Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования «Рыбинский государственный
авиационный технический университет имени П. А. Соловьева»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ
УПРАВЛЕНИЯ

Кафедра вычислительных систем

Реферат
по дисциплине:
«Вычислительные системы»
на тему:
«Архитектуры *ARM* и *x*86»

Выполнил: студент группы ИВМ-24 Морозов А. А.
Руководитель: кандидат технических наук, доцент Павлов Р. В.

Рыбинск 2024

Содержание

Введение

Современные вычислительные системы базируются на двух основных подходах к архитектуре процессоров – *ARM* и *x*86. Эти архитектуры определяют принципы организации команд, структуру процессоров и методы их взаимодействия с памятью и периферийными устройствами.

Архитектура *ARM* представляет собой пример подхода на основе принципов *RISC* (*Reduced Instruction Set Computing*), что подразумевает использование минимального набора команд, оптимизированных для выполнения базовых операций. Такой подход обеспечивает высокую энергоэффективность и простоту реализации. *ARM* широко применяется в мобильных устройствах, встраиваемых системах и решениях для интернета вещей (*IoT*), благодаря своей способности работать в условиях ограниченных ресурсов.

В свою очередь, *x*86 является представителем *CISC*-архитектуры (*Complex Instruction Set Computing*). Она предлагает сложный и обширный набор инструкций, что делает её особенно подходящей для настольных компьютеров, серверов и рабочих станций, где важна высокая производительность для выполнения ресурсоёмких задач. Исторически сложившаяся совместимость с предыдущими поколениями процессоров обеспечила *x*86 ведущую роль в индустрии персональных компьютеров.

Изучение этих двух подходов важно для понимания текущих трендов в проектировании вычислительных систем. На протяжении десятилетий развитие архитектур *ARM* и *x*86 определяло траекторию эволюции процессоров, формируя основу современных вычислительных устройств.

Инновации в архитектуре процессоров активно способствуют созданию новых продуктов и решений, начиная от высокопроизводительных серверов до компактных умных устройств. Процессоры на базе архитектур *ARM* и *x*86 постоянно совершенствуются, чтобы соответствовать растущим требованиям производительности, энергоэффективности и безопасности. Особое внимание

уделяется вопросам параллельной обработки данных, адаптации к специализированным нагрузкам и интеграции с системами искусственного интеллекта.

Кроме того, важно отметить роль глобальной конкуренции в области процессорных технологий. Производители стремятся внедрять новые функции, улучшать производственные процессы и снижать стоимость решений. Это позволяет создавать устройства, которые одновременно предлагают высокую производительность и экономичность, обеспечивая широкий спектр применения архитектур *ARM* и *x*86 в различных отраслях. На протяжении десятилетий развитие архитектур *ARM* и *x*86 определяло траекторию эволюции процессоров, формируя основу современных вычислительных устройств.

1 Основы работы микроЭВМ

Любая микропроцессорная система служит для обработки информации с целью получения требуемого результата. Обработка информации в МПС осуществляется микроЭВМ [1]. На рисунке 1.1 изображена типовая структура микроЭВМ

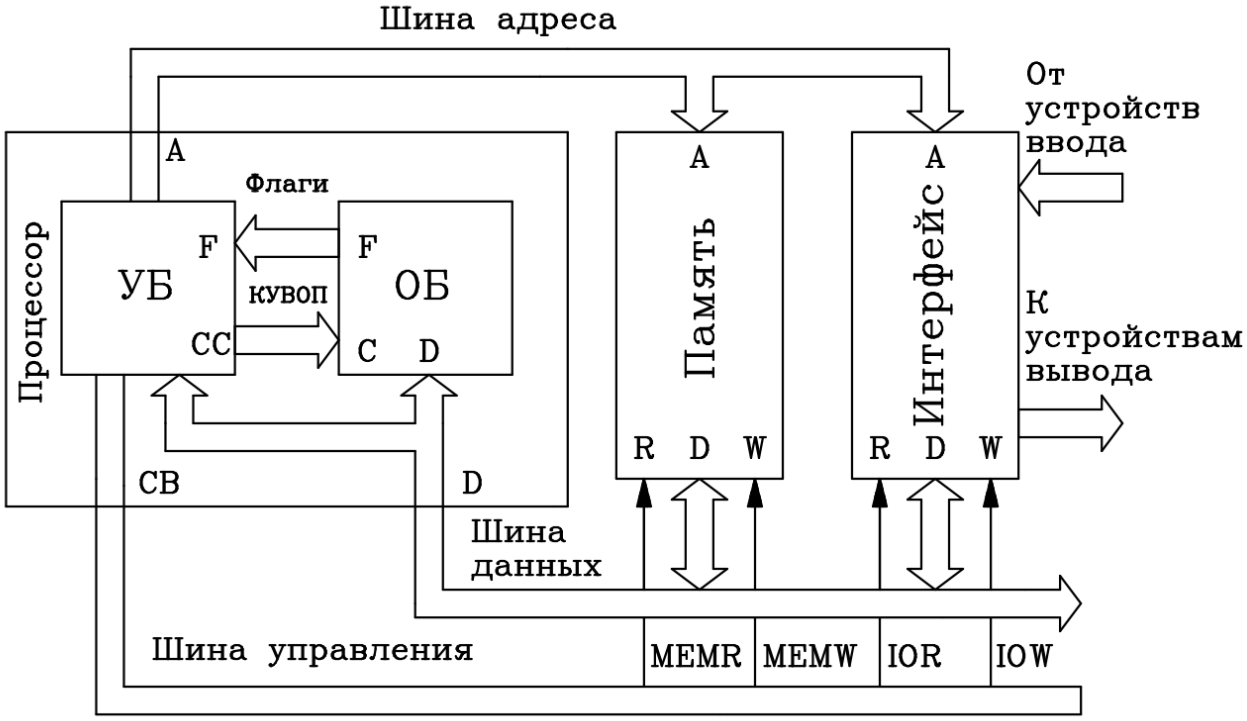

Рисунок 1.1 – Типовая структура микроЭВМ

Память микроЭВМ предназначена для хранения программы и промежуточных данных. Каждый элемент информации хранится в отдельной ячейке, доступ к которой осуществляется по адресу, закодированному в двоичной системе. Обращение может быть двух типов: чтение или запись, которые регулируются управляющими сигналами *MEMR* (чтение) и *MEMW* (запись).

Процессор микроЭВМ выполняет вычислительный процесс через два блока: операционный (ОБ) и управляющий (УБ). Операционный блок выполняет операции с данными, а управляющий блок обеспечивает их последовательное выполнение. Для этого процессор извлекает команду из памяти, адрес которой указан в программном счетчике. Затем УБ генерирует сигнал чтения *MEMR*, и команда поступает в процессор.

Если команда операционная, УБ направляет коды управления в ОБ для выполнения операции. Если требуется операнд из памяти, его извлечение также происходит через адрес, указанный в команде, с использованием сигнала MEMR. Результат операции записывается в память через сигнал MEMW, указывающий адрес ячейки назначения.

Для реализации условного перехода ОБ формирует флаги, которые передаются в УБ для принятия решения о следующем шаге. УБ затем определяет адрес следующей команды и продолжает выполнение. При управляющей команде УБ сразу направляет адрес следующей команды.

Интерфейс микроЭВМ соединяет устройства ввода/вывода (УВВ) с процессором или памятью. Обращение к УВВ аналогично обращению к памяти и регулируется управляющими сигналами *IOR* (чтение) и *IOW* (запись).

2 Начало развития архитектур

Архитектура процессоров прошла долгий путь эволюции. Первые процессоры были относительно простыми устройствами, способными выполнять ограниченный набор операций. Со временем требования к вычислительным системам росли, что стимулировало появление новых технологий и подходов к проектированию. На ранних этапах развития архитектур акцент делался на повышение производительности и совместимости с существующими системами. Это стало основой для появления *CISC*-архитектуры, которая предлагала широкий набор сложных команд для упрощения разработки программного обеспечения.

Однако с ростом сложности вычислительных задач появилась необходимость оптимизировать использование ресурсов процессора. Это привело к созданию концепции *RISC*, ориентированной на минимизацию набора команд и повышение эффективности их выполнения. Впоследствии обе архитектуры продолжали совершенствоваться, занимая свои ниши в различных сегментах рынка.

На сегодняшний день архитектуры *ARM* и *x*86 являются доминирующими в своих областях. *ARM*, следуя принципам *RISC*, нашла широкое применение в мобильных устройствах и встраиваемых системах благодаря своей энергоэффективности. В то же время *x*86, базирующаяся на принципах *CISC*, сохраняет лидирующие позиции в сегментах настольных компьютеров, серверов и высокопроизводительных систем.

2.1 Ранние архитектуры

Первая электронная вычислительная машина (ЭВМ) *ENIAC* была создана в США в 1946 году для расчета баллистических таблиц. В ней использовалось 17468 электронных ламп и 1500 электромагнитных реле. Эта машина успешно выдержала испытания, продемонстрировав невиданное до тех пор быстродействие (время сложения – 200 мкс, умножения – 2,8 мс, деления – 6 мс).

Однако она имела габаритные размеры 29 м × 5,8 м × 6 м, весила 30 т, потребляла 150 кВт электроэнергии в час, стоила 2,8 миллиона долларов и безотказно работала около 7 минут. Такие характеристики исключали возможность распространения подобных машин.

С развитием технологий в 1950-х и 1960-х годах появились первые процессоры, которые начали интегрировать несколько функций в одну единую систему. Эти ранние процессоры были предназначены для выполнения ограниченного набора операций, таких как сложение, вычитание и хранение данных в память. Эти устройства не обладали тем уровнем гибкости, который позже появился с разработкой архитектуры команд.

2.2 Переход от машинных кодов к *Assembler*

С момента появления первых электронных вычислительных машин программирование происходило напрямую на уровне машинных кодов, что представляло собой трудоемкий и сложный процесс. Машинный код – это низкоуровневый код, состоящий из двоичных инструкций, которые могут быть интерпретированы процессором и выполнены без дополнительных преобразований. Однако для программистов, работающих с такими кодами, процесс написания программ был крайне трудоемким и требовал высокой точности. В ответ на это была разработана ассемблерная программа, которая позволила упростить этот процесс, сделав программирование более доступным.

На рисунке 2.1 изображена памятка программиста для ЭВМ «Минск-22»

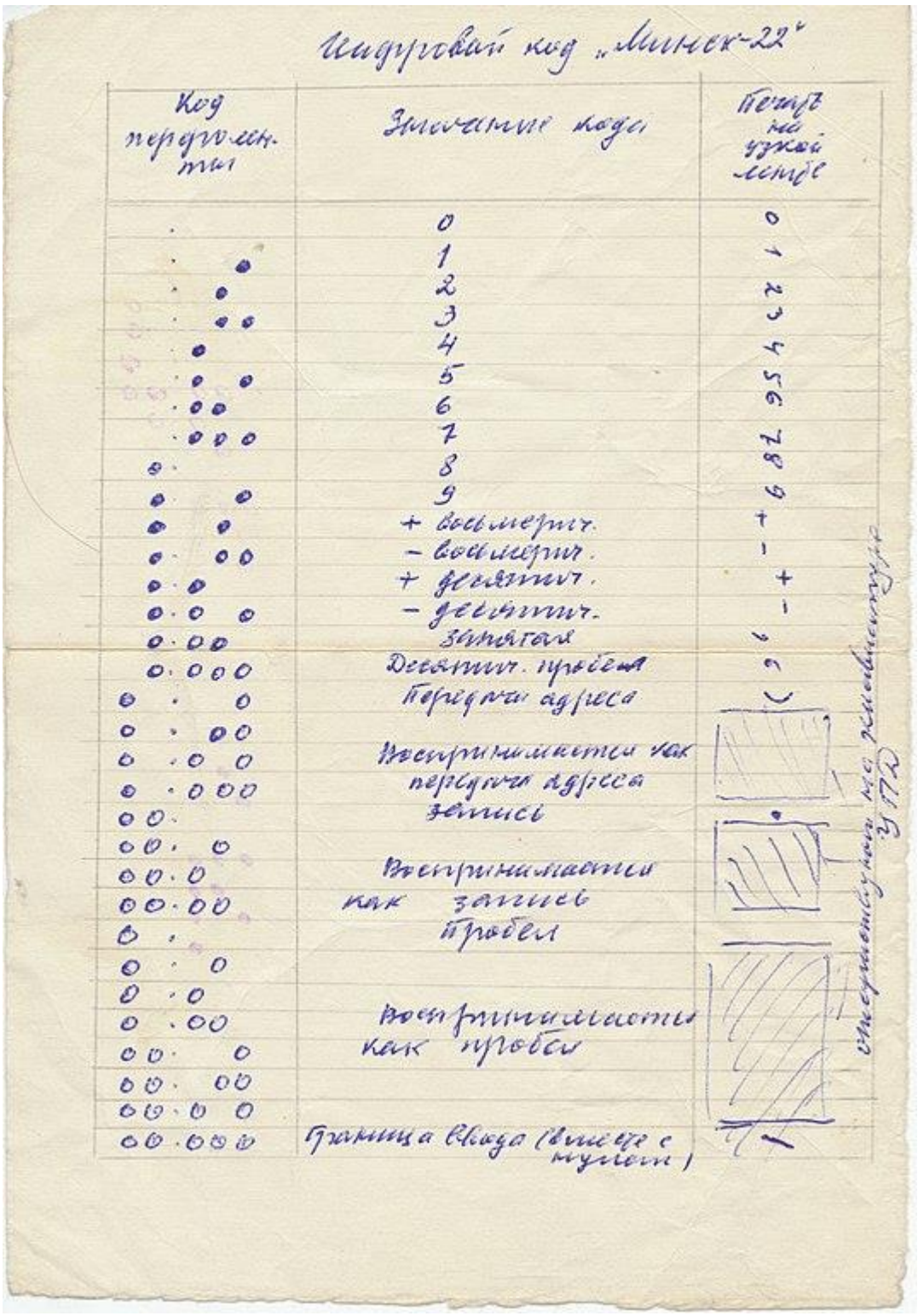

Рисунок 2.1 – Памятка программиста

Ассемблер (от английского *Assembler* – сборщик, компилятор) был разработан как промежуточный шаг между машинным кодом и более

высокоуровневыми языками программирования. Ассемблер предоставляет программистам возможность использовать мнемонические (читаемые человеком) инструкции вместо двоичных кодов машинных команд. Каждая ассемблерная инструкция соответствует одной машинной инструкции, но записана в виде символов, что значительно облегчает процесс программирования.

На рисунке 2.2 изображено сравнение машинного побайтового кода и кода на языке программирования *Assembler*.

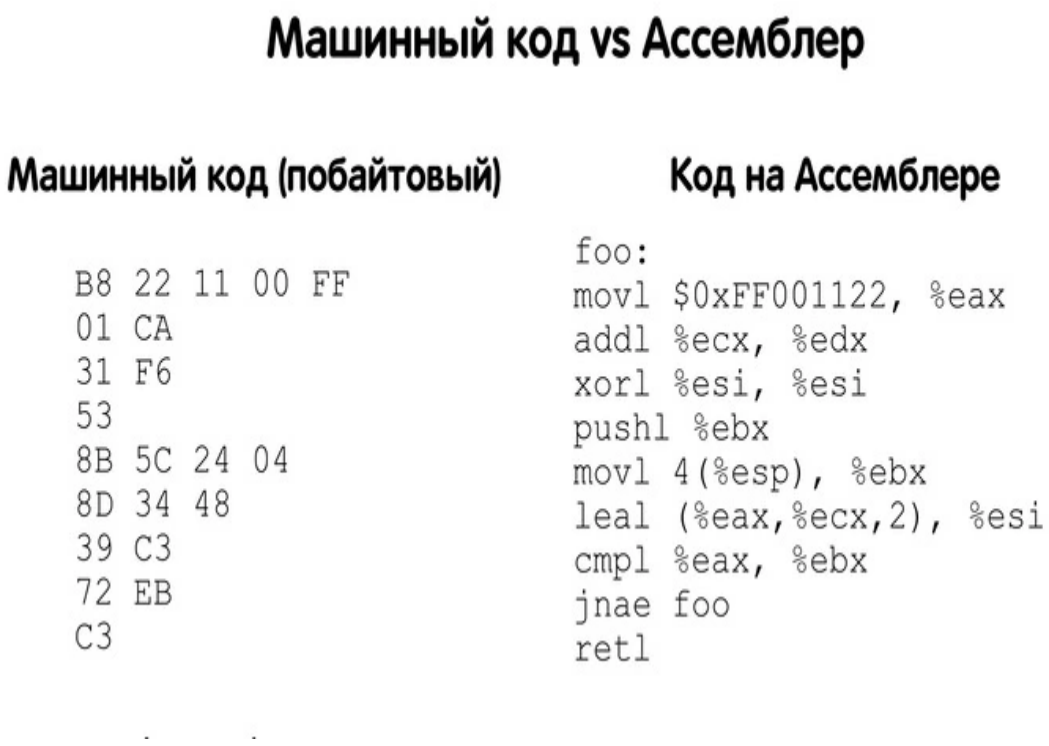

Рисунок 2.2 – Сравнение кодов

На фоне этих изменений продолжалась эволюция самих архитектур процессоров, что привело к созданию новых моделей, ориентированных на увеличение производительности и гибкости программирования. Одной из таких архитектур стала x86, использующая принципы *CISC* (*Complex Instruction Set Computer*).

3 Архитектура *x*86

«В конце 60-х годов калькуляторы представляли собой большие электромеханические машины размером с современный лазерный принтер и весили около 20 кг. В сентябре 1969 года японская компания *Busicom* обратилась к корпорации *Intel* с просьбой выпустить 12 несерийных микросхем для электронной вычислительной машины. Инженер компании *Intel* Тед Хофф (*Ted Hoff*), назначенный в качестве исполнителя этого проекта, решил, что можно поместить 4-разрядный универсальный процессор на одну микросхему, которая будет выполнять те же функции и при этом окажется проще и дешевле. Так в 1970 году появился первый процессор на одной микросхеме – 4004 на 2300 транзисторах.» [2].

Архитектура *x*86 была разработана компанией *Intel* в начале 1970-х годов. Первый процессор с архитектурой *x*86, *Intel* 8086, был выпущен в 1978 году, и с тех пор эта архитектура претерпела множество изменений и усовершенствований. В течение нескольких десятилетий архитектура *x*86 была основой для всех основных персональных компьютеров, и её поддержка сыграла ключевую роль в распространении вычислительных технологий в массовом сегменте. Процессоры на базе *x*86 обеспечивали необходимые вычислительные мощности для запуска операционных систем и приложений, став популярными в офисах, учебных заведениях и домах.

С каждым новым поколением процессоров *Intel x*86 архитектура продолжала развиваться, поддерживая обратную совместимость с предыдущими версиями. Эта совместимость с ранними версиями процессоров стала одной из главных причин её длительного успеха. Например, процессоры *Intel* 80386, выпущенные в 1985 году, уже поддерживали 32-битные вычисления, что обеспечивало большую производительность, по сравнению с предшествующими 16-битными версиями. Вскоре последовали новые улучшения, такие как процессоры Intel Pentium, которые добавили поддержку многозадачности и

вычислений с плавающей запятой, что сделало их ещё более мощными и универсальными.

3.1 *CISC* архитектура команд

Архитектура *CISC* (Complex Instruction Set Computing) относится к архитектуре процессора, в которой используется набор сложных инструкций, каждая из которых может выполнять несколько операций за один цикл. Это позволяет сократить количество команд, которые необходимо выполнить для реализации сложных вычислений, и тем самым улучшить эффективность работы программного обеспечения. В отличие от других архитектур, которые используют небольшой набор простых команд, *CISC* ориентирован на использование более сложных инструкций, которые могут выполнять несколько операций за один такт процессора. Этот подход позволил значительно улучшить производительность в ранние годы развития вычислительной техники, когда важнейшими проблемами были ограниченные ресурсы памяти и необходимость оптимизации кода.

К особенностям *CISC* архитектуры относят:

- использование переменной длины команд. Команды могут быть как очень короткими (например, всего в 1 байт), так и достаточно длинными, до 15 байт. Это даёт процессору гибкость в представлении команд и позволяет эффективно интегрировать сложные операции в одну инструкцию. Например, команда в архитектуре *x*86 может включать несколько шагов, таких как вычисление арифметической операции и обращение к памяти, что сокращает количество инструкций, необходимых для выполнения более сложных задач. За счёт этого код для программы имел довольно маленький объём и сами программы занимали мало места на диске, но также на выполнение 1 инструкции приходится больше 1 такта машинного времени;

- работа напрямую с памятью без необходимости предварительного получения данных из регистра процессора. В *x*86, например, существуют различные способы адресации [3]:

а) регистровая адресация;

б) непосредственная адресация;

в) прямая адресация;

г) базовая адресация.

- небольшое количество регистров. Из-за возможности работать напрямую с памятью в *CISC* архитектуре используется сравнительно малое количество регистров.

3.2 Совместимость с предыдущими поколениями

Архитектура *x*86 является одной из самых широко используемых и долговечных в мире вычислительных систем. Одной из её ключевых особенностей является обратная совместимость, что означает, что процессоры на базе x86 могут запускать программное обеспечение, написанное для более ранних версий процессоров. Эта совместимость не ограничивается только поддержкой старых инструкций, но также включает возможность эмуляции более ранних режимов работы, что позволяет запускать старые программы и операционные системы на современных машинах.

Процессоры *x*86 поддерживают несколько режимов работы, каждый из которых соответствует различным поколениям процессоров. Например, процессоры 16-битной архитектуры *x*86, появившиеся в 1980-х годах, использовали набор инструкций, который оставался совместимым с более поздними 32-битными и 64-битными версиями архитектуры. Это позволило пользователям запускать старые программы и операционные системы, даже если они были написаны для процессоров, значительно отличающихся по мощности и возможностям.

С развитием архитектуры *x*86 появились новые расширения, такие как 32-битные и 64-битные режимы работы. Процессоры с 32-битной архитектурой (*x*86) представляют собой эволюцию более старых 16-битных чипов, и они сохраняют полную совместимость с программным обеспечением, разработанным для более ранних процессоров. В свою очередь, 64-битные процессоры (*x*86-64) не только поддерживают 32-битные и 16-битные режимы, но и предлагают значительные улучшения по производительности, памяти и вычислительным возможностям.

Обратная совместимость архитектуры *x*86 означает, что старые программы, включая операционные системы и приложения, могут работать на новых процессорах без необходимости изменений в исходном коде. Для этого современные процессоры *x*86 оснащены специальными режимами работы, такими как режимы совместимости (*Long mode* и *Legacy mode*) и виртуализации, которые позволяют запускать старые 16-битные и 32-битные приложения на более мощных 64-битных процессорах. Это особенно важно для пользователей и организаций, которые используют старое программное обеспечение, которое не было обновлено или не имеет аналогов для более новых архитектур.

Благодаря поддержке множества режимов работы процессоры *x*86 остаются универсальными и устойчивыми к изменениям в индустрии. Например, современные процессоры *Intel* и *AMD* с архитектурой *x*86-64 могут без проблем запускать как старое 32-битное ПО, так и современные 64-битные приложения, что позволяет пользователям плавно переходить на новые технологии, не теряя совместимости с предыдущими версиями программного обеспечения.

3.3 Расширения для *x*86

Архитектура *x*86 прошла через множество улучшений и расширений, чтобы поддерживать более сложные вычисления, улучшать производительность и обеспечивать совместимость с новыми технологиями. Некоторые из этих расширений касаются обработка мультимедийных данных, научных

вычислений, а также параллельных вычислений. В частности, важными расширениями для *x*86 являются *MMX*, *SSE* и *AVX*.

3.3.1 Расширение *MultiMedia Extensions*

MMX (*MultiMedia Extensions*) был представлен компанией *Intel* в 1997 году как расширение для ускорения обработки мультимедийных данных, таких как аудио, видео и графика. Это расширение добавило новые *SIMD* (*Single Instruction, Multiple Data*) инструкции, что позволило процессорам одновременно обрабатывать несколько данных с помощью одной команды.

Основные особенности *MMX*:

- добавление 8 64-битных регистров (*MM*0 – *MM*7), которые могут хранить несколько 8-битных, 16-битных, 32-битных или 64-битных значений;

- *SIMD*-инструкции, позволяющие выполнять однотипные операции над несколькими данными одновременно (например, сложение нескольких чисел);

- предназначен для обработки мультимедийных данных, таких как аудио- и видеокодеки, а также графики.

3.3.2 Расширение *Streaming SIMD Extensions*

SSE (*Streaming SIMD Extensions*) был анонсирован в 1999 году с процессорами *Intel Pentium* III и стал следующим шагом в развитии *SIMD*-инструкций. *SSE* значительно улучшило возможности *MMX*, расширив поддержку более широких данных и более сложных вычислений.

Основные особенности SSE:

- добавление 8 128-битных регистров *XMM*0 – *XMM*7, которые могут обрабатывать 4 32-битных числа с плавающей запятой одновременно;

- совершенствование арифметических операций с плавающей запятой, что особенно полезно для научных и инженерных вычислений;

- новые инструкции для обработки данных с плавающей запятой (например, операции сложения, умножения, сравнения);

- совместимость с *MMX* (например, может работать с теми же самыми регистрами, но использует их для других типов данных).

3.3.3 Расширение *Advanced Vector Extensions*

Advanced Vector Extensions (*AVX*) – расширение системы команд *x*86 для микропроцессоров *Intel* и *AMD*, предложенное *Intel* в марте 2008. *AVX* значительно расширяет возможности *SIMD* и позволяет обрабатывать ещё большие объёмы данных, улучшая производительность в вычислениях, связанных с научными расчетами, машинным обучением и обработкой графики.

Основные особенности *AVX*:

- добавление 256-битных регистров *YMM*0 – *YMM*15, которые позволяют работать с более широкими данными и увеличивают пропускную способность за счёт обработки большего числа данных за один такт;

- поддержка чисел с плавающей запятой двойной точности (64 бита) и улучшенные инструкции для работы с такими числами;

- новые инструкции для более эффективных операций с векторами (например, операции свертки, умножение векторов).

AVX-инструкции используются для работы с числами с плавающей точкой, в дальнейшем появилось расширение *AVX*2 для работы с целочисленными переменными. В 2013 году было представлено расширение *AVX*-512, которое добавило 512-битовые регистры, но не получило популярности в потребительском сегменте из-за повышенного энергопотребления (рисунок 3.1)

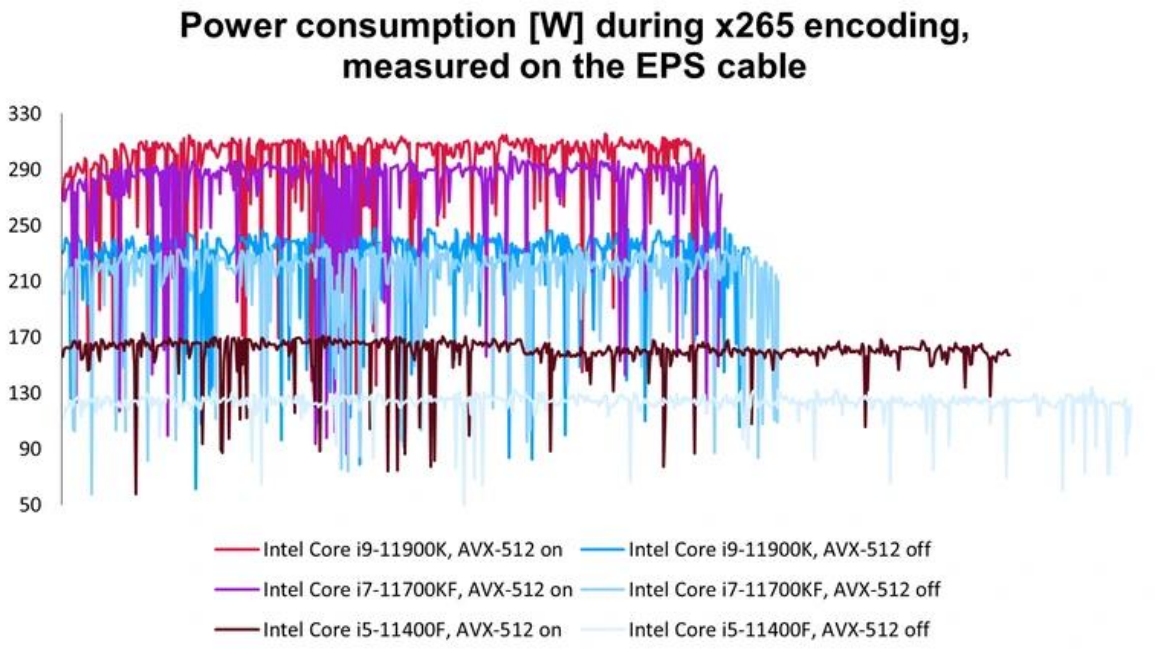

Рисунок 3.1 – Сравнение энергопотребления с *AVX*-512 и без него

Из вышеперечисленно можно сделать вывод, что процессоры на архитектуре *x86* широко используются в персональных компьютерах, серверах и встраиваемых системах благодаря своей совместимости с огромным количеством программного обеспечения и поддержке различных расширений, таких как *MMX*, *SSE*, *AVX*. Эти расширения значительно увеличивают производительность процессоров при выполнении вычислительных задач, таких как обработка мультимедийных данных, научные расчёты и машинное обучение.

В то же время, несмотря на свои многочисленные преимущества, архитектура *x86* сталкивается с вызовами в связи с ростом потребностей в более мощных и энергоэффективных процессорах. В последние годы активно развиваются альтернативные архитектуры, такие как *ARM*, которые демонстрируют большую эффективность в мобильных и встраиваемых устройствах. Тем не менее, *x86* остается доминирующей архитектурой на рынке настольных ПК и серверов, благодаря своей поддержке широкого спектра технологий и стандартов.

4 Архитектура *ARM*

Архитектура *ARM* (*Advanced RISC Machine*) является одной из самых популярных и широко используемых архитектур процессоров, которая нашла своё применение в самых разных областях, от мобильных и встраиваемых устройств до серверных решений и суперкомпьютеров. Разработанная в 1980-х годах компанией *Acorn Computers*, архитектура *ARM* изначально была ориентирована на создание высокопроизводительных, энергоэффективных процессоров, способных удовлетворять потребности быстро развивающегося рынка. Одним из основных вдохновителей при разработке архитектуры стал проект *Berkeley RISC*, который продемонстрировал возможности снижения сложности командных наборов процессора для улучшения производительности и повышения энергоэффективности. Вдохновленные этими идеями, инженеры *Acorn Computers* решили разработать процессор, который бы сочетал простоту, эффективность и производительность. Архитектура была задумана с учётом ограничений по энергозатратам, что позволило ей успешно конкурировать с более сложными архитектурами, такими как *x86*.

Первоначально архитектура называлась «*Acorn RISC Machine*» (*ARM*), что отражало её происхождение от компании *Acorn*. Однако в дальнейшем, после того как компания *ARM* отделилась от *Acorn*, аббревиатура была интерпретирована как «Advanced RISC Machine», чтобы подчеркнуть достижения и совершенство архитектуры. В отличие от более сложной и энергозатратной архитектуры *x86*, *ARM* использует принцип *RISC* (*Reduced Instruction Set Computing*), который направлен на минимизацию количества и сложности инструкций, что позволяет процессорам работать быстрее и эффективнее при меньшем потреблении энергии. Это было важным преимуществом для создания мобильных устройств, где экономия энергии имеет первостепенное значение, а также для встраиваемых решений, которые часто требуют высокой энергоэффективности и небольшой физической площади. Упрощённый набор инструкций позволил достигать высокой

производительности при значительно меньших затратах ресурсов, что стало важным фактором в расширении применения *ARM* в таких областях, как Интернет вещей (*IoT*) и автоматизация.

4.1 *RISC* архитектура команд

RISC (*Reduced Instruction Set Computer*) – это архитектурная концепция, основная цель которой заключается в упрощении набора машинных команд для повышения производительности и уменьшения затрат на обработку инструкций. Идея данной архитектуры начала формироваться в 1970-х годах, когда исследователи осознали, что сложные наборы инструкций (*CISC*) часто приводят к низкой эффективности выполнения программ. Одними из первых, кто начал активно развивать концепцию *RISC*, были Дэвид Паттерсон и Джон Хеннесси, работы которых стали фундаментом для дальнейших разработок.

Проект *Berkeley RISC*, запущенный в Калифорнийском университете в Беркли под руководством Паттерсона, стал одним из первых шагов к реализации концепции *RISC* на практике. Исследования, проводимые в рамках этого проекта, были сосредоточены на упрощении аппаратной части процессора и увеличении его производительности за счёт сокращения набора команд. В 1980 году был создан первый прототип процессора *RISC*-I, который имел всего 32 инструкции и мог выполнять их за один такт. Этот процессор продемонстрировал, что минималистский подход позволяет достичь высокой эффективности при низких затратах. В то же время проект *Stanford MIPS* под руководством Хеннесси также доказал жизнеспособность данного подхода, приведя к созданию одного из первых коммерчески успешных *RISC*-процессоров.

Ключевым принципом *RISC* является выполнение каждой инструкции за один такт процессора, что достигается за счёт оптимизации набора команд и использования конвейерной обработки данных. Такой подход позволяет эффективно использовать аппаратные ресурсы и значительно ускорять

выполнение программ. Помимо этого, архитектура RISC ориентирована на минимизацию избыточности инструкций и упрощение компиляции, что делает её привлекательной для разработки программного обеспечения.

В 1980-х годах *RISC*-архитектура начала находить практическое применение в коммерческих процессорах. Такие компании, как *IBM, Sun Microsystems* и *ARM*, начали разрабатывать свои собственные решения на основе этой концепции. Одной из первых значимых реализаций стал процессор IBM 801, разработанный для систем управления. Впоследствии *Sun Microsystems* представила архитектуру *SPARC*, а *ARM* выпустила свои первые процессоры для встраиваемых систем.

В рамках данной архитектуры разработчики акцентируют внимание на использовании регистров для хранения временных данных и минимизации операций с памятью. Это обеспечивает не только высокую скорость выполнения операций, но и снижение энергопотребления, что является важным фактором для мобильных устройств и встраиваемых систем. Сегодня архитектура *RISC* широко используется в процессорах, применяемых в различных сферах – от высокопроизводительных серверов до смартфонов и микроконтроллеров. Наиболее ярким примером процессоров, построенных на основе данной концепции, являются чипы семейства *ARM*, которые доминируют на рынке мобильных устройств. В то же время архитектура *RISC* остаётся предметом научных исследований и инженерных разработок, направленных на создание ещё более производительных и энергоэффективных процессоров.

RISC архитектура имеет множество особенностей.

4.1.1 Регистровые окна и файл

Архитектура *RISC* предполагает использование большого числа регистров общего назначения, что позволяет хранить временные данные и минимизировать обращения к оперативной памяти. Это значительно ускоряет выполнение операций, так как доступ к регистрам происходит гораздо быстрее, чем к памяти.

«Уже много лет известно, что главным препятствием высокой скорости выполнения команд является необходимость их загрузки из памяти. Для разрешения этой проблемы можно вызывать команды из памяти заранее и хранить в специальном наборе регистров.» [2]. Для работы с большим количеством регистров использовалось так называемое регистровое окно и регистровый файл.

В регистровом файле содержится реализация регистров процессора (массив ячеек, считываемых вертикально) и он делится на регистровые окна по какому-то количеству регистров (обычно по 32 регистра). Регистровые окна состоят из 3 полей (рисунок 4.1):

- левое поле для входных значений;
- среднее поле для локальных значений;
- правое поле для выходных значений.

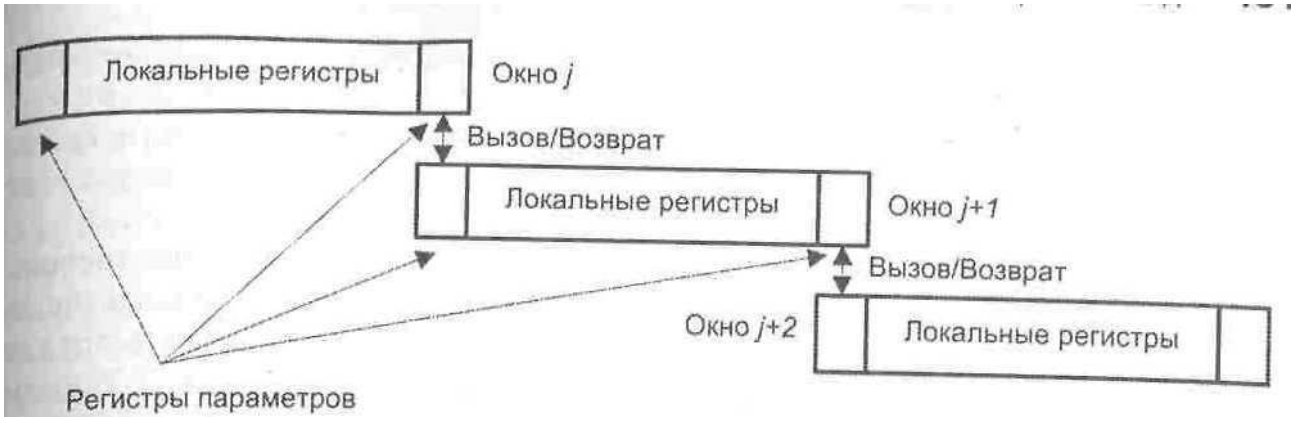

Рисунок 4.1 – Перекрытие регистровых окон

При каждом вызове процедуры ей выделяется регистровое окно и на это окно указывает специальный регистр «указатель текущего окна (*CWP, Current Window Pointer*)». Когда происходит вызов вложенной процедуры, то правое поле для выходных значений вызывавшей функции перекрывается и является левым полем для входных значений новой (вложенной) функции (рисунок 4.2) [4].

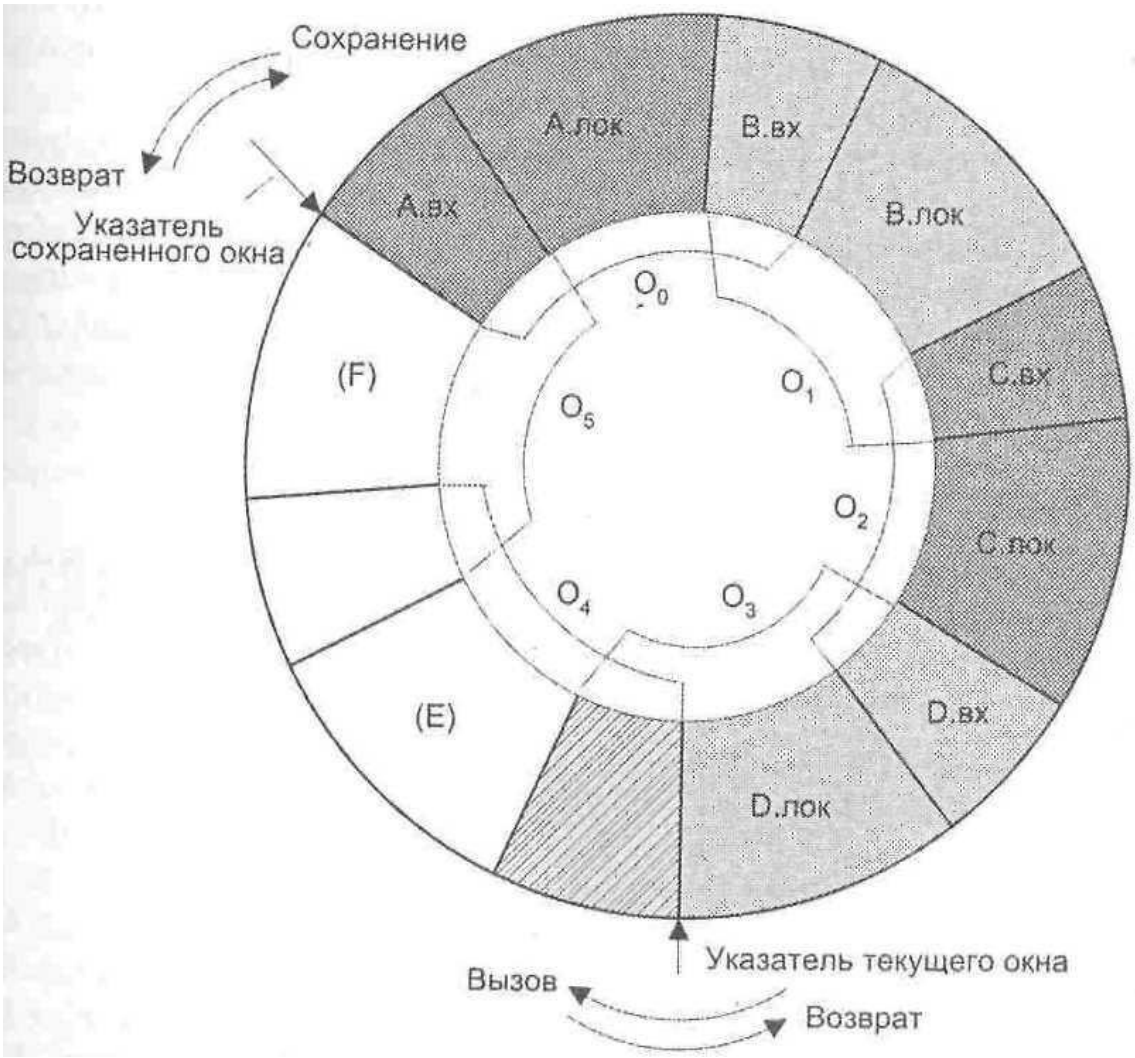

Рисунок 4.2 – Циклический буфер из пересекающихся регистровых окон

Главное достоинство регистровых окон – отсутствие работы со стеком при вызове подпрограмм. Кроме того, регистровые окна реализуют защиту памяти: подпрограмма не может изменить значения большинства регистров вызвавшей её программы. Среди минусов этого подхода можно отметить наличие максимальной глубины вложенности подпрограмм, при превышении которой накладные расходы при вызове подпрограмм резко возрастают.

Теоретически такой прием не исключен и в *CISC*. Однако устройство управления CISC-процессора оккупирует на кристалле более 50% площади, оставляя мало места для других подсистем, в частности для большого файла

регистров. Устройство управления *RISC*-процессора занимает порядка 10% поверхности кристалла, предоставляя возможность иметь большой регистровый файл.

4.1.2 Фиксированная длина команд

Все команды в *RISC* имеют одинаковую длину, обычно 32 бита, что существенно упрощает их обработку процессором. Фиксированная длина позволяет процессору легко определять границы инструкций и предсказывать адрес следующей команды в памяти, что особенно важно для реализации эффективного конвейера.

В отличие от *RISC*, архитектуры *CISC* (*Complex Instruction Set Computer*) используют команды переменной длины, которые могут занимать от нескольких байт до десятков байт. Это усложняет декодирование инструкций и приводит к увеличению времени их обработки.

Преимущество фиксированной длины команд также заключается в упрощении аппаратной части процессора, поскольку декодирование становится менее затратным и быстрее реализуемым.

Однако фиксированная длина может потребовать большего объёма памяти для хранения программ по сравнению с *CISC*, где сложные операции могут быть выражены одной компактной инструкцией. Несмотря на это, высокая скорость выполнения инструкций в *RISC* компенсирует возможный рост памяти.

Все команды в *RISC* имеют одинаковую длину, что упрощает их обработку. Такое решение позволяет легко предсказывать расположение следующей инструкции в памяти и ускоряет конвейерную обработку данных.

4.1.3 Конвейер команд

Конвейер в процессорах *RISC* разделяет выполнение инструкции. Это позволяет процессору одновременно обрабатывать несколько команд, значительно увеличивая его производительность.

При использовании конвейера команда обрабатывается уже не за два, а за большее количество шагов, каждый из которых реализуется определенным аппаратным компонентом, причем все эти компоненты могут работать параллельно.

На рисунке 4.3 изображен пятиступенчатый конвейер, где каждый блок выполняет свою функцию. Первая ступень (блок С1) извлекает команду из памяти и помещает ее в буфер, где она сохраняется до момента использования. Вторая ступень (блок С2) занимается декодированием команды, определяя ее тип и типы операндов. Третья ступень (блок С3) отвечает за нахождение операндов и их извлечение из регистров или памяти. Четвертая ступень (блок С4) выполняет команду. Наконец, пятая ступень (блок С5) записывает результат обратно в нужный регистр.

Рисунок 4.3 – Пятиступенчатый конвейер

На рисунке 4.4 показано, как конвейер работает во времени. В цикле 1 блок С1 обрабатывает команду 1, извлекая ее из памяти. В цикле 2 блок С2 декодирует команду 1, в то время как блок С1 вызывает команду 2 из памяти. В цикле 3 блок С3 извлекает операнды для команды 1, блок С2 декодирует команду 2, а блок С1 вызывает команду 3. В цикле 4 блок С4 выполняет команду 1, С3 извлекает операнды для команды 2, С2 декодирует команду 3, а блок С1 вызывает команду 4. Наконец, в цикле 5 блок С5 записывает результат выполнения команды 1 в регистр, а остальные ступени конвейера продолжают обработку следующих команд.

Рисунок 4.4 – Цикл конвейера

В *CISC* архитектуре тоже реализован конвейер, но создали его намного позже и функционально он сложнее чем у *RISC*.

4.2 Порядок байтов

Порядок байтов (*endianess*) определяет, как много байтов составного числа хранятся в памяти и в каком порядке. В зависимости от архитектуры процессора различают два основных порядка байтов: *big-endian* и *little-endian*. В режиме *big-endian* старший байт числа хранится по младшему адресу памяти, а младший — по старшему. В режиме *little-endian*, наоборот, младший байт сохраняется по младшему адресу, а старший – по старшему адресу.

Процессоры *ARM* поддерживают как *big-endian*, так и *little-endian* режимы, при этом большинство современных *ARM*-процессоров используют *little-endian* порядок байтов по умолчанию. Это означает, что при работе с числами младший байт будет располагаться по меньшему адресу, что является более удобным для многих приложений и стандартов. Тем не менее, *ARM*-процессоры позволяют переключаться между этими режимами, что может быть полезно для совместимости с другими архитектурами или при специфических требованиях.

Процессоры архитектуры x86 всегда используют *little-endian* порядок байтов. Это означает, что при взаимодействии с памятью младший байт числа будет размещён в первом адресе, а старший – в последнем. Такой порядок байтов стал стандартом для большинства современных ПК и серверных систем, обеспечивая совместимость с широким спектром программного обеспечения и периферийных устройств.

На рисунке 4.5 изображено сравнение двух порядков байтов.

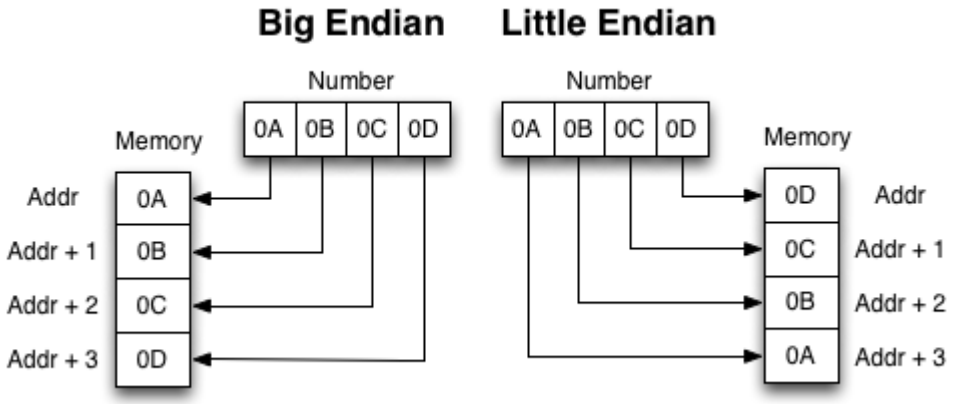

Рисунок 4.5 – Два порядка байтов

4.3 *ARM big.LITTLE*

Технология *ARM big.LITTLE* объединяет несколько типов ядер в 1 процессоре:

- *big* ядра, которые имеют высокую производительность и энергопотребление. Необходимы для трудоёмких задач;

- *LITTLE* ядра, которые имеют меньшую производительность и энергопотребление по сравнению с *big* ядрами. Используются для простых задач.

Важную роль в эффективном использовании архитектуры *big.LITTLE* играет планировщик задач, который отвечает за распределение задач между большими и маленькими ядрами. Планировщик оценивает текущие требования

к производительности и распределяет задачи таким образом, чтобы наилучшим образом использовать возможности каждого типа ядер. Это позволяет уменьшить потребление энергии, не теряя в производительности при изменении рабочих нагрузок.

Существует три основных способа управления ядрами в архитектуре *big.LITTLE*:

- *Heterogeneous Multi-Processing* (*HMP*) – все ядра, как большие, так и маленькие, активны одновременно. Планировщик может распределять задачи между любыми доступными ядрами, что обеспечивает высокую гибкость, но может быть менее энергоэффективным, так как более мощные ядра могут работать даже при минимальной нагрузке (рисунок 4.6);

- *Clustered switching* – ядра разделены на два кластера: один с большими ядрами и один с маленькими. Система переключается между этими кластерами в зависимости от нагрузки. Это улучшает энергоэффективность, так как большие ядра используются только для ресурсоёмких задач, а маленькие – для менее требовательных;

- *In-kernel switcher* включает в себя сочетание большого ядра с малым ядром, при этом в одном чипе может быть несколько одинаковых пар. Каждая пара работает как так называемое виртуальное ядро, и только одно реальное ядро (полностью) включается и работает в данный момент. «Большое» ядро используется при высокой нагрузке, а «малое» — при низкой. Когда нагрузка на виртуальное ядро изменяется (между высокой и низкой), включается новое ядро, состояние работы переносится, а старое ядро выключается, и обработка продолжается на новом ядре.

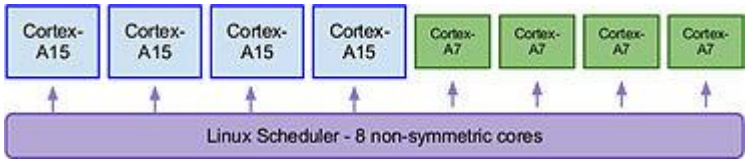

Рисунок 4.6 – *HMP* управление ядрами

Заключение

Архитектуры *ARM* и *x*86 представляют собой два основных подхода в проектировании микроЭВМ, каждый из которых имеет свои особенности и области применения. Архитектура *x*86, с её комплексной командной архитектурой *CISC*, традиционно используется в персональных компьютерах и серверах, обеспечивая высокую совместимость с предыдущими поколениями процессоров. Расширения, такие как *MMX*, *SSE* и *AVX*, позволяют улучшить производительность в задачах, требующих интенсивных вычислений, что делает её идеальной для широкого спектра приложений, включая мультимедиа и вычислительные ресурсоёмкие задачи.

В свою очередь, архитектура *ARM*, основанная на принципах *RISC*, отличается простотой и высокой энергоэффективностью, что делает её предпочтительным выбором для мобильных устройств и встроенных систем. Технология *big.LITTL*E в *ARM* позволяет эффективно использовать как мощные, так и энергоэффективные ядра, что обеспечит оптимальное распределение нагрузки и значительное сокращение потребления энергии при сохранении необходимой производительности.

Таким образом, обе архитектуры имеют свои сильные стороны, и их выбор зависит от специфики задачи. *x*86 продолжает доминировать в области высокопроизводительных вычислений и серверных решений, тогда как *ARM* становится всё более популярным в мобильных и энергоэффективных устройствах. Развитие обеих архитектур продолжает открывать новые возможности для технологий и приложений, требующих всё большей мощности и энергоэффективности.

Список использованных источников

1. Комаров В.М. Микропроцессорные системы: Учебное пособие. – 2 изд., перераб. и доп. – Рыбинск: РГАТА, 2004. – 173 с.

2. Таненбаум Э., Остин Т. Архитектура компьютера. 6-е изд. – СПб.: Питер, 2013. – 816 с.: ил.З.

3. Комаров В.М. Микропроцессорные системы: Учебное пособие. – 2 изд., перераб. и доп. – Рыбинск: РГАТА, 2004. – 163 с.

4. Орлов С. А., Цилькер Б. Я. Организация ЭВМ и систем: Учебник для вузов. 2-е изд. – СПб.: Питер, 2011. – 688 с.