, или систему команд. Главная задача разработчиков — найти тонкий баланс между простотой отдельных команд и их выразительной силой.

Такой подход позволяет:

Точно соответствовать целевому назначению и требованиям к производительности устройства (классический пример — выбор между архитектурами RISC и CISC).

Снизить сложность, энергопотребление и итоговую стоимость электронных компонентов.

Структурная организация компьютерных систем

Из-за предельной простоты и низкоуровневой природы машинных языков прямая работа с ними для человека чрезвычайно сложна, утомительна и чревата ошибками. Исторически это наблюдение привело к революционному подходу: компьютеры начали проектировать как многоуровневую иерархию абстракций. В такой модели каждый последующий, более высокий уровень опирается на функциональность предыдущего, более низкого.

Эта методика, известная как структурная организация

компьютера, позволяет приручить изначальную сложность и создавать вычислительные системы последовательно и логично.

СТРУКТУРИРОВАННАЯ ОРГАНИЗАЦИЯ КОМПЬЮТЕРА

Пропасть между намерением и кодом: природа разрыва между человеком и машиной

Между интуитивно понятными когнитивными моделями человека и холодной логикой вычислительных систем лежит объективная пропасть. В науке о взаимодействии человека и компьютера (HCI) этот феномен метко назван «пропастью исполнения» (*gulf of execution*) и «пропастью оценки» (*gulf of evaluation*).

Это фундаментальное несоответствие проявляется в двух параллельных реальностях:

На уровне человека. Мы оперируем смыслами. Наши задачи высокоуровневы, семантичны и подчинены конкретной цели: «найти похожие изображения», «составить итоговый отчет» или «понять настроение текста». Мы мыслим результатами, а не процессами.

На уровне машины. Компьютер мыслит иначе. Его реальность состоит из низкоуровневых, детерминированных и строго синтаксических операций: машинного кода, сухих математических вычислений и безупречной, но лишенной житейского контекста формальной логики.

Хотя это несоответствие не является фатальным, оно таит в себе серьезную опасность. Без должного посредничества

пользователь сталкивается с резким ростом когнитивной нагрузки, а вероятность ошибок неизбежно возрастает.

Ключ к преодолению этой пропасти — в создании элегантных, многоуровневых слоев абстракции. От языков программирования и компиляторов до современных интерфейсов (UI/UX) и алгоритмов искусственного интеллекта — все эти технологии выполняют одну критически важную роль. Они выступают в роли безупречных переводчиков, транслируя наши смелые человеческие цели в точные и исполнимые машинные инструкции.

Именно там, где абстракция становится невидимой, рождается по-настоящему гармоничное взаимодействие человека и технологии.

Создание удобных языков программирования: трансляция и интерпретация

Для решения проблемы сложности машинных команд разрабатываются более удобные для человека наборы инструкций. Если базовые машинные инструкции образуют язык **L0**, то новый пользовательский набор формирует язык **L1**. Поскольку физический компьютер способен выполнять исключительно код на языке **L0**, программы на **L1** требуют специальных методов запуска.

Существует **два основных подхода** к выполнению кода высокого уровня:

Трансляция: каждая команда языка **L1** предварительно заменяется эквивалентной последовательностью команд языка **L0**. Итоговая программа полностью состоит из машинных инструкций, загружается в память и выполняется аппаратно. Исходный код на **L1** при этом удаляется.

Интерпретация: специальная программа на языке **L0** (интерпретатор) принимает код на **L1** как входные данные. Она последовательно считывает, декодирует и мгновенно выполняет эквивалентные машинные команды без создания промежуточного файла.

Ключевое отличие методов заключается в роли управляющей программы: при трансляции работой компьютера управляет сгенерированный машинный код, а при интерпретации контроль осуществляет сам интерпретатор, для которого исходный текст является просто данными. На практике эти методы часто комбинируются.

Эволюция языков программирования: трансляция, интерпретация и виртуальные машины

«Железо» понимает только свой родной язык — машинный код, который мы для удобства назовем **L0**. Но писать на нем человеку мучительно сложно. Поэтому инженеры придумали надстройку — язык **L1**, который интуитивно понятен людям. Вот только физический процессор язык L1 не

знает. Ему нужен «переводчик».

Как заставить машину выполнить код высокого уровня? Есть два классических пути.

Трансляция (компиляция) Мы берем весь текст на L1 и заранее переводим его в машинный код L0. На выходе получается готовая исполняемая программа, состоящая исключительно из аппаратных инструкций. Она загружается в память и выполняется напрямую. Исходный код на L1 при этом бережно сохраняется — он необходим программисту для правок, отладки и последующих сборок.

Интерпретация Здесь нет готового исполняемого файла. Специальная программа на языке L0 — *интерпретатор* — принимает код на L1 как входные данные. Она читает его строка за строкой, на лету декодирует в машинные команды и тут же выполняет. *(Справедливости ради, современные интерпретаторы часто используют промежуточные представления или байт-код в оперативной памяти для ускорения работы).*

Ключевое отличие кроется в роли управляющей программы. При трансляции работой компьютера управляет сгенерированный машинный код. При интерпретации бразды правления держит сам интерпретатор, для которого ваш исходный текст — лишь данные для обработки. На практике эти подходы часто гибко комбинируются (яркий пример — JIT-компиляция).

Концепция виртуальных машин

Чтобы не запутаться в бесконечных процессах «перевода» и «чтения», в информатике придумали элегантную абстракцию **— виртуальную машину**.

Представьте, что мы создали некое идеальное устройство, машинным языком которого является L1. Назовем его виртуальной машиной **M1** (в противовес реальному «железу» — машине **M0**, говорящей на L0).

Даже если собрать такое устройство из кремния слишком дорого или технически нецелесообразно, программисты все равно могут писать код, *будто* M1 существует физически. А запуск этих программ на реальном компьютере M0 возьмут на себя трансляторы или интерпретаторы.

Многоуровневая архитектура вычислительных систем

Прыгнуть от голых нулей и единиц сразу к сложным абстракциям невозможно: разрыв между уровнями будет слишком велик, а перевод — неэффективен. Поэтому эволюция идет постепенно.

Сначала рождается язык, близкий к машинному (например, ассемблер). Он все еще требует понимания архитекту-

ры, но уже оперирует понятными мнемониками. Чтобы сделать программирование еще более человекоориентированным, создаются следующие этажи — язык **L2** (и виртуальная машина **M2**), затем **L3** и так далее.

Так формируется **многоуровневая структура** (см. рис. 1-1), где:

Рис. 1-1. Многоуровневая машина.

Нижний уровень представляет собой суровый машин-

ный код и аппаратные инструкции.

Верхний уровень максимально абстрактен и удобен для человека, хотя и сложен для программной реализации.

В этом теоретическом контексте понятия «уровень» и «виртуальная машина» выступают синонимами. Между ними существует фундаментальная взаимосвязь: машина определяет набор доступных инструкций (язык), а язык, в свою очередь, задает архитектуру устройства, способного этот язык выполнить.

Теоретически, используя современные технологии, можно отлить на кремнии специализированный процессор, который аппаратно понимает семантику языков вроде Java или Python. Но на практике это тупиковый путь. Машина теряет свою универсальность, а экономика не терпит узкоспециализированного «железа» ради одного языка. Практичная архитектура должна быть не только осуществимой, но и выгодной.

Правила выполнения кода и роли разработчиков

В этой многоэтажной конструкции из *n* виртуальных машин есть одно железное правило: напрямую электронными схемами выполняется только код нулевого уровня (L0) и микрокод. Всё, что выше, требует обязательной трансляции или интерпретации вниз, к реальному «железу».

Это деление диктует и роли в мире разработчиков:

Прикладные программисты творят на верхних этажах. Они пишут код, не задумываясь о том, как именно работают нижележащие интерпретаторы и трансляторы. Структура машины надежно укрывает их от сложностей, гарантируя успешное выполнение программ.

Системные архитекторы и создатели языков живут на нижних уровнях. Чтобы спроектировать эффективную систему, они обязаны досконально понимать, как работают все нижележащие абстракции.

Понимание принципов построения многоуровневых машин и специфики каждого отдельного этажа — это фундамент. Без него невозможно по-настоящему глубоко постичь архитектуру современных вычислительных систем.

Анатомия вычислений: Шестиуровневая архитектура компьютерной системы

Современный компьютер — это не монолит, а сложная иерархия абстракций. В основе его устройства лежит классическая шестиуровневая модель, где каждый «этаж» надстраивается над предыдущим, скрывая его внутреннюю сложность и предлагая новые возможности. Фундаментом этой пирамиды выступает **Уровень 0** (цифровая логика), на котором базируется **Уровень 1** (микроархитектура), обеспечивающий выполнение команд **Уровня 2** (архитектуры на-

бора команд, или ISA).

Стоит отметить, что существует и «уровень устройств», лежащий еще ниже нулевой отметки. Он редко фигурирует на схемах абстракции, поскольку принадлежит царству электротехники и физики полупроводников. Здесь, в мире отдельных транзисторов — базовых физических примитивов, — правят законы физики твердого тела, выходящие за рамки классической компьютерной архитектуры.

(Рис. 1–2. Шестиуровневая модель компьютера. Под

каждым уровнем указан метод его поддержки и название соответствующей программы).

Уровень 0: Цифровая логика

Это нижняя граница классического изучения компьютерных систем. Ключевые объекты здесь — логические вентили (элементы). Несмотря на то что физически они состоят из аналоговых компонентов (транзисторов), работают они как безупречные цифровые устройства. Каждый вентиль принимает на вход сигналы «0» или «1» и вычисляет простейшую логическую функцию (например, «И» или «ИЛИ»).

Комбинируя вентили, инженеры создают 1-разрядную ячейку памяти — триггер, способный хранить бит информации. Группировка таких ячеек (по 16, 32 или 64) формирует регистры, а их объединение с элементами управления рождает базовый вычислительный механизм.

Уровень 1: Микроархитектура

Следующая ступень иерархии. Здесь располагаются сверхбыстрая внутренняя память (обычно от 8 до 32 регистров) и арифметико-логическое устройство (АЛУ), ответственное за базовые математические и логические операции. Регистры связаны с АЛУ, образуя конвейер передачи данных.

Его главная задача проста и элегантна: выбрать один или два регистра, выполнить над ними операцию в АЛУ и записать результат обратно. Управление этим конвейером может быть программным (через микрокод) или чисто аппаратным.

Сегодня, в эпоху доминирования аппаратного управления, термин «уровень микропрограммирования» уступил место более точному — «уровень микроархитектуры».

Уровень 2: Архитектура набора команд (ISA)

Instruction Set Architecture (ISA) — это тот самый уровень, который описывается в официальной документации производителей (например, в «Справочнике по машинному языку»). Производители фиксируют здесь именно ISA, не вдаваясь в подробности нижележащих слоев. В этих руководствах задокументированы инструкции, которые интерпретируются либо микропрограммой, либо аппаратными схемами. Машины с разной ISA говорят на разных машинных языках и требуют собственных справочников.

Уровень 3: Машинный уровень операционной системы (Гибридный уровень)

Уникальный гибридный слой. Большинство его инструкций заимствованы с уровня ISA, но к ним добавляются принципиально новые возможности: расширенный набор команд, усовершенствованная организация памяти и поддержка многозадачности (одновременное выполнение нескольких программ).

Различия между архитектурами этого уровня куда существеннее, чем между уровнями 1 и 2. Новые функции реализуются через интерпретатор, роль которого выполняет операционная система. Инструкции, совпадающие с уровнем 2, выполняются напрямую (аппаратно или через микропро-

грамму), а новые обрабатываются уже самой ОС.

Рубеж прикладного программирования

Между уровнями 3 и 4 проходит принципиальная водораздельная черта:

Три нижних уровня (0, 1, 2) скрыты от рядового разработчика. Их цель — обеспечить работу интерпретаторов и трансляторов, поддерживающих этажи выше. Это вотчина системных программистов.

Уровни 4 и 5 созданы для прикладных программистов, решающих конкретные практические задачи.

Если уровни 2 и 3 традиционно опираются на *интерпретацию*, то уровни 4 и 5 поддерживаются методом *трансляции* (компиляции).

Языковая эволюция: от нулей и единиц к человеческому слову

Языки уровней 0, 1 и 2 сугубо машинные: они состоят из последовательностей нулей и единиц, идеальных для процессора, но нечитаемых для человека. Начиная с 4-го уровня, в игру вступают слова и аббревиатуры, понятные разработчику.

Уровень 4: Язык ассемблера. Символьное отражение машинного языка. Он позволяет описывать программы для уровней ISA и ОС в удобочитаемом виде. Специальная программа-транслятор (ассемблер) преобразует мнемонический код в машинный, после чего тот передается на выполнение соответствующей машине.

Уровень 5: Языки высокого уровня. Инструменты для решения прикладных задач (C, C++, Java, Python, PHP и др.). Программы на них обычно транслируются на уровень 3 или 4 с помощью компиляторов. Возможна и интерпретация: например, Java сначала компилируется в промежуточный байт-код (напоминающий ISA), который затем исполняется виртуальной машиной.

Архитектура и организация: две стороны одной медали

Компьютер — это иерархическая последовательность уровней, где каждый является самостоятельной абстракцией со своими уникальными объектами и операциями, опирающейся на предыдущий. В профессиональной и академической среде строго разграничивают два понятия:

Компьютерная архитектура охватывает лишь те аспекты системы, которые видны программисту: совокупность типов данных, операций, функций, регистров и методов адресации. Это ответ на вопрос: *«Что система делает?»* **Организация компьютера** (или его реализация) относится к физическим деталям, прозрачным для разработчика: тактовой частоте, физической технологии создания памяти, типам используемых логических вентилей. Это ответ на вопрос: *«Как она делает это физически?»*

Введение в многоуровневые машины: как стирается граница между «железом» и кодом

Введение в многоуровневые машины: как стирается граница между «железом» и кодоIBM 709м

Чтобы по-настоящему понять архитектуру многоуровневых вычислительных машин, необходимо проследить их историческую эволюцию и трансформацию количества, а главное — природы вычислительных уровней.

Программный код, написанный на машинном языке (условный уровень 1)*, исполняется электронными схемами (уровень 0) напрямую, без посредничества трансляторов или интерпретаторов. Эти схемы вместе с устройствами памяти и ввода-вывода формируют аппаратное обеспечение. Это не абстрактные алгоритмы, а осязаемая физика: интегральные микросхемы, печатные платы, кабели, блоки питания и периферия.

Программное обеспечение, напротив, состоит из алгоритмов (детальных инструкций) и их цифровых воплощений — программ. Хотя код физически хранится на жестких дисках или оптических носителях, сама суть ПО кроется в наборе команд, а не в материале, который их хранит.

В ранних вычислительных машинах рубеж между «железом» и софтом был проведен с хирургической точностью. Однако с развитием технологий, из-за постоянного добавле-

ния, удаления или слияния уровней, эта граница значительно размылась (Вахид, 2003)**, и сегодня их зачастую трудно различить. Ключевой концепцией здесь выступает **логическая эквивалентность** аппаратного и программного обеспечения.

Любая функция, реализуемая программно, может быть встроена непосредственно в «железо», как только она будет досконально изучена. По меткому выражению Карен Панетты Ленц: *«Аппаратное обеспечение — это просто застывшее программное обеспечение»*. Справедливо и обратное: любая аппаратная команда может быть эмулирована программно. Выбор в пользу того или иного подхода зависит от стоимости, быстродействия, надежности и ожидаемой частоты изменений. Строгих догм не существует: эти решения постоянно корректируются под влиянием технологической экономики и рыночного спроса.

Рождение микропрограммирования

Первые цифровые компьютеры 1940-х годов обладали лишь двумя уровнями: уровнем ISA (системы команд), на котором велось программирование, и уровнем цифровой логики, исполняющим эти программы. Схемы цифровой логики отличались высокой сложностью, трудоемкостью в проектировании и низкой надежностью.

В 1951 году исследователь Кембриджского университета Морис Уилкс предложил концепцию трехуровневого компьютера. Его целью было радикальное упрощение аппарат-

ной части и сокращение количества ненадежных вакуумных ламп (Wilkes, 1951). В этой архитектуре предусматривался встроенный неизменяемый интерпретатор — микропрограмма, которая выполняла программы уровня ISA путем их пошаговой интерпретации.

Поскольку аппаратному обеспечению теперь требовалось исполнять лишь микропрограммы с ограниченным набором команд (а не полный арсенал уровня ISA), объем необходимых электронных схем сократился. В эпоху вакуумных ламп такое упрощение напрямую вело к уменьшению их числа и, как следствие, к повышению общей надежности системы (снижению количества ежедневных сбоев).

В 1950-х годах было создано несколько подобных трехуровневых устройств, а в 1960-х их число возросло. К 1970 году концепция интерпретации уровня ISA посредством микропрограммы, а не прямой электронной обработки, стала доминирующей и применялась во всех основных вычислительных машинах того периода.

Зарождение операционных систем

На ранних этапах компьютеры были «открытыми», что требовало от программиста личного, почти физического управления машиной. Возле каждого устройства велся бумажный журнал регистрации. Специалист записывался на конкретный слот (например, с 3 до 5 утра, когда в машинном зале царила тишина), брал колоду 80-колоночных перфокарт и остро заточенный карандаш, вежливо вытеснял предыду-

щего пользователя и садился за консоль.

Процесс запуска программы на языке ФОРТРАН напоминал сложный ритуал:

Загрузка из библиотеки большой зеленой колоды карт с надписью *FORTRAN compiler* в считыватель и нажатие кнопки «ПУСК».

Загрузка собственной программы на ФОРТРАНе и нажатие кнопки «ПРОДОЛЖИТЬ» для ее считывания.

Повторное считывание программы, так как многим компиляторам требовалось два или более прохода по исходным данным.

Ожидание завершения трансляции. Этот этап вызывал нервное напряжение: при обнаружении ошибки процесс приходилось начинать заново. При успешном исходе компилятор записывал переведенную программу на машинном языке на новые перфокарты.

Загрузка полученной программы вместе с колодой библиотечных подпрограмм.

Запуск выполнения. Часто программа аварийно останавливалась на середине. Программист вручную манипулировал переключателями и изучал индикаторы консоли для диагностики. При удаче ошибка исправлялась, и цикл повторялся. При неудаче создавалась распечатка содержимого памяти (дамп ядра) для домашнего анализа.

Эта рутинная процедура заставляла специалистов глубоко изучать устройство машины и методы устранения частых

поломок. Однако машина часто простаивала из-за физической переноски карт или длительного поиска причин сбоев.

Автоматизация и первые операционные системы

Около 1960 года для сокращения потерь времени была предпринята попытка автоматизировать работу оператора. В памяти компьютера стала постоянно находиться специальная программа — операционная система (резидентный монитор). Программист добавлял к своей программе управляющие карты, которые считывались и исполнялись этой системой автоматически.

Характерным примером служит система FMS (FORTRAN Monitor System) для IBM 709. Процесс выглядел так:

Операционная система считывала карту задания (со знаком *), используя ее данные для учета (этот знак отличал управляющие карты от программных и данных).

Затем считывалась карта *FORTRAN, дающая команду загрузить компилятор ФОРТРАН с магнитной ленты.

Компилятор обрабатывал исходный код и возвращал управление операционной системе.

Система считывала карту *DATA, предписывающую выполнить скомпилированную программу, используя последующие карты в качестве входных данных.

Рисунок 1-3. Пример задания для операционной системы FMS.

Помимо автоматизации труда оператора, это стало первым шагом к созданию новой виртуальной машины. Карту *FORTRAN можно трактовать как виртуальную команду «скомпилировать программу», а карту *DATA — как виртуальную команду «выполнить». Этот скромный уровень, состоявший всего из двух инструкций, заложил фундамент для дальнейшего развития.

Эволюция операционных систем

В последующие годы операционные системы усложнялись. На уровень ISA добавлялись новые инструкции и функции, формируя видимость отдельного вычислительного уровня. Некоторые из этих команд дублировали инструкции ISA, но другие (особенно операции ввода-вывода) кардинально отличались. Изначально их называли макросами операционной системы или вызовами супервизора; современный термин для них — **системный вызов**.

Первые ОС работали в режиме пакетной обработки: они считывали колоды карт и выводили результаты на линейный принтер. Ожидание результата могло занимать несколько часов, что сильно затрудняло разработку ПО.

В начале 1960-х годов исследователи из Дартмутского колледжа и других организаций создали системы разделения времени. Они позволяли нескольким программистам взаимодействовать с центральным компьютером напрямую через удаленные терминалы, подключенные по телефонным линиям. Это обеспечивало почти мгновенный отклик системы, независимо от местоположения пользователя.

В контексте данной темы особый интерес представляют те компоненты ОС, которые интерпретируют функции уровня 3, отсутствующие на уровне ISA, а не механизмы разделения времени как таковые. Следует помнить, что современные ОС выполняют гораздо более широкий спектр задач, чем просто интерпретация добавленных функций.

Перенос функциональности в микрокод

Когда микропрограммирование стало стандартом (к 1970 году), разработчики осознали возможность добавления новых инструкций путем простого расширения микропрограммы. Фактически, это позволяло создавать «аппаратное обеспечение» (новые машинные команды) программными методами.

Это открытие спровоцировало взрывной рост наборов машинных команд в условиях конкуренции производителей. Многие новые инструкции не были строго необходимыми, так как их функции можно было реализовать комбинацией существующих команд. Однако выделенные инструкции работали быстрее. Классический пример: инструкция INC (инкремент), добавляющая 1 к числу, выполнялась быстрее, чем универсальная инструкция ADD, поэтому ее внедрили отдельно.

Аналогичным образом в микропрограмму были интегрированы следующие возможности:

инструкции для умножения и деления целых чисел;

инструкции для арифметики с плавающей запятой;

инструкции вызова и возврата из процедур;

инструкции для ускорения выполнения циклов;

инструкции для обработки символьных строк.

Оценив простоту добавления новых команд, разработчики начали внедрять в микропрограммы и более сложные функции:

средства ускорения вычислений с массивами (индексация и косвенная адресация);

функции перемещения программ в памяти после их запуска;

системы прерываний, сигнализирующие о завершении операций ввода-вывода;

механизмы переключения процессов, позволяющие приостановить одну программу и запустить другую минимальным числом команд;

специализированные инструкции для обработки аудио, графики и мультимедиа.

С течением времени добавлялось множество других приспособлений, преимущественно направленных на ускорение специфических операций.

Отказ от микропрограммирования и современные тенденции

Микропрограммы достигли пика популярности в 1960-х и 1970-х годах. Однако по мере их разрастания скорость выполнения неумолимо снижалась. В определенный момент исследователи пришли к выводу, что отказ от микрокода, сокращение набора команд и обеспечение прямого аппаратного управления оставшимися командами способны значительно ускорить работу машин. В некотором смысле проектирование компьютеров совершило полный круг, вернувшись к состоянию до изобретения микропрограммирования Уилксом (философия RISC).

Тем не менее, эволюция продолжается. Современные процессоры по-прежнему используют микропрограммирование, но иначе: для преобразования сложных внешних инструкций во внутренний микрокод (микрооперации), который затем исполняется на оптимизированном, сверхбыстром аппаратном обеспечении.

Главная цель этого исторического экскурса — продемонстрировать, что граница между аппаратным и программным обеспечением произвольна и постоянно меняется. То, что сегодня является программным обеспечением, завтра может стать аппаратным, и наоборот. Границы между различными уровнями архитектуры также крайне изменчивы. Для программиста, работающего на уровне ISA, конкретный способ физической реализации инструкции не имеет значения (за исключением ее быстродействия). Он может использовать команду умножения, не задумываясь, является ли она аппаратной или программной.

Конец ознакомительного фрагмента.