DOI: 10.32517/2221-1993-2025-24-1-53-63

Е. А. Еремин
Пермский государственный гуманитарно-педагогический университет, г. Пермь, Россия

МНОГОСЛОЙНАЯ МОДЕЛЬ КОМПЬЮТЕРА КАК ОРИЕНТИР ПРИ ОТБОРЕ МАТЕРИАЛА ДЛЯ КУРСА ИНФОРМАТИКИ

Аннотация

Многослойная модель компьютера — это представление его в виде последовательности слоев (уровней): от физического кодирования значений 0 и 1 до прикладного программного обеспечения. Для формирования целостной картины знаний каждый слой имеет определенное значение. В статье традиционно выделяемые слои модели сопоставляются с содержанием сложившегося в школе курса информатики. Показано, что в программе углубленного курса X—XI классов можно найти материал, в той или иной степени связанный с каждым из слоев. Тем не менее в свете возникшей острой необходимости в вычислительной технике отечественного производства и ее элементной базе, удельный вес изучения некоторых слоев следует увеличить. На основе проведенного анализа федеральных рабочих программ школьного курса информатики приводятся конкретные рекомендации по возможному совершенствованию содержания предмета. Статья будет полезна учителям информатики, а также школьникам, интересующимся компьютерами и желающим связать с ними будущую специальность.

Ключевые слова: слой, уровень, многослойная модель, компьютер, школьный курс, информатика, целостность, систематичность.

Информация об авторе

Еремин Евгений Александрович, канд. физ.-мат. наук, доцент, доцент кафедры прикладной информатики, информационных систем и технологий, факультет информатики и экономики, Пермский государственный гуманитарно-педагогический университет, г. Пермь, Россия; *e-mail*: evg_eremin@mail.ru

Достаточно однажды увидеть всю картину целиком, чтобы все стало на свои места, подобно тому как с последним десятком кусочков складывается большая мозаика-головоломка.

Стивен Кинг

1. Введение

Компьютер — одно из фундаментальных понятий курса информатики. О том, что при его изучении удобно пользоваться представлением в виде последовательно добавляемых слоев, известно давно. На наш взгляд, наиболее удачно для знакомства с указанной концепцией подходят классический учебник Э. С. Таненбаума «Архитектура компьютера» [15], а также недавно переизданная объемная книга С. Л. Харрис, Д. Харриса «Цифровая схемотехника и архитектура компьютера: RISC-V» [24] для подготовки разработчиков современной вычислительной техники.

В данной статье многослойная модель компьютера сопоставляется с содержанием школьного курса ин-

E. A. Eremin
Perm State Humanitarian Pedagogical University, Perm, Russia

MULTILAYER MODEL OF COMPUTER AS A REFERENCE POINT DURING SELECTING MATERIAL FOR THE INFORMATICS COURSE

Abstract

The multilayer model of computer is its representation as a sequence of layers (levels): from physical encoding of the values 0 and 1 to application software. Each layer has a specific meaning for forming a complete picture of knowledge. In the article the traditionally distinguished layers of the model are compared with the content of the established at the school informatics course. It is shown that in the program of advanced course for grades 10–11 the material that is to one degree or another related to each of the layers can be found. However, in light of the urgent need for domestically produced computing equipment and its element base, the proportion of studying some layers should be increased. Based on the analysis of the federal work programs for the school informatics course, specific recommendations are provided on possible improvement of the discipline content. The article will be useful for teachers, as well as for pupils interested in computers and wishing to connect with them future specialty.

Keywords: layer, level, multilayer model, computer, school course, informatics, integrality, systematicity.

Information about the author

Evgeniy A. Eremin, Candidate of Sciences (Physics and Mathematics), Docent, Associate Professor at the Department of Applied Informatics, Information Systems and Technologies, Faculty of Informatics and Economics, Perm State Humanitarian Pedagogical University, Perm, Russia; *e-mail*: evg_eremin@mail.ru

форматики. Показано, что традиционно выделяемые в модели слои в той или иной степени в сложившемся курсе присутствуют, хотя внимание к некоторым из них недостаточно (аргументация приводится в завершающей части после описания всех слоев).

Опираясь на свой продолжительный педагогический опыт, мы считаем, что значение основополагающих моделей, которые способны придать целостность содержанию учебного курса, недооценивается. В качестве аргументации сошлемся на глубокий анализ в публикации М. Е. Бершадского [2], где обсуждается роль понимания в усвоении материала и механизм формирования понимания. В частности, там показывается, что «ученик понимает новый материал, если в его сознании известные и новые предметы мышления могут быть соединены с помощью известных ученику видов связей» [там же]. Ситуация дополнительно осложняется тем, что, как свидетельствуют эксперименты, описанные в статье [30], интуитивно складывающаяся в голове студента структура знаний подчас носит весьма причудливый и далекий от систематичности характер. Вместо того чтобы взять за основу формирующейся картины наиболее общие понятия, студенты неосознанно берут наиболее важные с их собственной точки зрения; в роли таких «опорных» пунктов часто оказываются, в частности, яркие (но второстепенные) примеры. Авторы работы [30] назвали этот эффект «paths of lower-cost», что по смыслу лучше всего соответствует русской фразе «по линии наименьшего сопротивления». Из сказанного становится ясно, что наличие в голове ученика некоторой упорядоченной модели, на которую будет «ложиться» новый материал, существенно повышает систематизацию, а значит, и качество знаний.

Тем, кто получил образование в СССР, здесь наверняка вспомнятся известные слова К. Маркса о преимуществе самого плохого архитектора над наилучшей пчелой: человек «с самого начала отличается тем, что, прежде чем строить ячейку из воска, он уже построил ее в своей голове» [9]. В цитируемом высказывании подчеркивается значение внутренней модели внешнего мира как основы любой человеческой деятельности. Эта старая цитата сейчас неожиданно начинает играть новыми красками: как говорилось на пленарном заседании Всероссийского съезда учителей информатики, наличие собственной внутренней модели мира является главным отличием человеческого разума от современного искусственного интеллекта.

Таким образом, данная статья именно про *значение мировоззренческих основ* при планировании школьного курса информатики в противовес имеющемуся «кое-где у нас порой» подходу, когда изучение сосредоточено на *наборе сведений*, необходимых для решения двух-трех десятков типов контрольных задач, разработанных в целях итоговой оценки знаний по предмету.

Знакомство с описанной моделью будет полезно, в первую очередь, учителям информатики: целостное представление о предмете всегда помогает в преподавании. Кроме того, благодаря тщательному отбору материала и доступному его изложению, статья также может быть рекомендована и школьникам, которые интересуются компьютерами, и особенно тем, кто собирается связать с ними свою будущую профессию.

Замечание о терминологии.

В информатике часто используются два очень близких по значению термина — ***уровень*** (*англ.* level, layer) и ***слой*** (*англ.* layer, stratum). Если заглянуть в толковые словари русского языка, то для обсуждаемой сейчас темы лучше всего подойдут следующие формулировки:

Уровень *— это подразделение чего-нибудь целого, получаемое при его расчленении, а* ***слой*** *— это пласт, который лежит между другими пластами и имеет более или менее четкие границы с ними; то, что образовалось поверх другого, что, возникнув позже, покрывает собой другое.*

Видно, что смысл довольно близок. Но все же, по нашему мнению, термин «уровень» немного больше ассоциируется с положением в некоторой разветвленной иерархии объектов, а «слой» — с упорядоченной последовательностью частей определенного назначения и устройства. Именно поэтому модель в названии статьи названа *многослойной*, а не *многоуровневой*, как тоже вполне можно было бы написать.

Данная статья во многом опирается на учебник Э. С. Таненбаума [15]. В его оригинальном английском тексте в предисловии прямо написано: «We may view a computer <…> as a series of *layers* or *levels*, one on top of another», т. е. мы можем рассматривать компьютер как последовательность *слоев* или *уровней*, расположенных один над другим. Так что, следуя классику, оба обсуждаемых термина можно смело использовать как синонимы.

2. Особенности многослойных моделей

Многослойная модель полезна, когда мы имеем дело со сложным объектом. Разобраться во всех его деталях затруднительно (да и нужно ли?), поэтому важно выделить среди них наиболее значимые. При таком подходе, как правило, возникают группы понятий и принципов, которые относятся к некоторой общей области, — это и есть слои или уровни. Каждый слой включает в себя определенные базовые объекты, их функции и закономерности поведения. Каждый новый слой надстраивается над предыдущим и опирается на него. Строго говоря, для понимания работы более высокого слоя не обязательно знать все особенности нижележащего, но для формирования адекватной картины крайне полезно иметь представление о наиболее важных из них. Как писал М. Бен-Ари, «в каждом отдельном курсе вы будете учить определенному уровню абстракции; вы должны представить в явном виде жизнеспособную модель на один уровень ниже, чем тот, который вы учите» [27].

Если же требуется не просто освоить некоторый набор практических навыков, а сформировать общее систематическое представление об объекте, тогда понадобятся (хотя бы самые минимальные) сведения обо всех его уровнях. Здесь мы касаемся предмета постоянных дискуссий о том, насколько нужно понимать работу компьютера, чтобы им грамотно пользоваться. Сошлемся на мнение авторитетного специалиста, британского ученого Б. дю Бюлей: «Даже если не делается попыток представить картину того, что происходит “внутри”, ученики сформируют свою собственную. Без помощи [*учителя*] их картина будет довольно бедной, базирующейся на совпадениях и, может быть, недостаточной, чтобы объ-

яснить бóльшую часть наблюдаемого поведения. Пока все идет хорошо, это, возможно, не имеет значения, но понимание ошибок часто требует знаний вида “машина пыталась сделать это, ... но наткнулась на то-то ... и поэтому не знает, как продолжить”» [29].

При переходе от одного слоя к другому активно используется **принцип абстракции**, когда, не принимая во внимание второстепенные детали, удается свести сложный предмет к более простому для понимания. Как и у любого полезного средства, кроме положительного эффекта часто возникают и побочные. В статье [26] принцип упрощения путем создания многочисленных «защитных слоев» несколько иронично назван «капустным»: «Один из эффектов “кочана” заключается в снижении квалификационных требований к специалистам массовых профессий, причем не только к тем, кто занимается эксплуатацией информационных систем, но даже и к тем, кто ведет массовые направления разработок. Логическим результатом прогресса в таком нетребовательном к уровню образования направлении становится преимущественное положение “монтерских” знаний в сравнении со знаниями академическими. <...> Надо сказать, что ИТ не одиноки в этом отношении, <...> например, нечто подобное можно обнаружить, если сравнить электротехнику с соответствующими разделами физики».

Посмотрим на обсуждаемую тему под другим углом. Добавляя уровни абстракции, мы упрощаем картину и облегчаем ее понимание. С одной стороны, это, бесспорно, хорошо. С другой стороны, при этом мы перестаем видеть и теряем возможность использовать многие полезные свойства нижних слоев. Чтобы нагляднее это себе представить, вспомним языки программирования высокого уровня. Да, их гораздо легче освоить, чем встроенный язык процессора, на высоком уровне легче и быстрее пишется программа. Но зато при этом из-за появления дополнительных слоев падает быстродействие (как тут не вспомнить «медленный» Python), а из-за «удаления» от аппаратной части компьютера мы теряем возможность ею эффективно управлять. Так что какие-то знания о нижних слоях все же могут приносить пользу.

3. Структура многослойной модели компьютера

Перейдем теперь непосредственно к рассмотрению многослойной модели компьютера. Внимательно проанализировав уровни, предложенные в упомянутых выше учебниках [15] и [24], применительно к школьному курсу информатики, мы предлагаем сформировать модель из следующих слоев (рис. 1).

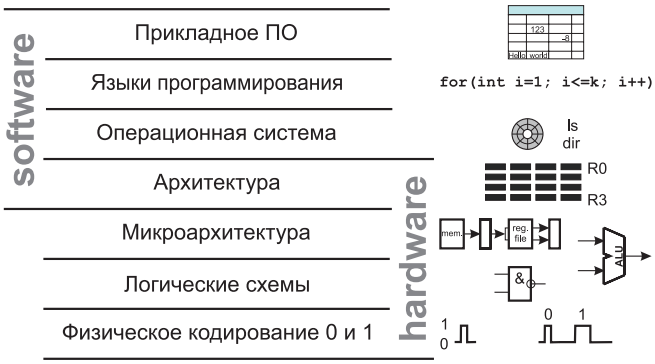

Рис. 1. Многослойная модель компьютера

Далее каждый слой модели будет рассмотрен подробнее.

Заметим, что аналогичная многоуровневая модель от физической передачи сигналов до взаимодействия прикладных программ существует и для компьютерных сетей; она называется OSI — Open Systems Interconnection (модель взаимодействия открытых систем) [10].

3.1. Физическое кодирование 0 и 1

Физическое кодирование — это методы преобразования нулей и единиц в физические сигналы (сейчас чаще всего электрические, но можно брать за основу и более современные оптические, а также устаревшие механические или гидравлические методы двоичного представления). Проще говоря, это способ, с помощью которого хранятся или передаются (что, как мы скоро увидим, не совсем одно и то же) нулевое и единичное значения каждого из битов. Очевидно, что данный слой устройства компьютера с мировоззренческой точки зрения необычайно важен, хотя непосредственно на практическую работу пользователя на компьютере почти не влияет.

Самый простой и самый распространенный способ физического кодирования состоит в том, что каждому из двух логических значений (0 или 1) ставится в соответствие определенный уровень напряжения. Важно подчеркнуть, что ради повышения надежности распознавания всегда допустимо некоторое отклонение от стандартного значения (рис. 2).

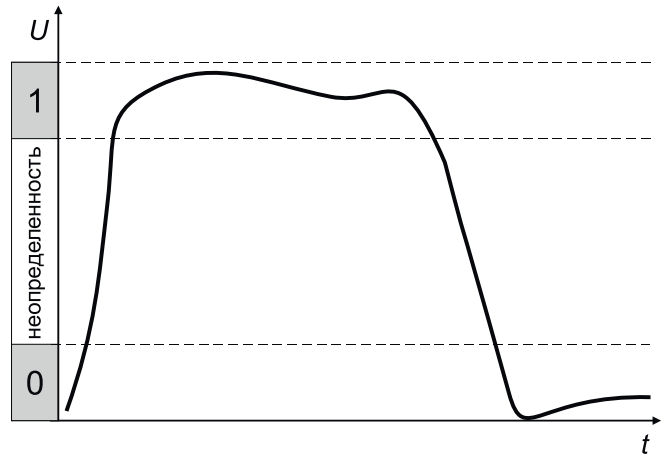

Рис. 2. Соответствие уровней напряжения с логическими 0 и 1

На рисунке 2 сигнал некоторое время попадает в область неопределенности (между 0 и 1). Это так называемый «переходный процесс», влияние которого легко исключается путем синхронизации по времени: считывание уровня включается с некоторой задержкой и выключается через достаточный для распознавания промежуток времени. В результате в момент переключения распознавание просто-напросто «не работает». Практическая реализация описанного способа не вызывает затруднений у инженеров.

Интересно отметить, что величина напряжения (в том числе даже знак!) часто зависит от выбранных ло-

ISSN 2221-1993 • **Informatics in School** • 2025 • Volume 24 No 1

гических микросхем [24]. Для нашего разговора о слоях это не совсем удобно, поскольку логические схемы на данном уровне пока еще не рассматривались.

Обсуждение вопросов физического кодирования также невозможно без разговора об аналоговых и цифровых сигналах. По сути, рассмотренный нами рисунок 2 показывает, как работает распознавание дискретных значений 0 и 1 в «непрерывном», т. е. аналоговом, диапазоне значений напряжения.

А теперь рассмотрим физическое кодирование при обмене данных между компьютерами. Тут появляется дополнительная проблема: если внутри одного компьютера все процессы строго синхронизированы (за этим следит устройство управления), то при передаче сигнала между разными компьютерами все сложнее. Представим себе, что 0 и 1 по-прежнему кодируются низким и высоким напряжением, как на рисунке 2. Очевидно, что здесь невозможно отличить полное отсутствие сигнала от передачи нуля — в обоих случаях напряжение низкое. На помощь приходит импульсное кодирование, при котором 0 и 1 представлены не уровнями напряжения, а переходом от одного уровня к другому. Понятно, что это снимает проблему опознавания отсутствия данных, поскольку в последнем случае изменений вообще нет. Приведем в качестве примера широко известный манчестерский код [17], в котором единица кодируется перепадом от 1 к 0, а ноль, напротив, переходом от 0 к 1. Кроме того, существуют и другие способы кодирования передаваемых данных: например, можно различать 0 и 1 по длительности импульса или по частоте, т. е. по количеству перепадов, зарегистрированных за стандартный промежуток времени [7].

3.2. Логические схемы

С переходом к следующему слою мы отвлекаемся от физических основ кодирования данных и оперируем исключительно абстрактными величинами 0 и 1. Такой подход дает полное право применять к анализу работы логических схем сугубо теоретическую науку под названием «Алгебра логики».

Для обработки двоичных данных используются самые разнообразные логические схемы. Простейшими из них являются *логические элементы*, каждый из которых выполняет элементарную логическую операцию над одним битом.

Выбор базовых элементов для вычислительных устройств является самостоятельной задачей. Это в теории можно взять за основу любые понравившиеся логические элементы. А для практического конструирования необходимо принять во внимание, что эти элементы придется изготовлять в больших количествах. Отсюда, в частности, вытекает, что базовые элементы должны быть максимально простыми, а их разновидностей должно быть как можно меньше.

Классический набор логических элементов — это И, ИЛИ, НЕ. Доказано, что на их основе можно построить любую сколь угодно сложную логическую функцию. Но, как оказывается, совсем не обязательно брать за основу *три* элемента — достаточно *одного*, правда, комбинированного. Таким универсальным базисом обычно служит элемент И-НЕ (отрицание конъюнкции, так называемый *штрих Шеффера*): используя его, несложно построить любой из трех перечисленных выше элементов [15, 18] и, следовательно, любую логическую функцию. Для электроники это означает, что любую логическую схему можно спроектировать на базе одного типового элемента. Попутно отметим, что схема И-НЕ требует меньше транзисторов, чем И [15].

Аналогичными свойствами обладает операция ИЛИ-НЕ (*стрелка Пирса*).

От традиционного обсуждения логических элементов, таблиц их истинности и практического использования (в частности, для сброса или установки необходимых битов) нетрудно перейти к более сложным логическим схемам, которые применяются в вычислительной технике. В школьном курсе обычно разбирают *триггер* и однобитный *сумматор* [1, 11]. Первый, будучи основой *регистра*, дает некоторое представление об устройстве памяти, а второй — об идеях работы *арифметико-логического устройства* (АЛУ). В качестве «информации к размышлению» для наиболее любознательных читателей призываем попутно задуматься над следующим фактом. Традиционный алгоритм сложения является последовательным, т. е. сначала складываются младшие разряды слагаемых, затем (с учетом возможного переноса) следующие, более старшие и т. д. Можно ли как-то ускорить этот процесс, обеспечив параллельное (одновременное) сложение нескольких разрядов сразу? Если потребуется подсказка, то наберите в поисковой машине запрос «*ускоренный перенос*».

3.3. Микроархитектура и архитектура

Поняв основные идеи, по которым строятся логические схемы компьютера, можно перейти к рассмотрению устройства компьютера в целом. Удобно обсуждать два следующих слоя вместе, поскольку они тесно связаны. К тому же уровень микроархитектуры для школьного курса информатики представляет не самый большой интерес и может считаться некоторым дополнением к важному уровню архитектуры.

Итак, что же такое архитектура? Самый простой, но весьма глубокий ответ можно сформулировать так: ***архитектура*** *— это то, как видит hardware программист*. Причем подчеркнем, что речь идет о профессиональном системном программисте, а не о студенте, начинающем изучать язык высокого уровня. Э. С. Таненбаум, предвидя возможные возражения по этому поводу, предлагает для четкости говорить о возможностях компилятора [15], но мы не будем усложнять наше обсуждение.

Для предотвращения путаницы надо обязательно подчеркнуть, что устоявшийся в компьютерной литературе термин «фон-неймановская архитектура» (в честь выдающегося математика Дж. фон Неймана, описавшего совместно с А. Берксом и Г. Голдстейном классические основы построения вычислительных устройств [28]) не вполне соответствует приведенному выше определению. Дело в том, что сформулированные в этом актуальном до сих пор труде фундаментальные принципы слишком общие и допускают различную конкретизацию. Приведем в качестве примера два варианта реализации архитектуры процессора — CISC и RISC [14], которые оба являются фон-неймановскими.

Что же в таком случае образует понятие архитектуры в общепринятом смысле? Прежде всего, это

ISSN 2221-1993 • **Информатика в школе** • 2025 • Том 24 № 1

система команд компьютера (точнее говоря, его процессора). Как образно пишет В. Столлингс, «областью, в которой “встречаются” программист и конструктор компьютеров, является набор машинных команд» [14]. Кроме того, сюда же необходимо добавить ряд связанных деталей, а именно: обрабатываемые данные и их форматы, структура собственной памяти процессора (его регистры), особенности организации ветвлений программ, методы адресации к памяти, способы обмена данными с внешними устройствами и некоторые другие важные вещи. Таким образом, мы видим, что ***архитектура** есть не что иное, как **самое общее**, очищенное от второстепенных подробностей, **описание устройства компьютера***.

Кроме того, есть еще одно обстоятельство, которое делает архитектуру достойной рассмотрения в образовательном курсе. Дело в том, что архитектура — это *наиболее стабильное и неизменное* описание компьютера. Обоснование данного утверждения достаточно простое: пользователи программного обеспечения (ПО) хотят максимального сохранения работоспособности привычных для них программ на новых моделях компьютеров. А это значит, что разработка ПО должна опираться на какие-то базовые неизменные основы, которые как раз и есть архитектура.

Рассмотрим в качестве примера семейство процессоров Intel. До недавнего времени их совместимость обеспечивалась благодаря соблюдению тщательно документированной 32-битной архитектуры IA-32 (IA — сокращение от Intel Architecture). Все более интенсивное использование возможностей 64-битных процессоров породило новую, 64-битную, архитектуру Intel64 (не путать с IA-64). Переход к новой архитектуре не всегда гладкий: часть очень старых программ перестала работать под новыми 64-разрядными версиями Windows.

Важной частью архитектуры является кодирование команд, т. е. правила их записи в форме набора нулей и единиц. Увидеть хотя бы раз такое представление команд необычайно важно для мировоззрения, поскольку без подобных примеров все утверждения о том, что программа состоит из двоичных кодов, выглядят голословно. К сожалению, в распространенных Intel-совместимых процессорах кодирование устроено достаточно сложно. В то же время в более современной архитектуре RISC-V (которая из всех имеющихся вариантов лучше всего подходит для реализации в отечественной вычислительной технике [23]) разобраться в двоичной записи нескольких простейших инструкций процессора вполне возможно [24]. Материалы по стандарту RISC-V можно найти на официальном сайте (https://riscv.org/), а также на сайте российского Альянса RISC-V (https://riscv-alliance.ru/).

Как подчеркивал Э. С. Таненбаум [15], нет четкой границы между включением инструкции в систему команд и оформлением ее в виде служебной программы. Прекрасным примером этого может служить наличие или отсутствие в разных моделях процессоров встроенных инструкций целочисленного умножения и деления, а также вещественной арифметики.

Из сказанного выше о сути архитектуры отчетливо видно, что она есть *теоретическое* описание компьютера. Реализовать *на практике* такое обобщенное описание можно различными способами. И тут на сцену выходит слой *микроархитектуры*, который является связующим звеном между логическими схемами и архитектурой. Здесь описывается, как именно в процессоре расположены и соединены друг с другом его функциональные узлы, необходимые для реализации архитектуры. Очевидно, что у каждой архитектуры может быть много различных микроархитектур, которые обеспечивают разную сложность, производительность и цену. Все они смогут выполнять одни и те же программы, но их внутреннее устройство может очень сильно отличаться.

Например, в неоднократно цитированном ранее учебнике [24] для современной свободно распространяемой архитектуры процессоров RISC-V рассматриваются три варианта реализации: однотактный, многотактный и конвейерный процессоры. Обсуждение особенностей каждого варианта увело бы нас слишком далеко. Скажем только, что конвейерное (параллельное) выполнение нескольких команд очень широко применяется в современных процессорах.

Подчеркнем, что архитектура и микроархитектура разрабатываются разными специалистами, что важно с точки зрения профориентационных целей.

Итак, мы видели, что слой микроархитектуры является чисто технологическим и для учебного курса не так интересен. Зато слой архитектуры, напротив, очень важен. Мы вернемся к этому вопросу, когда закончим рассмотрение оставшихся слоев модели.

3.4. Операционная система

Теперь представим себе компьютер, у которого реализованы все предыдущие слои. Мы имеем машину, которая уже способна выполнить программу, но только такую, что представлена в виде кодов команд из нулей и единиц. (При некоторой фантазии можно допустить, что программа вводится в виде восьмеричных цифр, как это было в школьных компьютерах УКНЦ, где такой ввод обеспечивался встроенной программой-монитором.)

Попутно предложим читателям любопытный вопрос: а вы не задумывались над тем, что абсолютно верные утверждения «компьютер всегда работает по программе» и «при включении питания в ОЗУ нет никакой программы» как-то плохо согласуются между собой? Разгадка здесь в том, что в любом современном компьютере есть стартовая программа, которая хранится в виде «готового к употреблению» двоичного кода в специальном устройстве памяти — ПЗУ (постоянном запоминающем устройстве). Именно она и начинает выполняться в момент включения компьютера. Стартовая программа умеет немного: протестировать компьютер и определить наличие у него наиболее важных устройств, а затем инициировать процесс загрузки программного обеспечения в ОЗУ. И что же ищется и загружается в первую очередь? Конечно же, операционная система (ОС), которая и образует следующий слой, обеспечивающий работу компьютера.

Таким образом, в нашем рассмотрении появляется первый слой, который не встроен жестко в конструкцию компьютера, а *загружается* в него. Иными словами, у пользователя теперь есть выбор, какую именно операционную систему поставить (а может, и какие несколько ОС).

Можно ли записать ОС в ПЗУ в качестве стартовой программы? Технически при современных возможностях производства микросхем это решаемая задача. Но так не делают, чтобы не потерять гибкость. И дело даже не столько в том, что продавать компьютеры с «зашитой» версией Windows 10 или Linux Ubuntu станет сложнее. Гораздо важнее, что существенно усложнится обновление ОС: при покупке, скажем, новейшего принтера вам, скорее всего, придется вызывать мастера для перепрограммирования ПЗУ с операционной системой. Не самая привлекательная перспектива...

Зачем нужна ОС и почему без предустановленной операционной системы практически не продаются компьютеры?

История развития ОС весьма поучительна. В первом поколении вычислительной техники операционных систем не было [16]. Программу загружал человек, а после ее выполнения и выдачи результатов машина останавливалась в ожидании ввода новой задачи. Сначала такой режим был приемлемым, тем более что лампы, на которых тогда были построены ЭВМ, часто выходили из строя, из-за чего требовался ремонт. Но с переходом ко второму поколению машины постепенно стали более надежными, так что люди начали задумываться по поводу долгого простоя дорогостоящего оборудования при переходе от одной задачи к другой. Появился так называемый *пакетный режим*, при котором сразу после завершения вычислений одной задачи, пока печатались результаты, автоматически загружалась следующая. Руководили этим процессом специальные управляющие *программы-мониторы*. Так сложилось, что третье поколение представляло собой огромные машины для коллективного пользования. Для них организация многозадачной работы стала еще более насущной, что и привело к появлению операционных систем, управляющих эффективным прохождением задач через машину. Заметим, что хотя компьютеры четвертого поколения в основном персональные, т. е. ими обычно пользуется только один человек, но они по-прежнему многозадачны: одновременно принимают набираемый текст, параллельно проверяя его синтаксис и стиль, воспроизводят музыку, «закачивают» обновления (часто даже без вашего ведома!) и делают параллельно еще много чего. Итак, *стремление к эффективному выполнению задач* служит главной причиной появления ОС.

Была и еще одна причина появления операционных систем. О ней вспоминают реже, что не снижает ее значимости. Дело в том, что до появления ОС все внешние устройства были достаточно примитивными: устройства ввода с перфокарт, перфолент и магнитных лент имели ограниченные возможности, и управление ими легко программировалось. Ситуация коренным образом изменилась с появлением магнитных дисков, где доступ к данным осуществляется значительно сложнее. Для упрощения ситуации была придумана идеология *файловой системы*, когда достаточно указать символическое имя файла, а система, пользуясь своими служебными данными, найдет и прочитает с диска нужные сектора. Кстати говоря, порядок чтения секторов тоже можно оптимизировать, что способна делать ОС.

Связь ОС с наличием дисковых накопителей наиболее отчетливо видна на примере микро-ЭВМ (персональных компьютеров), начиная с самой первой операционной системы CP/M [16]. В то время даже существовал термин «дисковая операционная система», фиксирующий факт такой связи.

Помимо названных выше функций реализации многозадачности и поддержки файловой системы операционная система имеет и еще целый ряд важных функций. Чтобы не загромождать изложение, выделим среди них только одну, имеющую непосредственное отношение к обсуждаемой многослойной модели. Речь идет о создании интерфейса между слоем прикладного ПО и аппаратным обеспечением; часто такой интерфейс называют API — Application Program Interface (программный интерфейс приложения). Благодаря ему прикладной программист, которому требуется, скажем, напечатать на бумаге текст, просто вызывает стандартную функцию API. А все необходимые действия по управлению выводом на конкретную разновидность принтера берет на себя ОС.

Таким образом, операционная система может рассматриваться как расширенная (еще одним слоем) машина, управлять которой значительно проще, чем это делается на уровне «железа». «Операционная система превращает уродливое аппаратное обеспечение в красивые абстракции», как несколько иронично гласит подпись под рисунком 1.2 в книге [16].

3.5. Языки программирования

На данном уровне обсуждения мы имеем машину, пригодную для разработки программного обеспечения. Сразу заметим, что возможность сохранения текста программы и результатов ее трансляции в файлы, которая обеспечивается слоем ОС, играет здесь не последнюю роль.

Очевидно, что для разработки новой программы необходима программа-ассистент, которая помогает преобразовать написанный человеком текст на некотором языке в набор нулей и единиц, понятный процессору. Такая служебная программа называется *транслятором*, т. е. переводчиком (с некоторого внешнего языка на «единственно правильный» язык машинного кода из 0 и 1). Существуют две базовые стратегии трансляции — *интерпретация* и *компиляция*; возможны также их комбинации. Вопрос этот многократно описан в литературе, так что сейчас мы не будем на нем останавливаться.

Языков программирования — великое множество. Наиболее простой из них (не случайно его называют *языком низкого уровня*) — **ассемблер**. Он позволяет записать последовательность машинных команд не в виде 0 и 1, а как некоторый более удобный для человека текст. Подчеркнем, что каждая строка ассемблерной программы преобразуется в двоичный код *одной* машинной инструкции.

Как выглядит запись команды на ассемблере? Например, так:

```
ADD x31, x6, x7
```

Перед нами команда сложения содержимого регистров x6 и x7, результат которой сохраняется в x31.

Часто ассемблер трактуют как язык, в котором двоичные коды заменяются определенным текстом. При этом говорится, что осмысленное (английское) слово

ADD гораздо легче запомнить, чем двоичное число. Не будем сейчас дискутировать по поводу простоты запоминания, особенно для тех, кто не знает английский (для примера взгляните на пару реальных команд: AUIPC и SLTIU). Подчеркнем только, что главное достоинство ассемблера совсем не в этом — гораздо важнее то, что адреса ячеек памяти можно заменять символическими именами (их называют *метками* или *идентификаторами*). Дело в том, что заранее удачно предугадать конкретные адреса в ОЗУ не так просто; особенно трудно бывает переделать программу в случае, когда была допущена ошибка и для ее устранения требуется переместить часть кода по памяти. Зато ассемблер справляется с этой задачей прекрасно. Он сначала планирует размещение двоичного кода в памяти, а затем составляет таблицу соответствия между метками и адресами памяти. Пользуясь этой таблицей, ассемблер быстро и безошибочно подставляет вместо меток актуальные числовые адреса. Естественно, что в случае любого изменения в тексте программы описанная процедура без труда повторяется заново.

Судите сами, насколько это важно. Если поставить метку в строке, где описывается место памяти для хранения *данных*, то эта метка фактически станет именем *переменной*. Если же помечена ячейка в *программе*, то на это место можно сделать переход, в том числе и условный. В результате получаем средство для организации *развилки*, *цикла* или *процедуры*, т. е. любой алгоритмической структуры.

Ассемблер незаменим в двух случаях: во-первых, когда необходимо написать эффективный и быстро работающий машинный код, а во-вторых, если нужен доступ к аппаратной части компьютера. Как вы догадываетесь, речь идет, в первую очередь, о драйверах и системных программах.

Конечно, и недостатки у ассемблера есть. Прежде всего, он *машинно-зависимый*, т. е. для каждого семейства машин свой. (В скобках заметим, что, зная и понимая один ассемблер, перейти к другому не так сложно.) Кроме того, программирование на ассемблере весьма трудоемко.

Поэтому естественным образом возникли **языки высокого уровня**, такие как Pascal, Java, Python и др. В них вводятся (машинно-независимые) правила записи алгоритмических конструкций, таких как ветвление, цикл, процедура и т. д. Компилятор анализирует текст программы и составляет подходящие блоки из машинных инструкций, которые реализуют соответствующие структуры для данного процессора. В итоге получается двоичный код, который сохраняется в виде исполняемого файла. Последний, в свою очередь, может быть запущен операционной системой.

Имея средства для программирования, можно создавать любое прикладное программное обеспечение. Но с точки зрения нашего обсуждения имеет особый интерес разработка системного ПО, особенно операционных систем. Трудность здесь в том, что для работы стандартного компилятора нужна ОС, а для реализации ОС нужен компилятор. Как же быть? Вот, например, как описывается процедура рождения ядра ОС Unix для машины PDP-7 [4]. Сначала *на другой машине* GE-635 готовилась бумажная лента с ассемблерным текстом программы, читаемая PDP-7. Затем эту ленту переносили и вводили в машину PDP-7. Введенная исходная программа транслировалась «автономным» ассемблером, который генерировал выходной файл и запускал его. Так продолжалось до тех пор, пока не были завершены примитивное ядро Unix, редактор, простой командный интерпретатор и несколько самых необходимых утилит (вроде команды Unix для копирования файлов).

Но затем Д. Ритчи придумал принципиально новый подход: он создал язык Си, который по возможностям не уступал ассемблеру, но в то же время содержал базовые конструкции языка высокого уровня (такой язык называют *машинно-ориентированным языком промежуточного уровня*, т. е., проще говоря, сильно расширенным ассемблером). В результате разработка ОС Unix стала возможной на языке Си. Язык этот и по сей день широко используется для написания системного ПО, в том числе операционных систем.

Сам Д. Ритчи позднее писал про свое детище: «Он, безусловно, удовлетворил потребности в таком языке реализации систем, который был достаточно эффективным, чтобы заменить ассемблер, и в то же время достаточно абстрактным и гибким, чтобы описывать алгоритмы и взаимодействия в широком спектре программных сред» (цит. по [3]).

Приведенные исторические подробности показывают, с каким трудом иногда приходится добавлять новые слои программного обеспечения.

3.6. Прикладное ПО

Наконец, последний, знакомый каждому пользователю слой прикладного программного обеспечения. Именно с ним чаще всего имеют дело в повседневной жизни. По этой самой причине особой необходимости в описании данного слоя нет. Заметим только, что для того, чтобы ваша любимая прикладная программа была реализована и заработала, необходимы все описанные ранее слои.

* * *

Итак, мы рассмотрели все существующие на сегодняшний день слои. Появятся ли новые? Несомненно! Уже сейчас видны контуры мощного ИИ-слоя (слоя искусственного интеллекта), который по мере своего развития почти наверняка «спрячет» под собой все программные слои. Быть может, тогда станет излишним учиться программировать и осваивать графический интерфейс прикладного ПО.

Интересно вспомнить, что современные достижения ИИ очень похожи на то, что обещали сделать (и до конца в свое время не сумели) японские ученые в ЭВМ пятого поколения: управление голосом, формулировка результата на естественном языке, самостоятельный выбор метода решения задачи и т. д. Впрочем, не будем опережать время и забегать слишком далеко вперед...

4. Сопоставление с содержанием школьного курса

Теперь, когда мы познакомились со всеми слоями модели компьютера, посмотрим, как эту модель целесообразно «спроектировать» на содержание школьного

ISSN 2221-1993 • **Informatics in School** • 2025 • Volume 24 No 1

Таблица

Сопоставление слоев модели и содержания школьного курса информатики

Слой	VII—IX базовый [19]	VII—IX углубленный [20]	X—XI базовый [21]	X—XI углубленный [22]
Физическое кодирование 0 и 1	Дискретность данных; двоичный код (алфавит); бит — двоичный разряд; логические элементы		+ ***передача***: сигнал, кодирование; хранение информации в ОЗУ	+ дискретные сигналы; дискретизация информации в цифровых системах
Логические схемы	Логические элементы; логические основы компьютера; поколения ЭВМ	+ сумматор	+ триггер; логическая схема по выражению и наоборот	+ многоразрядный сумматор; *микросхемы и технология их производства*
Микроархитектура (*)	Современные тенденции; параллельные вычисления; суперкомпьютеры; двоичная арифметика, логические операции (***АЛУ***)		+ многопроцессорные системы	+ автоматическое выполнение программы; шина
Архитектура	Основные функциональные блоки компьютера; двоичное представление (***форматы***) данных		+ хранение, объем памяти (***байт*** — ячейка); представление вещественных чисел (***форматы***)	+ архитектура Неймана и гарвардская; контроллеры, прямой доступ к памяти; переполнение, длинная арифметика
Операционная система	Файлы и папки; работа с ними	+ принципы построения файловых систем	+ операционная система (как разновидность ПО); администрирование	+ утилиты и драйверы; принципы размещения и именования файлов; шаблоны групп файлов
Языки программирования	Язык; система программирования; базовые конструкции; отладка		+ типы данных	+ компиляция и интерпретация; ООП; RAD-среды; обзор языков
Прикладное ПО	Состав ПО; примеры программ; практическая работа с прикладным ПО			

курса. Позиция автора заключается в том, что образованный ученик должен получить некоторое (пусть минимально необходимое) представление о *каждом* из перечисленных слоев.

Проанализируем, как обстоит с этим дело в сложившемся в школе курсе информатики. Для этого внимательно прочитаем федеральные рабочие программы 2023 года, доступные на сайте «Единое содержание общего образования» [19–22]. Перечисленные в них вопросы, связанные с каждым из выделенных слоев, выпишем в таблицу. В ней выделены столбцы, соответствующие требованиям к содержанию базового и углубленного курсов VII—IX и X—XI классов, что соответствует четырем действующим федеральным рабочим программам.

Текст, помеченный в таблице серым фоном, соответствует вопросам, где *можно* дать сведения по соответствующему слою. Но именно *можно*, особенно если знать о существовании этих связей. Например, рассказ о двоичности данных можно сопроводить сведениями об идеях кодирования 0 и 1 (но программы на этом не настаивают и даже не намекают на это). Поколения ЭВМ строятся на основе элементной базы, так что здесь будет уместно дополнение о том, что такое микропроцессоры и как их производят. Двоичное (и особенно байтовое) представление данных существенно влияет на архитектуру процессора. Мимоходом заметим, что даже в программе углубленного курса не упоминается о существовании в процессоре специально выделенного блока для обработки вещественных чисел — *математического сопроцессора*. А это важно, поскольку без такого обсуждения не каждый школьник обратит внимание на фундаментальный факт, что обработка вещественных чисел существенно отличается от обработки целых.

Из таблицы отчетливо видно, что «все серое» сосредоточено в верхней части таблицы, т. е. именно здесь возможна потеря важных сведений. Напрашивается вывод, что школьный курс слишком «накренен» в сторону практических применений компьютера и слабо освещает базовые идеи его устройства. Если следить за эволюцией содержания школьного курса, то это не удивляет.

Проанализируем содержание приведенной таблицы подробнее. Начнем с нескольких принципиальных замечаний.

Во-первых, только в углубленном курсе X—XI классов, и то курсивом (необязательный материал), есть явная ссылка на «*микросхемы и технологии их производства*». Иными словами, при изучении современного курса информатики даже просто сообщить школьникам о существовании такой отрасли, как производство чипов, по мнению программы, совсем не обязательно. Не

тут ли лежит корень нашего отставания в данной без преувеличения стратегической области?

Во-вторых, из программы начисто исчезло фундаментальное понятие «*система команд*». Причем ладно бы речь шла только о системе команд процессора, но и у исполнителей она тоже не упоминается! Между тем в традиционной методике преподавания информатики было принято, что «основной характеристикой исполнителя <...> *является система команд исполнителя* (СКИ)» [8]. Отсюда же, по-видимому, вытекает и невнимание к архитектуре компьютера, которая всегда начинается с системы команд процессора.

В-третьих, слово «*многоядерный*» также начисто отсутствует в программах курса. Важно ли это? Думается, в современных условиях очень даже важно. Купить в магазине новый компьютер с *одно*ядерным процессором уже невозможно, а понятных разъяснений, как работает *много*ядерный процессор и в чем его реальные преимущества, как говорится, «днем с огнем...». И это при том, что вопрос о параллельных вычислениях в программе прописан в явном виде.

И, наконец, еще одно замечание общего плана, не привязанное к конкретным строчкам программы, но навеянное общим анализом содержания программ [19–22]. Стратегия обучения школьников такова, что нынешний курс информатики незаметно распадается на две относительно самостоятельных части. С одной стороны, строгая абстрактная теория (двоичная система и арифметика для нее, алгебра логики и т. д.), хорошо подходящая для проверки на ЕГЭ. С другой — практические операции (работа с файлами средствами ОС и с прикладными пакетами, написание простых программ на языке высокого уровня и т. д.). Без сомнения, при тщательном обдумывании материалов урока знающий учитель сможет построить «мостики» между этими «половинками». Но располагает ли к этому программа?

Критиковать легче, чем предлагать пути преодоления недостатков. Поэтому приведем вполне конкретный пример, как видятся пути сближения названных частей курса. В частности, очень полезно, наглядно и интересно продемонстрировать школьникам на практике конкретные случаи хранения в памяти двоичных данных. С помощью языков программирования это удается сделать, экспериментируя с (двоичным!) содержимым переменных [6, 13, 25]. Вероятно, не очень сложно написать некие учебные приложения, которые отображают ту или иную информацию из памяти в двоичной форме. (Хотя это будет менее убедительно: откуда известно, что программа выводит данные именно в том виде, как они лежат в ОЗУ?) В любом случае, требуется внимание к проблеме, которое пока не чувствуется.

Отмеченные особенности, по нашему мнению, говорят о недостатках в стратегическом планировании современного школьного курса.

Еще раз обратимся к таблице и поищем в ней положительные моменты. Есть ли удачные совпадения между программой и изложенной многослойной моделью? Да, несомненно! Прежде всего, это три последние строки таблицы (операционная система и далее). Еще вполне представительно выглядит слой логических схем.

Если сосредоточить внимание только на последнем столбце, то обнаруживается неплохое соответствие по всем слоям. Исходя из этого, можно сказать, что углубленный курс X—XI классов [22] охватывает рассматриваемую многослойную модель вполне удовлетворительно.

На наш взгляд, наиболее «обделен вниманием» важный слой архитектуры. Поэтому обсудим его подробнее.

Начнем с того, что слой служит связующим звеном между hardware и software, что имеет важное методологическое значение для понимания взаимодействия между названными базовыми компонентами компьютера. Недостаточное внимание к вопросам, связанным с указанной тематикой, может приводить к непроизвольному разделению курса информатики на две слабо связанные части: теоретическую (про двоичную систему и логические элементы) и практическую (работа с пакетами и основы программирования) [5]. В результате многие ученики теряют из виду естественную связь между hard и soft и порой задают по поводу теории традиционный вопрос: «Зачем (или кому) это нужно?» И в самом деле: если на уроках не обращалось внимания на кодирование программы и данных, то для чего было изучать двоичную систему? Тем более непонятно, что с практической точки зрения дают разговоры об основах двоичной арифметики, если даже в [22] отсутствует термин «арифметико-логическое устройство».

Наконец, несколько личных впечатлений по поводу реакции аудитории на выступления по обсуждаемой тематике на нескольких последних всероссийских конференциях по преподаванию информатики. С сожалением можно констатировать, что предложения о необходимости знакомства в школьном курсе информатики с отдельными вопросами устройства компьютеров и современными специальностями, связанными с производством вычислительной техники, часто воспринимаются негативно. А как же без этого можно рассчитывать на развитие в этой области? Согласитесь, что все-таки надо не только ориентировать школьников на профессии, связанные с программированием и применением прикладного ПО, но и обращать внимание учащихся на перспективность работы в областях, связанных с проектированием и производством новых вычислительных устройств. Политика «Intel inside» по известным причинам дает сбои, а значит, жизненно необходимо ее чем-то заменить.

В качестве аргументации сошлемся на итоговое решение Всероссийской конференции «Преподавание информационных технологий в Российской Федерации», проходившей в 2024 году в Твери [12]. В пункте 23 там записано следующее: «Рекомендовать Минпросвещения России включить в школьную программу по учебному предмету “Информатика” (углубленный уровень) вопросы по архитектуре компьютера, функциям операционных систем, концепции и основным характеристикам языков программирования высокого и низкого уровней, предметно-ориентированных языков».

Завершая обсуждение, отметим, что слой микроархитектуры является скорее технологическим и поэтому менее значим. Если бы не высказанные выше соображения по поводу ориентации школьников на различные профессии в производстве hardware, то, возможно, данный слой можно было бы даже объявить частью слоя архитектуры.

5. Выводы

- Существуют минимум две важные причины, по которым стоит познакомиться с многослойной моделью компьютера и использовать ее в преподавании информатики. Во-первых, обострившаяся необходимость в разработке и развитии производства отечественной вычислительной техники и ее элементной базы; отсюда в профориентационных целях требуется дать школьникам более полное представление об этой стратегически важной отрасли. Во-вторых, учет базовых идей модели может оказаться полезным для систематизации знаний у школьников и формирования у них целостной картины предмета.
- Многослойный принцип действительно облегчает изучение многогранного и объемного материала о функциональных основах работы компьютера. При грамотном использовании он помогает увидеть и зафиксировать связи между выделенными слоями. Образно говоря, описываемый подход может стать *стержнем*, объединяющим отдельные темы курса информатики (кажущиеся, на первый взгляд, независимыми).
- «Потеря» хотя бы одного из слоев приводит к нарушению существующих взаимосвязей между темами учебного материала.
- Как следствие предыдущего, отрицательно влияет на систематичность знаний учащихся исключение из рассмотрения слоя компьютерной архитектуры, которая служит связующим звеном между программной и аппаратной частями. Это именно тот слой, где наиболее отчетливо проявляются связи между теорией (двоичная система счисления и др.) и лежащими выше и применяемыми на практике программными слоями. С указанной точки зрения полезно искать способы наглядной демонстрации двоичного представления различных типов данных и команд программы непосредственно в памяти компьютера или в файлах.
- В свете возможного направления развития отечественной вычислительной техники, вероятно, стоит познакомить школьников с существованием свободно распространяемого стандарта архитектуры компьютера под названием RISC-V. Можно порекомендовать показать несколько его наиболее простых команд, их кодирование и использование.
- Полезно также увеличить внимание к слою физического кодирования 0 и 1, учитывая его большое значение для мировоззрения (а также с точки зрения межпредметных связей).
- Многослойная модель позволяет просто и наглядно показать одну из причин постоянной потребности в росте производительности компьютерной техники: каждый добавляемый слой (или усложнение имеющегося) увеличивает время выполнения программы.
- Важно сказать ученикам, что, помимо упрощения освоения, рост числа слоев между пользователем и собственно компьютером имеет и отрицательные стороны: теряются возможности нижележащих слоев и падает эффективность взаимодействия прикладного ПО с аппаратной частью. Снижается также понимание логики поведения машины.

Список источников

1. *Андреева Е. В., Босова Л. Л., Фалина И. Н.* Математические основы информатики. Элективный курс: учебное пособие. М.: БИНОМ. Лаборатория знаний, 2005. 328 с. EDN: WLQBUT.

2. *Бершадский М. Е.* Педагогическая диагностика уровня понимания // Педагогические измерения. 2012. № 3. С. 60–88. EDN: OFVQZU.

3. *Богатырев Р.* Летопись языков. Си // Мир ПК. 2001. № 8. https://www.osp.ru/pcworld/2001/08/162103

4. Деннис Макалистэйр Ритчи. Между Unix и C // Хабр. 11 сентября 2024. https://habr.com/ru/companies/timeweb/articles/842238/

5. *Еремин Е. А.* Анализ содержательной линии «Компьютер» курса информатики с применением компьютерных средств представления знаний // Информатика («Первое сентября»). 2008. № 9. С. 8–18.

6. *Еремин Е. А.* Изучение средствами Delphi способов хранения в компьютерной памяти различных данных // Информатика («Первое сентября»). 2010. № 19. С. 4–23.

7. *Еремин Е. А.* Стандарт MSX для записи на МЛ // Информатика и образование. 1992. № 5-6. С. 56–58. EDN: TIFBDN.

8. *Лапчик М. П., Семакин И. Г., Хеннер Е. К.* Методика преподавания информатики: учеб. пособие для студ. пед. вузов. М.: Академия, 2001. 624 с. EDN: VFVKNW.

9. *Маркс К.* Капитал. М.: ИПЛ, 1969. Т. 1. С. 189.

10. *Олифер В. Г., Олифер Н. А.* Компьютерные сети, принципы, технологии, протоколы. СПб.: Питер, 2005. 864 с.

11. *Поляков К. Ю., Еремин Е. А.* Информатика. Углубленный уровень: учебник для 10 класса. В 2 ч. Ч. 1. М.: БИНОМ. Лаборатория знаний, 2013. 344 с. EDN: TUYVWD.

12. Решение Двадцать второй открытой Всероссийской конференции «Преподавание информационных технологий в Российской Федерации» (2024 год) // Преподавание информационных технологий в Российской Федерации. https://it-education.ru/conf2024/agenda/reshenie.php

13. *Романов С.* Использование памяти в Python // Хабр. 16 сентября 2013. https://habr.com/ru/post/193890/

14. *Столлингс В.* Структурная организация и архитектура компьютерных систем. М.: Вильямс, 2002. 896 с.

15. *Таненбаум Э. С.* Архитектура компьютера. СПб.: Питер, 2003. 704 с.

16. *Таненбаум Э. С., Бос Х.* Современные операционные системы. СПб.: Питер, 2015. 1120 с.

17. *Таненбаум Э. С., Фимстер Н., Уэзеролл Д.* Компьютерные сети. СПб.: Питер, 2023. 992 с.

18. *Токхейм Р.* Основы цифровой электроники. М.: Мир, 1988. 392 с.

19. Федеральная рабочая программа основного общего образования. Информатика (базовый уровень) (для 7–9 классов образовательных организаций) // Единое содержание общего образования. https://edsoo.ru/wp-content/uploads/2023/08/15_ФРП-Информатика-7-9-классы_база.pdf

20. Федеральная рабочая программа основного общего образования. Информатика (углубленный уровень) (для 7–9 классов образовательных организаций) // Единое содержание общего образования. https://edsoo.ru/wp-content/uploads/2023/08/16_ФРП_Информатика_7-9-классы_угл.pdf

21. Федеральная рабочая программа среднего общего образования. Информатика (базовый уровень) (для 10–11 классов образовательных организаций) // Единое содержание общего образования. https://edsoo.ru/wp-content/uploads/2023/08/21_ФРП-Информатика_10-11-классы_база.pdf

22. Федеральная рабочая программа среднего общего образования. Информатика (углубленный уровень) (для 10–11 классов образовательных организаций) // Единое со-

ISSN 2221-1993 • **Информатика в школе** • 2025 • Том 24 № 1

держание общего образования. https://edsoo.ru/wp-content/uploads/2023/08/22_ФРП_Информатика-10-11-классы_угл.pdf

23. *Фролов В. А., Галактионов В. А., Санжаров В. В.* Исследование технологии RISC-V // Труды ИСП РАН. 2020. Т. 32. № 2. С. 81–98. EDN: FYLLGM. DOI: 10.15514/ISPRAS-2020-32(2)-7.

24. *Харрис С. Л., Харрис Д.* Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. М.: ДМК Пресс, 2021. 810 с.

25. *Харрисон М.* Как устроен Python. Гид для разработчиков, программистов и интересующихся. СПб.: Питер, 2019. 272 с.

26. *Черняк Л. С.* Чему учиться? // Открытые системы. СУБД. 2003. № 2. https://www.osp.ru/os/2003/02/182640

27. *Ben-Ari M.* Constructivism in computer science education // Journal of Computers in Mathematics and Science Teaching. 2001. Vol. 20. No 1. P. 45–73.

28. *Burks A. W., Goldstine H. H., Neumann J.* Preliminary discussion of the logical design of an electronic computing instrument. Princeton, NJ: Institute for Advanced Study, 1946. P. 1–33. (Русский перевод: *Беркс А., Голдстейн Г., Нейман Дж.* Предварительное рассмотрение логической конструкции электронного вычислительного устройства // Кибернетический сборник. № 9. С. 7–67. М.: Мир, 1964.)

29. *Du Boulay B.* Some difficulties of learning to program // Journal of Educational Computing Research. 1986. Vol. 2. No 1. P. 57–73. DOI: 10.2190/3LFX-9RRF-67T8-UVK9.

30. *González R. L., García L. M. C., García M. M., Masa J. A.* Possibilities of “Nuclear Concepts Theory” on educational research, a review // Pixel International Conferences. https://conference.pixel-online.net/conferences//edu_future/common/download/Paper_pdf/SOE33-Gonzalez,Garcia,Garcia,Masa.pdf