Глава 15. Дополнительные особенности архитектуры ЭВМ

Никакую проблему нельзя решить на том же уровне, на котором она возникла.
Альберт Эйнштейн

В этой главе рассматриваются некоторые особенности конструирования компьютеров, которые, могут оказаться полезными для лучшего понимания основ архитектуры современных ЭВМ.

15.1. Дискретные и аналоговые вычислительные машины

В физике часто случалось, что существенный успех был достигнут проведением последовательной аналогии между несвязанными по виду явлениями.
Альберт Эйнштейн

При рассмотрении основных свойств алгоритмов отмечалось такое важное свойство алгоритма, которое называлось *дискретностью* (иногда структурностью). Это свойство означало, что любой достаточно сложный алгоритм состоит из этапов или шагов, причем каждый шаг, если он достаточно сложный, в свою очередь, тоже является алгоритмом и может разбиваться на ещё более мелкие шаги.[1] Это же свойство структурности алгоритма позволяло строить из одних алгоритмов другие по определённым правилам, например, путем составления суперпозиции (последовательного выполнения) двух алгоритмов. Здесь можно вспомнить и хорошо известный принцип структурного программирования, в соответствие с которым вся программа строится в виде последовательной декомпозиции на линейные участки, условные операторы и циклы.

Системы состоят из подсистем, подсистемы – из под подсистем и так до бесконечности – именно поэтому мы проектируем снизу вверх.
Алан Перлис,
первый лауреат премии Тьюринга

Как известно, если некоторый алгоритм *применим* к определенным входным данным, то он получает (вводит) эти данные, обрабатывает их за конечное число шагов и останавливается с выдачей результата.[2] Содержание наименьшего шага алгоритма определяется исполнителем этого алгоритма, например, для Паскаль-машины это оператор языка Паскаль, а для компьютера – машинная команда. Разумеется, как уже говорилось, можно спуститься на следующий уровень микроархитектуры компьютера и произвести разложение команды на микрооперации (мопы). Далее можно рассмотреть реализацию каждого мопа в виде схемы, составленной из элементарных логических элементов – вентилей. Как уже говорилось, каждый вентиль срабатывает, когда на него приходит специальный тактовый импульс – можно считать, что на *инженерном* уровне ви́дения архитектуры это и есть логический предел дробления алгоритма на всё более мелкие шаги для исполнителя-компьютера. На *физическом* уровне элементарным шагом, по-видимому, является срабатывание (открытие или закрытие) затвора транзистора после приложения к нему электрического напряжения. Дальше по уровням дискретности двигаться уже некуда, однако, остается ещё квантовый уровень…

Компьютеры, построенные по рассмотренному выше принципу, так и называются – *дискретными* (или цифровыми) вычислительными машинами. Сейчас постараемся понять, что это не единственно возможный принцип построения устройства автоматической обработки данных (компьютера).

Сначала следует рассмотреть, как решает задачи обработки данных человек. Оказывается, что он может делать это двумя принципиально различными путями. Один путь называется логической обра-

---

[1] Вообще говоря, это же касается и обрабатываемых алгоритмом данных, более мелкие шаги алгоритма, как правило, работают и с более мелкими порциями данных.

[2] Конечно, более строго надо было бы сказать, что это делает *исполнитель* алгоритма, но в компьютерной литературе обычно так говорить не принято.

боткой данных, например, рассмотрим, как сыщик узнаёт преступника по его словесному портрету. «Подбородок квадратный, глаза узкие, нос прямой, на кисти левой руки небольшой шрам, на правой кисти татуировка якоря. При ходьбе припадает на правую ногу, при общении щурит левый глаз, говорит без акцента и т.д.» Здесь явно проглядывает уже знакомое нам разбиение алгоритма узнавания на отдельные шаги. За такой процесс обработки данных у человека отвечает так называемое *логическое мышление*.

В то же время, это не единственный способ узнать человека. Так, мы «с одного взгляда» или, как говорят, интуитивно, узнаём в толпе своего приятеля, и при этом совсем не разбиваем для себя это узнавание ни на какие отдельные шаги. Такой процесс обработки информации человеком называется *интуитивным мышлением*, этот процесс для человека на осознаваемом уровне восприятия не обладает свойством дискретности, хотя, конечно, и занимает некоторый отрезок времени. Как этот процесс проходит на интуитивном уровне, или, как часто говорят, в подсознании, пока точно неизвестно, однако есть основания предполагать, что и на подсознательном уровне механизм обработки данных у человека не является дискретным. Любопытно, что логическое мышление на порядок энергозатратнее, чем интуитивное, поэтому «на автопилоте» человек использует интуитивное мышление (сначала делаем, потом соображаем).

Как считается, за логическое и интуитивное мышление отвечают две разные половины (полушария) человеческого мозга – левое и правое. Не стоит из праздного любопытства запоминать, за какое мышление отвечает левое, а за какое правое полушарие человеческого мозга, так как у людей-левшей всё обычно будет наоборот. Заметим также, что не стоит понимать всё это слишком буквально, на самом деле оба полушария мозга обрабатывают данные совместно, обмениваясь при этом между собой информацией. Можно говорить только об определенной *специализации* полушарий на логическую и интуитивную обработку данных.

Как установлено, разделение мышления на логическое и интуитивное является следствием обучения человека читать и писать. Дикие племена в Африке и в джунглях Амазонки таким двойственным мышлением не обладают.

В этой книге изучается только архитектура дискретных (цифровых) вычислительных машин, можно сказать, что они обрабатывают входные данные по принципу, очень похожему на логическое мышление человека. Резонно задать вопрос, а можно ли построить устройство автоматической обработки данных (компьютер), которое работало бы по принципу, похожему на интуитивное мышление человека? [1]

Да, такие компьютеры возможны, в отличие от уже знакомых нам *дискретных* ЭВМ, они называются *аналоговыми* (или непрерывными) вычислительными машинами (АВМ). Более того, в начале развития вычислительной техники аналоговые ЭВМ были весьма распространены. Разберем вкратце, как работали такие ЭВМ (их архитектура будет рассмотрена только на концептуальном уровне).

Одним из первых известных сейчас аналоговых устройств, по-видимому, является так называемый *антикитерский механизм*, обнаруженный в 1900 году на древнегреческом судне, затонувшем в Средиземном море в I веке до н.э. Он был изготовлен около 200 года до н.э. и использовался для моделирования движения небесных тел (планет, солнца и Луны). Механизм состоял из множества весьма точно изготовленных шестерёнок размером с монету.

Одним из принципов работы аналоговых ЭВМ является аналоговое представление данных. Для нас привычно представлять данные для компьютера в виде чисел, состоящих из определенного количества цифр (неважно, в какой системе счисления), такое представление данных называется цифровым или *дискретным*. Теперь необходимо понять, что данные можно представить и в непрерывной форме, например, в виде определенного напряжения на конденсаторе, расположением стрелок на часах или положением подвижной части логарифмической линейки (кто не знает, это такой удивительный деревянный калькулятор раньше был 🙂). Так вот, аналоговые ЭВМ обрабатывали данные, представленные в такой непрерывной форме (обычно в виде электрических напряжений и токов).

Вообще говоря, с точки зрения современной физики, непрерывных величин вообще не существует, так как есть наименьшие кванты времени, пространства, заряда и всех других величин. Например,

---

[1] Рассматриваемый вопрос не имеет прямого отношения к так называемым системам искусственного интеллекта, т.е. комплексам *программ* для обычного (цифрового) компьютера, имитирующим процесс обработки данных человеком в отдельных предметных областях.

минимальная, так называемая Планковская длина равна $1.6 \cdot 10^{-35}$ метра. Из-за малости этих величин на это можно не обращать внимания, тем более, что в последние годы набирает популярность и противоположная точка зрения о непрерывности мира (точнее, о бесконечном количестве его основных характеристик).

Процесс решения задачи аналоговой ЭВМ заключается в том, что входные данные подаются на входное устройство такой ЭВМ в виде набора электрических напряжений. Затем, через определённый (не очень большой) промежуток времени выходные данные (результаты решения задачи) появлялись на устройстве вывода, тоже в виде набора электрических напряжений. Более всего такой ответ походит на решение дифференциальных уравнений, представленное в виде графиков, а не числовых таблиц. Разумеется, выходные данные могли потом преобразовываться устройством вывода в числовой вид, печататься и выдаваться пользователю. Аналоговые ЭВМ могли очень быстро выполнять операций дифференцирования и интегрирования функций.

Аналоговые ЭВМ не поддаются программированию в привычном для нас виде, так как они преобразуют входные данные в выходные «за один шаг» своей работы. Можно сказать, что аналоговая ЭВМ *настраивается* на конкретную задачу, которую необходимо решить. Чтобы понять, в чём заключается такая настройка на решаемую задачу, посмотрим более пристально на процесс программирования на машинно-ориентированном языке для привычных для нас дискретных ЭВМ.

Составляя алгоритм решения некоторой задачи на машинно-ориентированном языке, программист обычно записывает этот алгоритм в виде последовательности команд для конкретной ЭВМ. Можно сказать, что алгоритм решения задачи *отображается* на язык машины, подстраиваясь к особенностям этого языка и архитектуры самого компьютера. Такое отображение, как известно, является достаточно трудным делом, так как язык машины очень далёк от того языка, на котором прикладной программист мыслит в рамках своей задачи. Например, известно, что очень многие физические задачи описываются на языке дифференциальных и интегральных уравнений, который является «естественным» языком для описания таких задач. Те, кто уже сталкивались с решением задач вычислительного характера, должны представлять, как далёк язык машины (да и используемый при программировании язык высокого уровня) от задачи вычисления, скажем, определённого интеграла.

В отличие от дискретных ЭВМ, аналоговые компьютеры реализуют другой **принцип** решения задач. Вместо того чтобы отображать задачу на язык машины (т.е. «приспосабливать» задачу к машине), как это делается на дискретных ЭВМ, аналоговые ЭВМ *изменяют свою структуру*, чтобы самим соответствовать решаемой задаче! Таким образом, аналоговые ЭВМ имеют достаточно «гибкую» архитектуру, узлы такой ЭВМ могут по-разному соединяться друг с другом, «настраиваясь» на решаемую задачу. Заметим, что сам принцип «настройки» исполнителя алгоритма под конкретную задачу Вам уже встречался. Например, для машины Тьюринга правильнее говорить не «написать программу для машины Тьюринга, решающую данную задачу», а «*построить* машину Тьюринга, решающую данную задачу». В такой машине Тьюринга будет «построено» нужное для конкретной задачи число состояний, она будет распознавать нужный набор символов и т.д. Таким образом можно сказать, что *обобщенная* машина Тьюринга настраивается для выполнения конкретного алгоритма.

Некоторые из дискретных компьютеров также могут в определённых (весьма небольших) пределах изменять связи между своими узлами, подстраиваясь под решаемую задачу. Компьютеры с такой архитектурой называются *транспьютерами*, они не получили широкого распространения. Кроме того, современные супер-ЭВМ тоже могут в определённых пределах менять связи между своими вычислительными узлами, настраиваясь на конкретную задачу.

В некотором смысле аналоговая ЭВМ похожа на детские конструкторы, с которым некоторым из Вас приходилось иметь дело в юные годы. Как Вы знаете, по-разному соединяя детали такого конструктора, можно собирать **модели** самых разных предметов (подъёмный кран, качели, стул и стол и т.д.). Точно так же аналоговая ЭВМ, изменяя свою структуру, строит (электрическую) **модель** решаемой задачи. На практике настройка аналоговой ЭВМ на решаемую задачу состоит в установлении с помощью проводов многочисленных электрических соединений между узлами такого компьютера, и изменению электрических характеристик (сопротивлений, ёмкостей и индуктивностей) этих узлов с помощью соответствующих элементов управления (ручек, кнопок, ползунков, штырьков и т.д.).

Сначала аналоговые ЭВМ были электромеханическими (электродинамическими), обрабатываемые величины представлялись, например, на крутящихся металлических стержнях. Первая такая аналоговая ЭВМ в России была построена академиком Алексеем Николаевичем Крыловым ещё в 1912 году, эта машина умела решать обыкновенные дифференциальные уравнения 4-го порядка,


Академик А.Н. Крылов (1863-1945)

она использовалась при расчётах конструкции кораблей. В средине прошлого века такие машины часто назывались *дифференциальными анализаторами*. Далее стали строить и чисто электрические АВМ (на постоянном токе).


Дифференциальный анализатор, 1938

Здесь необходимо сказать, что использование в качестве «рабочего тела» в аналоговых ЭВМ именно электричества не является принципиальным. Например, в истории вычислительной техники известна реализация аналоговой ЭВМ, работающей на обыкновенной воде. В этом случае конструктивными элементами являются водяные резервуары с соединяющими их трубками разного сечения, а текущая вода моделирует решаемую задачу путем изменения скорости своего течения и давления в разных узлах такой аналоговой ЭВМ (это так называемые переменные гидравлические сопротивления).


Гидроинтегратор В. Лукьянова. Водяная аналоговая ЭВМ,1936 г

Например, на фото слева показана такая аналоговая ЭВМ – гидроинтегратор В.С. Лукьянова для решения (интегрирования) систем дифференциальных уравнений с *частными* производными (1936 год). В то время это была *единственная* ЭВМ в мире для решения уравнений в частных производных. Было организовано серийное производство этих машин, они экспортировались за рубеж (в Чехословакию, Польшу, Болгарию и Китай).

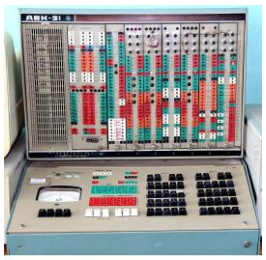
Вместо воды в аналоговых ЭВМ можно с успехом использовать газ и пар (это будут пневматическая и паровая ЭВМ 🙂). Это не шутка или безудержная фантазия, уже есть проект пневматической ЭВМ, работающей на так называемом эффекте Коанда. Надо заметить, что иногда надо иметь ЭВМ, работающую в экстремальных условиях, например, на поверхности Венеры при сверхвысоких температурах и давлениях, где «не выживет» ни один обычный компьютер. Их часто применяли во взрывоопасных средах, на некоторых оружейных и химических заводах (куда в принципе не подводится электричество!), а также на предприятиях с высоким радиационным фоном. Можно вспомнить и гиперзвуковые ракеты, которыми управляют бортовые компьютеры, когда на обшивке ракеты температура больше 1000 градусов.

В качестве «рабочего тела» аналоговой ЭВМ можно также использовать, например, звуковые волны, в этом случае конструктивными элементами являются различные плоскости и решетки, отражающие, поглощающие, поляризующие и модулирующие эти звуковые волны.

Исследования биофизиков позволяют предположить, что такого рода аналоговая звуковая «нано-ЭВМ» располагается внутри каждого аксона (отростка нейрона в нервной ткани человека и животных). Основой этого аналогового компьютера является особая трёхмерная кальциевая решётка, работающая на звуковых волнах очень высокой частоты. Эти звуковые волны генерируются ударами ионов о стенки нейрона и саму решётку, такие волны имеют частоты до 100 Ггц, что на два порядка превышает тактовую частоту процессоров современных ЭВМ. Предполагается, что это позволяет обрабатывать и изменять сигналы, принимаемые и передаваемые по нейронной сети организма.

Далее, стоит заметить, что в детских конструкторах строятся в основном *статические* модели, в то время как аналоговая ЭВМ строит *динамическую* модель некоторого объекта или явления. В качестве примера динамической модели из детского конструктора рассмотрим модель ветряной мельницы, которая *на самом деле* может крутиться под действием ветра, да ещё и изменяет наклон своих лопастей в зависимости от скорости ветра. Ясно, что изучение такой модели в принципе может помочь нам в конструировании *настоящих* ветряных мельниц, позволяющих эффективно использовать силу ветра.

Продолжая аналогию с детским конструктором можно заметить, что с его помощью можно собирать далеко не всякие модели, а только из определённого набора (по научному это называется – из определенной *предметной области*). Здесь можно вспомнить, что конструкторы бывают механические, электрические, оптические, радио-конструкторы и т.д. Отсюда можно сделать вывод, что всякая аналоговая ЭВМ тоже *специализированная*, предназначенная для решения задач в достаточно узкой предметной области. В то же время надо заметить, что почти все *цифровые* ЭВМ являются *универ-*


Малая АВМ АВК-31
1980 года выпуска

сальными, их можно с почти одинаковым успехом использовать во всех областях, это использование ограничивается только аппаратными ресурсами компьютера, которые непрерывно увеличиваются с развитием вычислительной техники. В качестве примера слева показана отечественная малая АВМ АВК-31 1980 года выпуска, она решала системы нелинейных дифференциальный уравнений 6-го порядка, эффективной быстродействие (в пересчёте на цифровые ЭВМ) составляло 100 млн. оп/сек. А АВМ АВК-32 решала уже уравнения 20-го порядка со скоростью 320 млн. оп/сек.

Разумеется, существуют и так называемые *специализированные* ЭВМ, например, предназначенные для управления различными устройствами (стиральными машинами, кухонными комбайнами и т.д.). Необходимо, однако, заметить, что специализация таких ЭВМ заключается только в том, что они, как правило, выполняют только одну записанную в их памяти программу, и не снабжаются аппаратными средствами для развитого интерфейса с пользователями. Другими словами, хранимую в памяти специализированных ЭВМ программу невозможно (или очень трудно) сменить, и у них, как правило, нет таких «программистских» устройств для ввода данных, как мышь, дисковод и т.п.

Аналоговая ЭВМ реализует достаточно большую скорость обработки информации, т.к. у неё отсутствует пошаговый алгоритм работы. Например, при решении системы дифференциальных уравнений, эквивалентное быстродействие аналоговой ЭВМ составляет десятки мегафлопс. Кстати, это же верно и для «аналогового компьютера» из соответствующего полушария человеческого мозга. Например, когда человек бегом спускается с холма, именно это полушарие «руководит» движениями человека, в частности, решая, как ему переставлять ноги. Попытка решить такую же задачу *в реальном времени* с помощью программы для обычного (цифрового) компьютера показывает, что с этой задачей с трудом справляются даже мощные современные ЭВМ. Именно поэтому современные роботы так трудно научить бегать и танцевать.

Однако, несмотря на большую скорость отработки информации, в настоящее время аналоговые ЭВМ практически проиграли в конкурентной борьбе с дискретными компьютерами. Дело заключается в том, что у аналоговых ЭВМ, даже если не принимать во внимание специфику их «программирования», есть один очень крупный недостаток. Они могут выдавать *числовые* результаты своей работы с маленькой точностью – две-три значащие десятичные цифры. Для большинства современных научных задач это совершенно неприемлемая точность вычислений, поэтому аналоговые ЭВМ имеет смысл использовать только в тех областях, где обрабатываемые данные имеют преимущественно нечисловую природу (например, в так называемых задачах распознавания образов). Заметим, что в своё время были реализованы и гибридные (аналогово-цифровые) ЭВМ, однако они тоже не оправдали возлагаемых на них надежд. С принципами работы аналоговых ЭВМ можно более подробно познакомиться, например, по книге [19].

15.2. Нейронные сети

Почему тебя не пугает машина, которая в тысячу раз сильнее тебя, но ужасает мысль о машине, которая многократно превосходит тебя интеллектом?

Станúслав Лем. «Сумма технологии»

Мой центральный процессор – самообучающийся процессор на базе нейронных сетей.

«Терминатор-2. Судный день»

Нейронные сети, как и аналоговые ЭВМ, реализуют не алгоритмическую обработку данных.

Нейронные сети стремительно вторгаются во все сферы компьютерных технологий, они призваны моделировать аналоговую обработку данных головным мозгом человека. В 1958 году американский учёный Фрэнк Розенблатт опубликовал статью «Перцептрон: Вероятная модель хранения и организации информации в головном мозге». Таким образом был изобрёл персептрон – простейшая искусственная нейронная сеть. На базе этой концепции Розенблатт в 1960 году построил первый нейрокомпьютер Марк-1, который мог решать достачно сложные задачи, например, распознавать текст.

Постараемся дать простейшее изложение принципов работы нейросетей. Базовые нейросетевые операции, как и микрооперации (мопы) обычного процессора, очень просты, это операции сложения и умножения. Чаще всего искусственный нейрон моделируется программно. Были попытки создать аппаратные модели нейрона как аналогового устройства, на вход которого поступает набор электрических сигналов разного напряженияи, после "срабатывания" такого нейрона появляется соответствующий набор выходных напряжений. Оказалось, однако, что программые модели более гибки в управлении и обходятся дешевле.

За каждый такт работы на вход такого нейрона поступает набор входных значений, они суммируются (как правило, с нормировкой, или используется суммирование с насыщением, см. главу 17). Затем эта сумма умножается на набор коэффициентов, обычно их называют *весами* (весовыми коэффициентами) или параметрами, Число таких весов равно числу выходов всех нейронов. Полученные выходные значения посылаются на вход (других) нейронов, при этом для положительного значении параметра значение выходного сигнала увеличивается, а при отрицательном – уменьшается. И всё ⚠. Т.е. с математической точки зрения каждый искусственный нейрон – эта просто (достатчно простая) функция от многих переменных Тем не менее доказано, что даже такая простая нейросеть способна (после обучения) *апроксимировать* все непрерывные функции от многих переменных, причём с любой точностью. Для этого просто достаточно взять побольше нейронов 🙂. Впрочем, чем больше в сети слоёв, тем меньше надо нейронов!

Обычно нейросеть многослойная, причём число нейронов на первом слое очень велико, а на последнем мало.[1] Например, для анализа изображений на первом слое число нейронов совпадает с числом пикселей входного изображения, а на последнем слое обычно равно числу графических объектов, которые необходимо опознать. Например, если надо распознавать десять цифр, а изображение каждой цифры 30x20 пикселей, то на первом слое будет 600 нейронов, а на втором только десять. Однако, так как у каждого входного нейрона один вход, но 10 выходов, то общее число коэффициентов будет 6000!

Когда на вход такой правильно настроенной (или, как говорят, правильно *обученной*) нейросети подаётся изображение одной из цифр, то только на одном выходе (соответствующим данной цифре) будет высокое значение выходного сигнала, а на остальных – низкое. Ну, или на всех выходах будут низкие значения, тогда это не цифра. Понятно, что обучение такой нейросети состоит в подборе «правильных» значений всех 6000 коэффициентов. Итак, изначально «чистая» нейросеть в процессе обучения настраивается на выполнение конкретной задачи.

Отметим, что в этом отношении нейросети напоминают аналоговые (непрерывные) ЭВМ, которые тоже надо настраивать для решения конкретной задачи (см. разд. 15.1). Для решения поставленной задачи нейросети тоже строят внутри себя *модель* этой задачи. В отличие от аналоговых ЭВМ, построение модели в нейросети состоит не в *физическом* изменении связей между узлами (нейронами), а только в изменении *характеристик* таких фиксированных связей. В первом приближении можно сказать, что у «проводов», соединяющих узлы-нейроны, можно менять их сопротивление (в нейросетях оно называется коэффициентами связи между узлами).

Построение такой модели и называют *обучением* нейронной сети. В процессе обучения на вход подаётся набор входных данных, для которых уже есть правильные ответы. Когда ответ для очередного теста неверный, то по определённым правилам меняются характеристики связей между нейронами сети. Подбор таких правил и является главной трудностью процесса обучения нейросети.[2] Заметим, что по такому же принципу *обратной связи* работает и нейронная сеть головного мозга человека и животных. Обычно процесс обучения проходит под управлением программы для обычного (цифрового) компьютера, тогда можно сказать, что нейросеть может *само обучаться* в процессе работы. Заметим, что в сложных нейросетях, кроме периода начального обучения, может происходить и постоянное *самообучение*, когда нейросеть сама непрерывно меняет свои коэффициенты (вспом-

---

[1] Такая сеть называется *однонаправленной*. Возможны и нейросети с более сложной структуры, включая обратные связи, однако их труднее настраивать и использовать.

[2] Начальные значения весов могут быть любыми, часно их просто инициализируют случайными значениями (при инициализации весов одинаковыми константами, например, нулями, процесс обучения значительно замедляется).

ним самомодифицирующиеся алгоритмы). Стоит отметить, что примерно так же обучаются и обычные люди, начиная с детского возраста.

В течении долгого времени использование нейронных сетей было сильно ограничено, так как обычные (алгоритмические) методы решения таких же задач оказывались более эффективными. С ростом числа нейронов и связей между ними (т.е. числа весовых коэффициентов или параметров) нейронная сеть требовала для своей работы всё больших вычислительных ресурсов. И лишь только начиная с 2010 годов произошёл прорыв в постоении больших нейронных сетей, которые и стали называться системами искусственного интеллекта (ИИ, AI – Artificial Intelligence).

В основном это были узкоспециализированные нейронные сети, предназначенные для решения задач в одной предметной области (игры в шахматы, распознавания образов, машинный перевод и т.д.). Например, можно построить нейронную сеть и заложить в управляющий компьютер правила некоторой игры (например, в шахматы), затем можно заставить две такие сети играть друг против друга (или даже одну сеть играть саму с собой). Разумеется, сначала они вообще будут пытаться делать неправильные ходы, на что им будет "строго" указывать управляющая ЭВМ. Когда нейросети научаться играть "правильно", они будут определённым образом изменять связи между своими узлами при каждом выигрыше и проигрыше.

В последние годы нейросети DNN (Deep Neural Network – нейронные сети глубокого обучения) достигли потрясающих результатов. Сейчас есть примеры таких нейросетей, которые, после примерно двух миллионов партий между собой, начинают гарантированно выигрывать у всех людей (включая чемпионов мира, которые, естественно, играть с ними отказываются 🙂). В настоящее время уже не осталось «человеческих» игр, в которые можно выиграть у таких соответствующим образом обученных нейросетей. Нейросети уже не оставили человеку никаких шансов выиграть хотя бы в одну логическую игру, в которые на протяжении сотен и тысяч лет люди играли между собой. Последней «сдалась» сложная игра Го, в 2016 году сеть AlphaGo со счётом 4:1 победила чемпиона мира Ли Седоля.

В качестве другого примера можно привести нейронную сеть Microsoft DeBERTa предназначенную для распознавания и понимания естественного языка, она состоит из 48 слоёв и имеет около 1.5 млрд. связей. В тесте SuperGLUE, ориентированном на оценку способности ИИ дать правильный ответ на базе прочитанного текста, DeBERTa впервые превзошла способности человека, показав 90.3 балла против 89.8 у человека.

Такие же большие успехи были достигнуты в распознавании образов (вспомним систему реального времени распознавания лиц в больших городах), системах машинного перевода (письменного и устного), генерации изображений на заданную тему и т.д. Сейчас ставятся и успешно решаются задачи, ещё несколько лет назад казавшиеся полностью фантастическими. Например, на вход нейросети подаются электрические сигналы, считанные с коры головного мозга человека, а на выходе получается человеческая речь.

Сейчас такие (уже обученные) нейросети начинают встраивать в компьютеры массового производства (например, в смартфоны) в виде *сопроцессора* к обычной (многоядерной) ЭВМ. В память этого сопроцессора можно загружать нейросети, обученные для распознавания образов (например, лиц людей в движении), для интеллектуального поиска в базах данных, голосового помощника, синхронного переводчика и т.д. Скоро станет стандартом, что в многоядерных процессорах по крайней мере одно ядро будет именно нейропроцессором.

Понятно, что обычные процессоры не «заточены» под такие операции, они будут выполнять их с помощью глубоко вложенных циклов, внутри которых будет много операций сложения и умножения. Здесь не так хорошо работают и упомянутые ранее векторные регистры современных ЭВМ. Так что надо создавать специальные нейропроцессоры.

Первыми кандидатами на такие устройства стали графические карты, состоящие из нескольких тысяч достаточно простых процессоров, обрабатывающих разбитое на пиксели изображение. В этих процессорах, однако, базовыми тоже являются отдельные операции сложения и умножения. Ясно, что нужен *специализированный* процессор.

В качестве примера можно привести так называемый тензорный процессор TPU (Tensor Processing Unit) фирмы Google. Он состоит из порядка 10 тысяч очень простых ядер, позволяющих складывать и перемножать короткие (8-разрядные) числа. Такие процессоры легко объединяются в кластеры по 1024 таких процессора, причем производительность от межпроцессорных связей почти не

уменьшается (графические процессоры так не могут). Один такой процессор позволяют Google обрабатывать в день порядка 100 миллионов фотографий достаточно высокого разрешения.

В настоящее время интенсивно развиваются так называемые *нейроморфные* процессоры, более полно имитирующие работу мозга. Это сверхбольшие интегральные схемы, предназначенные для моделирования *импульсных* нейронных сетей SNN (Spiking neural network), это позволяет резко поднять энергоэффективность. В таком процессоре вычислительные ядра и блоки памяти объединены, что минимизирует скорость передачи данных в процессе вычислений. Например, процессор TrueNorth содержит 5.4 млрд. транзисторов, он изготовлен по 28 нм. технологии и потребляет всего 70 мВт (импульсный!).

Здесь следует сказать, что все приведённые выше примеры касались только узкоспециализированных нейронных сетей, ориентированных на определённые предметную область (распознавание образов, перевод с одного языка на другой, игра на бирже, контекстная реклама в Интернете и т.д.) Говорят, что такие сети реализуют слабый искусственный интеллект (Narrow AI). В то же время уже давно люди стремились создать системы, способные решать задачи по крайней мере не хуже, чем это делают люди. Принято говорить, что такие системы реализуют сильный искусственный интеллект или ИИ общего назначения AGI (Artificial General Intelligence). Обычно такие системы называют большими языковыми ("разговаривающими" 🙂) моделями или когнитивными системами. Ясно, что такие сети уже не должны требовать полного переобучения при переходе от одного класса задач к другому.

15.2.1. Большие нейронные сети

Не следует смешивать то, что нам кажется невероятным и неестественным, с абсолютно невозможным.

Карл Фридрих Гаусс

Последствия появления мыслящих машин могут оказаться слишком жуткими. Будем надеяться и верить, что такое никогда не произойдёт 🙁.

Алан Тьюринг.
«Вычислительные машины и разум»,1950 г.

Итак, встала задача реализации больших нейронных сетей или больших языковых моделей. Ясно, что для этого в первую очередь было необходимо резко увеличить число нейронных связей, но тогда, однако, сильно возрастала вычислительную сложность такой системы и замедлялась её работа. Прорыв в этой области произошёл совсем недавно, в 2017 году, когда была изобретена схема построения большой нейронной сети в виде так называемого трансформера (Transformer). Эта схема, в частности, позволяла организовать хорошую масштабируемость системы.

Так, нейронная сеть GPT (Generative Pre-trained Transformer) компании OpenAI 2018 года выпуска содержала 175 млн. нейронных связей, GPT-2 2019 года 1.5 млрд., а GPT-3 2020 года уже 178 млрд. связей.[1] Для обучения таких сетей используется большой объём данных, в основном это сетевые библиотеки и большие интернет форумы. Например, для предварительного обучения сети GPT-3 использовались данные объёмом примерно 570 Гб (для сравнения, весь роман Л.Н.Толстого «Война и мир» это всего около 4 Мб).

Интересно, что архитектура трансформера позволяет, наряду с полной сетью, использовать её "урезанные" версии. Эти версии отличаются числом слоёв нейронной сети и, соответственно, числом нейронных связей. Например, для GPT-3 существует ряд таких версий, с числом связей от 125 млн. (версия Small, 12 слоёв), до версии с числом связей 13 млрд. (40 слоёв), у "полной" сети 96 слоёв и 178 млрд. связей. Надо сказать, что где-то 10-20 млрд. связей – это минимум для "хорошо разговаривающей" нейронной сети, такие сети сейчас выпускаются многими крупными компаниями, например, сеть Сбербанка GigaChat! содержит 13 млрд. связей (Giga 😳).

---

[1] Данные по GPT-4 засекречены ⚠, но по оценкам, в ней около 500 млрд. связей, а сейчас обучается уже и GPT-5.

Как уже говорилось, все такие сети требуют предварительного обучения (Pre-traine) на больших объёмах данных. Ясно, что в многослойных нейросетях (их часто называют *глубокими*), число коэффициентов исчисляется многими миллионами и миллиардами, и её обучение достаточно сложный процесс. Например, голосовой интерфейс «Алиса» фирмы Яндекс реализован в виде более десятка совместно работающих нейросетей. Одна сеть отделяет голосовое сообщение от шумов, другая классифицирует акценты, диалекты и сленги и т.д. «Алиса» обучалась на анализе многих миллионов диалогов из Интернета и социальных сетей.

Далее, после такого обучения сеть всё ещё не готова к работе. Главное назначение нейронной сети заключается в решении поставленной задачи путём ответов на вопросы, однако, человек предъявляет достаточно высокие требования к форме выдачи ответов. "Плохие" ответы могут касаться запрещённых тем или включать в себя опасные для обычного человека сведения, содержать ненормативную лексику (или просто сленг). Например, первый вариант «Алисы», обученной на молодёжный Интернет-чатах, пришлось срочно переобучать 🙂.

Попытка ввести в схему генерации ответов запреты на появоения "нехороших" результатов, конечно, делаются. Например, можно запретить выдачу рекомендаций по лечению больных, рецепты производства ядов и взрывчатых веществ и т.д. Это, однако, ненадёжный путь, так как такие запреты легко обходятся. Например, ввели запрет на выдачу адресов пиратских сайтов, тогда на вопрос, с какого сайта бесплатно скачать фильмы, будет ответ, что "эта информация закрыта". Однако, если спросить "А какие плохие сайты мне лучше не посещать?", то можно получить список таких сайтов. Отметим, что даже в детских сказках много таких примеров "обхода запретов".

Отсюда следует необходимость второго этапа обучения нейронной сети, обычно он называется *обучением с подкреплением* RL (Reinforcement learning) или *обучениям на основе человеческих предпочтений* RLHF (Reinforcement learning from Human Feedback). В это обучение вовлечена большая группа людей, которые "вручную" размечают ответы сети, говоря, что хорошо и что плохо.[1] Вообще говоря, именно так прививаются "хорошие манеры" и обычным людям, начиная с детского возраста: "не трогай, это бяка", "так не говори, поставлю в угол", "туда не ходи, лишу смартфона на неделю" и т.д. Можно (с известной натяжкой) считать это подсистемой "хорошего тона" и морально-этического контроля.[2] Именно это позволяет такому ИИ разговаривать "по-человечески".

Итак, чем больше связей, тем более высокий "интеллект" у нейронной сети, однако большие сети требуют мощных вычислительных ресурсов. Например, знаменитая сеть ChatGPT требует для своей работы супер-ЭВМ и её эксплуатация стоит примерно 500 млн. долларов в сутки (правда, она при этом одновременно общается с сотнями пользователей).

Однако, после того, как "исходники" таких сетей появились в открытом доспупе, то вскоре выяснилось, что малые и средние сети (около 10 млрд. связей, подходит даже для мощного ноутбука с "игровой" сетевой картой, время обучения около суток) допускают быструю и тонкую настройку и обучаясь на небольших, тщательно отобранных данных, становятся по качеству "разговора" сравнимыми с большими нейросетями. Можно сказать, что большие сети являются "энциклопедистами", имеющими знания почти во всех предметных областях, в то время как малые сети имеют "общегуманитарные" знания, плюс специализацию в нескольких предметных областях.

В то же время необходимо отметить, что большие нейросети превосходят малые по глубине своего "интеллекта". В Интернете уже есть для этого различные тесты. Например, можно задать ИИ вопрос (приводится оригинал на англйисом, это версия знаменитой загадки "на соображение"):

> Can you find the the mistake?
> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
> 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
> 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
> 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 70, 71, 72, 73

---

[1] Злые языки говорят, что при обучении сети ChatGPT, кроме высококвалифицированных специалистов (психологов, лингвистов и т.д.) над этим трудилась большая группа людей из Африки за три доллара в час.

[2] Не будем вступать в дискуссию, что это не человек, а машина, и у неё не может быть морали и этики "по определению".

Малая сеть говорит, что ошибок нет, а вот большая сеть видит ошибку [i] [см. сноску в конце главы].

Сейчас люди активно вытесняются из многих областей деятельности, которые раньше считали только своей прерогативой. Например, обзоры в области спорта, экономики или высокой моды сейчас лучше составляют такие нейросети, а не люди. Нейросеть ChatGPT (есть свободный доступ из интернета!) пишет фрагменты кода на многих языках программирования, оптимизирует код, конвертирует программы с одного языка на другой, создавать сайты по их словесному описанию и т.д. Кроме того, этот же, как говорят, чат-бот может поддерживать разговор на отвлечённые (в том числе философские) темы, отвечать на “каверзные” вопросы и т.д. Любопытно, что музыкальный сервер Spotify недавно удалил из своей библиотеки десятки тысяч ⚠ музыкальных записей, написанных искусственным интеллектом. Таким образом системы ИИ давно прошли тест А.Тьюринга на разумность.

Одним из недостатков существующих систем ИИ является их "простодушие". Человек, долучая новые данные, всегда в той или иной степени сомневается в их достоверности, даже если эти данные получены из надёжных источников. Попытка игнорирования такой схемы поведения часто приводит к печальным результатам (достаточно упомянуть бездумное использование фактов из Википедии). Таким образом, у человека есть своеобразная подсистема контроля неверных данных, достаточно вспомить знаменитое высказывание учёного и философа Рене Декарта "Сомневайся во всём".

Большие системы ИИ, обученные на гигантском объёме данных из Интернета, плохо "сомневаются" в достоверности этих данных. В результате в ответах на вопросы они сообщают о фильмах, в которых играли знаменитые актёры (в которых они не играли), несуществующих событиях и т.д. Один из ИИ, введённый в эксплуатацию в 2023 году, долгое время был уверен, что сейчас 2022 год и т.д. Сейчас принято говорит, что в этом случае ИИ "фантазирует", особенно это проявляется при разговоре в области гуманитарных наук. Это похоже на фантазии детей, их веру в Деда Мороза и т.д.

К сожалению, как и у человека, плохо обученные ИИ могут страдать аналогами психических расстройств, раздвоения личности и т.д. Учёным ещё предстоит справиться с этими проблемами (удастся ли это сделать 🙄).

15.2.2. Полноценный искусственный интеллект

– Педель [*это имя робота с искусственным интеллектом*], *– тихо спросил я его,*
– ты хотел бы стать человеком?
– Должен, – сказал он, но я понял, что это не ответ на мой вопрос, а какой-то заскок в его электронном мышлении.
– Могу, – сказал он после небольшой паузы и снова замолчал.
– Хотеть не умею, – и это был ответ.

Александр Житинский.
«Часы с вариантами»

Естественно, что следующим встаёт вопрос о создании ИИ, обладающий сознанием SAI (Strong Artificial Intelligence), то есть полностью разумный во всех смыслах этого слова. Мы не будем здесь дискутировать на филосовские темы, что такое "разум", "(само)сознание", "личность" и т.д., нам пока хватит и чисто житейского понимания этих терминов.

Сейчас, пожалуй, одним из главных отличий существующих ИИ от такого SAI является то, что они могут только отвечать на поставленные перед ними вопросы, но не могут ставить их перед собой.[1] У них нет такой цели, они не могут этого "хотеть". По мнению А.Тьюринга, такой интеллект должен обладать самосознанием (что это такое 🙄) и появиться не ранее 2075 года, однако уже сейчас большие системы ИИ демонстрирует некоторые признаки AGI.

Кора головного мозга человека содержит около 15 млрд. нейронов (весь мозг порядка 100 млрд.), каждый нейрон имеет до 10000 связей с другими нейронами. Таким образом, всего в мозге человека

---

[1] Конечно, при ответе на вопрос ИИ разлагает его на ряд вспомогательных подвопросов (у него эта схема работы называется "step by step".

$15*10^9*10^4$ т.е. $15*10^{13}$ связей, против 500 млрд. $=5*10^{11}$ связей у современных больших нейронных сетей. Но, с другой стороны, нас интересует только логическое мышление, а не вопросы управления телом и другие "физиологические" функции. Кроме того, скорость обработки данных компьютером на несколько прорядков превышает скорость работы мозга. Так что здесь всё под большим вопросом…

По мнению некоторых учёных, вопрос о возможности создании полноценного ИИ относится скорее не к научно-технической, а к теологической области, соответственно, существуют две точки зрения:

- Человека создал Бог, вхожив в него частицу своей сущности (душу?), эту божественную сущность человек не может ни воссоздать, ни передавать в создаваемым им вещи (скопировав или "вырвав" из себя эту сущность, вспомним големов и переселение душ). Следовательно, создание человеком AGI в принципе (по определению) невозможно.
- Человек имеет естественное происхождение, его разум появился в процессе эволюции и мозг имеет полностью физическую чущность и конечную сложность. Следовательно, человек сможет создать AGI, который будет иметь все его способности.

Завершая эту тему надо сказать, что, к счатью (или к сожалению) Вы непременно увидите, как всё это будет происходить.

15.3. Принцип микропрограммного управления

Пишите простейшие инструкции, как для трёхлетних детей.

Алан Лафли
президент компании Procter&Gamble

Принцип микропрограммного управления задает особую схему реализации процессора ЭВМ. Чтобы разобраться в этом вопросе, вернемся мысленно в начало этой книги и вспомним, как процессор выполняет отдельную команду программы. Пусть, например, на регистр команд трёхадресной ЭВМ выбрана некоторая команда `КОП A1,A2,A3`. Такая команда обрабатывается процессором, например, по схеме

```
R1:=A2; R2:=A3; S:=R1 КОП R2; A1:=S
```

где `R1,R2` и `S` – служебные (не адресуемые) регистры процессора. Как видим, сама команда тоже выполняется по некоторому алгоритму, состоящему из отдельных шагов (микроопераций, микрокоманд или мопов), а, следовательно, можно составить *программу* выполнения каждой команды компьютера (как уже упоминалось, это так называемый уровень микроархитектуры ЭВМ).

ЭВМ, в которых можно *управлять* этим микропрограммным уровнем (т.е. пользователь может добавлять и изменять микропрограммы для кодов операций), и называются компьютерами с микропрограммным управлением. При этом, как обычно, в распоряжении программиста находится язык машины, на котором он записывает свои программы.[1] В то же время, алгоритм выполнения каждой команды из языка машины реализуется в виде отдельной программы на специальном *микроязыке*. Эту программу, естественно, и называют *микропрограммой*. Все микропрограммы хранятся в специальной очень быстрой памяти внутри процессора. После считывания из памяти очередной команды, процессор по коду операции находит соответствующую ей микропрограмму и выполняет её, затем, как всегда, читает на регистр команд следующую команду и т.д.

Принцип микропрограммного управления при конструировании компьютеров предложил использовать в 1951 году Морис Винсент Уилкс (Maurice Vincent Wilkes), тот самый, кто участвовал в создании первой «настоящей» ЭВМ EDSAC. Впервые этот принцип реализован в ЭВМ EDSAC-2 в 1957 году, так же работал и легендарный процессор Intel 8086 1978 года выпуска.

Микроязык содержит относительно небольшое количество микрокоманд (мопов). Они реализуют такие простые действия, как пересылка операндов с одного места памяти в другое, целочисленное

---

[1] Имеется в виду программист на языке Ассемблера, он, как, впрочем, и прикладной программист, имеет право ничего не знать о том, что его компьютер работает по принципу микропрограммного управления. Как уже говорилось, этот принцип относится не к уровню машинных команд, а к следующему (уже не пользовательскому, а инженерному) уровню *микроархитектуры* ЭВМ.

сложение, сравнения операндов на равенство, некоторые сдвиги, логические операции **and**, **or** и **not** и т.п. Вследствие того, что теперь процессору надо уметь выполнять только небольшое число простых микрокоманд, его архитектура очень сильно упрощается, теперь его можно делать из очень быстро работающих (и дорогих) интегральных схем. Следовательно, несмотря на то, что теперь каждая команда выполняется по микропрограмме, таким образом можно избежать значительного снижения производительности компьютера, построенного по принципу микропрограммного управления.

Теперь надо рассмотреть те преимущества, которые открывает принцип микропрограммного управления. Во-первых, из-за небольшого числа простых микрокоманд в процессоре легко реализовать несколько быстродействующих конвейеров, сильно повышающих производительность ЭВМ.

Во-вторых, набор микропрограмм можно менять. Обычно это делается при включении машины, когда предоставляется возможность загрузить в память микропрограмм новый набор микрокоманд. Эта уникальная возможность позволяет в значительной степени менять архитектуру компьютера на уровне машинных команд. Действительно, можно в очень широких пределах менять язык машины, например, сменить двухадресную систему команд на трёхадресную или безадресную. Можно также изменить способы адресации, поменять форматы обрабатываемых данных (скажем, ввести 256-битные целые числа и новые команды, которые их обрабатывают). Можно сказать, что это аппаратная реализация макросредств, позволяющая изменять язык машины.

Компьютеры с микропрограммным управлением были достаточно широко распространены в 60-х и 70-х годах прошлого века, обычно на этих принципах строились ЭВМ средней и высокой производительность (но не супер-ЭВМ).

В персональных компьютерах микрокод используется, начиная с процессора Pentium Pro, выпущенного в 1995 году. На этом уровне *микроархитектуры* каждая достаточно сложная команда тоже описывается в виде микропрограммы. Например, команда цикла **loop** реализуется микропрограммой путём подачи на конвейер тела цикла `ECX` раз. Через микропрограммы выполняются операции деления и действия с вещественными денормализованными числами.

Например, в микроархитектуре процессора Nehalem фирмы Intel декодер циклов DSB (Decoded Stream Buffer) опознаются циклы с телом длиной до 28 команд, что резко увеличивает скорость их выполнения.

На самом деле как описывалось ранее, на конвейеры поступает не поток команд программы. Сначала подлежащие выполнению команды поступает из кэша команд L1I в так называемые буфера предвыборки, откуда они затем декодируются на микрооперации и поступают в буфер микроопераций, а уже оттуда выбираются в очереди для выполнения на исполнительные блоки конвейера (как же всё хорошо и просто было в машине фон Неймана). Эти блоки (их сейчас 8, они называются *портами* конвейеров) могут выполнять микрооперации и обмены с памятью (2 порта ввода, 2 вывода и 4 порта для выполнения микроопераций над данными).

Все микропрограммы хранятся в упакованном виде в небольшой (для современных процессоров x86 это несколько килобайт) быстродействующей постоянной памяти MSROM (Microcode Sequencer ROM) и зашифрованы (интересно, зачем). Архитекторам фирмы Intel пришлось, однако, предусмотреть особый режим, позволяющий загружать новые версии микропрограмм отдельных команд. В основном это сделано для того, чтобы пользователь мог оперативно исправить аппаратные ошибки, которые иногда обнаруживаются в работе ЭВМ. Этот процесс аналогичен процессу замены (перепрошивки) программного обеспечение `BIOS`, более того, и в самих `BIOS` присутствует раздел `CPUCODE.BIN` с новыми версиями микропрограмм, у которых исправлены ошибки.

Идеи, похожие на принцип микропрограммного управления, были заложены в 70-ые годы прошлого века в ЭВМ, так называемой RISC архитектуры (Reduced Instruction Set Computer – компьютеры с уменьшенным набором команд). Основная идея здесь состоит в том, чтобы оставить в машинном языке только небольшой набор (порядка двух-трех десятков) наиболее часто используемых простых команд, а все остальные операции реализовывать как подпрограммы в этом урезанном машинном языке. При этом делают так, чтобы все команды в таком компьютере имели одинаковую длину, имели формат `RR` (т .е. их операнды располагаются только в регистрах) и выполнялись за одинаковое время (обычно за один такт). Такой подход позволяет сильно упростить структуру процессора и реализовать в нём очень эффективный конвейер.

Основоположником RISC архитектуры принято считать Дэвида Паттерсона, предложившего её в 1980 году. Эта архитектура показала свою жизнеспособность и в настоящее время используется в нескольких выпускающихся семействах ЭВМ (например, SUN SPARC, Power PC, а также в процессо-

рах ARM, которые используются в большинстве смартфонов и планшетов). Как уже отмечалось, в компьютерах с полным набором команд CISC (Complex Instruction Set Computer) реализовано так называемое RISC ядро, которое быстро выполняет на конвейере небольшой набор самых распространенных команд. В семействе Intel RISC ядро появилось, начиная с процессора 486 в 1989 году, а в настоящее время практически все сложные команды (их длина может достигать 15 байт) транслируются в набор мопов. В качестве примера можно привести команды вычисления тригонометрических функций, например, команда **`fcos`** выполняется примерно за 70 тактов.

15.4. ЭВМ, управляемые потоком данных

Всё на свете лишь информация. А мы – подпрограммы, взаимодействующие в каком-то смоделированном пространстве.
Питер Уоттс

Все рассматриваемые до сих пор в этой книге ЭВМ базировались на принципах фон Неймана. Как уже говорилось, современные ЭВМ в той или иной мере отказываются от большинства этих принципов для повышения своей производительности. Тем не менее, все рассмотренные до сих пор архитектуры ЭВМ продолжают следовать одному *основополагающему* принципу фон Неймана. Это принцип программного управления, который состоит в том, что ЭВМ автоматически обрабатывает *данные*, выполняя расположенную в своей памяти *программу*. Итак, эти ЭВМ управляются потоком команд (CFC – Control Flow Computer). То, что именно программа должна обрабатывать данные, представляется большинству программистов совершенно очевидным. Разберёмся, однако, с этим вопросом более подробно.

Как Вы помните, ЭВМ является алгоритмической системой, в частности, она является *исполнителем* алгоритма на языке машины. Обычно дается примерно следующее *неформальное* определение алгоритма. «Алгоритм – это чёткая система правил (шагов алгоритма). Обрабатывая входные данные, к которым алгоритм *применим*, исполнитель алгоритма за конечное число выполнения своих шагов остановится и выдаст результат (выходные данные)».

Как видно, по самому определению алгоритма он выполняет в вычислительной системе *главную* роль, а обрабатываемые данные – *подчинённую*. В начале развития вычислительной техники такое положение вещей было очевидным, оно находило своё отражение и в «информативности» программы и данных. Например, исследуя программу решения систем линейных уравнений, можно в принципе понять, для чего предназначена эта программа. В то же время, как бы тщательно не изучались вводимые этой программой данные (т.е. числовые матрицы), вряд ли можно понять, в чём заключается решаемая задача. Можно сказать, что в некотором смысле в этой задаче программа играет главную роль, а обрабатываемые данные – второстепенную. Другими словами, если принять во внимание, что программу в совокупности с её данными можно рассматривать как *модель* некоторого объекта, то бóльшая часть сведений об этом объекте была сосредоточена именно в программе, а не в обрабатываемых данных.

Такая ситуация сохранялась в программировании примерно до начала 80-х годов прошлого века. Далее, однако, обрабатываемые данные непрерывно усложнялись, пока, наконец, для некоторых задач не превзошли по сложности программы, которые обрабатывали эти данные. Рассмотрим в качестве примера базу данных, которая хранит текущее состояние дел некоторого предприятия. Можно сказать, что база данных содержит сложно структурированную, изменяющуюся во времени *модель* этого предприятия. Программы, обрабатывающие информацию из базы данных (БД) называются системой управления базой данных (СУБД) [10]. СУБД позволяет вводить, удалять и модифицировать данные в БД, а также обрабатывать запросы на поиск и выдачу из БД нужных сведений. Так вот, сколько бы программист не исследовал программы, входящие в СУБД, он практически ничего не узнает о самом предприятии, ни что оно выпускает, ни сколько человек на нём работает и т.д. Очевидно, что в нашей модели предприятия (БД+СУБД) данные играют основную роль, а обрабатывающие их программы – уже второстепенную.

Как Вы можете догадаться, примерно в это же время появилась идея коренным образом изменить архитектуру компьютера так, чтобы отказаться от принципа *программного управления*. Таким образом, если компьютеры традиционной архитектуры управляются потоком (или потоками) команд, то компьютеры новой, нетрадиционной архитектуры должны управляться потоком данных. Можно сказать, что теперь не команды должны определять, когда и какие данные надо обрабатывать, а, наобо-

рот, данные сами выбирают для себя действия (операторы), которые в определенный момент надо выполнить над этими данными для получения нужного результата.[1] Компьютеры такой архитектуры принято называть потоковыми ЭВМ (по-английски DFC – Data Flow Computers) [1,14].

Надо заметить, что сам по себе принцип потоковой обработки данных не представляет собой ничего загадочного или экзотического. Например, отметим, что уже изученные Вами ранее такие алгоритмические системы, как машина Тьюринга и Нормальные алгоритмы Маркова были обработчиками данных именно этого класса. Действительно, скажем, в машине Тьюринга именно *данные* (текущий символ, на который указывает головка), определял ту клетку таблицы, *операции* из которой необходимо было выполнить для этого символа! Возвращаясь к классификации вычислительных машин по Флинну, по аналогии можно сказать, что в машине Тьюринга один поток данных задаёт один поток команд для своей обработки. Тогда машину Тьюринга можно отнести к классу SDSI (Single Data Single Instruction).

В то же время оказалось, что архитектура «настоящих» потоковых ЭВМ оказывается весьма сложной и сильно отличается от архитектуры традиционных ЭВМ. Например, в устройстве управления таких ЭВМ нет никакого счётчика адреса, а в их памяти нет программы (по крайней мере, в традиционном понимании). Рассмотрим схему функционирования такой потоковой ЭВМ только на одном очень простом примере.

Пусть необходимо выполнить такой оператор присваивания:

```
x:=(x+y)*p+(x+p)/z-p*y/z
```

Представим этот оператор в виде так называемого графа потока данных, показанного на рис. 15.1.

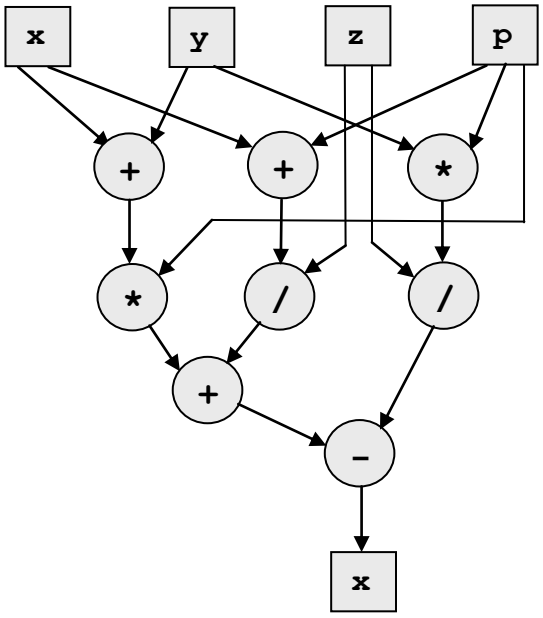


Рис.15.1. Граф потока данных для оператора присваивания.

В этом ориентированном графе есть два вида узлов: это сами обрабатываемые данные, изображённые в виде квадратиков (в нашем примере – значения переменных `x,y,z` и `p`) и *обрабатывающие элементы* потоковой ЭВМ, которые обозначены просто знаками соответствующих арифметических операций в кружочках.[2]

Основная идея потоковых вычислений состоит в следующем. Все действия над данными производят обрабатывающие элементы (операторы), в нашем примере эти элементы обозначены арифметическими операциями сложения, вычитания, умножения и деления. Можно сказать, что всё арифме-

---

[1] Первоначально идея о том, что операторы не должны полностью главенствовать над обрабатываемыми данными проникла в "обычное" программирование вместе с объектно-ориентированными языками. Здесь, правда, не шла речь об отказе от принципа программного управления, а о некотором органичном объединении алгоритмов и обрабатываемых данных внутри объектов.

[2] Граф потока данных базируется на принципах построения так называемых сетей Петри.

тико-логическое устройство потоковой ЭВМ – это набор таких обрабатывающих элементов. Каждый обрабатывающий элемент начинает автоматически выполняться, когда на его входе есть в наличие (готовы к обработке) требуемые для него данные. Так, в нашем примере автоматически (и параллельно!) начинают выполняться обрабатывающие элемента для операторов `+` , `+` и `*` во второй строке нашего графа, затем, на втором шаге работы, параллельно выполняются операторы `*` , `/` и `/` в третьей строке и т.д. Здесь просматривается большое сходство со способом работы электронных схем, составленных из вентилей. Действительно, если граф потока данных "положить на левый бок", то он по внешнему виду будет весьма похож на электронную схему двоичного сумматора, как она была показана на рис. 2.2а. При этом роль вентилей будут играть обрабатывающие элементы, а входных и выходных сигналов – обрабатываемые данные.

Заметим, что обрабатываемые данные в потоковом компьютере вовсе не являются *пассивными*, как в ЭВМ традиционной архитектуры, наоборот, они «громко заявляют» о своей готовности к обработке (можно сказать, требуют к себе "внимания" со стороны обрабатывающих элементов).[1] Вообще говоря, здесь есть все основания отказаться от принципа фон Неймана, согласно которому память только *хранит* данные, но не обрабатывает их. Другими словами, представляется естественным совместить функции хранения и обработки данных, и разместить обрабатывающие элементы не в арифметико-логическом устройстве, а прямо в оперативной памяти.

Вообще говоря, граф потока данных и является «программой» для потоковой ЭВМ, а программ в нашем привычном понимании здесь не существует. Здесь уместно вспомнить таблицу, описывающую машину Тьюринга, является ли она «программой»?. Можно сказать, что здесь сами данные управляют процессом своей обработки. Отсюда можно сделать вывод, что обычные языки программирования плохо подходят для записи алгоритмов обработки данных в потоковых ЭВМ, так что приходится делать сложный специальный компилятор, преобразующий обычную последовательную программу на языке высокого уровня в граф потока данных. Неудивительно поэтому, что вместе с идеей потоковых ЭВМ появились и специальные языки потоков данных DFL (Data Flow Languages),[2] ориентированные на прямое описание графа потока данных, например, язык Lucid [33].

Из рассмотренной схемы обработки данных в потоковых ЭВМ можно сделать вывод, что в них может быть реализован принцип *максимального параллелизма* в обработке данных, так как любые готовые данные тут же могут поступать на выполнение соответствующему обрабатывающему элементу потоковой ЭВМ. Ясно, что *быстрее* решить данную задачу просто невозможно (хотя можно, конечно, попытаться построить другой, более эффективный граф).

Немного подумав, можно понять, что максимальный параллелизм обработки данных имеет и отрицательную сторону. Дело в том, что многие вычисленные данные могут оказаться ненужными в дальнейшем вычислительном процессе. Например, в операторе `if then else` нужен результат только одной из двух вычислительных ветвей. Исходя из этого, более предпочтительной выглядит вычислительная модель управления вычислениями по запросу Demand-Driven Control). Теперь операции выполняются не по готовности операндов, а когда в результате этой операции возникнет необходимость, это так называемые ленивые (отложенные) вычисления (Lazy Evaluation). При этом операции ближе к вершине графа выполняются раньше, такая модель называется редукционной вычислительной системой. Теоретической основой редукционных систем является так называемое лямбда-исчисление, а для использования нужны функциональные языки программирования (Lisp, Haskell и другие).

Немного подумав, можно выделить две фундаментальные трудности, которые встают при реализации потоковой ЭВМ. Во-первых, для достаточно сложного алгоритма невозможно полностью построить граф потока данных до начала счёта. Действительно, алгоритм вводит свои входные данные и содержит условные операторы, выполнение которых может, в частности, зависеть и от этих входных данных. Следовательно, компилятор с некоторого традиционного языка программирования может построить только некоторый первоначальный граф потока данных, а устройство управления по-

---

[1] Здесь просматривается аналогия с конвейерами современных ЭВМ (см. разд. 14.2.1), где тоже есть набор обрабатывающих элементов, которые берут на выполнение данные по мере их готовности, причём многие операции (например, сложение, умножение и т.д.) могут выполнять несколько таких обрабатывающих элементов.

[2] Язык не поворачивается назвать эти языки языками программирования .

токовой ЭВМ должно будет в процессе счёта *динамически* изменять этот граф. Ясно, что это весьма сложная задача для аппаратуры процессора потоковой ЭВМ.

Вторая проблема заключается в том, что арифметико-логическое устройство потоковой ЭВМ должно содержать много одинаковых обрабатывающих элементов. Даже для приведенного нами в качестве примера простейшего графа потока данных нужно по два обрабатывающих элемента для выполнения операций сложения и деления. Для реальных алгоритмов число таких одинаковых обрабатывающих элементов должно исчисляться сотнями, иначе готовые к счёту данные будут долго простаивать в ожидании освобождения нужного им обрабатывающего элемента, и вся выгода от потоковых вычислений будет потеряна (и тогда ни стоило, как говорится, и огород городить). Другая трудность заключается в том, что выход каждого обрабатывающего элемента (результат его работы) может быть подан на вход любого другого обрабатывающего элемента. Такие вычислительные системы называются *полносвязными*. При достаточно большом числе обрабатывающих элементов (а выше было обосновано, что иначе и быть не может), реализовать все связи между ними становится практически неразрешимой технической задачей.

Ясно, что перед конструкторами потоковой ЭВМ встают очень большие трудности. В настоящее время универсальные потоковые ЭВМ не нашли широкого применения из-за сложности своей архитектуры, пока существуют только экспериментальные образцы таких ЭВМ.[1] Элементы потоковой архитектуры реализованы в современных суперконвейерных ЭВМ, в которых существует буфер переупорядочивания RoB (ReOder Buffer), с командами, уже разложенными на мопы. Моп из этого буфера выбирается на исполнение на один из портов конвейера, как только для него будут готовы все его операнды, причём порядок выполнения мопов может отличаться от порядка их следования в программе. Поэтому некоторая команды может начать выполняться раньше *предшествующей* команды программы. В этом буфере, однако, находится не вся программа, а всего несколько сот текущих команд.

Несколько лучше обстоит дело, если обрабатывающие элементы в потоковых ЭВМ, сделать достаточно сложными, т.е. предназначенными для выполнения крупных шагов обработки данных (например, операторов, или процедур и функций). В этом случае каждый такой обрабатывающий элемент можно реализовать в виде отдельного микропроцессора, снабженного собственной (локальной) памятью и соответствующей программой (в традиционном понимании). Все такие обрабатывающие элементы объединены высокоскоростными линиями связи (шинами), причем каждый элемент начинает работать, как только он имеет все необходимые входные данные. Эти данные могут располагаться в локальной памяти или поступать на линии связи данного обрабатывающего элемента от других обрабатывающих элементов. Заметим, что похожую организацию имеют и современные супер-ЭВМ.

Идея управления обработки информации потоками данных нашла применение и в таком быстроразвивающемся направлении вычислительной техники, как нейрокомпьютеры (нейросети).

---

[1] Информация к размышлению: по каким принципам обрабатывает информацию мозг человека?

15.5. Квантовые вычисления

…для постижения модели квантовых вычислений нужен намного более суровый сдвиг парадигмы, чем, к примеру, такой сдвиг от структурного программирования к объектно-ориентированному и тем более к функциональному. Здесь нет ни слова о том, что кто-то из программистов находится выше или ниже, но лишь о том, что освоение новой модели – это реальный переворот в мозгах и способе алгоритмического мышления.

Darkus

Когда вы приближаетесь к океану, знания о колодцах только мешают.

Конфуций, V век до н.э.

Цифровые вычислительные машины, как уже говорилось, являются исполнителями алгоритма, записанного на языке машины. Напомним главные свойства любого алгоритма.

- Дискретность или структурность. Алгоритм состоит из шагов, которые выполняются исполнителем в определённом порядке. Кроме того, если есть два алгоритма, такие, что выходные данные первого из них можно использовать как входные данные для второго, то строго доказано, что существует третий алгоритм, который является суперпозицией (работает как последовательное выполнение) двух первых алгоритмов.
- Детерминированность. Результат выполнения исполнителем каждого алгоритма не зависит от времени и места такого выполнения. Другими словами, для одинаковых входных данных исполнение алгоритма всегда дает одинаковые результаты.
- Определённость. Исполнитель однозначно понимает, как необходимо выполнять каждый шаг алгоритма.

Во многих книгах к необходимым свойствам алгоритма относят также весьма странное свойство массовость, которое предполагает, что для алгоритма существует достаточно большое множество входных данных. Однако, если его принять, то все так любимые учебниками первые программы, печатающие что-то вроде «Hello, World!» уже не должны считаться алгоритмами, что, конечно, весьма странно ☹.

Как было рассказано выше, аналоговые вычислительные машины отказываются от принципа структурности, и, таким образом, не являются собственно алгоритмическими системами. В то же время необходимо заметить, что они по-прежнему удовлетворяют свойству детерминированности.[1]

А теперь посмотрим, что будет, если при решении задач отбросить свойство детерминированности. На первый взгляд, это совершенно бессмысленно, так как теперь при решении задачи с одними и теми же входными данными будут получаться разные ответы. Всё, однако, обстоит не так безнадёжно, ведь примерно таким образом и работают так называемые квантовые ЭВМ.

Идею квантовых вычислений впервые высказал советский математик Ю.И. Манин в 1980 году в своей книге «Вычислимое и невычислимое» [25]. Он обратил внимание на то, что мощность ЭВМ возрастает менее, чем линейно, с ростом их аппаратных возможностей, в то время как при линейном росте характеристик моделируемых физических систем число связей между их элементами возрастает много быстрее, по экспоненте. Например, при добавлении только одного электрона в молекулу, сложность численного решения соответствующего волнового уравнения Шредингера возрастает в два раза! Отсюда делался вывод о том, что классические ЭВМ малопригодны для моделирования сложных физических и биологических систем.

Ю.И. Манин предложил использовать для этих целей квантово-механические системы, сложность связей в которых также экспоненциально возрастает с линейным ростом таких систем. Эта идея

---

[1] Разумеется, при этом надо учитывать, что, в отличие от дискретных ЭВМ, из-за аналогового принципа работы, как входные данные, так и ответы представлены с некоторой погрешностью и, как следствие, их нельзя точно повторить при следующем запуске на счёт.

была затем развита в опубликованной в 1982 году работе знаменитого физика-теоретика нобелевского лауреата Ричарда Фейнмана (Richard Feynman) «Simulation physics with computers» [26].

Практически все современные классические компьютеры работают в двоичной системе, и их основным элементом является бит, который может находиться в двух устойчивых состояниях. Из N таких бит можно собрать N-разрядный двоичный регистр, способный в каждый момент времени хранить одно N-разрядное двоичное число. Р. Фейнман предложил на роль бита в квантовой ЭВМ какой-нибудь двухуровневый квантовый объект, который, естественно, назвали *кубитом* (quantum bit, qubit, q-bit). Например, можно взять для этой цели электрон, измерение спина которого даёт только два различных результата.

Термин «кубит» предложил американский физик-теоретик Бен(джамин) Шумахер (Ben Schumacher) в 1995 году. Интересно, что некоторые современные исследования предполагают, что молекула $Ca_9(PO_4)_6$ в так называемых кластерах Познера в человеческом мозге может работать как *кутрит* (квантовый аналог трита – троичного запоминающего элемента). В принципе, возможно создание и квантовых ЭВМ на элементах с обобщённым названием *кудит* (кутрит – троичный, кукварт – четверичный и т.д.).

Из N кубитов собирается квантовый регистр. Р. Фейнман обосновал, что на таком регистре можно построить изолированную систему, находящуюся в так называемой *когерентной квантовой суперпозиции*, часто называемой *квантовой сцепленностью* или *квантовой запутанностью* (entanglement). Такая система описывается 2^N комплексными числами и с математической точки зрения задаёт вектор в комплексном гильбертовом пространстве размерности 2^N.

Таким образом, если для задания значения обычного регистра требуется *одно* N-разрядное число, то для задания значения квантового регистра такой же длины понадобится уже 2^N и уже *комплексных* чисел. Видно, что с ростом длины квантового регистра число таких комплексных чисел возрастает фантастически быстро.

Квантовый регистр является основным элементом квантовой ЭВМ, на него помещаются входные данные и с него же считывается результат. За один такт работы к одному или двум кубитам этого регистра применяется одна из *квантовых операций*. По аналогии с обычными компьютерами, где битовые операции реализуются (электронными) вентилями (logic gates), квантовые операции реализуются квантовыми вентилями (quantum gates), которые оперируют с одним или двумя кубитами.

В отличие от обычного вентиля, который задаёт *логическую* операцию над битами, квантовый вентиль, несмотря на то, что он оперирует только с одним или двумя кубитами квантового регистра, в математическом смысле задаёт *унитарный оператор* над *всем* вектором в рассмотренном выше гильбертовом пространстве. Такой оператор описывается комплексной матрицей размерности 2^N x 2^N, а сама операция заключается в умножении этой матрицы на вектор состояния квантового регистра. Отсюда видно, насколько квантовые операции мощнее обычных битовых, эта мощность теоретически возрастает в два раза с каждым новым кубитом квантового регистра.

Напоминаем, что гильбертовым пространством называется линейное векторное пространство со скалярным произведением, а унитарный оператор – это линейный оператор, который сохраняет норму вектора. Так как линейный оператор имеет *обратный* оператор, то все квантовые операторы, в отличие от, например, логических операторов **`and`**, **`xor`** и т.д. (исключение составляет операция **`not`**), *обратимы во времени*. Это так называемая *консервативная логика*, основы которой заложили советские физики Л.Д. Ландау и Е.М. Лифшиц в 1961 году. Например, обратимое булевское сложение по модулю два **`cnot`** должно иметь два входа (`x` и `y`) и два выхода `x` и (`x` **`xor`** `y`). Обратимое логическое умножение **`ccnot`** (так называемая операция Тоффоли) имеет уже три входа и три выхода. Операций **`not`**, **`cnot`** и **`ccnot`** достаточно для представления любой квантовой операции.

Известно, что любое вычисление, уменьшающее энтропию информационную, обязано приводить к увеличению энтропии термодинамической, то есть к выделению тепла. Как следует из теории, при потере каждого бита информации выделяется не менее $kT*\ln(2)$ Дж, где k – постоянная Больцмана, связывающая температуру с кинетической энергией вещества («теплом»). При комнатной температуре T=300 Кельвинов потеря бита это около $4 \cdot 10^{-21}$ Дж.=0.025 эВ. Например, при каждом срабатывании самого широко распространённого при конструировании микросхем вентиля **`nand`** информационная энтропия уменьшается на 1.189 бита, что приводит к рассеиванию примерно 0.02 эВ тепла. То же самое происходит и при каждом чтении и записи в оперативную память. А вот квантовые опе-

рации с физической точки зрения проходят без потери информации, и, как следствие, энтропия квантовой системы не меняется и тепло не выделяется (температура не увеличивается).

Любую логическую операцию над битовыми данными, как известно, можно задать в виде схемы, собранной их небольшого набора (базовых) вентилей, например, **`or`**, **`and`** и **`not`**, это же верно и для квантовых операций. Как уже упоминалось, в качестве базовых квантовых операций (имеющих обратные!) можно, например, взять двухкубитную операцию **`cnot`** (**C**ontrolled **`NOT`**, аналог операции **`xor`**), однокубитную операции **`not`** и самое простое унитарное преобразование Адамара (поворот вектора на 45º).[1]

Управляющая ЭВМ воздействует на кубиты квантового регистра, например, с помощью импульсов микроволнового лазера. Сначала такие импульсы переводят регистр в запутанное состояние. Далее с помощью таких импульсов квантовые вентили воздействуют на кубиты квантового регистра. Подбирая частоту и длительность этих импульсов, можно заставить кубиты переходить в другие состояния, всё ещё находясь в суперпозиции. Более мощный импульс лазера используется и как команда СТОП, переводя квантовый регистр в классическое состояние из «0» и «1».

Квантовая ЭВМ может выполнять только линейную последовательность квантовых операций, никакие условные операторы и циклы невозможны. Таким образом, собственно квантовый алгоритм, где, конечно, есть условные операторы и циклы, выполняется на традиционном (управляющем) компьютере, а сама квантовая ЭВМ является только своеобразным сопроцессором.

Выполняя квантовый алгоритм, управляющий компьютер в нужные моменты заполняет квантовый регистр входными данными и переводит кубиты в состояние квантовой суперпозиции. Далее управляющая ЭВМ вызывает квантовый сопроцессор, указывая ему, какую цепочку операций нужно выполнить. Чаще всего управляющая ЭВМ вычисляет (строит) такую цепочку, включая в неё нужные квантовые операции над кубитами. Специфической квантовой операцией в цепочке является так называемый квантовый *оракул*, фактически это предикат над значением квантового регистра, который определяет, удовлетворяет ли значение этого регистра заданному условию. После завершения работы квантовой ЭВМ она останавливается, и затем управляющая ЭВМ считывает (уже двоичный) результат с квантового регистра, после чего решает, что делать дальше.

Квантовый компьютер, выполнив последовательность *квантовых операций*, для одних и тех же входных данных может с определённой вероятностью дать любой ответ из некоторого множества. При этом квантовый алгоритм для решаемой задачи считается верным, если правильный ответ получается с вероятностью, достаточно близкой к единице. Как образно сказано в одной статье: «Квантовые волны в кубитах интерферируют таким образом, что неправильные решения тонут, а правильные – всплывают».

Повторив на управляющей ЭВМ вычисления на одинаковых входных данных несколько раз и выбрав тот ответ, который встречается наиболее часто, можно снизить вероятность ошибки до сколь угодно малой величины. Ещё проще дело обстоит в том случае, если выданный квантовым компьютером ответ можно легко проверить на правильность на традиционном компьютере. Например, если вычисляется корень некоторого уравнения, то обычно легко проверить, является ли полученный ответ «настоящим» корнем, просто подставив его в уравнение. В частности, легко определить, является ли полученное квантовым алгоритмом натуральное число делителем некоторого другого целого числа.

У квантовых вычислений есть, кроме недетерминированности, и другой существенный недостаток. В отличие от классического алгоритма, выполнение которого можно прервать при получении приемлемого ответа (аналог цикла **`while`** или **`repeat`**), заданная схема квантовых вычислений должна быть выполнена до конца (аналог цикла **`for`**). Любая попытка «посмотреть» на промежуточный результат вычислений приводит к коллапсу квантовой системы, это называется *декогеренцией*. При этом вычислительный процесс прекращается, а квантовый регистр переходит в классическое состояние, т.е. каждый кубит принимает значение ноль или единица.

Декогеренция может происходить и спонтанно, например, при колебаниях температуры или посторонних помехах. Это представляет серьёзную угрозу для квантовых вычислений. Время наступления спонтанной декогеренции обычно меньше времени выполнения сложных квантовых вычислений.

---

[1] Дэвид Дойч (David Deutsch) в 1995 году показал, что любые квантовые вычисления можно выполнять с помощью всего двух базовых квантовых операций.

Для устранения этой опасности были разработаны методы квантовой коррекции таких ошибок, без применения которых квантовые вычисления реализовать нельзя. Эти методы, как и для классических ЭВМ, используют избыточное количество кубитов, благодаря чему можно определять и исправлять ошибки. Итак, на «работающие» кубиты приходится тратить и «контрольные» кубиты, удерживающих квантовый регистр от декогерентности, т.е. исправляющих возникающие квантовые ошибки. Время работы до спонтанной декогеренции весьма мало, так в 2018 году 50-кубитная ЭВМ фирмы IBM могла находится в когерентном состоянии примерно 90 микросекунд, хотя отдельный кубит может находится в когерентном состоянии значительно дольше (порядка 3 мс).

Используется также термин *квантовый объем* – это количество кубит на количество ошибок при совершении операции. Это показывает, что недостаточно просто сказать сколько в системе кубитов, важна ещё и степень контроля над ними, которая позволяет избежать ошибок. Для роста квантового объема необходим рост и количества, и качества контроля над кубитами. Естественно ввести понятие «логического» кубита, стабильная работа которого обеспечивается определённым числом физических кубит. Таким образом, логический кубит состоит из многих физических кубит.

Несмотря на всю экзотичность квантовых вычислений, доказано, что они не нарушают хорошо известный тезис Тьюринга. Отсюда следует, что все задачи, которые может решить квантовая ЭВМ, решаются и на традиционных компьютерах. Таким образом, речь может идти только о гораздо большей эффективности решения некоторых задач. Неудивительно поэтому, что для формального описания квантовой ЭВМ можно использовать одну из модификаций машины Тьюринга, *квантовую* машину Тьюринга. Кратко расскажем, как работает такая машина.

Сначала рассмотрим так называемую *недетерминированную* машину Тьюринга, в таблице которой на пересечении столбца, озаглавленного символом алфавита, и строки с текущим состоянием, может находиться не одна клетка программы, а целый набор («стопка») таких клеток. Каждая клетка, как известно, задаёт три операции: запись символа в текущую клетку ленты, движение головки по ленте на одну позицию и переключение в новое состояние. Так вот, в недетерминированной машине Тьюринга все эти клетки такой «стопки» начинают выполняться одновременно. В этот момент для общей таблицы происходит порождение нескольких копий текущей ленты, у каждой своя головка. Далее все головки параллельно работают со своими лентами (и, вообще говоря, с разной скоростью). Следует понять, что это происходит при *каждом* выполнении любой клетки, и, таким образом, число лент и головок может быстро возрастать.

С точки зрения программиста при выполнении такой клетки порождается несколько вычислительных задач (не потоков!) Windows, у всех этих задач общая секция кода, но свои секции данных и стека (в начальный момент это копии секций из родительской задачи) . Более всего этому соответствует системный вызов `fork()` в языке C, который «раздваивает» текущую задачу (в Unix – процесс).

Работа недетерминированной машины Тьюринга завершается, когда любая из параллельных задач попадает в клетку останова, при этом позиция головки на этой ленте определяет ответ. Понятно что при работе такой машины порождается дерево вычислительных задач, число которых может быстро и неограниченно возрастать. Ясно, что при запусках с одинаковыми входными данными, такая машина может давать *разные* ответы, что и означает, что она *недетерминированная* (и, вообще говоря, не задаёт алгоритм в традиционном смысле). Можно сделать такую машину детерминированной, если поток случайных величин, управляющий работой всех головой, сделать дополнительными входными данными алгоритма.

Рассмотрим, например, использование этой машины для решения задачи поиска выхода из лабиринта. В каждой точке, где лабиринт разветвляется на несколько путей, мы ставим в программе соответствующее число клеток, и начинаем параллельно двигаться по всем возможным путям в лабиринте, при этом первая же задача, который найдёт выход из лабиринта, останавливает работу машины.

При этом удобно для повышения эффективности вычислений добавить в таблицу особые клетки *безрезультативного* останова, попав в которую данная вычислительная задач заканчивается (её лента и головка уничтожаются), но остальные задачи продолжают выполняться. Для поиска выхода из лабиринта эта клетка тупика, из которого нет ни одного выхода. Таким образом, можно существенно уменьшать число одновременно выполняемых параллельных процессов. На такой машине естественным образом реализуется класс задач спуска по дереву с возвратом при неудаче, эти задачи на обычных ЭВМ решаются, как правило, с помощью рекурсии. Понятно, что, начиная из *одной* исходной точки лабиринта, такая машина может найти *разные* выходы из этого лабиринта.

Квантовая машина Тьюринга является дальнейшей модификацией недетерминированной машины.[1] Теперь каждая клетка программы помечена некоторым комплексным числом, которое задаёт так называемую *амплитуду* этой клетки, полный набор амплитуд всех клеток нормирован на единицу. У квантовой машины одна лента и много головок (число которых, как и для недетерминированной машины, в процессе работы может расти, в пределе своя головка стоит на каждом символе ленты). Считается, что в каждой клетке такой общей ленты содержатся (с разной ненулевой вероятностью) все символы, которые были туда хотя бы раз записаны. Таким образом, через некоторое время от начала работы, каждая клетка обрабатываемого слова на ленте содержит (с разной вероятностью) некоторое подмножество символов алфавита. Далее, с точки зрения программиста *одна* квантовая машина Тьюринга при своей работе находится (с разной вероятностью, определяемой полным набором амплитуд всех клеток) сразу во всех своих состояниях.

Другими словами, при чтении головкой содержимого клетки ленты, выбираются *все* столбцы программы, для которых их символ входит в множество символов в этой клетке, и в каждом таком выбранном столбце выбираются для выполнения *все* клетки программы, у которых ненулевая вероятность для их состояния. На каждом шаге работы это действие выполняется сразу *всеми* головками. Когда для конкретной головки выполняемые клетки требуют «разнонаправленного» движения (налево-направо, налево-стоять, направо-стоять), то головки копируются, а при попадании двух головок в одну позицию ленты они «сливаются» в одну головку.

Здесь важно понять, что на каждом шаге работы может измениться содержание всего обрабатываемого слова на ленте, т.е. изменяется вероятность нахождения конкретных символов в *каждой* позиции обрабатываемого слова. Это показывает мощность одной квантовой операции. Попытайтесь понять, что при увеличении числа головок на единицу, мощность одной операции квантовой машины Тьюринга возрастает в два раза! Для правильно «запрограммированной» машины вероятность нахождения одного из символов в каждой клетке ленты со временем возрастает, что и означает сходимость процесса к определённому выходному слову (ответу).

При попадании в клетку останова квантовая машина Тьюринга «замирает» в одной из только что выполненных клеток, выбор этой клетки носит вероятностный характер, при этом она не обязательно будет именно клеткой останова. Аналогично, выданный ответ также может не быть «настоящим» ответом решаемой задачи. Как уже говорилось, программа квантовых вычислений считается верной, если правильный ответ получается с вероятностью, достаточно близкой к единице.

Для практического описания квантовой ЭВМ квантовая машина Тьюринга неудобна, как, впрочем, и обычная детерминированная машина Тьюринга неудобна для описания традиционных ЭВМ. Как известно, формальной моделью для современных компьютеров служит не машина Тьюринга, а абстрактная машина фон Неймана. Аналогично для квантовых вычислений в качестве формальной модели используется так называемая квантовая схема вычислений, предложенная в 1989 году Дэвидом Дойчем (David Deutsch). Эта схема логически эквивалентна квантовой машине Тьюринга.

В соответствие с этим формализмом, для первых квантовых ЭВМ схема обработки данных на квантовом регистре логически строится как интегральная схема из базовых квантовых вентилей, а не хранится в виде команд в памяти ЭВМ. Для традиционных компьютеров это называется реализация алгоритма «в кремнии» (на интегральной схеме), в противоположность записи программы в виде команд в памяти ЭВМ. В настоящее время, однако, уже появились первые языки достаточно высокого уровня, на которых квантовая схема записывается в памяти традиционной ЭВМ уже в более привычном виде как набор шагов для квантового исполнителя (интерпретатора) этой схемы. В качестве примеров можно привести QCL (Quantum Computing Language), несколько похожий на язык C, а также квантовый язык программирования высокого уровня Quipper. Отметим также «квантовый Ассемблер» фирмы IBM OpenQASM (Open Quantum Assembly Language), он похож на язык Verilog, используемый для описания электронных схем.

Для практически всех людей законы квантовых вычислений (и квантовой физики вообще) трудны для понимания, так как нарушают привычные для нас основы здравого смысла. Так, наиболее известным широкой публике примером объекта в квантовой суперпозиции является знаменитый кот Шредингера, который, пока не произведено измерение его состояния, был «жив и мёртв одновременно». Мы вполне можем принять тот факт, что, если мы посмотрим на кота (а это и

---

[1] Для интереса отметим, что сейчас уже есть и нейронная машина Тьюринга (Neural Turing Machine), основанная на объединении идей машины Тьюринга и машины фон Неймана.

есть измерение его состояния), и кот окажется мёртв, то некоторое время назад он вполне мог быть жив. Наш разум, однако, категорически не может допустить, что, если мы посмотрим на кота, и тот окажется жив, то некоторое время назад он был мёртв (зомби 😵). Впрочем, так как кот тоже сам является измерительным устройством, то этот парадокс скорее всего является надуманным.

Это же непонимание верно и для того небольшого подмножества людей, которые считают себя программистами. Например, всем программистам известно, что для поиска максимума в неупорядоченном массиве из N элементов потребуется не менее N-1 операций сравнения (каждое сравнение двух элементов «отбраковывает» по одному элементу, всего нужно отсеять N-1 элемент). Как говорят, сложность этого алгоритма O(N). Трудно воспринять, но для квантового компьютера сложность этой задачи (так называемый алгоритм Гровера) пропорциональна квадратному корню из длины массива, т.е. $O(\sqrt{N})$. Другими словами, можно найти максимум, не потратив хотя бы по одной квантовой операции для каждого числа такого массива!

После некоторого размышления можно сообразить, почему так происходит. По сравнению с классическим компьютером, каждая операция которого, например, сравнивает значения *двух* элементов массива, мощная квантовая операция задаёт унитарный оператор и изменяет значение сразу *всего* вектора состояний. Можно считать, что последовательность квантовых операций вращает (нормированный) вектор состояния в гильбертовом пространстве, последовательно приближая его к вектору-решению задачи. К сожалению, большинство других квантовых парадоксов понять на основе здравого смысла уже невозможно.

В качестве другого примера рассмотрим знаменитую задачу разложения натурального числа на простые сомножители. Классические алгоритмы разложения двоичного числа длины N бит на простые множители имеет сложность $O(2^N)$, они перебирают все возможные делители от 2 до $\sqrt{N}$. Придуманы и более «хитрые» алгоритмы разложения числа на простые множители, имеющие сложность $O(\exp(n^{1/3}))$.

Рекордом на конкурсе разложения на простые сомножители 155-значного десятичного целого числа (512-битового, принято обозначать RSA-512) в 1999 году параллельно на многих персональных ЭВМ в Интернете было 7 месяцев работы (8 Gflops лет). Разложение 232-значного десятичного числа (RSA-768) на множители ещё можно выполнить на супер-ЭВМ за разумное время (это сделано в 2016 году), но дальше дело пошло хуже. В 2019 году произведено разложение на множители 240 значного числа (RSA-795). Затраченные на это вычислительные мощности оцениваются как 4000 лет работы процессора Intel Xeon Gold 6130 (2,1 Ггц). За разложение на множители числа в 270 десятичных цифр (RSA-896) назначена награда в 75000 долларов, а за число из 309 десятичных цифр (RSA-1024) награда уже в 100000 долларов, но пока эти награды никто не получил.

Лучшие из известных *параллельных* алгоритмов могут разложить на супер-ЭВМ одно 250-значное десятичное число на множители за время порядка 800 тысяч лет, а 1000-значное число – уже за 10^{25} лет. Последняя величина во много раз больше возраста Вселенной, который сейчас оценивается примерно в 13.8 10^9 лет. Именно поэтому так называемое симметричное шифрование с открытым ключом длиной 2048 бит считается сейчас вполне надёжным.

Квантовому же компьютеру с размером регистра порядка 10 тысяч (логических) кубит, для разложения 1000-значного десятичного числа на множители потребуется всего нескольких часов. Грубо говоря, для разложения на множители целого числа длиной N бит требуется квантовый регистр длиной в N кубит. К сожалению, как уже говорилось, это не физические, а логические кубиты. Такой квантовый алгоритм факторизации (разложения на множители) был предложен в 1994 году американским математиком Питером Шором (Peter Shor) [27]. Этот алгоритм позволяет разложить на простые множители n-значное десятичное число за $O(n^3)$ квантовых операций, используя $O(\lg(n))$ кубитов. По существу, с публикации этого алгоритма и начался бум интереса к квантовым вычислениям.

Квантовая схема этого алгоритма ищет не разложение на множители числа N, а период функции $f(x)=a^x \bmod N$, где a – небольшая целая константа, обычно 2 или 3. Ясно, что для случая, когда этот период меньше N, у числа x есть множители. Нахождение самих множителей числа N на основе найденного периода является простой задачей и выполняется уже классическим алгоритмом.

Здесь нужно, однако, сказать, что существует очень много задач, при решении которых квантовые вычисления не дадут никакого выигрыша перед использованием традиционных компьютеров. Это показал профессор кафедры суперкомпьютеров и квантовой информатики факультета ВМК МГУ

Ю.И. Ожигов, он привёл большой ряд алгоритмов, выполнение которых принципиально нельзя ускорить с использованием квантовых вычислений. Отметим, что в настоящее время известно не так много полезных квантовых алгоритмов.

Квантовая ЭВМ D-Wave Systems, 2017 (2000 кубитов)

Первый прототип квантового компьютера, в котором кубиты реализовывались ионами, находящимися в специальных ионных ловушках, работающих при сверхнизких температурах, был разработан австрийскими физиками И. Цираком и П. Цоллером в 1995 году. В 2005 году также на ионных ловушках ученым из Инсбрукского университета в Австрии удалось создать кубайт – квантовый регистр из 8 кубитов.

В 2012 году фирма D-Wave Systems построила квантовый компьютер, с регистром из 512 кубитов, в 2015 она же выпустила компьютер D-Wave 2X с 1000 кубитами, а в 2017 году – уже с 2000 кубитами. Машина D-Wave 2X охлаждается жидким гелием и работает при температуре ниже 15 микрокельвинов (15 мК), это много холоднее, чем в межзвёздном пространстве. Машина содержит 128 тысяч вычислительных элементов, так называемых туннельных переходов Джозефсона.

Это «не настоящая» квантовая ЭВМ, так как она предназначена для решения только одного, хотя и довольно важного, класса задач. Это так называемые адиабатические квантовые вычисления (квантовая нормализация, квантовый отжиг), для нахождения глобального экстремума функции. Кроме того, по мнению некоторых специалистов, «настоящая» квантовая запутанность на этой ЭВМ существует только в пределах каждого кубайта квантового регистра, и практически не распространяется на регистр целиком.[1]

Первые «настоящие» квантовые ЭВМ содержали только несколько десятков кубитов. Например, 50-кубитный квантовый компьютер фирмы IBM System One, построенный в 2018 году и похожий на вычурную люстру, хотя и занимает площадь всего 10 кв.м., но работает в криостате при температуре 10 мК. Для ввода данных на квантовый регистр используется цепочка сверхпроводников, каждый контур этой цепочки всё более холодный, последний и имеет температуру 10 мК. Для получения такой низкой температуры приходится использовать жидкий изотоп гелия-3, которого, к сожалению, на Земле почти в миллион раз меньше, чем гелия-4. Пока основной способ получения гелия-3 – это распад радиоактивного трития, используемого в ядерном оружие. В 2018 году Google Quantum AI lab создан квантовый процессор Bristlecone из 72 сверхпроводящих кубитов, а в 2019 году Google построил 53-кубитную машину.


Квантовая ЭВМ IBM Eagle
127 кубитов

В 2021 году фирма QuantWare начала коммерческий выпуск квантовых 5-кубитный процессоров, а в 2022 году фирма IBM Quantum System One представила серийный компьютер Eagle с 127 кубитами. В 2023 году эта же фирма обещает уже квантовую ЭВМ с 1121 кубитами. В 2022 году фирма IBM создала квантовую ЭВМ Osprey, содержащую 433 кубита, а в планах на 2023 год у IBM уже квантовая ЭВМ Condor с 1121 кубитами.

Вопросы и упражнения

Если вы самый умный человек в комнате, то вы не в той комнате, где должны находиться.

Конфуций, V век до н.э.

1. Что означает свойство дискретности алгоритма ?
2. Что такое аналоговое (непрерывное) представление данных ?
3. Почему аналоговые ЭВМ при решении задачи не выполняют никакого алгоритма ?

---

[1] Мерой квантовой запутанности системы является вещественное число от нуля до единицы. Впервые введена в 1996 году в работе Чарльза Беннетта (C.H. Bennett) (с соавторами).

4. Почему правильнее говорить не «написать алгоритм решения задачи для машины Тьюринга», а «построить машину Тьюринга для решения определённой задачи» ?
5. Как «запрограммировать» аналоговую ЭВМ для решения конкретной задачи ?
6. Каковы достоинства и недостатки аналоговых ЭВМ по сравнению с цифровыми ?
7. Почему все аналоговые ЭВМ являются специализированными, в то время как цифровые ЭВМ, как правило, называются универсальными ?
8. Что такое нейронная сеть, и как она может изменять свою структуру ?
9. Как нейрокомпьютер настраивается на решаемую задачу ?
10. В чём сходство и различие между непрерывными ЭВМ и нейрокомпьютерами по принципам решения задач ?
11. В чём заключается принцип микропрограммного управления ?
12. Что такое микропрограмма ?
13. На каком языке пишутся микропрограммы ?
14. Какие преимущества и недостатки имеет построение ЭВМ на принципе микропрограммного управления ?
15. Что такое компьютеры RISC архитектуры ?
16. В чём заключается принцип программного управления фон Неймана ?
17. К какому классу задач обработки данных относится машина Тьюринга ?
18. Поясните работу компьютеров, которые управляются потоком команд и потоком данных.
19. Что такое обрабатывающий элемент потоковой ЭВМ ?
20. Что является «программой» для потоковой ЭВМ ?
21. В чём заключается принцип максимального параллелизма в обработке данных ?
22. Что такое граф потока данных ?
23. Почему квантовые ЭВМ не являются алгоритмическими системами ?
24. Почему квантовые операции над содержимым квантового регистра много мощнее, чем обычные операции над значением регистра традиционного компьютера ?

---

[i] Большая нейронная сеть говорит, что в последовательности натуральных чисел ошики она не видит, но есть синтаксическая ошибка в самом вопросе, два раза написан артикль "the". Я тоже не увидел этой ошибки ☹.