Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

Пермский национальный исследовательский политехнический университет

(ПНИПУ)

Факультет: Электротехнический (ЭТФ)

Направление: 09.03.04 – Программная инженерия (ПИ)

Профиль: Разработка программно-информационной систем (РИС)

Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ

Зав. кафедрой ИТАС: д-р экон. наук, проф.

_____ Р.А. Файзрахманов

«_____»_____ 2023 г.

КУРСОВАЯ РАБОТА

по дисциплине

«Организация ЭВМ и систем»

на тему

«Структурно-алгоритмическое проектирование ЭВМ»

Студент: _____ Дерябин Кирилл Николаевич

(подпись, дата)

Группа: РИС-21-1бзу

Дата сдачи _____

Дата защиты _____

Оценка _____

Руководитель КР: к.т.н., доц. каф. ИТАС Погудин А.Л.

_____

(подпись, дата)

Пермь 2023

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет
(ПНИПУ)

Факультет: Электротехнический (ЭТФ)
Направление: 09.03.04 – Программная инженерия (ПИ)
Профиль: Разработка программно-информационной систем (РИС)
Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ
Зав. кафедрой ИТАС: д-р экон. наук,
проф.
_____ Р.А.
Файзрахманов
«_____»_____
2023 г.

ЗАДАНИЕ
на выполнение курсовой работы

Фамилия, имя, отчество: Дерябин Кирилл Николаевич
Факультет Электротехнический Группа РИС-21-1бзу
Начало выполнения работы: 16.05.2022
Контрольные сроки просмотра работы: 27.05, 04.06, 09.01, 11.01
Защита работы: 13.01.2023

1. Наименование темы: «Структурно-алгоритмическое проектирование ЭВМ».

2. Исходные данные к работе (проекта):

Объект исследования – Арифметико-логическое устройство (АЛУ).

Предмет исследования – Алгоритм работы и структура АЛУ.

Цель работы (проекта) – разработать операции сложения, вычитания, сравнения и поразрядного «И».

3. Содержание:

3.1 Исследование предметной области курсовой работы

3.2 Анализ исходных данных задания на курсовую работу

3.3 Спецификация устройства на уровне «черного ящика»

3.4 Представление устройства в виде операционной и управляющей частей

3.5 Разработка структуры устройства

3.6 Составление алгоритма работы устройства.

3.7 Разработка микропрограммы работы устройства

3.8 Составление полной спецификации устройства

3.9 Составление фрагмента функциональной схемы устройства

3.10 Разработка временной диаграммы работы устройства

3.11 Контрольный пример

Руководитель КР:	(подпись, дата)	к.т.н., доц. каф. ИТАС Погудин А.Л.
Задание получил:	(подпись, дата)	К.Н. Дерябин

КАЛЕНДАРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ
КУРСОВОЙ РАБОТЫ

№ пп	Этапы работы	Объём этапа, %	Сроки выполнения	
			Начало	Конец
1.	Исследование предметной области	10	16.05.22	20.05.22
2.	Устройство управления	5	23.05.22	27.05.22
3.	Сложение	5	30.05.22	03.06.22
4.	Вычитание	5	06.06.22	10.06.22
5.	Сравнение	5	13.06.22	17.06.22
6.	Логическое «И»	5	20.06.22	08.07.22
7.	Адресация	5	11.07.22	15.07.22
8.	Разработка устройства.	5	18.07.22	22.07.22
9.	Анализ исходных данных задания на курсовую работу	5	25.07.22	05.08.22
10.	Спецификация устройства на уровне «черного ящика»	5	08.08.22	12.08.22
11.	Представление черного ящика в виде операционной и управляющей частей	5	15.08.22	19.08.22
12.	Разработка структуры операционной части устройства	5	22.08.22	09.09.22
13.	Составление схемы алгоритма работы устройства и его микропрограммы	5	12.09.22	16.09.22
14.	Разработка схемы алгоритма работы	5	19.09.22	23.09.22

15.	Составление полной спецификации устройства	5	26.09.22	14.10.22
16.	Разработка фрагмента функциональной схемы	5	17.10.22	11.11.22
17.	Контрольный пример	5	14.11.22	09.12.22
18.	Временная диаграмма работы УУ	5	12.12.22	16.12.22
19.	Оформление курсовой работы	5	23.12.22	12.01.23
20.	Защита курсовой работы		13.01.23	

Руководитель КР: _____ (подпись, дата) к.т.н., доц. каф. ИТАС Погудин А.Л..

Задание получил: _____ (подпись, дата) Дерябин Кирилл Николаевич

РЕФЕРАТ

АРИФМЕТИКО-ЛОГИЧЕСКОЕ УСТРОЙСТВО (АЛУ), УПРАВЛЯЮЩЕЕ УСТРОЙСТВО (УУ), СЛОЖЕНИЕ, ВЫЧИТАНИЕ, СРАВЕНЕНИЕ, ПОРЯЗРЯДНОЕ ЛОГИЧЕСКОЕ «И» В СОСТАВЕ ИНТЕРПРЕТИРУЕМОЙ ВИРТУАЛЬНОЙ МАШИНЫ.

Цель работы – разработать операции сложения, вычитания, сравнения, логического «И» для работы в составе интерпретируемой виртуальной восьмиразрядной машины.

При разработке виртуальной машины будут использованы стандартные операции сложения, вычитания, сравнения чисел и логической операции «И». Так же будет разработан специальный набор инструкций с индивидуальными кодами операций, которые будут интерпретированы виртуальной машиной в процессе работы для обработки операций машинным кодом платформы и архитектуры (x86), под которую будет разрабатываться виртуальная машина.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ ИСПОЛЬЗУЕМЫХ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СОКРАЩЕНИЙ И ТЕРМИНОВ

АЛУ	Арифметико-логическое устройство
МО	Микрооперация
МПР	Микропрограмма

ВВЕДЕНИЕ

Цель работы – разработать операции сложения, вычитания, сравнения, логического «И» для работы в составе интерпретируемой виртуальной восьмиразрядной машины.

При разработке виртуальной машины будут использованы стандартные операции сложения, вычитания, сравнения чисел и логической операции «И». Так же будет разработан специальный набор инструкций с индивидуальными кодами операций, которые будут интерпретированы виртуальной машиной в процессе работы для обработки операций машинным кодом платформы и архитектуры (x86), под которую будет разрабатываться виртуальная машина.

Будут приведены примеры работы виртуальной машины.

Сложение и вычитание двоичных чисел

Прямой код числа — это представление беззнакового двоичного числа. Если речь идет о машинной арифметике, то, как правило на представление числа отводится определенное ограниченное число разрядов. Диапазон чисел, который можно представить числом разрядов n равен 2^n.

Обратный код числа, или дополнение до единицы (one's complement) — это инвертирование прямого кода (поэтому его еще называют инверсный код). То есть все нули заменяются на единицы, а единицы на нули.

Дополнительный код числа, или дополнение до двойки (two's complement) — это обратный код, к младшему значащему разряду которого прибавлена единица.

Предположим, что для работы с двоичными числами есть тетрада (4 бита). Таким образом можно представить 16 чисел ($2^4 = 16$) в диапазоне от 0 до 15.

00 – 0000

…

15 – 1111

Беззнаковые числа представляются в прямом коде, но для арифметических задач требуются и отрицательные числа. Поэтому отдадим отрицательному диапазону 8 чисел, а другие 8 чисел останутся для положительного диапазона. В таком случае получим знаковое число с диапазоном значений от -8 до +7. Для различия положительных и отрицательных чисел выделяют старший разряд числа, который называется знаковым (sign bit). 0 в этом разряде говорит нам о том, что это положительное число, а 1 — отрицательное.

Для представления чисел со знаком используется дополнительный код. Таким образом -7 в дополнительном коде получается, как прямой код 7 = 0111, обратный код 7 = 1000, дополнительный код 7 = 1001. Обратим внимание на то, что прямой код 1001 представляет число 9, которое отличается от числа -7 ровно на 16, или 2^4. Или, что тоже самое, дополнительный код числа «дополняет» прямой код до 2^n, то есть 7+9=16.

При таком представлении отрицательного числа операции сложения и вычитания можно реализовать одной схемой сложения, при этом очень легко определять переполнение результата.

Пара примеров.

7-3=4
0111 прямой код 7
1101 дополнительный код 3
0100 результат сложения 4

-1+7=6
1111 дополнительный код 1
0111 прямой код 7
0110 результат сложения 6

Арифметическое переполнение определяется по двум последним переносам, включая перенос за старший разряд. При этом если переносы 11 или 00, то переполнения не было, а если 01 или 10, то было. При этом, если переполнения не было, то выход за разряды можно игнорировать.

Рассмотрим несколько примеров с переносами

7+1=8
00111 прямой код 7
00001 прямой код 1
01110 переносы
01000 результат 8 — переполнение
Два последних переноса 01 — переполнение

-7+7=0
00111 прямой код 7
01001 дополнительный код 7
11110 переносы
10000 результат 16 — но пятый разряд можно игнорировать, реальный результат 0.

Два последних переноса 11 и перенос в пятый разряд можно отбросить, оставшийся результат, ноль, арифметически корректен. Опять же проверять на переполнение можно простейшей операцией XOR двух бит переносов. Обратный код дополняет число до 2^n-1, или до всех 1, потому и называется дополнением до 1.

Алгоритмы операций сложения, вычитания, сравнения, побитового «И»

В каждом из алгоритмов будет присутствовать операнд C, который сыграет роль аккумулятора и сохранит в себя результат операции. На самом деле в зависимости от инструкции, этот приемник может быть любым регистром, в котором предполагается разместить итоговые значения вычислений. Операнды A и B будут значениями, над которыми требуется провести какие-то действия.

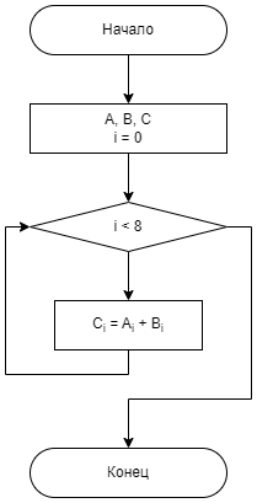

Рисунок 1 – Алгоритм операции сложения двух операндов A и B

На данной схеме изображено повторение действия сложения разрядов. Все биты поочередно складываются и направляются в результат.

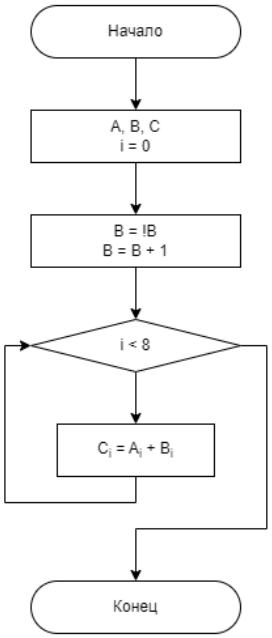

Рисунок 2 – Алгоритм операции вычитания двух операндов A и B

Алгоритм вычитания выполняется сложением, предварительно преобразовав число B в обратный код, а далее, прибавив к нему 1 - в дополнительный. Таким образом, получается аналогично сложить два числа и получить результат который бы выполняло вычитание.

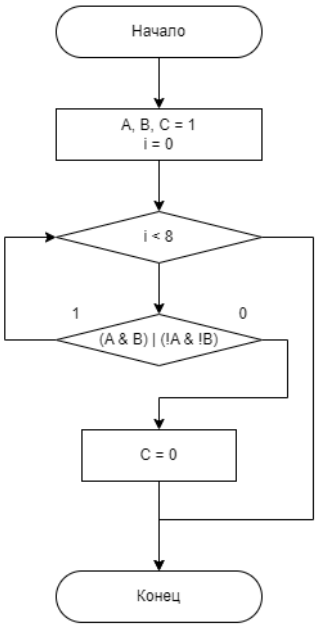

Рисунок 3 – Алгоритм операции сравнения двух операндов A и B

Предположим, что изначально C = 1 (A и B равны). Далее проверим каждый разряд числа используя логическое выражение XNOR (A & B) | (!A & !B). Если хоть один бит операнда отличается от второго, сравнение заканчивается, устанавливая флаг равенства C в 0.

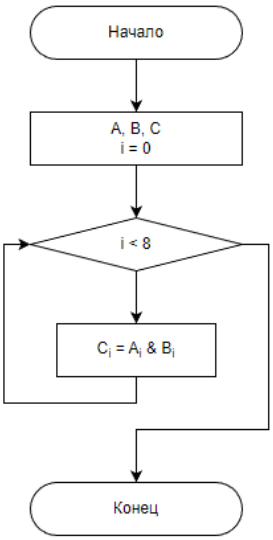

Рисунок 4 – Алгоритм поразрядного «И»

Аналогично операциям сложения и вычитания, выполняется поразрядное «И» для каждого разряда числа. Результат операции сохраняется в C.

Виртуальные машины и их назначение

Виртуальная машина – программная или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы. Наиболее часто виртуальная машина подразумевает под собой эмулятор архитектуры процессоров x86-64 для запуска каких-то гостевых ОС и основной ОС. Для этого используется аппаратная виртуализация, которая должна поддерживаться целевым процессором. Так же по мимо виртуальных машин для запуска систем, существуют программные виртуальные машины, которые значительно медленнее и используются для более простых задач.

Назначение программной виртуальной машины в основном направлено на кроссплатформенное выполнение программ. Виртуальная машина пишется индивидуально под каждую аппаратную платформу, но с соблюдением всех кодов операций виртуального процессора, что позволяет в конечном итоге запускать исполняемый файл написанный один раз – на всех системах и платформах. Это самый главный плюс виртуальных машин, но также есть и минус, это возможные временные издержки при выполнении, что может немного замедлять написанную программу.

С развитием виртуальных машин этому минусу было уделено довольно большое влияние, и вскоре были разработаны различные методики борьбы с падением производительности.

Виды программных виртуальных машин

Чтобы разобраться с падением производительности приложений, разберем виды виртуальных машин.

Первый и самый простой вид виртуальных машин – интерпретаторы. Виртуальный процессор имея все те же самые регистры перемещается по сегменту кода и выполняет коды операций путем сравнения. Весь путь интерпретации инструкций происходит как правило в одном цикле до завершения со стороны исполняемой программы или ошибке, и такие машины как правило выполняются в системе в одном потоке и не разделяются.

Второй вид виртуальных машин – виртуальные машины с JIT компилятором. Часто используемые куски кода компилируются в машинный код чтобы выполняться быстрее.

Управляемый и неуправляемый код

Неуправляемый код – это машинный код, выполняемый на аппаратной платформе. Вся проблема состоит в том, что при написании кода на языках asm/C/C++ и подобных языках программирования, которые впоследствии будут скомпилированы в машинный код, вся ответственность за используемые ресурсы программы и безопасность отвечает целиком и полностью программист. Если компилятор в состоянии анализировать возможную ошибку или предупредить программиста о непредсказуемом поведении на этапе компиляции программы, то появление логических ошибок и ошибок работы с памятью или привилегиями выявляются в «режиме отладки». При отладке, конечно же помогает информация о сохранении символов, которая фактически сопоставляет исходный код с адресами, чтобы дать программисту возможность наглядно и пошагово выполнять программу, не бегая взглядом по дизассемблированному коду, анализ которого в больших проектах может занять довольно много времени, а в удобно читаемом виде ориентироваться в своем же исходном коде в период проведения отладки. Естественно, всю эту полезную информацию подготавливает компилятор в период компиляции программы. Эта информация часто нужна на этапе тестирования приложения и вскоре после стабильной версии становится бесполезной. Без этой информации при возникновении необработанной исключительной ситуации программа как правило автоматически снимается системой с исполнения оставляя дамп памяти для запуска его в отладчике. Естественно, при отсутствии pdb файла (portable database) с символами, отладчик указывает на адрес возникновения проблемы в дизассемблере. Приходится анализировать содержимое регистров, естественно понимая шаблоны генерации кода определенным компилятором языка высокого уровня.

В случае с языками C/C++, самая страшная ошибка безопасности — это переполнение буфера. При переполнении массива на стеке, происходит перезапись адреса возврата, который был занесен в стек при исполнении инструкции CALL. Этим можно воспользоваться, переместив выполнение в другой участок кода после выполнения инструкции возврата из процедуры (RET), или даже выполнить свой код. Грамотное выполнение шелл кода может дать возможность даже запустить программу или вызвать какие-нибудь API функции системы. Допустим в Windows PEB (Process Environment Block) лежит по адресу FS:[30h] в x86, или в регистре GS по смещению 60h в 64-х разрядной архитектуре. Воспользовавшись данными в PEB мы можем узнать всю информацию с адресами всех загруженных в процессе модулей, крайне нужной для хакеров из которых как правило выступает ntdll.dll. Используя функции ntdll можно выдать себе необходимые права для запуска драйвера и получить права ядра.

Управляемый код – является байт кодом, который управляется средой выполнения (виртуальной машиной). Типичным представителем такого подхода является .NET. Плюс данного решения в том, что оно избавляет программиста от долгих раздумий о возникшей ошибке и делает понимание ошибки простым. Так же, такой подход является безопасным, в связи с тем, что если допустим мы не указали размер массива, и передали его в функцию копирования той же строки которая больше по размеру чем размер нашего массива, средой выполнения сразу же будет сгенерировано исключение и в понятном виде выведено в MessageBox, что даст программисту довольно быстро понять ошибку, а так же даст возможность пользователям программы сообщить разработчику о возникновении какой то проблемы, что является весьма удобным способом.

Так же, среда выполнения может предоставлять такие важные службы как автоматизированное управление памятью. В этом случае работа программиста совсем упрощается, что вовсе избавляет от пристальной слежки за памятью и в конечном итоге ускоряет процесс разработки. Чего не сказать про производительность. Сборщики мусора могут требовать определенное количество ресурсов ЦП при выяснении, какой объект нужен, а какой уже можно освободить. Чтобы оптимизировать работу программ, C# компилируется в «промежуточный байт код» (IL) еще называемый PCODE, компиляцию

которого в машинный берет на себя среда выполнения, реализованная под определенную аппаратную платформу.

.NET (Dot Net)

.NET (ранее известна как .NET Core) — это модульная платформа для разработки программного обеспечения с открытым исходным кодом, разработанная компанией Micorosft. Поддерживает такие операционные системы как Windows, Linux, macOS. Поддерживает такие языки программирования как C#, Visual Basic .NET и F#.

.NET основана на .NET Framework. Платформа .NET отличается от неё модульностью, кроссплатформенностью, возможностью применения облачных технологий, и тем, что в ней произошло разделение между библиотекой CoreFX и средой выполнения CoreCLR.

.NET — модульная платформа. Каждый её компонент обновляется через менеджер пакетов NuGet, а значит можно обновлять её модули по отдельности, в то время как .NET Framework обновляется целиком. Каждое приложение может работать с разными модулями и не зависит от единого обновления платформы.

CoreFX — это библиотека, интегрированная в .NET. Среди её компонентов: System.Collections, System.IO, System.Xml.

CoreCLR — это среда выполнения, включающая в себя RyuJIT (JIT-компилятор), встроенный сборщик мусора и другие компоненты.

Java

Рисунок 5 – Логотип Java

Java Virtual Machine (сокращенно Java VM, JVM) — виртуальная машина Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (javac). JVM может также использоваться для выполнения программ, написанных на других языках программирования. Например, исходный код на языке Ada может быть скомпилирован в байт-код Java, который затем может выполниться с помощью JVM.

JVM является ключевым компонентом платформы Java. Так как виртуальные машины Java доступны для многих аппаратных и программных платформ, Java может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа. Использование одного байт-кода для многих платформ позволяет описать Java как «скомпилируй единожды, запускай везде» (compile once, run anywhere).

Виртуальные машины Java обычно содержат интерпретатор байт-кода, однако, для повышения производительности во многих машинах также применяется JIT-компиляция часто исполняемых фрагментов байт-кода в машинный код.

Обзор архитектуры JVM на базе версии Java SE 7 представлен ниже

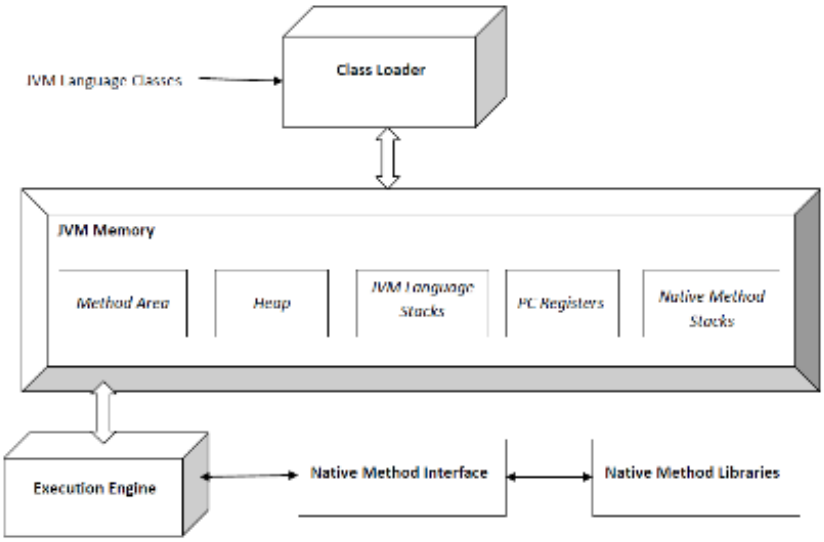

Рисунок 6 – Архитектура JVM

В начале развития платформы Java существовали две конкурирующие реализации Java VM. Первая реализация от Sun Microsystems, вторая от Microsoft, специально оптимизированная для выполнения Java кода на платформе Windows.

Сегодня Java до сих пор является очень востребованным языком программирования, используемым для написания кроссплатформенных приложений.

Скриптовый язык Pawn

Наиболее ярким примером виртуальных машин-интерпретаторов является свободная виртуальная машина встраиваемого скриптового языка Pawn, авторами которой является компания CompuPhase.

Pawn скрипт компилируется в PCODE, коды операций виртуальной машины которые интерпретируются. Код скомпилированного скрипта является полностью управляемым, способным прервать выполнение в любом месте и дать всю информацию о произошедшей проблеме, в отличие от неуправляемого машинного кода.

С реализацией сборщика Марка Питера, виртуальная машина стала поддерживать JIT и стала еще быстрее, а сам PCODE скомпилированного скрипта стал играть роль промежуточного кода.

Компания по сей день поддерживает этот язык и выпускает как обновления, так и занимается исправлением известных ошибок. Так как проект открытый, участники проекта выполняют все исправления и обновления в свое свободное время.

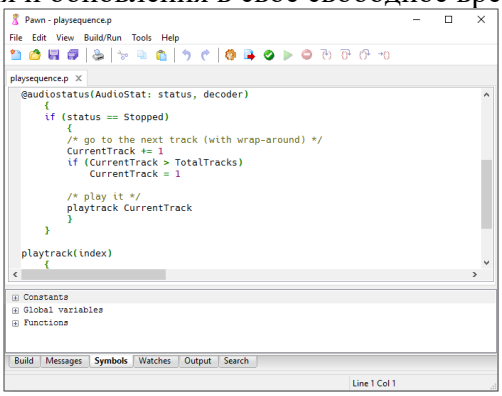

Рисунок 7 – Pawn IDE

Спустя некоторое время после выхода и некоторых исправлений языка, его стали использовать для написания плагинов и модов под различные игры, яркими примерами которых являются Multi Theft Auto, Counter Strike 1.6.

Далее в 2005 году некоммерческое сообщество разработчиков Alliedmods, занимающаяся разработкой утилит для плагинов на различные игры, берет на вооружение язык Pawn с его виртуальной машиной и выпускает AmxModX. Синтаксис и принципы работы виртуальной машины по сей день остаются оригинальными, чего не сказать про следующую платформу для игр на движке Source – Source Mod. В данном проекте Alliedmods переработали синтаксис языка и варианты его отладки. Компилятор был переписан полностью на C++. Сейчас виртуальная машина использует исключительно JIT, если реализации под целевую аппаратную платформу существуют. В противном случае, включается интерпретатор.

Так же существуют множество проектов, которые предпочитают поддерживать кроссплатформенные плагины/модули, и они используют именно виртуальную машину Pawn за ее простоту и понятность.

Реализация собственной интерпретируемой виртуальной машины

В ходе данной работы был разработан набор команд для обработки виртуальным процессором. Далее была реализована интерпретируемая виртуальная машина, которая обрабатывала написанную на байт-коде тестовую программу. Проект написан на языке C 99.

Виртуальная машина содержит 4 регистра общего назначения, 3 сегментных регистра, и четыре служебных регистра, нужных для сохранения адреса возврата перед вызовом процедуры, флагов, указателя на границу стека и указателя на текущую инструкцию.

Рассмотрим таблицу регистров виртуальной машины с более подробной информацией ниже

Имя регистра	Разрядность	Назначение	Примечание
A (accumulator)	8 бит	Аккумулятор, общее назначение	Доступны программе
B		Общее назначение	
C			
D			
CS (code segment)	16+ бит	Сегментный регистр кода	Недоступны программе
DS (data segment)		Сегментный регистр данных	
SS (stack segment)		Сегментный регистр стека	
IP (instruction pointer)		Смещение от начала сегмента кода до текущей исполняемой инструкции	
SP (stack pointer)		Указатель на границу стека	
PIP (previous instruction pointer)		Адрес возврата Устанавливается инструкцией CALL	
FLAGS	8 бит	Регистр флагов	

Доступ к некоторым регистрам не был организован чтобы продемонстрировать базовую концепцию виртуальной машины. Если доступ будет необходим, его можно легко реализовать.

Виртуальная машина в данный момент использует разделенную память под каждый сегмент, но при необходимости можно составить плоскую модель памяти сложив все

размеры всех сегментов и выделив один блок памяти на весь исполняемый файл. Этот вариант упростит освобождение памяти сделав его единоразовым вызовом функции free.

Контекст интерпретатора

Все регистры и некоторые служебные данные хранятся в контексте виртуальной машины. Контекст – состояние всех регистров в данный момент с дополнительной специальной информацией нужной для интерпретатора.

Рассмотрим структуру контекста интерпретатора.

```
/* virtual processor context */
typedef struct vcpu_context_s {
	int number_of_externals;
	vm_external_func_def_t *p_extrns;
	vm_callbacks_dt_t *p_callbacks;
	int vm_flags;
	int code_size;
	int stack_size;
	int data_size;
	struct {
		union {
			struct { register_t A, B, C, D; };
			register_t regs[4];
		};
		unsigned char *CS, *DS, *SS;
		int IP, SP;
		flags_t FLAGS;
		register_t PIP;
	} cpuregs;
} vcpu_context_t;
```

Планируется добавить реализации внешних функций, чтобы программа, работающая в виртуальной машине, могла получать, например имя процесса виртуальной машины, или открывать файл и записывать туда данные. Количество этих внешних функций определяется полем number_of_externals. Сам указатель на массив функций и имен содержит адрес начала массива для доступа к этой информации интерпретатору.

```
typedef struct vm_callbacks_dt_s {
	VM_DEBUG_INTRUCTION_EXEC (*vm_debug_instruction_step)(vcpu_context_t *p_context);
	VM_DEBUG_INTRUCTION_EXEC (*vm_breakpoint_raised)(vcpu_context_t *p_context);
	void                     (*vm_instruction)(vcpu_context_t *p_context);
} vm_callbacks_dt_t;
```

vm_callbacks_dt_t – тип таблица диспетчеризации, содержащая указатели на нужные функции. Эта таблица хранит указатели на такие функции как vm_debug_instruction_step, vm_breakpoint_raised и vm_instruction. Если функции не заданы, по умолчанию они равны NULL и будут игнорированы интерпретатором с помощью проверки адреса перед вызовом.

vm_debug_instruction_step – функция шага выполнения (для отладки). Может возвращать 4 состояния.

VM_EXEC_NEXT – говорит о том, что виртуальная машина может продолжить исполнение инструкций после обработанной.

VM_EXEC_ENABLE_STEPEXEC – включает пошаговое выполнение после обработки текущей инструкции.

VM_EXEC_DISABLE_STEPEXEC – выключает пошаговое выполнение после обработки текущей инструкции.

VM_EXEC_TERMINATE – завершает выполнение инструкций и устанавливает статус завершения работы интерпретатора VM_MANUALLY_TERMINATED.

vm_breakpoint_raised – вызывается если интерпретатор встречает точку останова (OP_BRK). Дальнейшее исполнение определяется все теми же состояниями.

Вернемся к полям структуры vcpu_context_s. Поле vm_flags флаги текущего состояния интерпретатора, которые могут иметь следующие значения:

VM_FEXEC – флаг выполнения. Пока данный флаг выставлен, интерпретатор продолжает выполнение. Бит флага может быть изменен на 0 при ошибке сегментации или неизвестном коде операции что приведет к завершению выполнения.

VM_FSTEPEXEC – флаг шага. Пока данный флаг выставлен, интерпретатор вызывает функцию пошагового выполнения и ожидает обратной передачи управления.

Следующее поле code_size содержит размер сегмента кода, stack_size – размер стека, а data_size – размер данных (не используется).

Далее в структуре cpuregs находятся все регистры. К регистрам общего назначения из интерпретатора можно обращаться непосредственно по именам A, B, C, D, либо по индексам в массиве этих же регистров regs, т.к они находятся внутри объединения (union).

Поля CS, DS, SS являются указателями, в которых хранятся адреса начала сегментов либо выставленных вручную, либо установленных загрузчиком исполняемого файла.

Краткая таблица кодов операций с описанием

Байты инструкции	Мнемоника	Назначение	Влияние на FLAGS
0 Размер 1 байт	nop	Ничего не выполняет	
Коды операций 1 – 4 (с константой) Размер 2 байта Пример: 1 10 1 – код операции 10 - значение 5 – 16 (с регистром) Размер 1 байт	mov r, R/imm8	Запись константы в регистр A Запись значения из регистра R в регистр r	Не реализовано
Коды операций 17 – 28 Размер 1 байт	xch r, r	Обмен регистров значениями	
Коды операций 29-32 (с константой) Размер 2 байта Пример: 29 10 29 – код операции 10 – значение 33 – 48 (с регистром) Размер 1 байт	add r, R/imm8	Складывает регистр r с константой. Результат отправляется в регистр r. Складывает регистр r с регистром R. Результат отправляется в регистр r.	
Коды операций 49 – 52 (с константой) Размер 2 байта Коды операций 53 – 68 (с регистром)	sub r, R/imm8	Вычитает из регистра r константу и помещает результат в регистр r. Вычитает из регистра r значение в регистре R и	

Размер 1 байт		помещает результат в регистр r.	
Коды операций 69 – 72 (с константой) Размер 2 байта Коды операций 73 – 84 (с регистром) Размер 1 байт	cmp r, R/imm8	Выполняет сравнение регистра r с константой путем вычитания. Выполняет сравнение регистра r с регистром R путем вычитания	Влияет на 1 бит флага (zero flag) при нулевом результате. Влияет на 2 бит флага (sign flag) если первый операнд меньше второго.
Коды операций 85 – 88 (с константой) Размер 2 байта Коды операций 89 – 100 (с регистром) Размер 1 байт	and r, R/imm8	Выполняет логическое И между всеми битами регистра r и (регистра R / константы) и записывает результат в первый операнд.	Не реализовано
Код операции 101 (с константой) Размер 2 байта Коды операций 102 – 105 (с регистром) Размер 1 байт	push r/imm8	Поместить операнд в стек Завершает выполнение интерпретатора с ошибкой VM_STACK_OVERFLOW в случае выхода за границу SS.	
Код операции 106 (с константой) Размер 2 байта Коды операций 107 – 110 (с регистром) Размер 1 байт	pop (r)	Извлечь из стека Завершает выполнение интерпретатора с ошибкой VM_STACK_OVERFLOWL в случае выхода за границу SS в низ по адресам.	
Код операции 111 (с константой) Размер 2 байта Коды операций 112 – 115 (с регистром) Размер 1 байт	jmp r/imm8	Выполнить безусловный переход относительно текущего IP на значение регистра r/константы. Если требуется выполнить переход назад, аргументом должно быть отрицательное число.	
Код операции 116 (с константой) Размер 2 байта Коды операций 117 – 119 (с регистром) Размер 1 байт	jnz r/imm8	Переход по адресу из (регистра r/константы) пока аккумулятор не 0.	

Код операции 120 (с константой) Размер 2 байта Коды операций 121 – 123 (с регистром) Размер 1 байт	jz r/imm8	Переход по адресу из (регистра r/константы) пока аккумулятор 0.	
Код операции 124 (с константой) Размер 2 байта Коды операций 125 – 128 (с регистром) Размер 1 байт	je r/imm8	Переход по адресу из (регистра r/константы) если выставлен флаг нуля	
Код операции 129 (с константой) Размер 2 байта Коды операций 130 – 133 (с регистром) Размер 1 байт	jne r/imm8	Переход по адресу из (регистра r/константы) если не выставлен флаг нуля	
Код операции 134 (с константой) Размер 2 байта Коды операций 135 – 138 (с регистром) Размер 1 байт	jl r/imm8	Переход по адресу из (регистра r/константы) если меньше (выставлен флаг знака)	
Код операции 139 (с константой) Размер 2 байта Коды операций 140 – 143 (с регистром) Размер 1 байт	jle r/imm8	Переход по адресу из (регистра r/константы) если меньше либо равно (выставлен флаг знака или флаг нуля)	
Код операции 144 (с константой) Размер 2 байта Коды операций 145 – 148 (с регистром) Размер 1 байт	jg r/imm8	Переход по адресу из (регистра r/константы) если больше (не выставлен флаг знака)	
Код операции 149 (с константой) Размер 2 байта Коды операций 150 – 153 (с регистром) Размер 1 байт	jge r/imm8	Переход по адресу из (регистра r/константы) если больше или равно (не выставлен флаг знака или выставлен флаг нуля)	

Код операции 154 (с константой) Размер 2 байта Коды операций 155 – 158 (с регистром) Размер 1 байт	call r/imm8	Вызов процедуры по адресу из (регистра r/константы) Прежнее значение IP заносится в регистр PIP и может быть восстановлено инструкцией ret	
Коды операций 159 – 162 (с регистром) Размер 1 байт	inc r	Инкремент значения регистра	
Коды операций 163 – 166 (с регистром) Размер 1 байт	dec r	Декремент значения регистра	
Код операции 167 Размер 1 байт	ret	Возврат из процедуры	
Специфичные операции			
Код операции 169 Размер 1 байт	brk	Точка останова Прерывает выполнение программы до ответа отладчика.	Не реализовано
Код операции 170 Размер 1 байт	hlt	Останов. Прерывает выполнение программы без возможности возобновления.	

Выполнение байт-кода виртуальной машиной

Выполнение байт кода построено путем массива функций-обработчиков, индексами в котором выступает код операции. Стоит отметить, что инструкции brk (OP_BRK) и hlt (OP_HALT) обрабатываются интерпретатором напрямую, поэтому они не находятся в этом массиве. Однако, эти две инструкции можно получить в обратном вызове vm_instruction.

В файле vm_defs.h есть тип данных указателя на функцию обработчик. Выглядит она следующим образом

typedef int(*instruction_handler_pfn)(int opcode, vcpu_context_t *p_vmctx);

Первым аргументом передается текущий код операции, вторым указатель на созданный контекст виртуальной машины. В связи с тем, что контекст содержит в себе все необходимые данные, обработчик может выполнять любые действия с виртуальной машиной включая изменение служебных флагов интерпретатора, так же полностью манипулировать всеми регистрами.

Для примера рассмотрим обработчик инструкции mov для записи константы в аккумулятор.

```
int vm_ih_mov_a_const(int opcode, vcpu_context_t *p_vmctx)
{
	p_vmctx->cpuregs.A = p_vmctx->cpuregs.CS[p_vmctx->cpuregs.IP];
	return sizeof(register_t); //size of instruction parameter
}
```

Функция должна возвращать количество байт данных, чтобы интерпретатор мог прибавить к IP размер инструкции + размер аргумента. Как можно видеть, записываем в регистр A значение лежащее после кода операции в сегменте кода.

Сам же интерпретатор реализован в функции vm_start_execution, параметром которой должен быть существующий и настроенный контекст, а возвращаемое значение будет статусом завершения. Статусов завершения несколько:

VM_ERROR_NONE – программа успешно завершена и никаких ошибок не произошло.

VM_ERROR_ACCESS_VIOLATION – в ходе работы программы произошла ошибка сегментации. Инструкция обратилась к несуществующему адресу памяти либо IP вышел за пределы размера сегмента кода.

VM_ERROR_UNKNOWN_INSTRUCTION – код операции не может быть обработан потому что не существует. Эта ошибка может возникать если кодом операции является отрицательное число либо число, превышающее MAX_OPCODES.

VM_MANUALLY_TERMINATED – выполнение было прервано по желанию пользователя. Этот код завершения может возвращаться в случае, если в функциях обратного вызова шагового выполнения или обработчика точки останова было возвращено состояние VM_EXEC_TERMINATE.

VM_STACK_OVERFLOWL – SP вышел за пределы сегмента стека в левую сторону.

VM_STACK_OVERFLOW – SP вышел за пределы сегмента стека в правую сторону.

Интерпретатор получает текущий код операции на который в данный момент указывает IP, и обрабатывает инструкции brk и hlt.

Если работа программы не была прервана, проверяется существование такого кода операции. Если код операции существует, вызывается нужный обработчик под индексом, которым является код операции, предварительно сохранив код операции и увеличив значение IP, чтобы автоматически иметь позицию на аргументе кода операции. Если аргумента нет, в обработчике возвращаем 0, т. к. после выполнения обработчика к IP будет прибавлено его возвращаемое значение.

Далее цикл повторяется до тех пор, пока IP не достигнет конца сегмента кода, либо он не будет прерван самой программой или ошибкой.

Инициализация контекста виртуальной машины

Контекст как уже говорилось выше, служит для сохранения важной информации о таких вещах как регистры и служебные данные интерпретатора. Инициализируем контекст.

Я рекомендую использовать memset из string.h чтобы заполнить структуру контекста нулями во избежание возможных ошибок. Пример инициализации:

```
vcpu_context_t context;
memset(&context, 0, sizeof(context));
context.vm_flags |= VM_FEXEC;
context.code_size = sizeof(code);
context.cpuregs.CS = (char *)code;
context.stack_size = sizeof(stack);
context.cpuregs.SS = (char *)stack;
context.p_callbacks = адрес таблицы дисперчеризации;

int status = vm_start_execution(&context);
if (status != VM_ERROR_NONE) {
	printf("VM_EXECUTION: execution finished with errors!. Error code: %d\n", status);
	return 1;
}
```

Не забываем выставить флаг VM_FEXEC, говорящий интерпретатору что требуется выполнять код, переданный в контексте, иначе, выполнение будет сразу завершено. Если требуется пошаговое выполнение, добавляем флаг VM_FSTEPEXEC. После этого интерпретатор перед выполнением каждого кода операции будет вызывать функцию vm_debug_instruction_step в таблице диспетчеризации.

Тестовая программа, написанная кодами операций виртуальной машины

Рассмотрим код программы, выполняемой на виртуальной машине и разберемся как он работает по шагам.

```
    static char code[] = {
                                        //      start:
            OP_MOV_A_CONST, 10,         //0+2       mov a, 10
            OP_MOV_B_CONST, 10,         //2+2       mov b, 10
            OP_ADD_A_B,                 //4         add a, b
            OP_XCH_A_B,                 //5         xch a, b

            OP_PUSH_A,                  //6         push a
            OP_PUSH_B,                  //7         push b
            OP_CALL_CONST,      12,     //8+2       call proc1
            OP_POP_A,                   //10        pop a
            OP_HALT,                    //11        hlt

                                        //      proc1:
            OP_POP_B,                   //12        pop b
            OP_POP_A,                   //13        pop a

                                        //      label1:
            OP_DEC_A,                   //14        dec a
            OP_INC_B,                   //15        inc b
            OP_JNZ_CONST, 14,           //16+2      jnz label1
            OP_PUSH_B,                  //18        push b
            OP_RET,                     //19        ret
            OP_BRK                      //20        brk
    };
```

1. В регистры A и B записываются два числа 10.
2. Регистр A складывается с регистром B, результат сохранен в A.
3. Выполняем обмен значениями между регистром A и B.
4. Помещаем значения регистров A и B в стек
5. Выполняем вызов процедуры по жесткому адресу (адреса рассчитаны вручную)
6. Выполнение переносится в процедуру proc1 где мы получаем из стека в регистры B и A ранее записанные данные в обратном порядке (принцип LIFO). Обратим внимание на SP и IP регистры.
7. Делаем декремент числа в регистре A и инкремент числа в B.
8. Повторяем прыжок на адрес инструкции «dec a» до тех пор, пока значение в A не станет равно нулю.
9. Регистр A стал равен 0. Поместим результат регистра B в стек.
10. Вызовем возврат из процедуры и попадем на адрес 10
11. Прочитаем данные из стека в регистр A.
12. Выполним останов.

Таким образом, в итоге мы получаем значение 20 в регистрах A и B. Посмотрим состояния регистров.

```
op OP_DEC_A on addr 14  (rA: 6 | rB: 14 | rC: 0 | rD: 0 | IP: 14 | SP: 0 | PIP: 10 )
op OP_INC_B on addr 15  (rA: 5 | rB: 14 | rC: 0 | rD: 0 | IP: 15 | SP: 0 | PIP: 10 )
op OP_JNZ_CONST on addr 16  (rA: 5 | rB: 15 | rC: 0 | rD: 0 | IP: 16 | SP: 0 | PIP: 10 )
op OP_DEC_A on addr 14  (rA: 5 | rB: 15 | rC: 0 | rD: 0 | IP: 14 | SP: 0 | PIP: 10 )
op OP_INC_B on addr 15  (rA: 4 | rB: 15 | rC: 0 | rD: 0 | IP: 15 | SP: 0 | PIP: 10 )
op OP_JNZ_CONST on addr 16  (rA: 4 | rB: 16 | rC: 0 | rD: 0 | IP: 16 | SP: 0 | PIP: 10 )
op OP_DEC_A on addr 14  (rA: 4 | rB: 16 | rC: 0 | rD: 0 | IP: 14 | SP: 0 | PIP: 10 )
op OP_INC_B on addr 15  (rA: 3 | rB: 16 | rC: 0 | rD: 0 | IP: 15 | SP: 0 | PIP: 10 )
op OP_JNZ_CONST on addr 16  (rA: 3 | rB: 17 | rC: 0 | rD: 0 | IP: 16 | SP: 0 | PIP: 10 )
op OP_DEC_A on addr 14  (rA: 3 | rB: 17 | rC: 0 | rD: 0 | IP: 14 | SP: 0 | PIP: 10 )
op OP_INC_B on addr 15  (rA: 2 | rB: 17 | rC: 0 | rD: 0 | IP: 15 | SP: 0 | PIP: 10 )
op OP_JNZ_CONST on addr 16  (rA: 2 | rB: 18 | rC: 0 | rD: 0 | IP: 16 | SP: 0 | PIP: 10 )
op OP_DEC_A on addr 14  (rA: 2 | rB: 18 | rC: 0 | rD: 0 | IP: 14 | SP: 0 | PIP: 10 )
op OP_INC_B on addr 15  (rA: 1 | rB: 18 | rC: 0 | rD: 0 | IP: 15 | SP: 0 | PIP: 10 )
op OP_JNZ_CONST on addr 16  (rA: 1 | rB: 19 | rC: 0 | rD: 0 | IP: 16 | SP: 0 | PIP: 10 )
op OP_DEC_A on addr 14  (rA: 1 | rB: 19 | rC: 0 | rD: 0 | IP: 14 | SP: 0 | PIP: 10 )
op OP_INC_B on addr 15  (rA: 0 | rB: 19 | rC: 0 | rD: 0 | IP: 15 | SP: 0 | PIP: 10 )
op OP_JNZ_CONST on addr 16  (rA: 0 | rB: 20 | rC: 0 | rD: 0 | IP: 16 | SP: 0 | PIP: 10 )
op OP_PUSH_B on addr 18  (rA: 0 | rB: 20 | rC: 0 | rD: 0 | IP: 18 | SP: 0 | PIP: 10 )
op OP_RET on addr 19  (rA: 0 | rB: 20 | rC: 0 | rD: 0 | IP: 19 | SP: 1 | PIP: 10 )
op OP_POP_A on addr 10  (rA: 0 | rB: 20 | rC: 0 | rD: 0 | IP: 10 | SP: 1 | PIP: 10 )
op OP_HALT on addr 11  (rA: 20 | rB: 20 | rC: 0 | rD: 0 | IP: 11 | SP: 0 | PIP: 10 )
HALT Instruction at address 0xb  vCPU program has terminated!
VM_EXECUTION: Execution completed succesfully!

D:\source\repos\processor\Debug\processor2.exe (процесс 6168) завершает работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, установите параметр "Сервис" -> "Параметры" -> "Отладка" ->
 "Автоматически закрыть консоль при остановке отладки".
Чтобы закрыть это окно, нажмите любую клавишу…
```

Рисунок 8 – Состояния регистров виртуальной машины во время выполнения инструкций

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Стрыгин В.В., Щарев Л.С., Основы вычислительной микропроцессорной техники и программирования. Москва «Высшая школа» 1989. – 479 с
2. Павловская, Т.А. C/C++. Программирование на языке высокого уровня: Учеб. пособие. – СПб.:Питер, 2007. – 461 с.
3. Жмакин, А. П. Архитектура ЭВМ: 2-е изд., перераб. и доп.: учеб. пособие. — СПб.: БХВ-Петербург, 2010. — 352 с.
4. Потапов, В.И., Шафеева, О.П., Червенчук, И.В. Основы компьютерной арифметики и логики: Учеб. пособие. – Омск: Изд- во ОмГТУ, 2004. – 172 с
5. Потапов, И. В. Элементы прикладной теории цифровых автоматов: учеб. пособие / И. В. Потапов. – Омск:

Интернет ресурс: https://planetcalc.ru