# Вступление

Дата рождения - 18.09.2016.

Ну привет. Знаешь, а ты везунчик. Почему? Потому что ты сейчас читаешь самый шикарный русскоязычный док по ромхакингу. Свежие версии качай из темы на сайте или спрашивай лично у меня.

Меня зовут Кай, я автор этого учебника. Свой первый хак я сделал 11го мая 2014го, это была игра Tecmo World Cup Soccer. С августа 2015го делаю хаки для турниров группы Nekketsu. В июле 2016го выпустил нашу фирменную многоигровку. Я работаю только с эмулятором FCEUX.

Я тоже был новичком, поэтому этот учебник будет написан так, каким бы я хотел видеть его в свой первый день обучения. Он расчитан на пошаговое обучение, с подробным разбором каждого этапа. Если ты не планируешь всерьез заниматься ромхакингом, возможно тебе подошли бы более краткие статьи из интернета, поскольку информации здесь будет очень много.

Я буду признателен, если ты напишешь мне и:
• укажешь на ошибки, даже орфографические. Я пишу довольно грамотно, но знаки препинания могу ставить по смоему усмотрению (авторские знаки типа).
• попросишь уточнить какой-то момент из статьи. Предварительно убедись, что ты прочитал и усвоил инфу из предыдущих разделов.
• предложишь идею по улучшению учебника или полезную статью. Статьи я добавляю сюда, пока я не уделял этому разделу много времени.
• захочешь меня как-то отблагодарить. Лайки и донаты приветствуются.

И небольшое напутствие перед обучением: меняй в игре все что пожелаешь, если считаешь что игра станет лучше чем она есть сейчас.

Сэнсэй Кайлыч наставляет молодых ромхакеров

## Все необходимое для работы

Скачиваешь архив по этой ссылке **https://yadi.sk/d/kmyUQXnl35deRt**

**FCEUX** - эмулятор, на котором ты будешь работать. В нем есть все что нужно для ромхакинга. Здесь я описывать его не буду, более подробно я расскажу о нем в ходе твоего обучения. В моей сборке 2 exe'шника с разными версиями, я в основном пользуюсь старой версией 2.2.**2** по некоторым причинам, и в большинстве случаев примеры в учебнике будут с нее. Храни ромы в папке **ROMS**, чтобы не валялись где попало на диске.

В архиве лежит папка ***Программы***. В нее я скинул то, что может тебе пригодиться, и чем пользуюсь я сам.

**Ammy Admin** - для отображения своего экрана собеседнику, аналолична Team Viewer, но с ограниченным функционалом. Некоторые антивирусы могут ругаться. С этой прогой я смогу оказать тебе небольшую консультацию онлайн. Для голосового общения можно использовать **Discord** https://discord.gg/pjqAQGg или группу **RaidCall** 4749936. Я пока не знаю буду ли я вообще кого-то консультировать лично. Ты пиши если что, там посмотрим.

**HashTab** - для вычисления контрольных сумм файлов. После установки в свойствах файла появится вкладка с хеш-суммами, откуда их легко скопировать и сравнить с другим файлом. Эта программа пригодится тебе при выкладывании хаков на **romhacking.net** для указания всей необходимой инфы про оригинальный ром.

**HexCmp** - для сравнения байтов двух ромов. Удобно искать баги, вызванные своим же кодом. Если у тебя в старой версии рома все стабильно, а в более свежей версии начались баги, сравниваешь оба рома и находишь причину.

**HxD** - отличный альтернативный Hex-редактор. Его я использую чтобы быстро выделить нужные мне байты рома для переноса их в мою многоигровку, потому что делать это через встроенный Hex-редактор эмулятора очень геморройно. Также с его помощью можно расширить ром. Для всего остального мне хватает эмуляторного.

**Lunar IPS** - для создания патча. Патч - это файл, который содержит в себе информацию о различии между двумя ромами. Например ты взял оригинальный ром, изменил в нем что-то, и сохранил отдельно. Теперь у тебя 2 разных рома, оригинал и твой хак. С помощью этой программы ты создаешь патч, который уже содержит в себе информацию о том, что ты изменил, и ты можешь наложить этот патч на оригинальный ром, чтобы из оригинального рома получился твой хак. Патчи считаются более-менее легальным способом выкладывания своих хаков, поскольку патч не содержит в себе игру, а только отличия одного рома от другого. То есть ты не нарушаешь закон выложив этот патч. Закон нарушит тот, кто им воспользуется. На сайтах вроде **romhacking.net** тебе потребуется выложить именно патч, поскольку выкладывать сам хак запрещено.

**YY-CHR** - для редактирования графики. Очень удобная программа, про работу с ней я расскажу в будущем.

**Калькулятор** - стандартный калькулятор Windows 7. В нем жмешь ***Вид*** - ***Программист***. Этим калькулятором ты будешь пользоваться очень часто.

И последняя, но не менее важная вещь. Тебе нужна возможность быстро что-то записать в любой момент времени. У меня есть своя приватная группа Вконтакте. В ней созданы отдельные темы для тех игр, которыми я занимаюсь. Если я что-то придумаю, я сразу это записываю. Также помни, что твои хаки на вес золота. Рекомендую использовать бэкапы на облаке с регулярной синхронизацией, я сам пользуюсь облаком **mail.ru**. Всегда храни все версии своих хаков, и старайся сохранять отдельные хаки как можно чаще, особенно пока ты еще учишься. Чем больше промежуточных сохранений, тем проще искать и исправлять свои ошибки.

## Уровень ромхакинга

Здесь ты можешь выбрать то, чему хочешь научиться. Учебник расчитан на людей с нулевым или минимальным опытом и знанием ромхакинга. Каждый уровень спроектирован с учетом того, что ты уже прочитал и усвоил информацию с предыдущих уровней. Это также означает что я очень редко буду повторять материал предыдущих уровней в новых статьях.

Я не буду утверждать что уровень мастера из этого документа это максимально возможный уровень ромхакинга. Нет, уровень мастера - это лично мой нынешний предел знаний и опыта. На самом деле я бы не назвал себя мастером, но я достаточно компетентен, чтобы написать этот учебник, и могу делать хаки среднего уровня. В общем, освоишь мой уровень мастера - чувак, ты крут, мне больше нечему тебя научить.

Я заранее скажу, что я могу сам не знать некоторых, даже достаточно примитивных вещей. Если я чего-то не знаю, или в чем-то не уверен, я так и скажу чтобы не вводить тебя в заблуждение. Со временем я могу узнать что-то новое, или мне кто-то подскажет или поправит меня, и тогда я отредактирую и дополню статьи учебника.

Не забывай кликать на каждый раздел, даже на оглавление, везде содержится полезная инфа. И очень важно, чтобы делал в точности все шаги из реальных примеров, а не просто читал сам текст. Помни, что все тут составлено как часть твоего обучения.

Ну погнали, че.

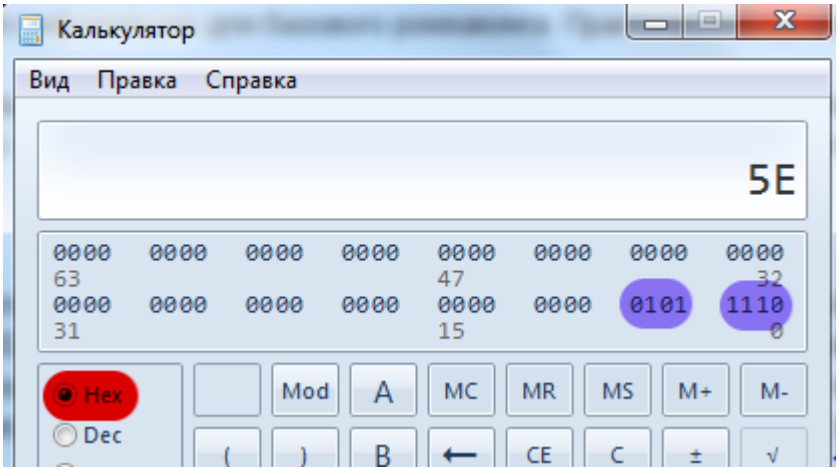
## Новичок

На этом уровне ты получишь всю необходимую предварительную информацию для базового ромхакинга.

Полагаю, ты уже знаешь что такое ROM-файл. Ром - это образ игры для эмулятора. Если точнее, это двоичный файл, содержащий копию данных из микросхемы картриджа. Чтобы поиграть в игру, тебе надо найти ром этой игры. Для эмулятора Денди (Nintendo Entertainment System, или коротко NES) нужен ром-файл с расширением ***.nes***.

Для начала уточню, что отныне работать предстоит с шестнадцатеричной системой счисления (***HEX***). Отличия от привычной нам десятичной (***DEC***) в том, что в ***HEX***, чтобы добраться от 00 до 10, нужно перебрать 16 чисел: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0**A**, 0**B**, 0**C**, 0**D**, 0**E**, 0**F**, и только потом идет **10**. 0A в ***HEX*** это 10 в ***DEC***, 0B в ***HEX*** это 11 в ***DEC***. 10 в ***HEX*** это 16 в ***DEC***. Чтобы все это не запоминать, для удобства перевода из ***DEC*** в ***HEX*** и наоборот тебе нужен калькулятор. Также помни, что байт 00 тоже используется в ***HEX***, поэтому если в игре отсчет идет например с 00 до 05, это **6** разных чисел, а не 5 (00, 01, 02, 03, 04, 05).

**Байт** - число в ***HEX***.
**0325** - адрес под номером 0325 (в некоторых документах можно встретить обозначение 0325**h**).
**5E** - байт 5E.
**01011110b** - разбор байта 5E на биты (двоичная система ***BIN***). Буква **b** означает, что этот набор из 8 цифр является разбором байта на биты. Биты всегда указываются с седьмого по нулевой, слева направо. Разбор байта на биты тебе может понадобиться в будущем.

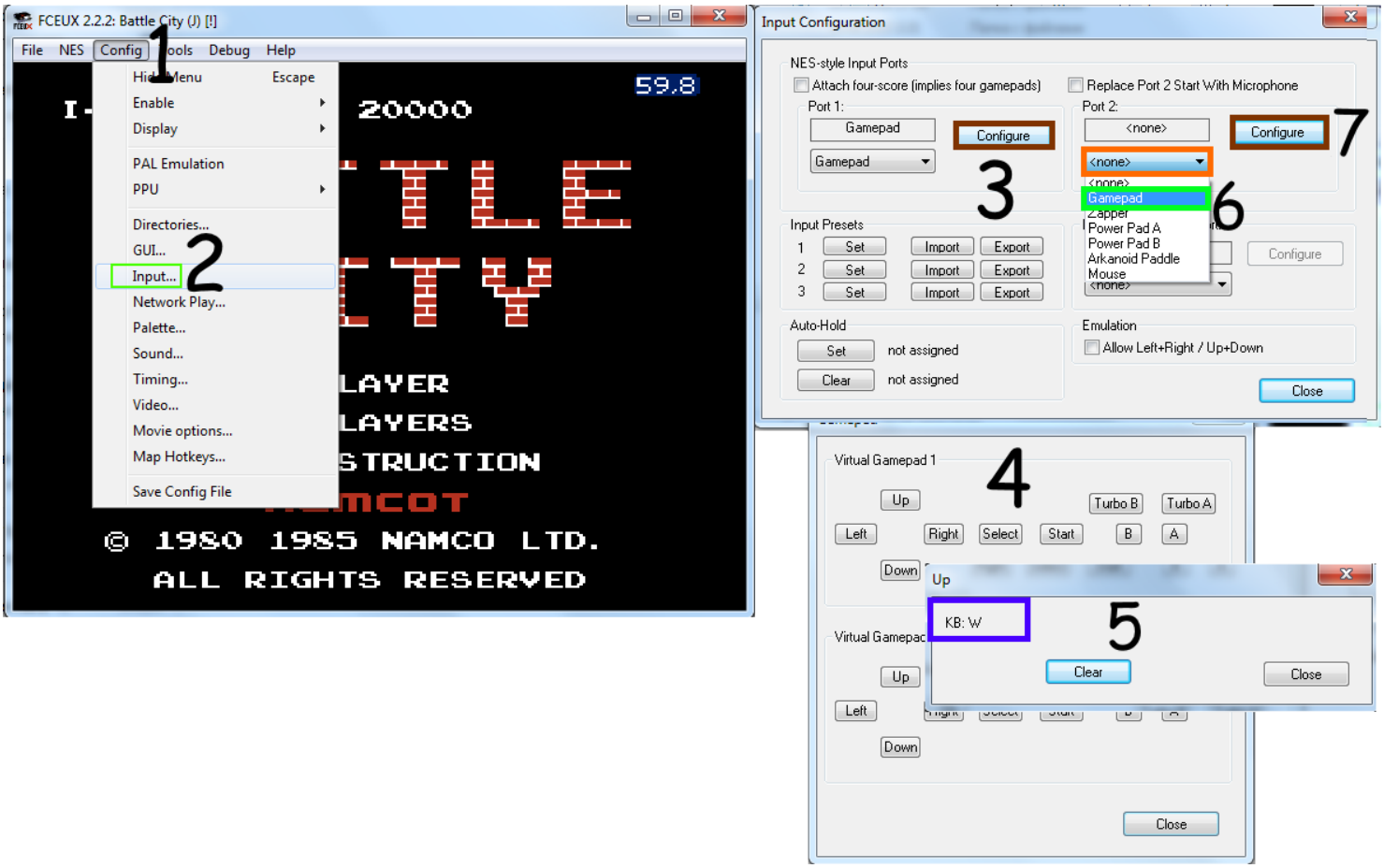



**0 1 0 1 1 1 1 0 b**
7 6 5 4 3 2 1 0 - порядковый номер, используется обозначение бит7, бит6... бит0

## Настройка эмулятора

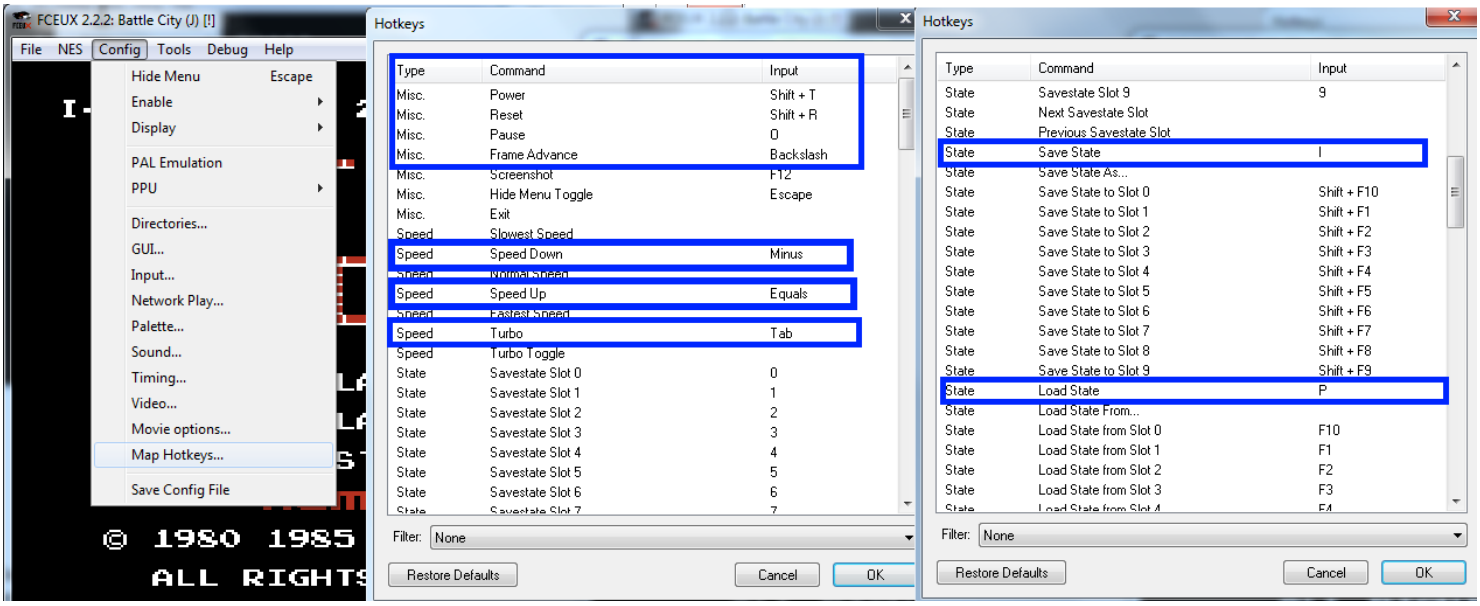
Чтобы освоить интерфейс, сначала открой ром. Для обучения я предлагаю тебе игру ***Battle City*** (танчики), который ты найдешь в папке ***ROMS***. Запусти ***fceux.exe 2.2.2***, нажми ***File*** - ***Open ROM***, выбери ***Battle City (J) [!].nes***. Смотри видео или читай инструкцию по настройке.


Первым делом настраиваешь управление под себя. Эмулятор у тебя уже запущен, и на экране танчики. В эмуляторе жмешь *Config* - *Input* (**1**, **2**). Там где *Port 1* жмешь *Configure* (**3**), откроется окно (**4**). Кликаешь мышкой на кнопку, допустим *Up*, откроется маленькое окошко (**5**), после открытия 2 раза жмешь на желаемую клавишу, окно автоматически закроется. Переходишь к следующей кнопке, и так пока не настроишь все в графе Virtual Gamepad 1. Жмешь внизу *Close*. Затем там где *Port 2* жмешь на <*none*> (**6**), выбираешь *Gamepad*. Затем жмешь на *Configure* (**7**) и аналогично настраиваешь управление для Virtual Gamepad 2, уже на другие кнопки. В конце жмешь *Close*.

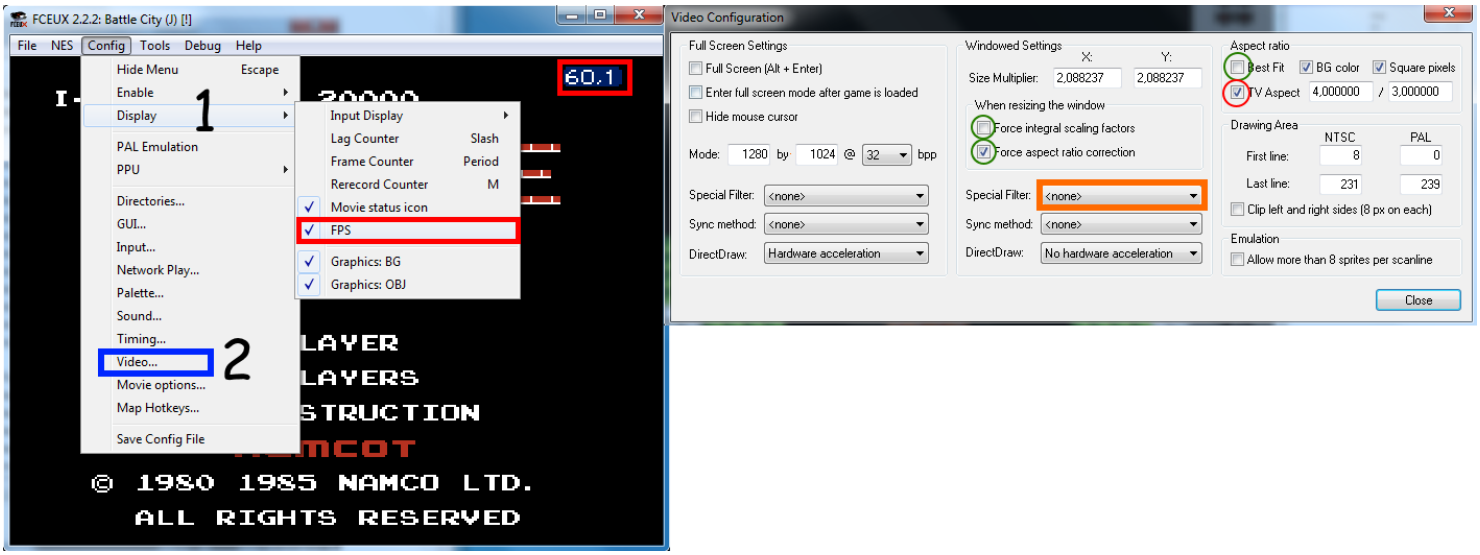



Теперь надо настроить горячие клавиши эмулятора для твоего удобства. Жмешь Config - Map Hotkeys, ищешь в списке выделенные команды с картинки. Если что-то хочешь поменять, кликаешь дважды по команде, при появлении маленького окна жмешь один раз на нужную клавишу или комбинацию клавиш, окно закроется. Если после настройки управления у тебя клавиши совпадают с горячими клавишами эмулятора, находишь команду в списке, два раза кликаешь на нее, в маленьком окне жмешь *Clear*. На скриншоте показаны мои настройки, остальными клавишами я не пользуюсь.

• **Power** - программный сброс эмулятора, аналогичен полному выключению эмулятора из розетки и последующему мгновенному включению.
• **Reset** - аппаратный сброс эмулятора, аналогичен нажатию кнопки Reset на оригинальной приставке.
• **Pause** - пауза в эмуляторе. Работа эмулятора полностью замораживается.
• **Frame Advance** - первое нажатие аналогично *Pause*, при повторном нажатии эмулятор сделает шаг в 1 кадр. Чтобы снять паузу, нужно нажать кнопку, назначенную на *Pause*.
• **Speed Down** - замедлить скорость работы эмулятора.
• **Speed UP** - увеличить скорость работы эмулятора.
• **Turbo** - очень быстрая скорость работы эмулятора, для длительного эффекта нужно удерживать кнопку.
• **Save State** - сохраниться.
• **Load State** - загрузиться.




Далее по желанию делаешь отображение FPS на экране (**1**). Затем заходишь в настройки видео (**2**). Если выставишь зеленые галочки как у меня, сможешь удобно менять размер окна эмулятора без изменения соотношения размера сторон, а красная - мне нравится соотношение сторон TV, тут уж кому как. По желанию выставь фильтр изображения на экране, если тебе не нравится оригинальная пиксельная графика.

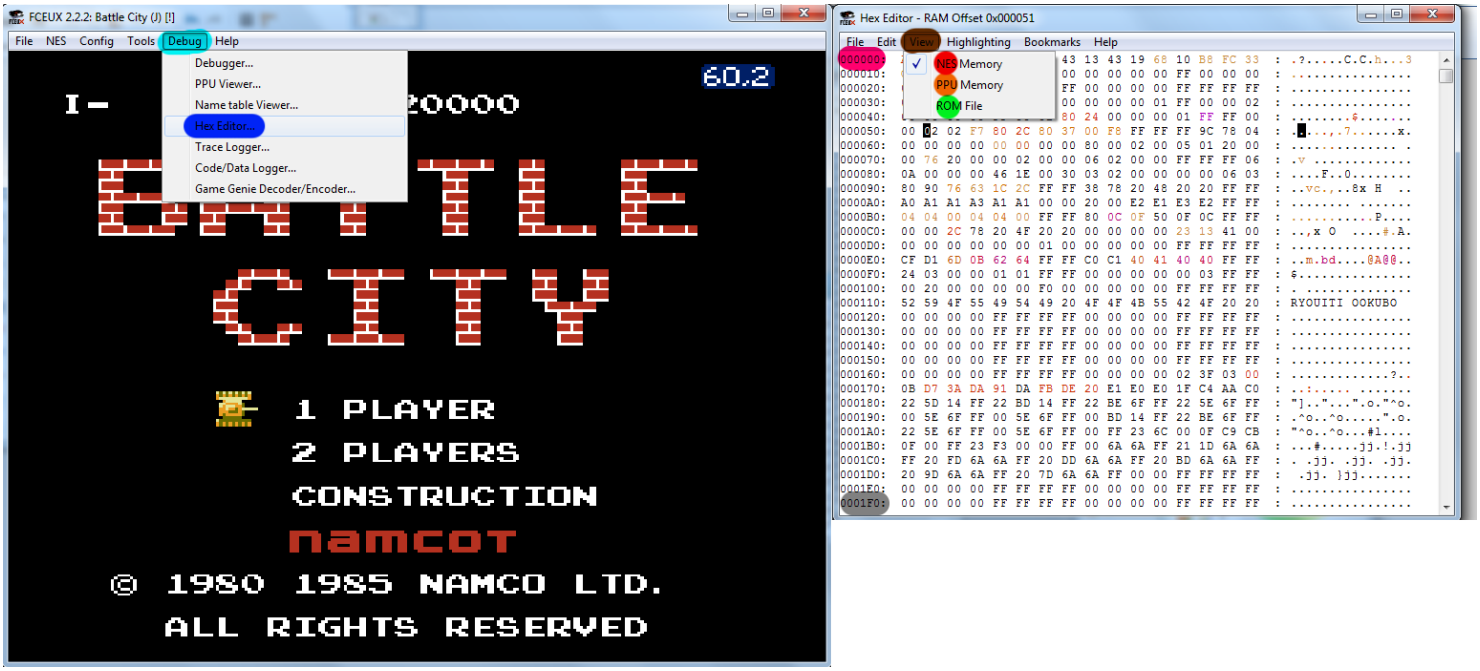



На этом все. Можешь немного поиграть, проверить управление в эмуляторе и горячие клавиши. Выбор слота для сохранения/загрузки клавишами 1, 2... 0.

## Окно Hex Editor

Сам эмулятор можно использовать просто для игры, но ты собираешься научиться ромхакингу, поэтому разберем вспомогательные инструменты. Перейдем к изучению двух важнейших дополнительных окон эмулятора - **Hex Editor** и **Debugger** (про него будет рассказано в разделе с командами процессора). Во время работы с обоими окнами тебе нужно переключить раскладку клавы на **английскую**, чтобы ты мог прописывать байты буквами. Начнем с Hex Editor, в дальнейшем буду называть его просто хекс.

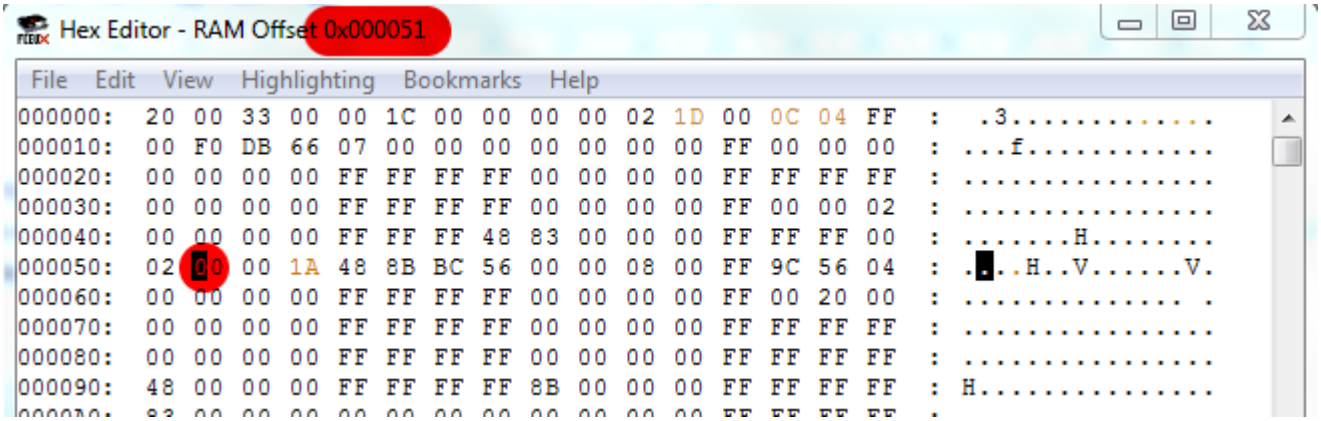
В эмуляторе жми **Debug** - **Hex Editor**. Я предпочитаю чтобы окно было растянуто от **000000** до **0001F0** (32 строчки). Кстати говоря, дальше я не буду называть первые два нуля. Я буду говорить 0000 и 01F0.




Во вкладке **View** можно переключиться на 3 разных режима, все они нужны для ромхакинга. В будущем я буду называть их **NES**, **PPU** и **ROM** соответственно. Рассмотрим каждый в отдельности.

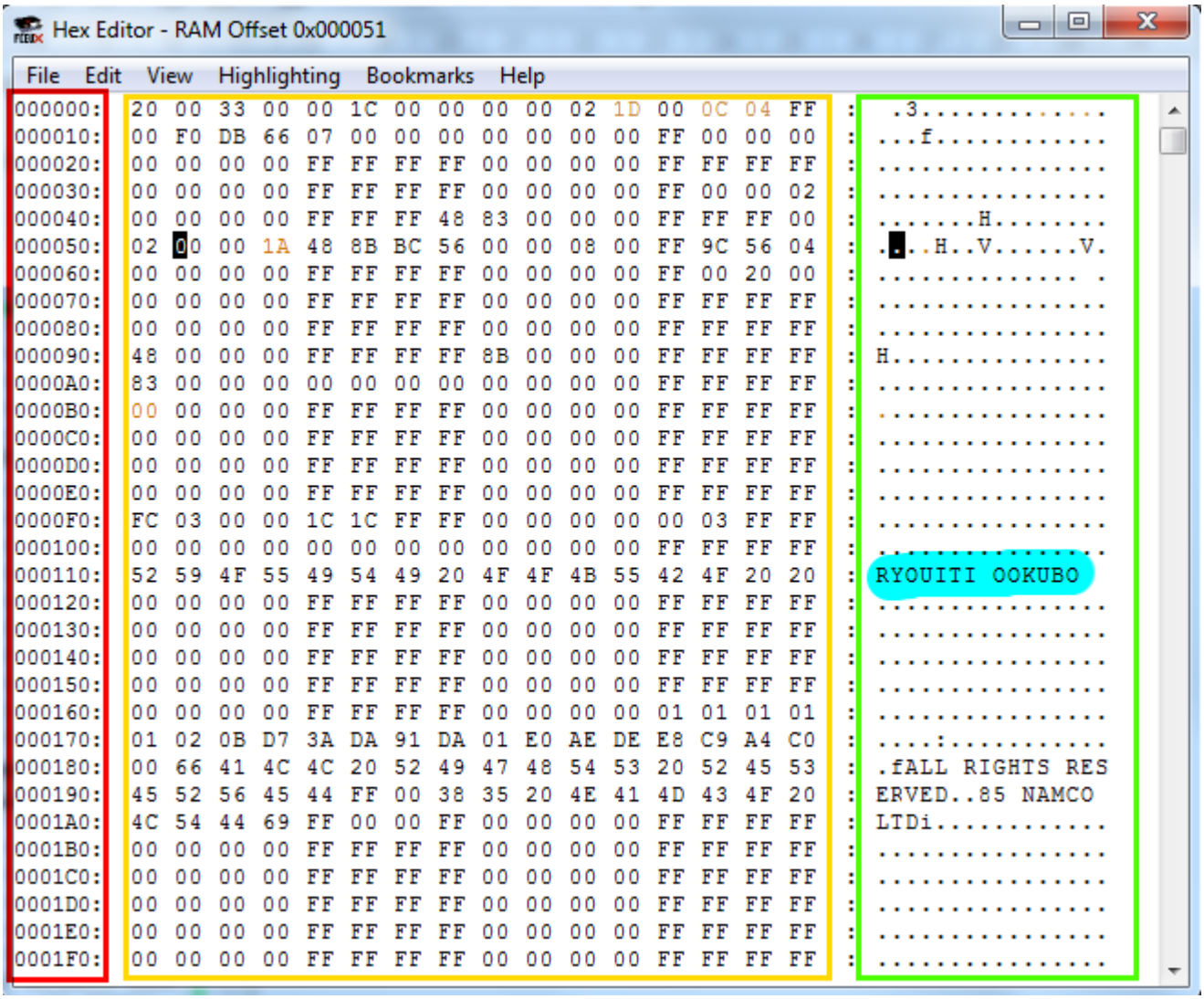
## NES Memory

Итак, что ты тут видишь? Большая таблица, по 16 байтов в строчке. Некоторые байты постоянно меняются. Такие байты по умолчанию подсвечиваются разными цветами, чтобы проще было что-то найти. Местоположение каждого байта называется ***адрес***. В данный момент мой курсор находится на строчке 0050, 2й байт по счету = адрес **0051**. По адресам можно перемещаться стрелочками на клаве.

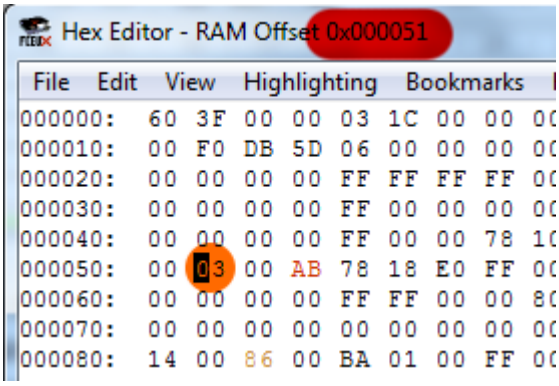


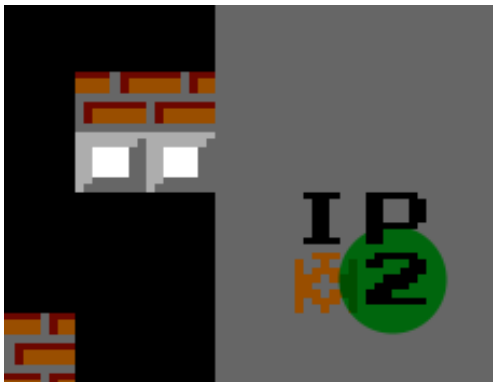

Все, что ты тут видишь - оперативная память (**RAM**, или по-русски **ОЗУ**). RAM - энергозависимая часть системы компьютерной памяти, в которой во время работы компьютера хранится выполняемый машинный код, а также входные, выходные и промежуточные данные, обрабатываемые процессором (википедия). В данном случае компьютер это эмулятор. Если проще, RAM - это книжная полка со множеством отделов (адресов), и в каждой полке хранится одна книга (байт). В оперативке хранятся байты, которые могут меняться редко, часто, или не меняться вообще, все зависит от кода игры. Адреса могут отвечать за координату объекта по горизонтали или по вертикали, это может быть счетчик жизней, это может быть таймер, а байт это соответственно некое количество.

Окно хекса разделено на 3 части - **начальные адреса**, **адреса** и **перевод байтов в символы**, как например фраза **RYOUITI OOKUBO**. **Здесь** можно писать только цифры и буквы A, B, C, D, E, F, а **здесь** цифры, буквы и некоторые символы.



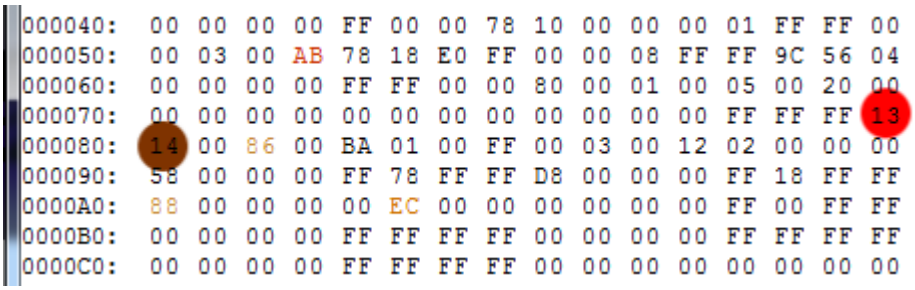

Продемонстрирую тебе что можно делать с этими адресами. Начни игру 1 Player, выбери 1й уровень. Сохранись когда на экране появится первый вражеский танк. Количество твоих жизней находится по адресу **0051**. Сейчас там записан байт **03**, а на экране справа указано **2** жизни. Иногда **байт** в оперативке может превышать на 1 **количество** жизней на экране, как в случае с танчиками, это зависит от кода игры.




В танчиках игра регулярно проверяет этот адрес на количество жизней, и отображает это количество на экране. Попробуй заменить байт в адресе на 04, и на экране сразу отобразится 3 жизни. Если тебя убьют, **байт** автоматически уменьшится на 1. Также игра регулярно проверяет адрес на то, что твои жизни уже закончились. Поэтому если ты пропишешь 00, сразу же вылезет надпись Game Over. Примечательно то, что ты можешь продолжать двигаться своим танком во время этой надписи. Это можно было бы назвать багом, но разработчики в 1985 году не могли знать что ты сегодня будешь менять количество жизней напрямую через оперативку, так что тут все нормально. Мы их прощаем.

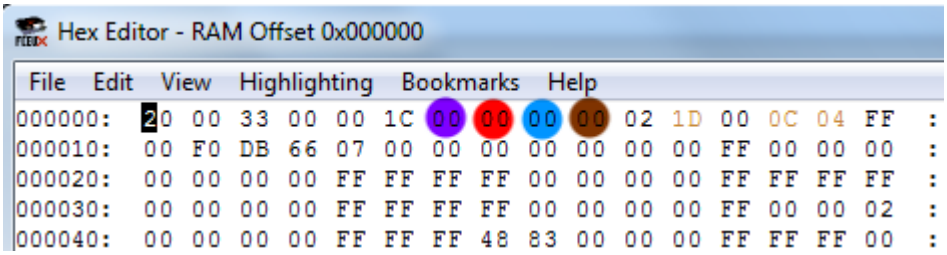
Загрузи сохранение. Найди 2 адреса - **007F** и **0080**.




Оба адреса отвечают за количество вражеских танков, но у каждого своя задача. **007F** это счетчик танков, которые еще не появились, то есть которые остались в запасе. Они отображаются справа на экране. **0080** это общий счетчик танков на уровне. При загрузке **туда** пишется байт 14, это 20 в десятичной системе, то есть на каждом уровне 20 вражеских танков. Оба адреса по-своему важны, но адрес **0080** в приоритете для прохождения игры, поскольку именно его игра проверяет чтобы закончить раунд когда там будет написано 00. Адрес **007F** второстепенный, ты можешь например обнулить его (прописать 00), танки больше появляться не будут, но раунд не закончится. Кстати, на экране, в отличии от счетчика жизней, счетчик танков будет некорректно отображать количество при изменении **байта** через оперативку. Это зависит от кода игры, опять же разработчики не видели в этом необходимости, оригинальная игра работает как часы. Можешь побаловаться с этими двумя адресами.

Еще один пример. Выбей бонус из танка, но не подбирай его, сохранись. Найди адрес 0088, этот адрес отвечает за тип бонуса. В игре всего 7 бонусов, это байты от 00 до 06. Попробуй поменять байт, увидишь что бонус также меняется. Остальные байты не предусмотрены, начнет меняться графика бонуса на всякий бред, а при взятии такого бонуса игра может повиснуть.

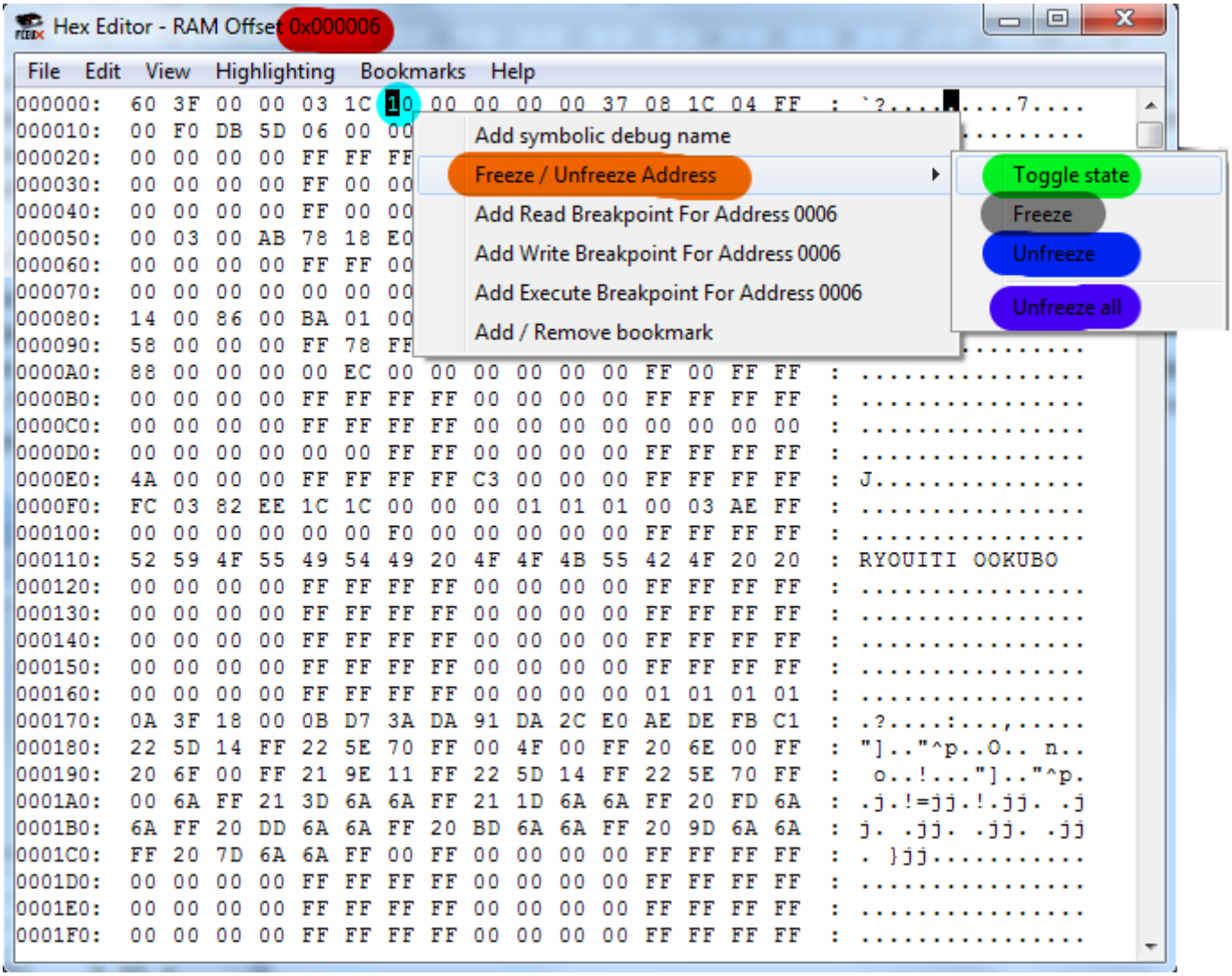
И последние несколько адресов в этом уроке. Обрати внимание на адреса 0006-0009. Это адреса, которые отображают нажатые кнопки.




Адрес **0006** и **0008** отвечают за кнопки первого джойстика. В адресе **0006** показано удержание кнопок, если их нажать и не отпускать, он нужен для передвижения танка. В **0008** показано одноразовое нажатие кнопки, даже при удержании, этот адрес служит для стрельбы. Адреса **0007** и **0009** имеют такое же значение, но для второго джойстика. Игра считывает эти адреса и заставляет твой танк двигаться и стрелять соответственно. Ручная запись байтов по этим адресам ни к чему не приведет, код чтения джойстиков регулярно что-то в них записывает. Работа с адресами джойстиков тебе понадобится для изменения управления в игре, или добавления нового действия на разные кнопки. Например тебе вдруг захотелось сделать так, чтобы при нажатии кнопки Select тебе добавлялась +1 жизнь. Ок, не проблема, научишься. И как среди всей этой кучи искать все эти адреса, на которых ты сегодня практиковался, ты тоже скоро научишься.

Как видишь, байты в оперативке можно менять вручную. Это не всегда работает, иногда байт сразу же меняется обратно, как в случае с джойстиками. Однако есть способ насильно скормить игре нужный байт. Ставишь эмулятор на паузу, прописываешь в адресе нужный тебе байт, затем на нем правой кнопкой мыши - **Freeze** - **Toggle**. Все, байт зафиксирован, теперь он отмечен синим цветом, можешь снять паузу. Приведу пример на джойстике. Делаешь паузу, по адресу **0006** пропиши **10** (это байт нажатой кнопки ***Вверх***), и **заморозь** его. Кстати, я не в курсе чем отличается **Toggle** от **Freeze**. Помню только то, что в

первые дни моего знакомства с хексом **Freeze** как-то хуже работал чем **Toggle**, по крайней мере мне так запомнилось, короче отныне я пользуюсь только **Toggle**. В общем, я буду говорить ***заморозить*** байт в адресе, подразумевая **Toggle**. Вернемся к примеру. Если **заморозить** байт **10** и снять паузу, твой танк будет самостоятельно двигаться вперед, а ты в это время сможешь только стрелять.




Аналогично можно делать и с другими адресами, только не обязательно ставить эмулятор на паузу если и без нее все прекрасно меняется. Если **заморозить** адрес с жизнями - жизни станут безлимитными, адрес с количеством врагов - враги никогда не закончатся, адрес с бонусом - бонус всегда будет появляться один и тот же. Чтобы отменить этот эффект, нужно нажать на нужном адресе и выбрать **Unfreeze** (чтобы разморозить конкретный адрес) либо **Unfreeze all** (это разморозит все замороженные адреса, и в этом случае можно предварительно кликать на любом адресе).

Теперь рассмотрим распределение памяти в оперативке.
**0000-07FF** - основная оперативка. Вот некоторое типичное распределение памяти в этом диапазоне с которым я чаще всего сталкиваюсь. Но в каждой игре может быть по-разному.
0000-00FF - адреса с нажатиями джойстика, таймеры, счетчик кадров, адреса для рандома в игре, и прочие адреса, используемые для загрузки экранов игры.
0100-01FF - стек (про него поговорим позже). Стек по этим адресам будет **всегда**.
0200-02FF - адреса с атрибутами спрайтов (про спрайты будет рассказано в разделах с графикой). Для спрайтов отведено 16 строчек, их ни с чем не спутаешь. В танчиках атрибуты спрайтов тоже по этим адресам, это хорошо видно во время игры когда на экране несколько танков.
0300-06FF - адреса с координатами объектов и их характеристики.
0700-07FF - адреса для музыки.

**0800-1FFF** - основная оперативка дублируется 3 раза подряд. Тебе предстоит работать только с оригинальной (основной).

**2000-5FFF** - адресные регистры. На самом деле в процессоре не так много регистров. Про них я расскажу на более высоких уровнях.

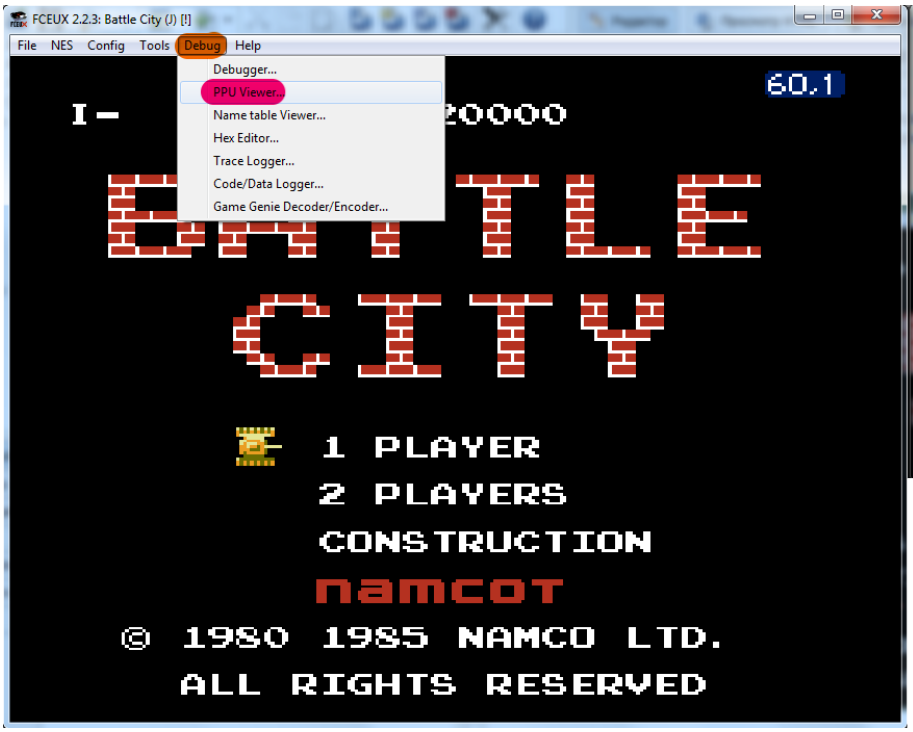
**6000-7FFF** - память батарейки, если она есть. Батарейка нужна чаще всего в тех играх, в которых можно сохраняться. Но ее также можно использовать для удобного ромхакинга, про это будет рассказано на более высоких уровнях.

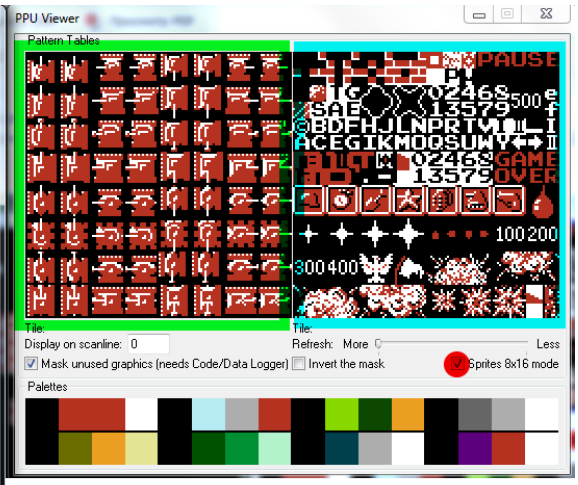
**8000-FFFF** - здесь расположены байты, которые в данный момент загружены в память процессора. По этим адресам находится сам код игры, а также различные байты данных, которые могут использоваться в основной оперативке, например скорость объектов, высота прыжка, звуки мелодий, текст, и многое другое. Область 8000-FFFF размером всего **32кб**. Поскольку процессор Денди умеет обращаться только к адресам в диапазоне 0000-FFFF, а большинству крупных игр для работы недостаточно 32кб памяти, игры используют так называемое переключение банков, а метод и возможность переключения зависят от маппера. Обо всем этом будет рассказано на более высоких уровнях.

## PPU Memory

По адресам 0000-1FFF никогда не меняй байты вручную без предварительного сохранения, потому что изменения нельзя откатить комбинацией клавиш ***Ctrl+Z***.

Здесь находится видеопамять - графика заднего фона и движущихся объектов (спрайтов), а также цветовая палитра.
Про все это будет рассказано в разделах с графикой. Если хочешь уже сейчас мельком взглянуть как выглядит графика, открой fceux 2.2.**3**, выбери **Debug** - **PPU Viewer**, и поставь **галочку**.




Правой кнопкой мыши на **левом** и **правом** окне ты изменишь цвет.

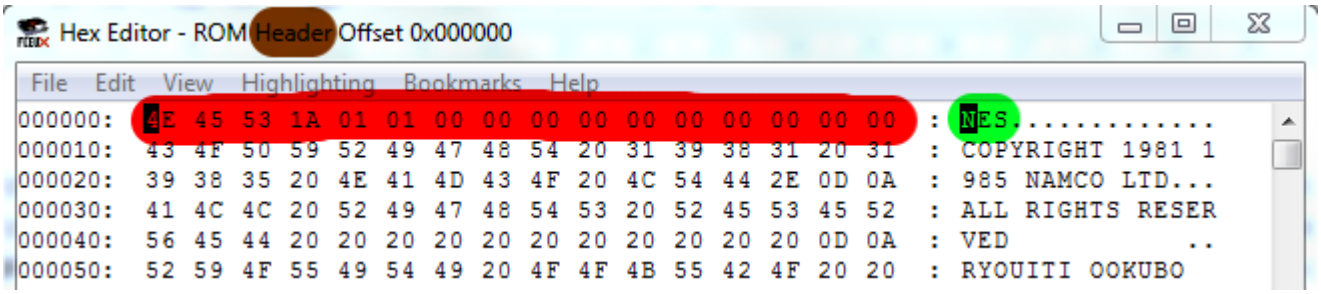
## ROM File

Здесь ты увидишь то, из чего состоит ром. В окне эмулятора нажми ***Help*** - ***Message Log***. Ты увидишь вот такую инфу:

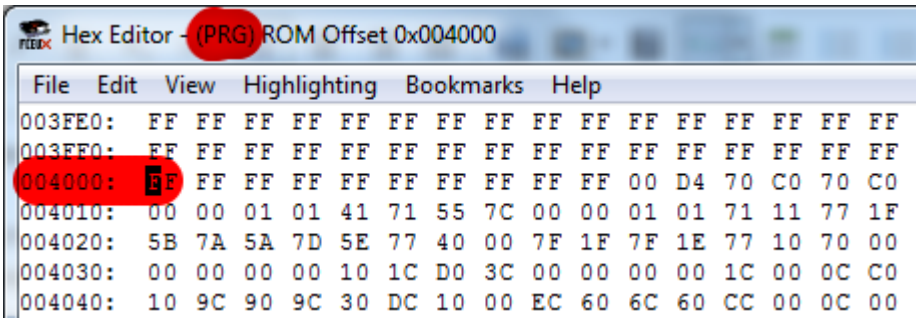
```
PRG ROM:    1 x 16KiB
CHR ROM:    1 x  8KiB
ROM CRC32:  0x7e053e64
ROM MD5:  0xa41a8a46771160e97ee5967365c07c83
Mapper #:  0
Mapper name: NROM
Mirroring: Horizontal
Battery-backed: No
Trained: No
```

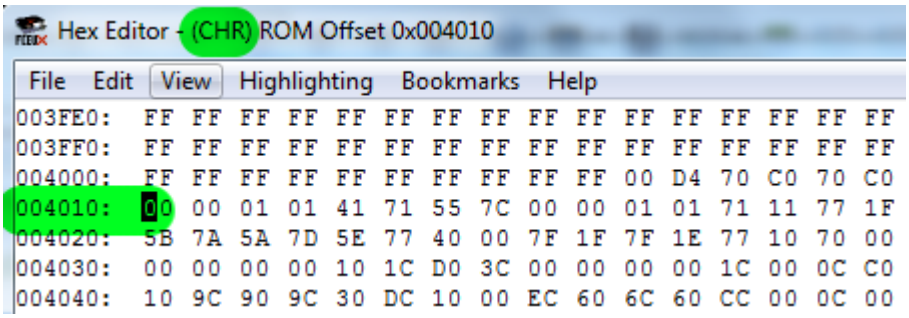
Разберем что значит каждый пункт.
**PRG ROM** - размер памяти PRG, то есть код игры и различные байты данных.
**CHR ROM** - размер памяти CHR, то есть байты для графики - задний фон и спрайты (объекты на экране).
В обоих случаях число перед **x** означает количество банков, в данном случае всего 1 банк PRG и CHR. Это очень немного, но для танчиков вполне достаточно.
**ROM CRC32** и **ROM MD5** - контрольные суммы, которые эмулятор почему-то вычисляет неправильно.
**Mapper #** и **Mapper name** - номер и название маппера.
**Mirroring** - метод отображения фона в видеопамяти.
**Battery-backed** - наличие батарейки.
**Trained** - точно не знаю что это, тебе это не пригодится.

Эти данные, за исключением контрольной суммы, эмулятор считывает из рома по адресам **0000-000F**. Эти **16** байтов называются **iNES header**, или заголовок, или хэдер. Они нужны чтобы эмулятор понимал как именно ему эмулировать данный ром. Первые 3 байта обязательно составляют слово **NES**. В картридже нету этой строчки, она там просто не нужна. Хэдер удобно редактировать через эмулятор ***Nestopia*** при помощи ***File - Edit iNes Header***, или может есть какие-то другие специальные программы, не в курсе. Хекс редактор эмулятора fceux не позволяет его редактировать, однако любой другой сможет. Заранее скажу, что если просто изменить хэдер, ничего хорошего из этого не выйдет, игра скорее всего перестанет запускаться, нужны дополнительные манипуляции с ромом.




В ROM всегда сначала идут байты RPG, затем CHR. Из лога мы видим что размер рома состоит из 16кб PRG (адреса 0010-4000), а CHR 8кб (адреса 4010-6000). CHR очень легко отличается от PRG по внешнему виду, это придет с опытом. Можно просто кликнуть на интересующее место в ROM, и окно эмулятора скажет тебе где ты сейчас, в **PRG** или **CHR**.

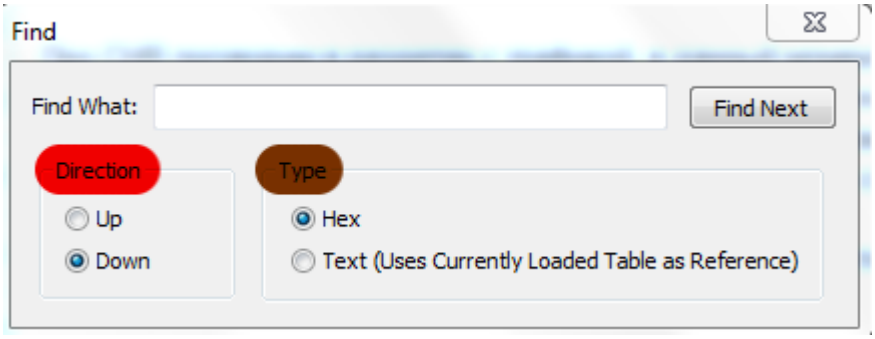



Кстати, не во всех играх присутствует **CHR**, иногда игры загружают байты из **PRG** напрямую в видеопамять, в этом случае такие байты тоже легко можно найти на глаз.

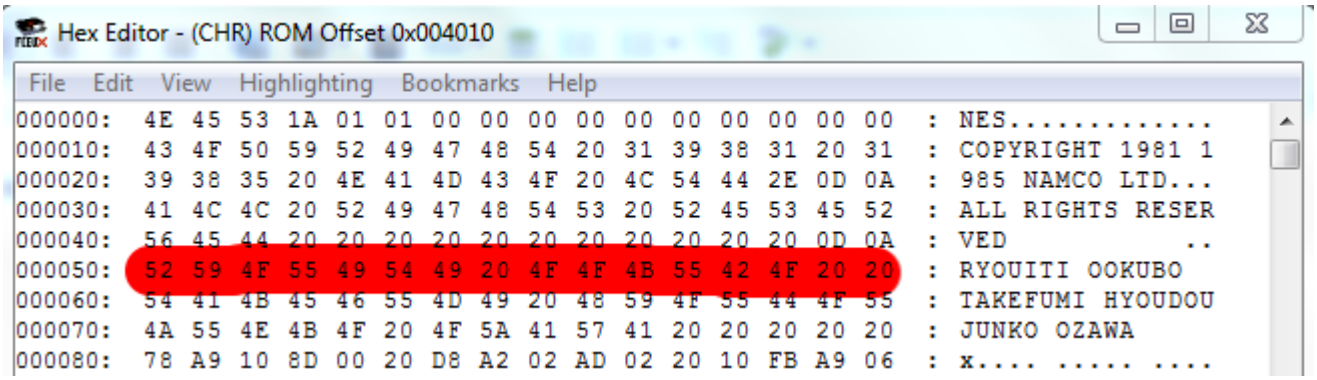
Про CHR поговорим в разделах с графикой, в данный момент нас интересует PRG. Как я уже говорил, здесь находится код и данные игры, а ромхакинг заключается в том, чтобы заменять некоторые байты на свои, к примеру увеличить количество жизней, или прописать свой собственный код чтобы игра вела себя по-другому. Измененые байты будут подсвечиваться красным цветом, отменять изменения можно комбинацией клавиш ***CTRL+Z***. Для того, чтобы знать что и как нужно менять в роме, тебе нужно понимать команды процессора, про которые ты узнаешь в следующем разделе. За исключением хэдера, любые другие байты можно редактировать.

Начнем с простого, освоим поиск (работает в любом из трех разделов ***View***). Нажми комбинацию ***Ctrl+F***, появится окошко.



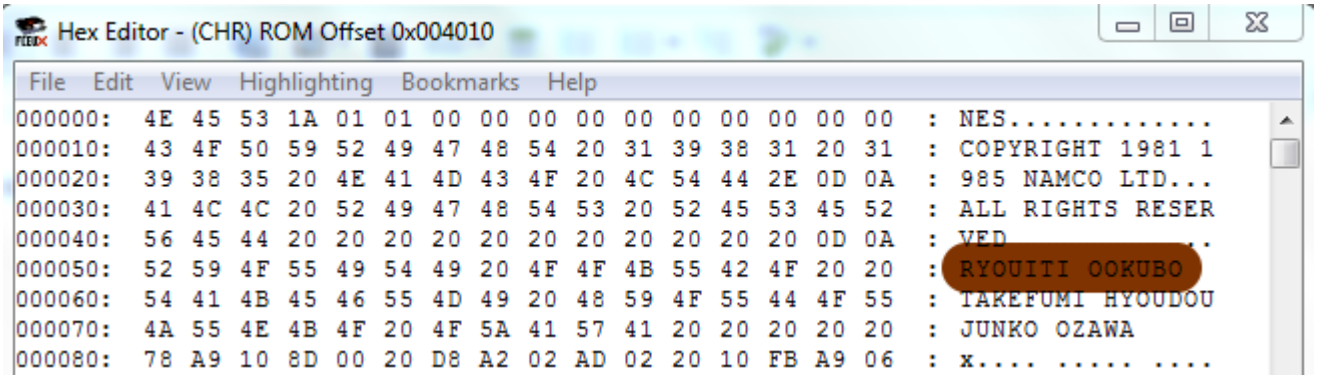

**Direction** - в какую сторону искать, ***Up*** - вверх, ***Down*** - вниз. Теперь **Type**, это режим поиска. ***Hex*** - поиск по байтам, ***Text*** - поиск по тексту.

Для тренировки возьмем надпись RYOUITI OOKUBO. Окно с поиском можно не закрывать, просто отодвинь в сторону. Сначала выдели **байты** этих слов по адресам 0050-005F и нажми ***Ctrl+C*.**



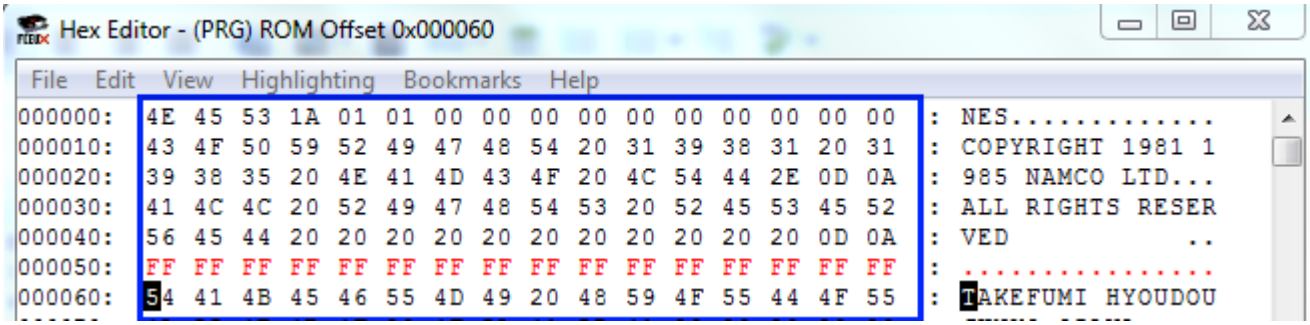

Переключись на NES, вставь байты в поиск ***Ctrl+V***, выбери режим ***Hex*** и нажми кнопку ***Find Next***, тебя перекинет на местоположение результата поиска. Ты найдешь 6 результатов поиска, дальше поиск пойдет по кругу.

Теперь скопируй саму надпись **RYOUITI OOKUBO**, но уже справа, где буквы, и сделай те же действия поиска повторно, выбрав режим ***Text***.



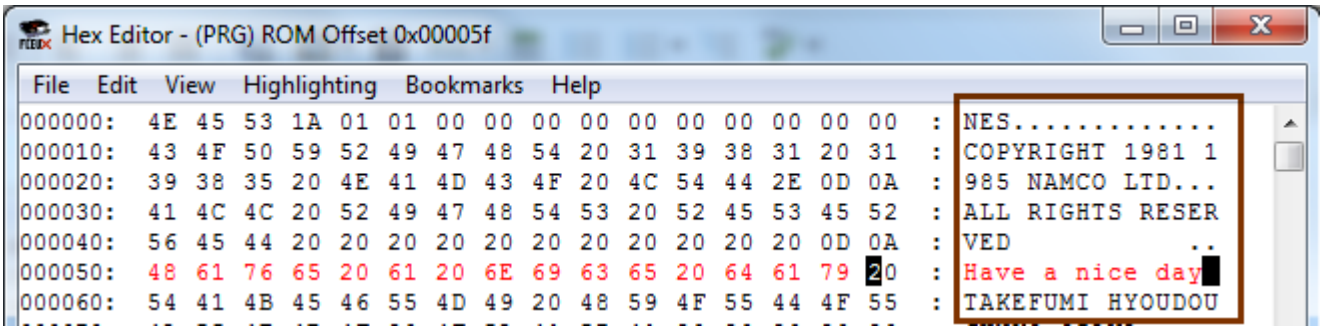

После поиска переключи назад на режим ***Hex***. Поиск в неправильном режиме будет выдавать ошибку.

Затри эту фразу, прописав байты **FF** в **левой** части. Затем повторно пропиши **FF** поверх уже написанных **FF**.




Удерживай ***Ctrl+Z*** для отката изменений.

Теперь попробуй прописать любую **фразу** на английском **справа**, уложившись в эту строчку.




Откати изменения.

## Список команд процессора

В процессоре множество команд, но я буду говорить лишь про те, которыми пользуюсь сам. Также я не буду использовать все разновидности команд.

Изучение команд является самым сложным этапом для новичка, но если ты осилишь этот шаг, то дальше все пойдет как по маслу.

Распечатай этот список, пригодится. Скриншот уже лежит у тебя в папке с названием **6502.png**.

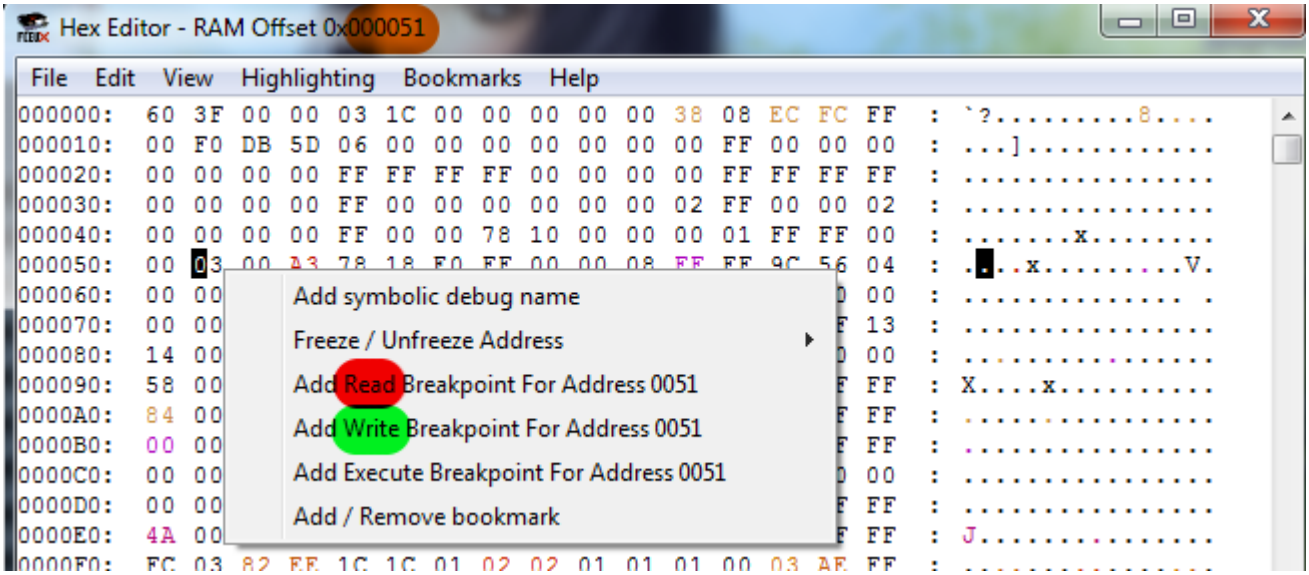
| | LDA | | STA | | CMP | | INC | |
|---|---|---|---|---|---|---|---|---|
| **A9** | #$байт | **8D** | $АДРЕС | **C9** | #$байт | **EE** | $АДРЕС | **69** |
| **AD** | $АДРЕС | **9D** | $АДРЕС,X | **CD** | $АДРЕС | **FE** | $АДРЕС,X | **6D** |
| **BD** | $АДРЕС,X | **99** | $АДРЕС,Y | **DD** | $АДРЕС,X | | | **7D** |
| **B9** | $АДРЕС,Y | | | **D9** | $АДРЕС,Y | **E8** | **INX** | **79** |
| | | | **STX** | | | **C8** | **INY** | |
| | **LDX** | **96** | $адрес,Y | | **CPX** | | | |
| **A2** | #$байт | **8E** | $АДРЕС | **E0** | #$байт | | **DEC** | **0A** |
| **AE** | $АДРЕС | | | **EC** | $АДРЕС | **CE** | $АДРЕС | **0E** |
| **BE** | $АДРЕС,Y | | **STY** | | | **DE** | $АДРЕС,X | **1E** |
| | | **94** | $адрес,X | | **CPY** | | | |
| | **LDY** | **8C** | $АДРЕС | **C0** | #$байт | **CA** | **DEX** | **8A** |
| **A0** | #$байт | | | **CC** | $АДРЕС | **88** | **DEY** | **AA** |
| **AC** | $АДРЕС | **29** | **AND** | | | | | **98** |
| **BC** | $АДРЕС,X | **49** | **EOR** | **2C** | **BIT** | **EA** | **NOP** | **A8** |
| | | | | | | | | |
| **F0** | **BEQ** | **10** | **BPL** | **90** | **BCC** | **18** | **CLC** | **4C** |
| **D0** | **BNE** | **30** | **BMI** | **B0** | **BCS** | **38** | **SEC** | **6C** |

В будущих статьях будут использоваться примеры с использованием уже изученных команд, поэтому читай внимательно и следуй инструкциям.

Полный список команд ты найдешь в статье Команды процессора 6502 (Bnu). Текстовый вариант в файле **6502.txt**.

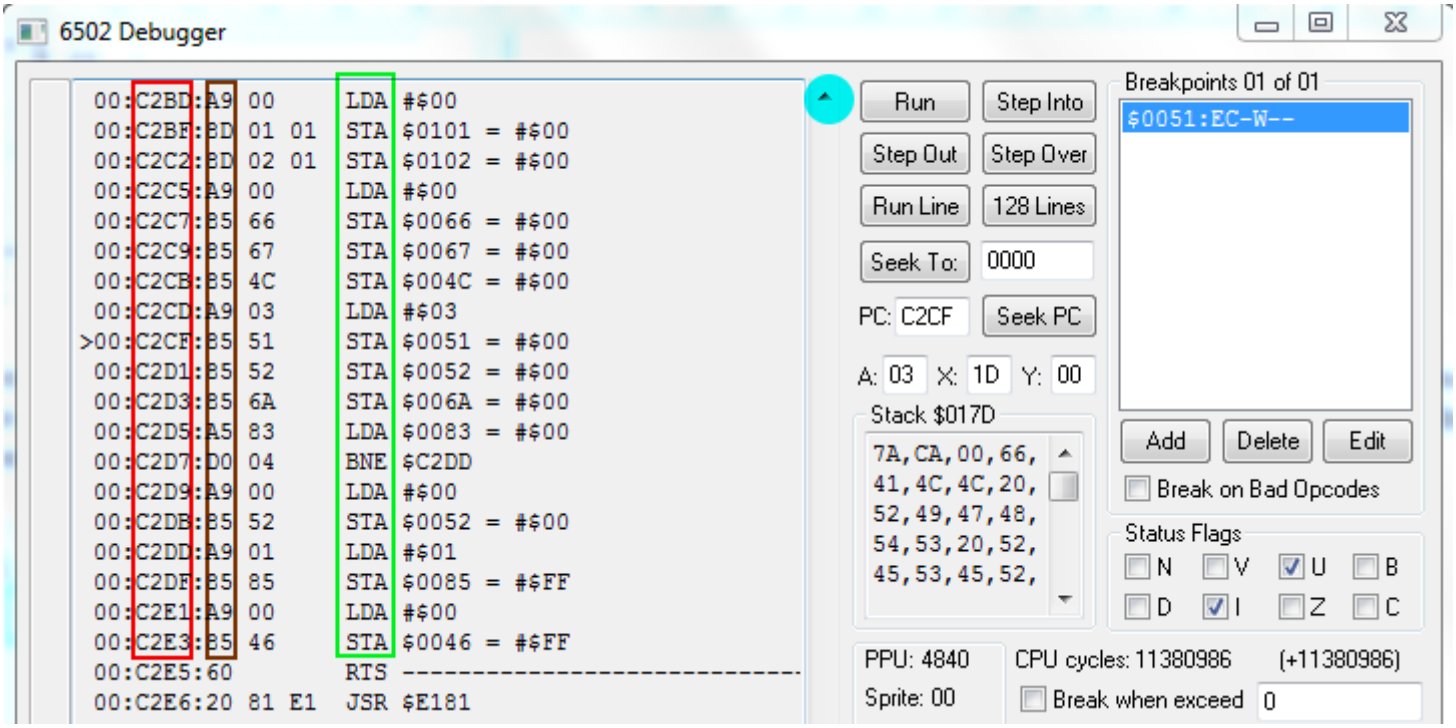
## Команды загрузки и записи, изучение дебаггера

В танчиках сделай рестарт, теперь ты находишься в меню выбора режима игры. Поставь эмулятор на паузу, найди в NES адрес **0051** (счетчик жизней). Нажми на него правой кнопкой - **Add Write Breakpoint**. Для основной оперативки бывают 2 вида бряка - на **чтение** и на **запись**. Они вылавливают команды загрузки и записи соответственно по нужным тебе адресам. Сейчас нам интересен момент **записи** количества жизней в этот адрес, поэтому выбирай **Write**.



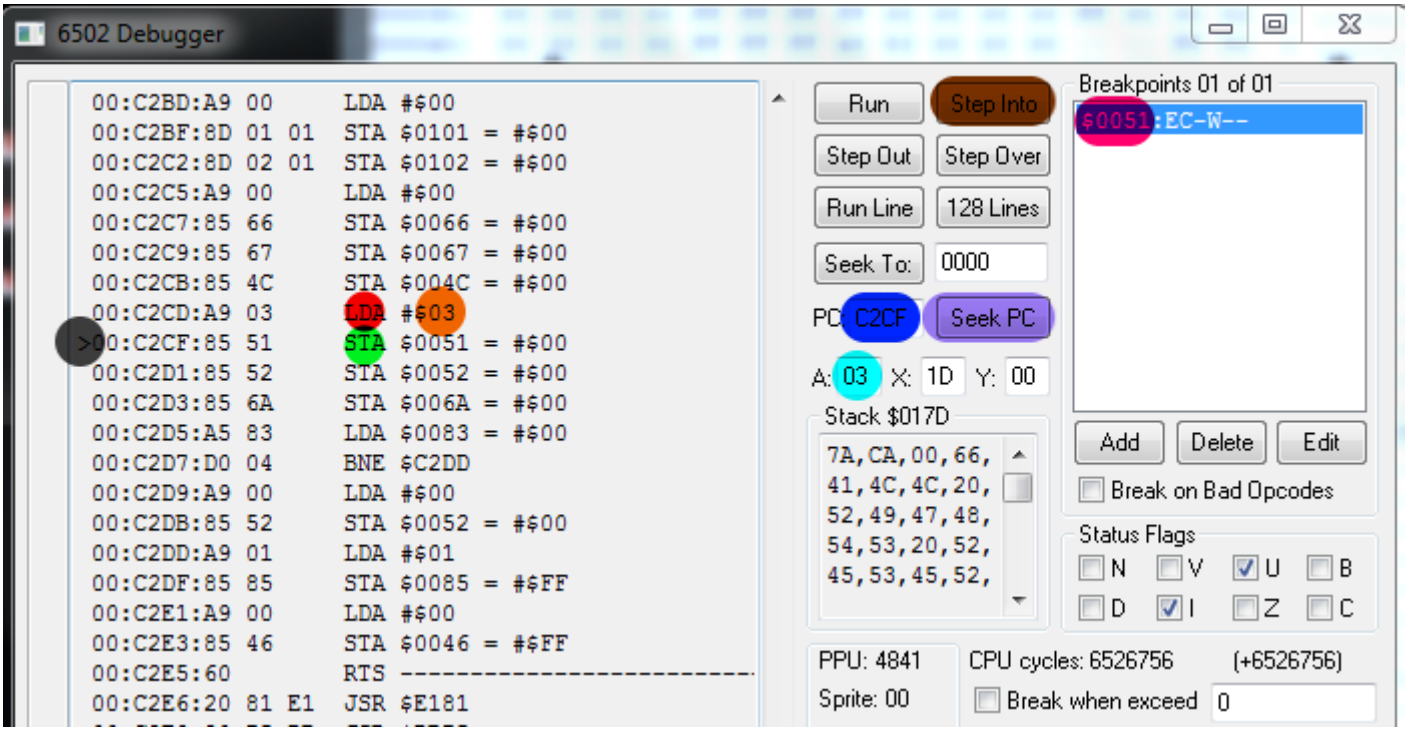

У тебя откроется окно дебаггера. Пока что сверни это окно, теперь переключись на эмулятор, сними паузу и нажми кнопку ***Start***. Если ты все сделал правильно, то на экране все еще будет меню с выбором режима, вылезет окно дебаггера, а эмулятор поставится на паузу.

Взглянем на окно дебаггера. Здесь очень много всего, так что будем потиху разбирать самое важное для тебя. Промотай **стрелочкой** текст чуть повыше, чтобы выглядело примерно как у меня на скрине. Окно можешь растянуть как тебе удобнее.



Разберем левую часть окна. **Адреса команд** (всегда указаны по тем же адресам, что и в NES), **байты команд** и перевод этих **байтов** в **текстовый вариант**. У каждой команды процессора есть свой **номер** (**байт**). Дебаггер распознает эти **байты** и отображает их **здесь**.

**Стрелочка** сейчас находится там, где собирается выполниться код. Адрес рядом со **стрелочкой** аналогичен **адресу** рядом с кнопкой **Seek PC**. Промотай экран на несколько страниц вверх или вниз, нажми эту **кнопку**, и дебаггер вернет тебя на позицию **стрелочки**.

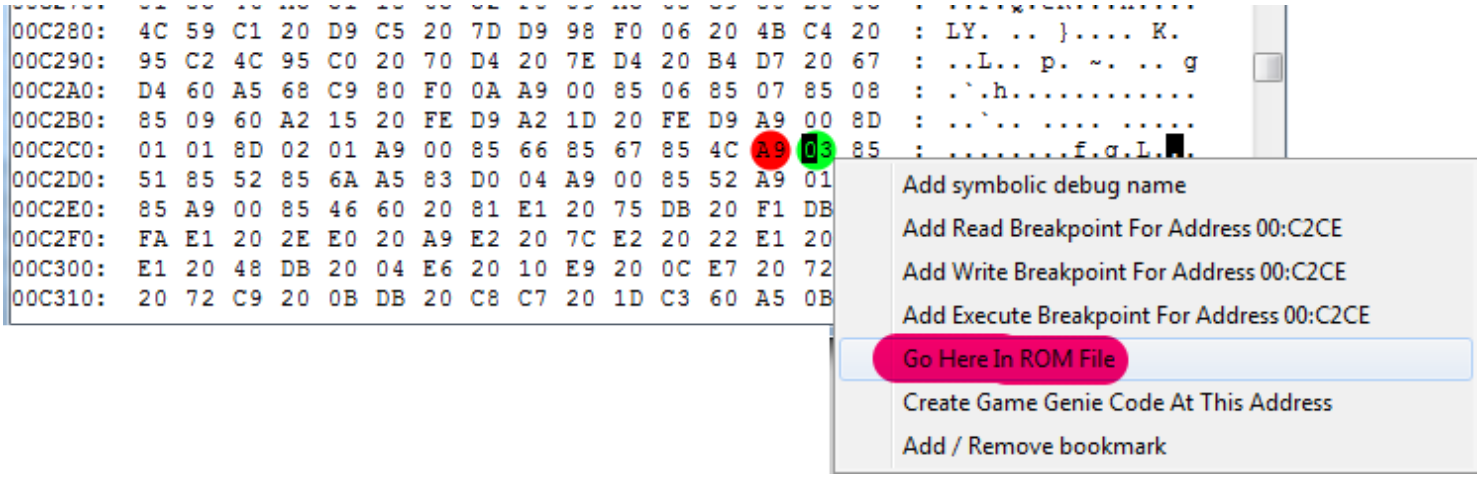



При помощи **бряка** мы выловили момент **попытки** записать количество жизней в адрес 0051, но этого **еще не произошло**. Команда **LDA** расшифровывается как ***загрузить байт в A*** (Load to A). **A** - это один из трех важнейших регистров, он используется для хранения байтов и для различных операций с этим байтом. В дебаггере ты увидишь регистр A его рядом с байтом **03**, как и другие 2 регистра - **X** и **Y**. Справа от каждого регистра находится байт, который уже был ранее в них загружен. Эти байты можно менять вручную, что иногда полезно для тестирования кода. Байт **03** оказался в регистре A при помощи команды **LDA** #$**03**. Команда **STA** - ***записать байт из A в определенный адрес*** (Set A to). STA $0051 = записать байт из A в адрес 0051 (адрес с жизнями первого игрока). Нажав кнопку **Step Into**, код сделает 1 шаг вперед и выполнит команду. Далее идет следующая команда - STA $0052 - записать этот же байт в 0052 (адрес с жизнями второго игрока). Снова нажми **Step Into**, код сделает еще 1 шаг вперед и выполнит эту команду.

Сейчас твоя задача - увеличить начальное количество жизней. Ты уже нашел нужную команду (LDA # $03), теперь надо найти и поменять байт 03 на какой-то другой. Есть несколько способов найти этот байт, покажу самый простой.

```
 00:C2CD:A9 03     LDA #$03
 00:C2CF:85 51     STA $0051 = #$03
 00:C2D1:85 52     STA $0052 = #$03
>00:C2D3:85 6A     STA $006A = #$00
```

Слева от LDA адрес **C2CD**. Кликни в дебаггере 1 раз на этом адресе, он выделится, скопируй его. Переходишь в хекс, жмешь ***Ctrl+G***, откроется окошко, вставь этот адрес и нажми ***OK***.



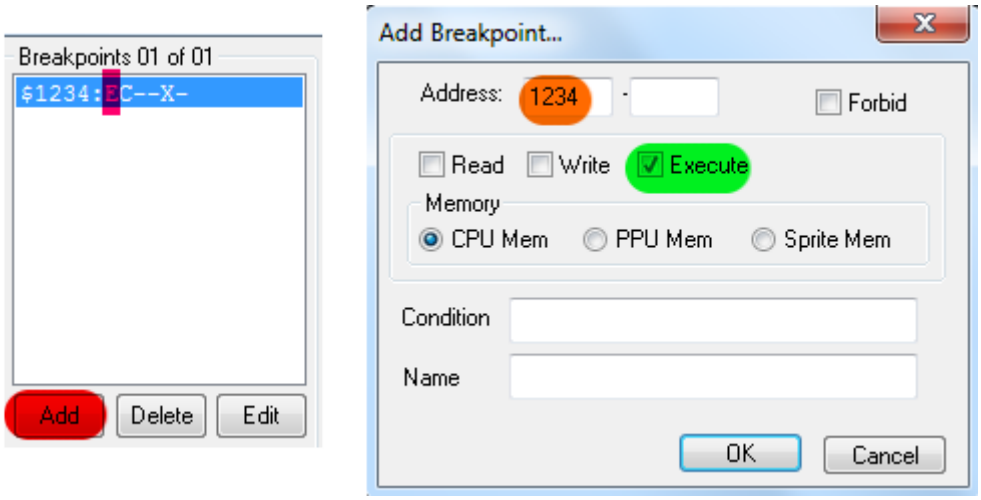

**A9** - байт команды LDA, **03** - байт с жизнями. Кликаешь на **байте** правой кнопкой - **Go Here In ROM File**. Тебя перекинет на **реальный** адрес этого байта в ROM (этот адрес всегда будет отличаться от адреса в NES). Меняешь байт **03** например на 05, затем в хексе ***File*** - ***Save Rom As***, сохраняешь ром под другим именем. Если использовать Save Rom вместо Save Rom As, измененые байты сразу же запишутся в ром, с которым ты работаешь, а также эти байты снова станут черными, и ты уже не сможешь отменить изменения. Открой еще одно окно эмулятора, открой этот ром и проверь, изменилось ли количество жизней.

Поздравляю с твоим первым хаком!

## Небольшая важная остановка

Для твоего обучения я сделал специальный хак, его ты найдешь в папке ROMS под названием **Battle City [BZK].nes**. Никаких изменений в геймплее, хак служит только для демонстрации работы команд.

Итак, что тебе нужно будет делать. Открываешь ром и дебаггер через ***Debug*** - ***Debugger***. В каждом из разделов с командами будет фраза **Execute**, после которой будет идти некий адрес. В дебаггере **создаешь** новый бряк, **здесь** прописываешь этот адрес (1234 просто для примера), ставишь **галочку** - ***ОК***.



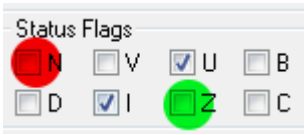

**Execute** переводится как ***выполнить***. Бряк сработает в тот момент, когда код будет проходить по команде, расположенной по этому **адресу**. Буква **E** говорит о том, что бряк активен, для активации/деактивации кликни по бряку дважды.

Кстати, кнопка ***Run*** в дебаггере снимает паузу с эмулятора. В моем хаке код будет бесконечно проходить по твоим брякам 60 раз в секунду, то есть каждый кадр, так что если ты что-то упустил - активируй бряк и жми кнопку, бряк мгновенно сработает снова.

Мои коды будут работать с адресами оперативки в диапазоне 0120-015F. Это адреса стека, про который я уже обещал рассказать тебе позднее. Заранее скажу, что использование адресов стека - занятие не самое безопасное, но код танчиков вроде как не затрагивает этот диапазон. Еще ты можешь вручную прописывать адрес рядом с Seek PC и жать кнопку, это переместит выполнение кода по этому адресу. Изредка может возникнуть такая ситуация, что во время прокрутки кода тебя перекинет в совершенно другое место. Я не в курсе с чем это связано. Если это случилось, ты можешь нажать Run и дождаться следующего срабатывания бряка, либо вручную переместить выполнение кода на прежний адрес.

## Подробное изучение команд

Сначала я расскажу про ***регистр статуса процессора***. Это еще один очень важный регистр. Байт, находящийся в нем, разбивается на биты, и каждый бит отражает состояние процессора и свойства результатов предыдущих операций. В статусе процессора биты называются **флагами**. Флаг - индикатор того, что было/не было произведено определенное действие. Он как лампочка на панели управления, которая загорается или тухнет в зависимости от чего-то, к чему она привязана. Если бит = 0, то привязанный к нему флаг будет деактивирован (очищен), если бит = 1, флаг будет активен (выставлен). Я предпочитаю слова ***активен*** и ***очищен***. Сам байт тебе не понадобится разбивать на биты, эмулятор уже сделал это за тебя и отобразил на отдельной панели в дебаггере. Кстати, ты можешь кликать по этим галочкам, меняя вручную состояние флагов, иногда полезно для тестирования кода.




Во время загрузки байта командами загрузки обновляется статус процессора, а конкретно флаги **N** (бит7) и **Z** (бит1). Флаг **N** указывает на то, положительный или отрицательный это байт. Процессор считает положительными байтами 00-7F, а отрицательными - 80-FF. Если байт положительный, флаг **N** очищается, если отрицательный - активируется. Флаг **Z** указывает на то, что байт = или ≠00. Если байт = 00, флаг **Z** активируется, если ≠00 - очищается. Оба флага никак не влияют на команды загрузки и записи, но они окажут влияние на другие команды. Для сокращения я буду писать N-, N+, Z-, Z+.

**Execute 8100**

Я буду переписывать команды из дебаггера сюда и объяснять их. Читай и пошагово прокручивай код. Некоторые читатели могут читать учебник вдали от компа, так что скриншоты команд прилагаются. Скрин может слегка отличаться от того, что ты видишь у себя в дебаггере прямо сейчас.

```
 00:80FF:FF        UNDEFINED
>00:8100:A9 00     LDA #$00
 00:8102:85 FF     STA $00FF = #$FF
 00:8104:A9 95     LDA #$95
 00:8106:85 FF     STA $00FF = #$FF
 00:8108:8D FF 00  STA $00FF = #$FF
 00:810B:A2 01     LDX #$01
 00:810D:A0 20     LDY #$20
 00:810F:8E 23 01  STX $0123 = #$01
 00:8112:8C 24 01  STY $0124 = #$20
 00:8115:BD 2E 01  LDA $012E,X @ $012E = #$FF
 00:8118:99 2C 01  STA $012C,Y @ $012C = #$00
 00:811B:A0 2F     LDY #$2F
 00:811D:96 FF     STX $00FF,Y @ $00FF = #$FF
 00:811F:BC 34 01  LDY $0134,X @ $0134 = #$00
 00:8122:A2 17     LDX #$17
 00:8124:A0 9D     LDY #$9D
 00:8126:A1 4E     LDA ($4E,X) @ $63FF = #$00
 00:8128:B1 8A     LDA ($8A),Y @ $0000 = #$00
 00:812A:60        RTS ----------------------
 00:812B:FF        UNDEFINED
```

LDA #$**00**
STA $00FF
Ничего примечательного, похожий код ты уже видел. Грузим **00** в A, при этом Z+ (байт = 00), N- (это положительный байт). Пишем **00** в 00FF.

LDA #$**95**
STA $00FF

STA $00FF
Z- (байт ≠00), N+ (отрицательный байт). **95** 2 раза подряд пишется в 00FF. При таком коде вторая команда просто перезапишет **95** по этому адресу, по сути она не делает ничего полезного, но сами команды STA отличаются друг от друга. В первой используются 2 байта - **85 FF**, а во второй уже 3 байта - **8D FF 00**. **85** и **8D** - разновидности команды STA. Смысл этих команд один и тот же, но команда **85** может работать только с диапазоном адресов 0000-00FF, а **8D** - 0000-FFFF, то есть по максимуму. **85** используется для экономии места, и разработчики игр активно этим пользуются, что слегка усложняет ромхакинг, если тебе нужно поменять адрес на более крупный, и не только. Адрес для команд с двумя байтами вроде **85** вычисляется так: по умолчанию адрес 0000, к нему прибавляется **FF** = 00**FF**. В процессоре адрес для любых команд с тремя байтами вроде **8D** вычисляется по-другому: **FF** и **00** переворачиваются местами = **00FF**. В дебаггере сразу отображается итоговый адрес, а после него - байт, который в нем уже записан. Я предпочитаю использовать команду **8D**, и прочие команды процессора, которые охватывают весь диапазон. На скрине со списком команд отображены в основном команды, охватывающие максимальный диапазон, они подписаны как ***АДРЕС***. Там есть и другие команды с нулевым адресом вроде **85**, они подписаны как ***адрес***, там их немного, но они там только из-за того, что не существует другой разновидности.

LD**X** #$01
LD**Y** #$20
Байт можно загружать не только в A, но также в **X** и **Y**, для каждого регистра своя команда. Байты появятся в дебаггере рядом с этими регистрами. Z-, N-, им не важно какой из трех регистров ты используешь.

STX $0123
STY $0124
Записываем байты из этих регистров по разным адресам.

LDA $012E,**X**
STA $012C,**Y**
Для вычисления адреса в командах загрузки и записи тебе может пригодиться любой из двух регистров **X** и **Y**, которые я называю регистрами координат, потому что они могут использоваться для уточнения конечных адресов. Сейчас **X** = 01, **Y** = 20. Адрес вычисляется так: 012E + **X** = 012F, 012C + **Y** = 014C. Дебаггер отображает итоговый адрес, однако в течении всего кода **X** и **Y** могут несколько раз меняться, пока код не дойдет до нужного тебе места, и соответственно будут пересчитываться адреса, это нужно учитывать при просмотре кода. Правильным итоговым адресом можно считать тот адрес, который дебаггер отображает в момент выполнения команды. Причем в разное время могут быть разные байты в **X** и **Y**, но для того эти команды и существуют, чтобы одной такой командой охватить несколько адресов. При грамотном использвании этих команд можно значительно сократить размер кода.

LDY #$2F
Обновим Y.

STX $00FF,**Y**
LDY $0134,**X**
**X** и **Y** могут использовать друг друга для вычисления конечного адреса для загрузки и записи. Для команд загрузки STX,**Y** и STY,**X** существуют лишь варианты с нулевым адресом, а для максимального адреса нельзя использовать противоположный регистр. Это может показаться немного неудобным, но на то они и вспомогательные регистры, у них ограниченные возможности. Записи по адресам лучше проводить через A.

LDX #$17
LDY #$9D
Обновим байты в регистрах.

LDA ($4E,X)
LDA ($8A),Y

Это 2 сложные команды загрузки. Есть 2 аналогичные команды записи, но я их в своем хаке не прописывал, ведь они могли повлиять на работу игры. Этими командами я сам почти не пользуюсь, но ты можешь встретить их во время просмотра кода игры, и тебе нужно понимать как в них вычисляется адрес. Сейчас я не могу знать что творится в твоей оперативке, поэтому просто покажу механику.

Разберем команду LDA ($**4E**,**X**). Итоговый адрес вычисляется так:
1. по умолчанию адрес 0000.
2. 0000 + **4E** + **X** = адрес**1**.
3. читается байт**1** из адреса**1**.
4. читается байт**2** из адреса2 (адрес**1** + 01).
5. байт**1** и байт**2** меняются местами = итоговый адрес, отсюда загружается байт в A.

Теперь разберем команду LDA ($**8A**),**Y**. Вот как вычисляется итоговый адрес:
1. по умолчанию адрес 0000
2. 0000 + **8A** = адрес**1**.
3. читается байт**1** из адреса**1**.
4. читается байт**2** из адреса2 (адрес**1** + 01).
5. оба байта меняются местами = адрес**3**.
6. адрес**3** + **Y** = итоговый адрес, отсюда загружается байт в A.

A1 использовал **X** для вычисления адреса на начальном этапе, а B1 использовал **Y** в самом конце. Чтобы было проще запомнить, обрати внимание на отображение команд в дебаггере: **X** находится в скобке (помогает вычислению адресов, откуда будут считываться байты), а **Y** вынесен за скобки (помогает вычислить итоговый адрес). Как в реальной математике - то, что находится в скобках, вычисляется первее. Ты прикинь, вероятно вот он, тот самый случай, про который тебе в школе твердили учителя - применение знаний на практике. Не зря же ты домашку делал.

RTS -----
Эта команда завершает отрезок кода, про нее я расскажу в командах со стеком. Палочки после команды просто для удобства восприятия.

А пока на этом все. В этой статье были показаны не все разновидности, но это не страшно, в дебаггере все равно показывается что это за команда, а разновидность легко понять по количеству байтов и по методу вычисления конечного адреса. И еще раз напомню про обновление состояния флагов во время загрузки байтов.

## Команды сравнения

Команды сравнения бывают **CMP**, **CPX** и **CPY** для каждого регистра соответственно. Само сравнение происходит с помощью вычитания из регистра некого байта, а результат этого вычитания будет отображаться во флагах N, Z и C, то есть команды сравнения нужны как раз для того, чтобы повлиять на эти флаги. Напомню для чего нужен каждый флаг:

• N - активен если байт или результат отрицательный (80-FF), очищен если положительный (00-80)

• Z - активен если байт или результат = 00, очищен если ≠00

• C - его мы еще не изучали. Он активен если байт больше или = байту, с которым тот сравнивался, очищен если меньше. На этот флаг можно повлиять не только сравнением, но об этом в других статьях.

А для того, чтобы использовать состояние этих флагов в своем коде, существуют 6 команд, по 2 на каждый флаг.

**Execute 8200**

```
 00:81FF:FF        UNDEFINED
>00:8200:A9 99     LDA #$99
 00:8202:A2 12     LDX #$12
 00:8204:A0 01     LDY #$01
 00:8206:8D 30 01  STA $0130 = #$99
 00:8209:8E 31 01  STX $0131 = #$12
 00:820C:8C 32 01  STY $0132 = #$01
 00:820F:AD 31 01  LDA $0131 = #$12
 00:8212:C9 12     CMP #$12
 00:8214:F0 03     BEQ $8219
 00:8216:8D 34 01  STA $0134 = #$00
 00:8219:D0 05     BNE $8220
 00:821B:A9 00     LDA #$00
 00:821D:8D 34 01  STA $0134 = #$00
 00:8220:AE 32 01  LDX $0132 = #$01
 00:8223:E0 05     CPX #$05
 00:8225:30 03     BMI $822A
 00:8227:8C 35 01  STY $0135 = #$FF
 00:822A:A2 03     LDX #$03
 00:822C:BC 2D 01  LDY $012D,X @ $0130 = #$99
 00:822F:E0 03     CPX #$03
 00:8231:C0 03     CPY #$03
 00:8233:10 03     BPL $8238
 00:8235:AD 32 01  LDA $0132 = #$01
 00:8238:C9 02     CMP #$02
 00:823A:B0 C4     BCS $8200
 00:823C:90 01     BCC $823F
 00:823E:60        RTS ----------------------
 00:823F:B9 50 01  LDA $0150,Y @ $01E9 = #$00
 00:8242:60        RTS ----------------------
 00:8243:FF        UNDEFINED
```

LDA #$99
LDX #$12
LDY #$01
STA $0130
STX $0131
STY $0132
Подготовка кода. Это значит что я специально прописал эти команды для дальнейшей демонстрации работы кода.

LDA $0131
В A загрузили 12. N-, Z-. C не обновляется при загрузке.

CMP #$12

Сравниваем A с байтом 12. Результат = 00, N-, Z+. C+, потому что A оказался больше или = 12. В A все еще 12, потому что команды сравнения не меняют байты в регистрах, они только оказывают влияние на флаги.

**BEQ 8219**
**STA** $0134
**BEQ** - проверка на = 00, то есть команда проверяет на Z+. Если активен, условие команды выполняется, код перепрыгнет **STA** и окажется на адресе **8219**, который в дебаггере прописан после **BEQ**. Разберемся откуда взялся **8219**. **BEQ** состоит из двух байтов - сама команда F0 + байт **03** (расстояние прыжка). **BEQ** прописан по адресу 8214. Адрес прыжка высчитывается так: 8214 + 02 + **03** = **8219**. Недалекие прыжки легко посчитать на глаз. Команда **STA** = 8D + 34 + 01 = 3 байта, и если ты хочешь перепрыгнуть только **STA**, то прописываешь **03**. При промотке кода **STA** перепрыгивается. Код уже становится интересней, это больше не загрузка + запись, а запись при каком-то условии.

**BNE** $**8220**
LDA #$00
STA $0134
Z все еще +. **BNE** проверяет на ≠00, то есть на Z-. Условие не выполнено, код запишет 00 в 0134. Если бы условие было выполнено, код прыгнул бы на **8220** (8219 + 02 + 05).

LDX $0132
CPX #$05
**BMI** $822A
STY $0135
X = **01**. **01** - 05 = FC. В процессоре нету таких чисел как ***минус 4***, поэтому если байт меньше вычитаемого, мысленно прибавляй к нему 100. Получится 1**01** - 05 = FC. N+, Z-. C-, потому что X оказался меньше 05. **BMI** проверяет на отрицательный байт. N+, условие выполнено, прыжок.

LDX #$03
Подготовка кода.

LDY $012D,X
**CPX** #$03
**CPY** #$03
BPL $8238
LDA $0132
Загрузили 99 в Y. Но дальше идет **CPX**, которому не интересен Y, ведь он проверяет X. Нужно быть внимательным при прописывании кода и не путать команды. **CPX** проверит X, 03 - 03 = 00, N-, Z+, C+. **CPY** проверит Y, 99 - 03 = 96, N+, Z-, C+. BPL проверяет на положительный байт. N+, условие не выполнено, A = 01.

CMP #$02
**BCS** $**8200**
**BCC** $823F
RTS -----
**LDA** $0150,Y
RTS -----
01 - 02 = FF. N+, Z-, C-. **BCS** проверяет на C+. Условие не выполняется - код идет дальше. Кстати, прыжок уже в обратную сторону, на **8200**. После **BCS** прописан C4. Дело в том, что при помощи всех этих 6 команд можно прыгать не только вперед (положительными байтами), но и назад (отрицательными байтами). Если ты поменяешь байт после команды и кликнешь на дебаггер, он обновит адрес для прыжка, поэтому ты можешь корректировать адрес для прыжка сколько угодно, пока не попадешь на нужный.
К счастью, условие **BCS** не выполняется, иначе мы бы прыгнули в самое начало и все началось бы по новой. Таким образом можно даже зациклить код, если он составлен неправильно. **BCC** проверяет на C-, условие выполнено, прыжок на **LDA**. Еще один вариант зацикливания - если прописать FE после **BCC**, и при выполнении условия она будет бесконечно прыгать сама на себя.

Итак, ты узнал про команды сравнения и чтения флагов для составления условий в своих кодах. Для флага C обязательна предварительная команды сравнения, потому что команды загрузки на него не влияют, а других способов влияния ты пока не знаешь.

## Команды увеличения и уменьшения на 01

Первые 2 команды - **INC** (увеличить байт в адресе на 01) и **DEC** (уменьшить байт в адресе на 01). Команды не умеют работать напрямую с A, только с адресами. Результат этих команд повлияет на флаги N и Z как обычно.

**Execute 8300**

```
 00:82FF:FF        UNDEFINED
>00:8300:A2 01     LDX #$01
 00:8302:A0 07     LDY #$07
 00:8304:A9 00     LDA #$00
 00:8306:8D 37 01  STA $0137 = #$FF
 00:8309:EE 37 01  INC $0137 = #$FF
 00:830C:CE 37 01  DEC $0137 = #$FF
 00:830F:DE 36 01  DEC $0136,X @ $013B = #$77
 00:8312:CA        DEX
 00:8313:C8        INY
 00:8314:A9 77     LDA #$77
 00:8316:99 37 01  STA $0137,Y @ $013A = #$00
 00:8319:88        DEY
 00:831A:E8        INX
 00:831B:E0 05     CPX #$05
 00:831D:D0 F7     BNE $8316
 00:831F:60        RTS ----------------------
 00:8320:FF        UNDEFINED
```

LDX #$01
LDY #$07
LDA #$00
STA $0137
Подготовка кода. В итоге N-, Z+.

INC $0137
0137 = 01. N-, Z-.

DEC $0137
0137 = 00. N-, Z+

DEC $0136,X
0137 = FF. N+, Z-. После этих команд A остается без изменений

Теперь команды, работающие напрямую с X и Y. Результат отобразится в этих регистрах.

DEX
INY
Уменьшаем X на 01 и увеличиваем Y на 01.

LDA #$**77**
**STA** $0137,Y
**DEY**
**INX**
CPX #$**05**
**BNE** $**8316**
**RTS**
Код, который будет записывать **77** по различным адресам. Это пример того, как X может работать в качестве счетчика. Сначала мы грузим **77** в A, и **пишем** его в 013F, поскольку Y = 08. Далее мы **уменьшаем** Y на 01, и **увеличиваем** X на 01. Если X еще **≠05**, мы прыгаем на команду **записи** A в адрес, который уже изменился благодаря **DEY**. В итоге у нас будут использованы 5 различных адресов пока X

увеличивается от 00 до 04, потому что при **05** код **завершится**. В данном примере **BNE** можно заменить на BCC. Сам прыжок можно было делать на 8314, а не на **8316**, но A на протяжении этого отрезка кода останется без изменений, поэтому нет смысла повторно загружать его перед **STA**.

## Арифметические команды

Первые 2 команды - сложение и вычитание. **ADC** - прибавить к A некий байт, **SBC** - вычесть из A некий байт. Результат повлияет на N и Z как обычно. У обеих команд есть свои особенности с флагом C, будем подробно разбирать.

**Execute 8400**

```
 00:83FF:FF        UNDEFINED
>00:8400:18        CLC
 00:8401:38        SEC
 00:8402:A2 00     LDX #$00
 00:8404:A0 00     LDY #$00
 00:8406:18        CLC
 00:8407:A9 01     LDA #$01
 00:8409:69 02     ADC #$02
 00:840B:38        SEC
 00:840C:7D 50 84  ADC $8450,X @ $8455 = #$FF
 00:840F:E8        INX
 00:8410:7D 50 84  ADC $8450,X @ $8455 = #$FF
 00:8413:E9 03     SBC #$03
 00:8415:18        CLC
 00:8416:ED 58 84  SBC $8458 = #$25
 00:8419:E8        INX
 00:841A:C8        INY
 00:841B:A9 44     LDA #$44
 00:841D:18        CLC
 00:841E:7D 50 84  ADC $8450,X @ $8455 = #$FF
 00:8421:A9 44     LDA #$44
 00:8423:F9 58 84  SBC $8458,Y @ $845B = #$FF
 00:8426:A9 21     LDA #$21
 00:8428:0A        ASL
 00:8429:0A        ASL
 00:842A:0A        ASL
 00:842B:8D 2C 01  STA $012C = #$00
 00:842E:4E 2C 01  LSR $012C = #$00
 00:8431:4E 2C 01  LSR $012C = #$00
 00:8434:4E 2C 01  LSR $012C = #$00
 00:8437:4E 2C 01  LSR $012C = #$00
 00:843A:4E 2C 01  LSR $012C = #$00
 00:843D:60        RTS ----------------------
 00:843E:FF        UNDEFINED
```

CLC
SEC
Предварительно разберем 2 команды, которые вручную меняют состояние C. CLC = C-, SEC = C+.

LDX #$00
LDY #$00
Подготовка кода.

CLC
LDA #$01
ADC #$02
Очищаем C, делаем сложение 01 + 02 = 03, результат отобразится в A.

**SEC**
ADC $8450,X
**Активируем** C, делаем сложение 03 + 23 = 27, что вроде как неправильно, ведь должно быть 26. Дело в том, что ADC проверяет на C, и если **C+**, команда прибавляет лишние **01**, в итоге 03 + 23 + **01** = 27. Также C-, дальше объясню почему.

INX
ADC $8450,X
Увеличили X на 01. 27 + E2 = **1**09, отобразится как 09. Результат перевалил за FF, следовательно C+. В предыдущем примере мы не привысили FF, поэтому там был C-.

SBC #$03
C+. 09 - 03 = 06. Команда SBC по взаимодействию с C зеракльна ADC. При C+ она сделала обычное вычитание. Результат не перевалил за 00, следовательно C+.

CLC
SBC $8458
**C-**. 06 - 25 - **01** = E0. Результат перевалил за 00, C-.

INX
INY
Подготовка кода.

LDA #$44
CLC
ADC $8450,X
CLC может быть прописана до или после LDA, это не важно, потому что LDA не влияет на C, важно чтобы перед ADC. Вообще, CLC можно ставить где угодно до ADC, если ты убедился что ничто другое не повлияет на C после CLC. Результат = 20, C+.

LDA #$44
SBC $8458,Y
Результат также = 20, C+. Обе команды взаимозаменяемые, ты в принципе можешь использовать любую из них для одинаковых целей, как тебе удобней.

Вторые 2 команды - умножение и деление. **ASL** - умножить байт на 2, **LSR** - разделить байт на 2. Результат повлияет на N и Z как обычно. Также команды влияют на C, каждая по-своему.

LDA #$21
ASL
C+. Результат = 21 * 2 = 42. Мы работаем с A, поэтому в нем отображается результат. N-, Z-. C-, потому что результат не перевалил за FF.

ASL
42 * 2 = 84. N+, Z-, C-.

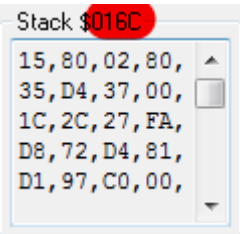
ASL
STA $012C
84 * 2 = **1**08 = 08. N-, Z-. C+, потому что результат перевалил за FF. Пишем A в 012C.

LSR $012C x 5
1. 08 / 2 = 04. N-, Z-. C-, потому что мы делим четное число. Сейчас мы работаем с адресом, а не с регистром, поэтому A без изменений.
2. 04 / 2 = 02. N-, Z,. C-.
3. 02 / 2 = 01. N-, Z,. C-.
4. 01 / 2 = 00. N-, Z+. При делении нечетного байта разультат всегда округляется в меньшую сторону. И раз это нечетный байт, то C+.
5. 00 / 2 = 00. N-, Z+, C-.

Эти особенности команд с флагом C сделаны для расширения возможностей вычисления, однако новичку они скорее будут мешать на начальном этапе. Я сам не могу вспомнить чтобы где-то их использовал. Главное запомнить, что для привычного сложения нужен предварительный C-, а для вычитания C+.

## Команды со стеком, прыжки и прочие команды

Наконец-то разберемся что такое стек. Это адреса 0100-01FF, которые используется для хранения байтов и для их извлечения из стека обратно, по принципу ***последним зашел - первым вышел***. Содержимое стека можно посмотреть в дебаггере.




**Адрес** указывает на адрес, в который будет помещен следующий байт для хранения, после чего адрес уменьшится, а при извлечении байта адрес увеличится. Этот адрес можно менять вручную при помощи команды TXS (которую мы изучать не будем). Со стеком нужно работать очень аккуратно, чтобы не испортить оригинальное течение кода игры, хотя здесь нет особых сложностей. Не используй стек в качестве свободных адресов, если ты не уверен что эти адреса не будут использоваться игрой в будущем.

Также стек применяется для хранения адресов прыжков с возвратом. Но сначала разберем хранение байтов.

**Execute 8400**

```
 00:84FF:FF        UNDEFINED
>00:8500:A9 24     LDA #$24
 00:8502:A2 85     LDX #$85
 00:8504:A0 11     LDY #$11
 00:8506:48        PHA
 00:8507:8A        TXA
 00:8508:48        PHA
 00:8509:98        TYA
 00:850A:48        PHA
 00:850B:A9 01     LDA #$01
 00:850D:A2 02     LDX #$02
 00:850F:A0 03     LDY #$03
 00:8511:68        PLA
 00:8512:A8        TAY
 00:8513:68        PLA
 00:8514:AA        TAX
 00:8515:68        PLA
 00:8516:08        PHP
 00:8517:28        PLP
 00:8518:20 30 85  JSR $8530
 00:851B:20 35 85  JSR $8535
 00:851E:4C 3A 85  JMP $853A
 00:8521:FF        UNDEFINED
 00:8522:FF        UNDEFINED
 00:8523:FF        UNDEFINED
 00:8524:EA        NOP
 00:8525:EA        NOP
 00:8526:EA        NOP
 00:8527:EA        NOP
 00:8528:EA        NOP
 00:8529:60        RTS ----------------
 00:852A:FF        UNDEFINED
 00:852B:FF        UNDEFINED
 00:852C:FF        UNDEFINED
 00:852D:FF        UNDEFINED
 00:852E:FF        UNDEFINED
 00:852F:FF        UNDEFINED
 00:8530:8D 27 01  STA $0127 = #$24
 00:8533:60        RTS ----------------
 00:8534:FF        UNDEFINED
 00:8535:8E 28 01  STX $0128 = #$85
 00:8538:60        RTS ----------------
 00:8539:FF        UNDEFINED
 00:853A:8C 29 01  STY $0129 = #$11
 00:853D:6C 27 01  JMP ($0127) = $8524
 00:8540:FF        UNDEFINED
```

LDA #$24
LDX #$85
LDY #$11
Подготовка кода.

PHA
Помещаем A в стек. Байт 24 отобразится в стеке в дебаггере. В A байт без изменений.

**TXA**
**PHA**
TXA - предварительно **копируем** байт из X в A, потому что в стек нельзя напрямую помещать X и Y. Теперь в обоих регистрах одинаковый байт. При копировании байта между регистрами обновляются N и Z. Теперь **помещаем** A в стек. В дебаггере отображено сначала **85**, затем 24, потому что **85** мы засунули последним, и сейчас он первый в списке на извлечение.

TYA

PHA
Аналогично сохраняем в стеке Y через A.

LDA #$01
LDX #$02
LDY #$03
Обновим байты в регистрах для демонстрации.

**PLA**
**TAY**
**PLA**
**TAX**
**PLA**
Сначала **вытаскиваем** из стека байт 11 и **перемещаем** его в Y. Затем **вытаскиваем** 85 и **перемещаем** в X. И последним **вытаскиваем** 24, который раньше был в A. В итоге мы сохранили по очереди байты из 3х регистров в стеке, и вытащили их из него в обратном порядке. Байты в регистрах поменялись на те, которые были извлечены из стека. Если вытащить больше/меньше байтов в конце своего кода, игра может в последствии зависнуть, поэтому тебе нужно убедиться что ты в стеке ничего не забыл, а также что ты не стащил из него ничего лишнего.

**PHP**
**PLP**
**PHP** - отдельная команда для регистра статуса процессора. Как ты помнишь, байт в этом регистре разбивается на биты и каждый бит отвечает за свой флаг, но байт остается байтом. **PHP** сохраняет этот байт в стеке, а **PLP** извлекает его оттуда. Изредка могут быть такие ситуации, когда тебе нужно полностью сохранить состояние флагов, для этого и нужны эти 2 команды. Кстати, стек не запоминает откуда был байт, который в нем хранится, поэтому в теории можно например поместить любой байт при помощи PHA, а вытащить его уже **PLP**, и флаги станут соответственными.

JSR $8530
JSR - прыжок с сохранением адреса прыжка, или прыжок с возвратом. 2 байта после команды меняются местами и получается адрес прыжка. При выполнении команды тебя перекинет на адрес 8530, а в стеке появятся 2 байта - 1A и 85. Дело в том, что JSR прописана по адресу 8518. Старший байт 85 записывается в стек первым, а к младшему 18 прибавляется 02 и он записывается последним.

STA $0127
RTS
В данный момент ты находишься в ***подпрограмме*** благодаря JSR, то есть JSR создает подпрограмму (это просто для технической инфы). RTS - возврат из этой подпрограммы. Команда считывает 2 последних байта из стека и составляет из них адрес 85 + (1A + 01) = 851B. Таким образом, находясь в подпрограмме, по байтам из стека ты можешь вычислить откуда был прыжок. Но для этого есть еще один способ. Найди в дебаггере кнопку ***Step Out***. При нажатии на нее тебя вернет к месту после JSR, а конкретнее по адресу из последних двух байтов стека. Если чуть более подробно, то при нажатии этой кнопки код будет выполняться до того момента, пока не дойдет до команды RTS, и выполнит эту команду (байты в стеке к этому времени уже могут отличаться от тех, которые ты видишь в стеке до нажатия кнопки).

JSR $8535
Еще один прыжок, но давай не будем прыгать, а разберем еще одну кнопку - ***Step Over***. Эта кнопка перепрыгивает команды JSR, но не просто перепрыгивает, а выполняет весь код из подпрограммы и возвращает тебя на следующую команду после JSR. Обе кнопки не всегда работают как надо, но это не баг, не буду в это углубляться.

JMP $853A
Это обычный прыжок, но без возврата, то есть в стеке не появится новых байтов, хотя команды RTS работают как надо, поэтому RTS не стоит ставить где попало.

STY $0129

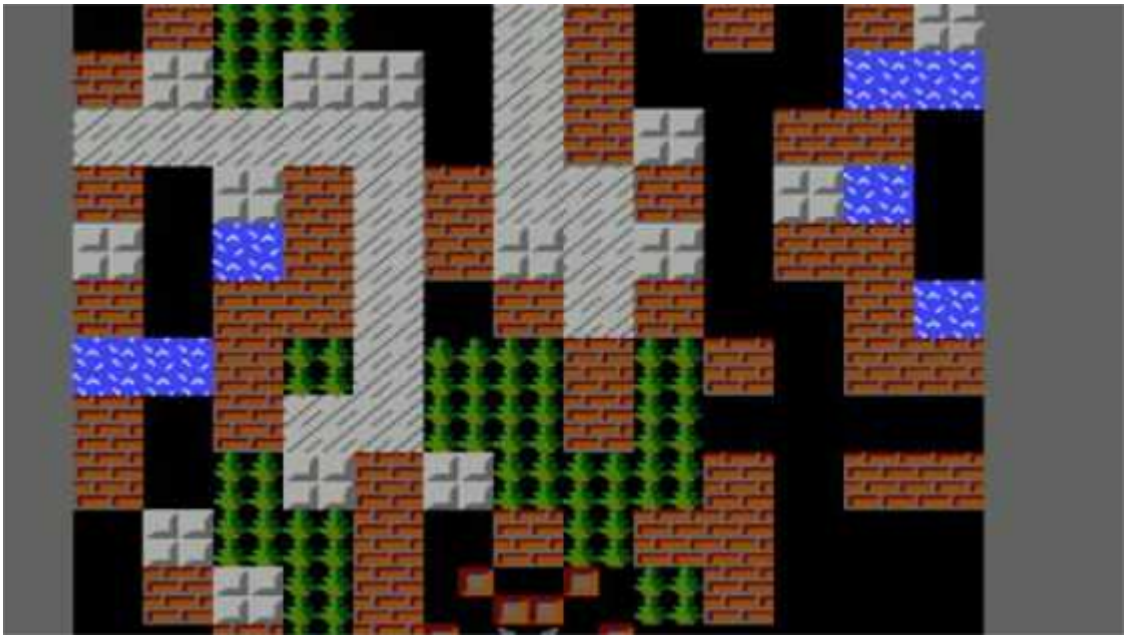
JMP ($0127)
Этот прыжок (6C) отличается от предыдущего (4C). После команды прописаны 27 01. Меняем байты местами и получаем 0127. 0127 = 24, 0128 = 85. Меняем байты местами, получаем 8524 - это адрес прыжка.

NOP x 5
NOP - отсутствие команды. При чтении этой команды эмулятор ничего не делает. Он просто шагнет на эту команду, посмотрит на нее, и пойдет дальше. Команда может быть полезна в основном для затирания кода. Допустим, ты написал много кода, и понял что что-то в нем лишнее. Ты можешь переписать свой код так, чтобы исключить из него этот лишний кусок, что займет у тебя некоторое время, а можешь просто на месте лишнего кода прописать много этих самых NOP. В общем, делай так, как тебе удобнее. Твое решение может зависеть от размера лишнего кода, от времени, которое тебе потребуется на перезапись, от того, что возможно ты потом захочешь вернуть все как было изначально, а также от других факторов. Ах да, еще классическое применение NOP - начальный читерский ромхакинг, затирание всяких DEC, чтобы жизни не отнимались. Кстати, не всегда можно просто затереть эту команду, нужно смотреть на то, какой код идет дальше, иногда лучше DEC заменить на LDA #$байт.

Ну и команда UNDEFINED. Есть несколько байтов, которые дебаггер отобразит вот так, это значит что команда не определена, то есть такой команды не существует. При написании кода убедись, что твой код не шагает по таким вот UNDEFINED, потому что результат может быть непредсказуемым. **BOOM!!!** Если в дебаггере под бряками поставить галочку ***Break on Bad Opcodes***, то дебаггер будет вызываться при шаге кода по таким неопознанным командам. Это может быть если ты неправильно указал координаты для прыжков и команд сравнения, или если ты использовал например JMP, а код завершаешь JSR, и все покатилось к чертям. В общем, отлично подходит для поиска багов, рекомендую постоянно включать эту галочку.

## Закрепление знаний

Если ты освоил все команды из статей и их особенности, для тебя не составит труда прочитать код игры и понять как он работает. По отдельности эти команды не так интересны, но когда ты пишешь свой собственный код - это захватывает. Поначалу будет трудно, я помню как сам учился, иногда на составление нескольких команд в правильном порядке уходил не один час, и все ради того, чтобы добавить какую-то мелочь. Тут уже дело в опыте, чем больше кодов ты напишешь, тем быстрее ты будешь писать новые, научишься их оптимизировать и совершать меньше ошибок. Если у тебя есть игра, которой ты хочешь заняться уже сейчас - замечательно. В моем случае я значительно повысил свой навык во время работы над танчиками, и вот что получилось.



Моей задачей было превратить унылые танчики в реальный шутер. В скорем времени я планирую переделать этот хак с нуля и улучшить его, потому что сейчас мне проще начать все заново, чем разгребать свои на тот момент примитивные коды и искать баги.

А напоследок я и предлагаю выполнить несложные ромхакерские задания с использованием уже изученных команд и посмотреть как это делаю я.

*По адресу 007F находится счетчик оставшихся в запасе танков. 4й, 11й и 18й танки - бонусные. Свободное место в роме начинается с FF50. В адресе 000B байт увеличивается на 01 каждый кадр.*

Задание 1. Сделать все 20 танков бонусными.

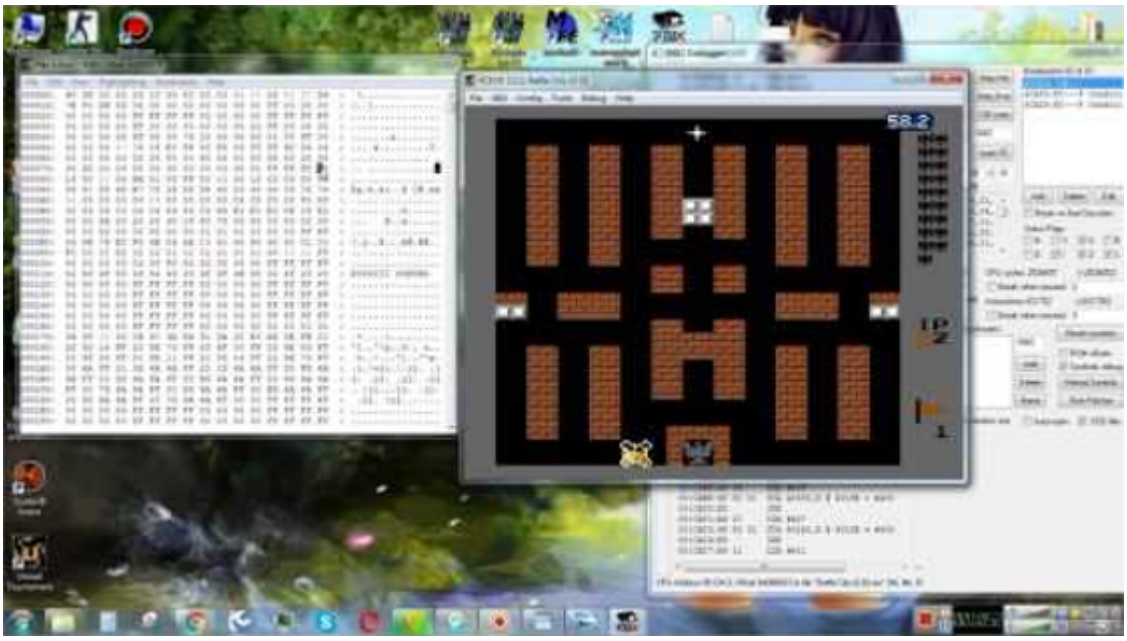
Задание 2. Запретить появление бонусных танков для режима 2 игрока.

Задание 3. Сделать появление бонусного танка с шансом 50% и 25%.

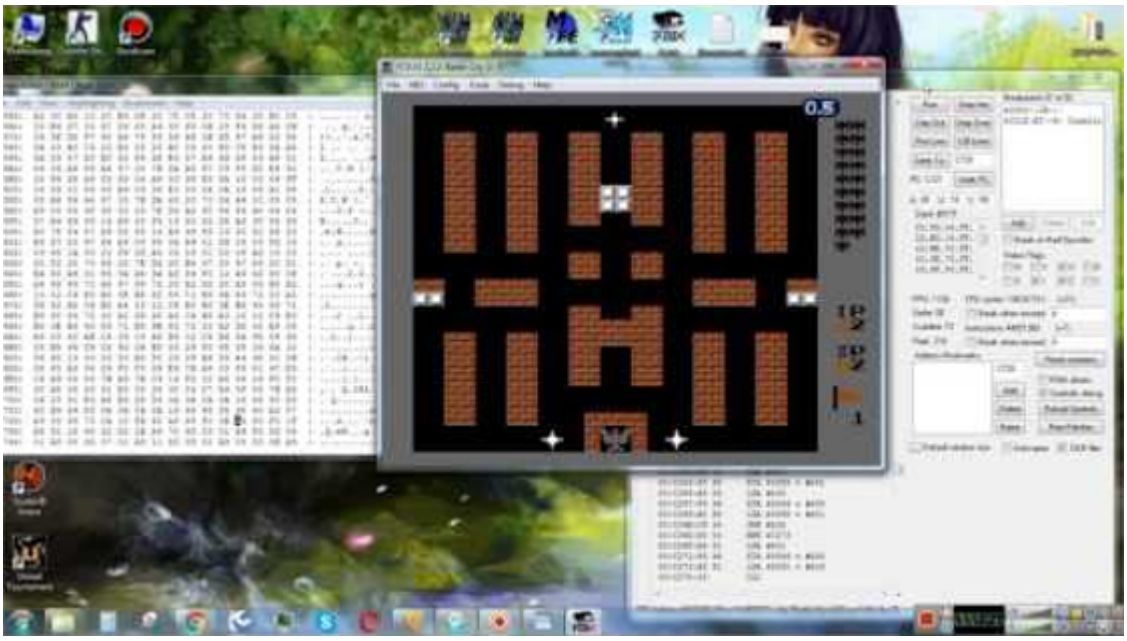
Задание 4. При появлении бонусного танка начислять +1 жизнь первому игроку.

Задание 5. Сделать бонусными любые 7 танков на уровне с промежутком минимум в 1 танк.

Задание 6. Появление 6 врагов в режиме 1 игрок.



Задание 7. При нажатии кнопки Select игроку начисляется +1 жизнь.



Теперь тебе остается только практиковаться, шпионить интернет в поисках ромхакерских статей от других авторов и ждать следующей версии учебника. С моим произношением у меня не очень хорошо получается снимать видео, но я постараюсь записать еще.

А пока ты переваривaешь полученную инфу, оставь отзыв на форуме и напиши, насколько тебе было понятно читать написанное, а также чему бы ты хотел научиться в будущем, и предложи свои идеи. Чем больше у тебя вопросов, тем больше статей я смогу для тебя написать. Ромхакинг жив!

## Ученик

Голень слизня...

## В разработке

## Адепт

...филе слизня...

## В разработке

## Эксперт

...и здесь нет слизня.

## В разработке

## Мастер

Futurama, 1й сезон, 7я серия.

## В разработке

# Статьи из интернета

Здесь собраны обычные и полезные статьи, большинство из них мне посоветовали на форуме. В основном это ссылки, я почти ничего из этого не читал, так что с уровнем ромхакинга я могу ошибаться, но это не так важно. Я их обязательно все почитаю и надеюсь подчерпну что-то полезное. Вообще, статьи я пытался читать только когда начинал раздупляться в ромхакинге, но ни одна из них мне не понравилась, поэтому я в основном разбирался самостоятельно.

Раздел со статьями нужен чтобы компенсировать нехватку информации в учебнике, ведь пока что готов только раздел для новичков. В будущем некоторые из статей могут быть переделаны мною и интегрированы в программу обучения, и тогда в них отпадет необходимость.

Самый лучший сайт по технической инфе **nesdev.com** (на английском). Я не пользуюсь поиском на этом сайте, просто гуглю инфу с указанием этого сайта.

# Новичок

Статьи уровня новичок.

## Ссылки на статьи

Первый урок. Введение, часть 1 (lancuster) **https://vk.cc/61OkYQ**

Первый урок. Введение, часть 2 (lancuster) **https://vk.cc/61Olec**

Азы использования Code/Data Logger (lancuster) **https://vk.cc/61OmqS**

Использование Game Genie кодов в ромхакинге и создание своих собственных GG кодов (lancuster) **https://vk.cc/61OmcN**

# Команды процессора 6502 (Bnu)

6502 Microprocessor

Revision 1.02 by _Bnu.

Most of the following information has been taking out of the "Commodore 64 Programmers Reference Manual" simply because it was available in electronic form and there appears to be no difference between this documentation and the 6502 documentation, they are both from the 6500 family after all. I've made changes and additions where appropriate.

In theory you should be able to use any code you can find for emulating the 6510 (the C64 processor).

## THE REGISTERS INSIDE THE 6502 MICROPROCESSOR

Almost all calculations are done in the microprocessor. Registers are special pieces of memory in the processor which are used to carry out, and store information about calculations. The 6502 has the following registers:

### THE ACCUMULATOR

This is THE most important register in the microprocessor. Various machine language instructions allow you to copy the contents of a memory location into the accumulator, copy the contents of the accumulator into a memory location, modify the contents of the accumulator or some other register directly, without affecting any memory. And the accumulator is the only register that has instructions for performing math.

### THE X INDEX REGISTER

This is a very important register. There are instructions for nearly all of the transformations you can make to the accumulator. But there are other instructions for things that only the X register can do. Various machine language instructions allow you to copy the contents of a memory location into the X register, copy the contents of the X register into a memory location, and modify the contents of the X, or some other register directly.

### THE Y INDEX REGISTER

This is a very important register. There are instructions for nearly all of the transformations you can make to the accumulator, and the X register. But there are other instructions for things that only the Y register can do. Various machine language instructions allow you to copy the contents of a memory location into the Y register, copy the contents of the Y register into a memory location, and modify the contents of the Y, or some other register directly.

### THE STATUS REGISTER

This register consists of eight "flags" (a flag = something that indicates whether something has, or has not occurred). Bits of this register are altered depending on the result of arithmetic and logical operations. These bits are described below:

| Bit No. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | S | V | | B | D | I | Z | C |

Bit0 - C - Carry flag: this holds the carry out of the most significant bit in any arithmetic operation. In subtraction operations however, this flag is cleared - set to 0 - if a borrow is required, set to 1 - if no borrow is required. The carry flag is also used in shift and rotate logical operations.

Bit1 - Z - Zero flag: this is set to 1 when any arithmetic or logical operation produces a zero result, and is set to 0 if the result is non-zero.

Bit 2 - I: this is an interrupt enable/disable flag. If it is set, interrupts are disabled. If it is cleared, interrupts are enabled.

Bit 3 - D: this is the decimal mode status flag. When set, and an Add with Carry or Subtract with Carry instruction is executed, the source values are treated as valid BCD (Binary Coded Decimal, eg. 0x00-0x99 = 0-99) numbers. The result generated is also a BCD number.

Bit 4 - B: this is set when a software interrupt (BRK instruction) is executed.

Bit 5: not used. Supposed to be logical 1 at all times.

Bit 6 - V - Overflow flag: when an arithmetic operation produces a result too large to be represented in a byte, V is set.

Bit 7 - S - Sign flag: this is set if the result of an operation is negative, cleared if positive.

The most commonly used flags are C, Z, V, S.

## THE PROGRAM COUNTER

This contains the address of the current machine language instruction being executed. Since the operating system is always "RUN"ning in the Commodore VIC-20 (or, for that matter, any computer), the program counter is always changing. It could only be stopped by halting the microprocessor in some way.

## THE STACK POINTER

This register contains the location of the first empty place on the stack. The stack is used for temporary storage by machine language programs, and by the computer.

ADDRESSING MODES

Instructions need operands to work on. There are various ways of indicating where the processor is to get these operands. The different methods used to do this are called addressing modes. The 6502 offers 11 modes, as described below.

1) Immediate
In this mode the operand's value is given in the instruction itself. In assembly language this is indicated by "#" before the operand.
eg. LDA #$0A - means "load the accumulator with the hex value 0A"
In machine code different modes are indicated by different codes. So LDA would be translated into different codes depending on the addressing mode. In this mode, it is: $A9 $0A

2 & 3) Absolute and Zero-page Absolute
In these modes the operands address is given.
eg. LDA $31F6 - (assembler)
$AD $31F6 - (machine code)
If the address is on zero page - i.e. any address where the high byte is 00 - only 1 byte is needed for the address. The processor automatically fills the 00 high byte.
eg. LDA $F4
$A5 $F4
Note the different instruction codes for the different modes.
Note also that for 2 byte addresses, the low byte is store first, eg. LDA $31F6 is stored as three bytes in memory, $AD $F6 $31.
Zero-page absolute is usually just called zero-page.

4) Implied
No operand addresses are required for this mode. They are implied by the instruction.
eg. TAX - (transfer accumulator contents to X-register)
$AA

5) Accumulator
In this mode the instruction operates on data in the accumulator, so no operands are needed.
eg. LSR - logical bit shift right
$4A

6 & 7) Indexed and Zero-page Indexed
In these modes the address given is added to the value in either the X or Y index register to give the actual address of the operand.
eg. LDA $31F6, Y
$D9 $31F6
LDA $31F6, X
$DD $31F6
Note that the different operation codes determine the index register used.
In the zero-page version, you should note that the X and Y registers are not interchangeable. Most instructions which can be used with zero-page indexing do so with X only.
eg. LDA $20, X
$B5 $20

8) Indirect
This mode applies only to the JMP instruction - JuMP to new location. It is indicated by parenthesis around the operand. The operand is the address of the bytes whose value is the new location.
eg. JMP ($215F)

| Assume the following - | byte | value |
|---|---|---|
| | $215F | $76 |
| | $2160 | $30 |

This instruction takes the value of bytes $215F, $2160 and uses that as the address to jump to - i.e. $3076 (remember that addresses are stored with low byte first).

9) Pre-indexed indirect
In this mode a zer0-page address is added to the contents of the X-register to give the address of the bytes holding the address of the operand. The indirection is indicated by parenthesis in assembly language.
eg. LDA ($3E, X)
$A1 $3E

| Assume the following - | byte | value |
|---|---|---|
| | X-reg. | $05 |
| | $0043 | $15 |
| | $0044 | $24 |
| | $2415 | $6E |

Then the instruction is executed by:
(i) adding $3E and $05 = $0043
(ii) getting address contained in bytes $0043, $0044 = $2415
(iii) loading contents of $2415 - i.e. $6E - into accumulator

Note a) When adding the 1-byte address and the X-register, wrap around addition is used - i.e. the sum is always a zero-page address.
eg. FF + 2 = 0001 not 0101 as you might expect.
DON'T FORGET THIS WHEN EMULATING THIS MODE.
b) Only the X register is used in this mode.

10) Post-indexed indirect
In this mode the contents of a zero-page address (and the following byte) give the indirect addressm which is added to the contents of the Y-register to yield the actual address of the operand. Again, inassembly language, the instruction is indicated by parenthesis.
eg. LDA ($4C), Y
Note that the parenthesis are only around the 2nd byte of the instruction since it is the part that does the indirection.

| Assume the following - | byte | value |
|---|---|---|
| | $004C | $00 |
| | $004D | $21 |
| | Y-reg. | $05 |
| | $2105 | $6D |

Then the instruction above executes by:
(i) getting the address in bytes $4C, $4D = $2100
(ii) adding the contents of the Y-register = $2105
(111) loading the contents of the byte $2105 - i.e. $6D into the accumulator.
Note: only the Y-register is used in this mode.

11) Relative

This mode is used with Branch-on-Condition instructions. It is probably the mode you will use most often. A 1 byte value is added to the program counter, and the program continues execution from that address. The 1 byte number is treated as a signed number - i.e. if bit 7 is 1, the number given byt bits 0-6 is negative; if bit 7 is 0, the number is positive. This enables a branch displacement of up to 127 bytes in either direction.

| eg | bit no. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | signed value | unsigned value |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | value | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | -39 | $A7 |
| | value | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | +39 | $27 |

Instruction example:

```
  BEQ $A7
  $F0 $A7
```

This instruction will check the zero status bit. If it is set, 39 decimal will be subtracted from the program counter and execution continues from that address. If the zero status bit is not set, execution continues from the following instruction.

Notes: a) The program counter points to the start of the instruction after the branch instruction before the branch displacement is added. Remember to take this into account when calculating displacements.

b) Branch-on-condition instructions work by checking the relevant status bits in the status register. Make sure that they have been set or unset as you want them. This is often done using a CMP instruction.

c) If you find you need to branch further than 127 bytes, use the opposite branch-on-condition and a JMP.

## MCS6502 MICROPROCESSOR INSTRUCTION SET - ALPHABETIC SEQUENCE

| | |
|---|---|
| ADC | Add Memory to Accumulator with Carry |
| AND | "AND" Memory with Accumulator |
| ASL | Shift Left One Bit (Memory or Accumulator) |
| BCC | Branch on Carry Clear |
| BCS | Branch on Carry Set |
| BEQ | Branch on Result Zero |
| BIT | Test Bits in Memory with Accumulator |
| BMI | Branch on Result Minus |
| BNE | Branch on Result not Zero |
| BPL | Branch on Result Plus |
| BRK | Force Break |
| BVC | Branch on Overflow Clear |
| BVS | Branch on Overflow Set |
| CLC | Clear Carry Flag |
| CLD | Clear Decimal Mode |
| CLI | Clear interrupt Disable Bit |
| CLV | Clear Overflow Flag |
| CMP | Compare Memory and Accumulator |
| CPX | Compare Memory and Index X |
| CPY | Compare Memory and Index Y |
| DEC | Decrement Memory by One |
| DEX | Decrement Index X by One |

| | |
|---|---|
| DEY | Decrement Index Y by One |
| EOR | "Exclusive-Or" Memory with Accumulator |
| INC | Increment Memory by One |
| INX | Increment Index X by One |
| INY | Increment Index Y by One |
| JMP | Jump to New Location |

MCS6502 MICROPROCESSOR INSTRUCTION SET - ALPHABETIC SEQUENCE

| | |
|---|---|
| JSR | Jump to New Location Saving Return Address |
| LDA | Load Accumulator with Memory |
| LDX | Load Index X with Memory |
| LDY | Load Index Y with Memory |
| LSR | Shift Right One Bit (Memory or Accumulator) |
| NOP | No Operation |
| ORA | "OR" Memory with Accumulator |
| PHA | Push Accumulator on Stack |
| PHP | Push Processor Status on Stack |
| PLA | Pull Accumulator from Stack |
| PLP | Pull Processor Status from Stack |
| ROL | Rotate One Bit Left (Memory or Accumulator) |
| ROR | Rotate One Bit Right (Memory or Accumulator) |
| RTI | Return from Interrupt |
| RTS | Return from Subroutine |
| SBC | Subtract Memory from Accumulator with Borrow |
| SEC | Set Carry Flag |
| SED | Set Decimal Mode |
| SEI | Set Interrupt Disable Status |
| STA | Store Accumulator in Memory |
| STX | Store Index X in Memory |
| STY | Store Index Y in Memory |
| TAX | Transfer Accumulator to Index X |
| TAY | Transfer Accumulator to Index Y |
| TSX | Transfer Stack Pointer to Index X |
| TXA | Transfer Index X to Accumulator |
| TXS | Transfer Index X to Stack Pointer |
| TYA | Transfer Index Y to Accumulator |

The following notation applies to this summary:

| | | | |
|---|---|---|---|
| A | Accumulator | EOR | Logical Exclusive Or |
| X, Y | Index Registers | fromS | Transfer from Stack |
| M | Memory | toS | Transfer to Stack |
| P | Processor Status Register | -> | Transfer to |
| S | Stack Pointer | <- | Transfer from |
| / | Change | V | Logical OR |
| _ | No Change | PC | Program Counter |
| + | Add | PCH | Program Counter High |
| /\ | Logical AND | PCL | Program Counter Low |
| - | Subtract | OPER | OPERAND |
| | | # | IMMEDIATE ADDRESSING MODE |

Note: At the top of each table is located in parentheses a reference number (Ref: XX) which directs the user to that Section in the MCS6500 Microcomputer Family Programming Manual in which the instruction is defined and discussed.

ADC Add memory to accumulator with carry ADC

Operation: A + M + C -> A, C

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| / | / | / | _ | _ | / |

(Ref: 2.2.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | ADC #Oper | 69 | 2 | 2 |
| Zero Page | ADC Oper | 65 | 2 | 3 |
| Zero Page,X | ADC Oper,X | 75 | 2 | 4 |
| Absolute | ADC Oper | 6D | 3 | 4 |
| Absolute,X | ADC Oper,X | 7D | 3 | 4* |
| Absolute,Y | ADC Oper,Y | 79 | 3 | 4* |
| (Indirect,X) | ADC (Oper,X) | 61 | 2 | 6 |
| (Indirect),Y | ADC (Oper),Y | 71 | 2 | 5* |

* Add 1 if page boundary is crossed.

AND "AND" memory with accumulator AND

Operation: A /\ M -> A

N Z C I D V
/ / _ _ _ _

(Ref: 2.2.3.0)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | AND #Oper | 29 | 2 | 2 |
| Zero Page | AND Oper | 25 | 2 | 3 |
| Zero Page,X | AND Oper,X | 35 | 2 | 4 |
| Absolute | AND Oper | 2D | 3 | 4 |
| Absolute,X | AND Oper,X | 3D | 3 | 4* |
| Absolute,Y | AND Oper,Y | 39 | 3 | 4* |
| (Indirect,X) | AND (Oper,X) | 21 | 2 | 6 |
| (Indirect,Y) | AND (Oper),Y | 31 | 2 | 5 |

* Add 1 if page boundary is crossed.

ASL ASL Shift Left One Bit (Memory or Accumulator) ASL

Operation: C <- |7|6|5|4|3|2|1|0| <- 0

N Z C I D V
/ / / _ _ _

(Ref: 10.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Accumulator | ASL A | 0A | 1 | 2 |
| Zero Page | ASL Oper | 06 | 2 | 5 |
| Zero Page,X | ASL Oper,X | 16 | 2 | 6 |
| Absolute | ASL Oper | 0E | 3 | 6 |
| Absolute, X | ASL Oper,X | 1E | 3 | 7 |

BCC BCC Branch on Carry Clear BCC

Operation: Branch on C = 0

N Z C I D V
_ _ _ _ _ _

(Ref: 4.1.1.3)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BCC Oper | 90 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BCS BCS Branch on carry set BCS

Operation: Branch on C = 1

N Z C I D V
_ _ _ _ _ _

(Ref: 4.1.1.4)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BCS Oper | B0 | 2 | 2* |

+----------------+----------------------+---------+---------+----------+
* Add 1 if branch occurs to same  page.
* Add 2 if branch occurs to next  page.

BEQ BEQ Branch on result zero BEQ

N Z C I D V

Operation: Branch on Z = 1 _ _ _ _ _ _

(Ref: 4.1.1.5)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BEQ Oper | F0 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to next page.

BIT BIT Test bits in memory with accumulator BIT

Operation: A /\ M, M7 -> N, M6 -> V

Bit 6 and 7 are transferred to the status register. N Z C I D V
If the result of A /\ M is zero then Z = 1, otherwise M7/ _ _ _ M6
Z = 0

(Ref: 4.2.1.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Zero Page | BIT Oper | 24 | 2 | 3 |
| Absolute | BIT Oper | 2C | 3 | 4 |

BMI BMI Branch on result minus BMI

Operation: Branch on N = 1 N Z C I D V

_ _ _ _ _ _

(Ref: 4.1.1.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BMI Oper | 30 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 1 if branch occurs to different page.

BNE BNE Branch on result not zero BNE

Operation: Branch on Z = 0 N Z C I D V

_ _ _ _ _ _

(Ref: 4.1.1.6)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BMI Oper | D0 | 2 | 2* |

+---------------+----------------------+---------+---------+----------+
* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BPL BPL Branch on result plus BPL

Operation: Branch on N = 0 N Z C I D V
– – – – – –
(Ref: 4.1.1.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BPL Oper | 10 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BRK BRK Force Break BRK

Operation: Forced Interrupt PC + 2 toS P toS N Z C I D V
– – – 1 – –
(Ref: 9.11)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | BRK | 00 | 1 | 7 |

1. A BRK command cannot be masked by setting I.

BVC BVC Branch on overflow clear BVC

Operation: Branch on V = 0 N Z C I D V
– – – – – –
(Ref: 4.1.1.8)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BVC Oper | 50 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BVS BVS Branch on overflow set BVS

Operation: Branch on V = 1 N Z C I D V
– – – – – –
(Ref: 4.1.1.7)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Relative | BVS Oper | 70 | 2 | 2* |

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

CLC CLC Clear carry flag CLC

Operation: 0 -> C

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| _ | _ | 0 | _ | _ | _ |

(Ref: 3.0.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | CLC | 18 | 1 | 2 |

CLD CLD Clear decimal mode CLD

Operation: 0 -> D

| N | A | C | I | D | V |
|---|---|---|---|---|---|
| _ | _ | _ | _ | 0 | _ |

(Ref: 3.3.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | CLD | D8 | 1 | 2 |

CLI CLI Clear interrupt disable bit CLI

Operation: 0 -> I

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| _ | _ | _ | 0 | _ | _ |

(Ref: 3.2.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | CLI | 58 | 1 | 2 |

CLV CLV Clear overflow flag CLV

Operation: 0 -> V

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| _ | _ | _ | _ | _ | 0 |

(Ref: 3.6.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | CLV | B8 | 1 | 2 |

CMP CMP Compare memory and accumulator CMP

Operation: A - M

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| / | / | / | _ | _ | _ |

(Ref: 4.2.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | CMP #Oper | C9 | 2 | 2 |
| Zero Page | CMP Oper | C5 | 2 | 3 |
| Zero Page,X | CMP Oper,X | D5 | 2 | 4 |
| Absolute | CMP Oper | CD | 3 | 4 |
| Absolute,X | CMP Oper,X | DD | 3 | 4* |
| Absolute,Y | CMP Oper,Y | D9 | 3 | 4* |
| (Indirect,X) | CMP (Oper,X) | C1 | 2 | 6 |
| (Indirect),Y | CMP (Oper),Y | D1 | 2 | 5* |

* Add 1 if page boundary is crossed.

CPX — CPX Compare Memory and Index X — CPX

Operation: X - M

N Z C I D V
/ / / _ _ _

(Ref: 7.8)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | CPX *Oper | E0 | 2 | 2 |
| Zero Page | CPX Oper | E4 | 2 | 3 |
| Absolute | CPX Oper | EC | 3 | 4 |

CPY — CPY Compare memory and index Y — CPY

Operation: Y - M

N Z C I D V
/ / / _ _ _

(Ref: 7.9)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | CPY *Oper | C0 | 2 | 2 |
| Zero Page | CPY Oper | C4 | 2 | 3 |
| Absolute | CPY Oper | CC | 3 | 4 |

DEC — DEC Decrement memory by one — DEC

Operation: M - 1 -> M

N Z C I D V
/ / _ _ _ _

(Ref: 10.7)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Zero Page | DEC Oper | C6 | 2 | 5 |
| Zero Page,X | DEC Oper,X | D6 | 2 | 6 |
| Absolute | DEC Oper | CE | 3 | 6 |
| Absolute,X | DEC Oper,X | DE | 3 | 7 |

DEX — DEX Decrement index X by one — DEX

Operation: X - 1 -> X

N Z C I D V
/ / _ _ _ _

(Ref: 7.6)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | DEX | CA | 1 | 2 |

DEY DEY Decrement index Y by one DEY

Operation: X - 1 -> Y

N Z C I D V
/ / _ _ _ _

(Ref: 7.7)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | DEY | 88 | 1 | 2 |

EOR EOR "Exclusive-Or" memory with accumulator EOR

Operation: A EOR M -> A

N Z C I D V
/ / _ _ _ _

(Ref: 2.2.3.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | EOR #Oper | 49 | 2 | 2 |
| Zero Page | EOR Oper | 45 | 2 | 3 |
| Zero Page,X | EOR Oper,X | 55 | 2 | 4 |
| Absolute | EOR Oper | 4D | 3 | 4 |
| Absolute,X | EOR Oper,X | 5D | 3 | 4* |
| Absolute,Y | EOR Oper,Y | 59 | 3 | 4* |
| (Indirect,X) | EOR (Oper,X) | 41 | 2 | 6 |
| (Indirect),Y | EOR (Oper),Y | 51 | 2 | 5* |

* Add 1 if page boundary is crossed.

INC INC Increment memory by one INC

N Z C I D V
Operation: M + 1 -> M / / _ _ _ _

(Ref: 10.6)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Zero Page | INC Oper | E6 | 2 | 5 |
| Zero Page,X | INC Oper,X | F6 | 2 | 6 |
| Absolute | INC Oper | EE | 3 | 6 |
| Absolute,X | INC Oper,X | FE | 3 | 7 |

INX INX Increment Index X by one INX

N Z C I D V
Operation: X + 1 -> X / / _ _ _ _

(Ref: 7.4)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | INX | E8 | 1 | 2 |

INY — INY Increment Index Y by one — INY

Operation: X + 1 -> X

N Z C I D V
/ / _ _ _ _

(Ref: 7.5)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | INY | C8 | 1 | 2 |

JMP — JMP Jump to new location — JMP

Operation: (PC + 1) -> PCL
(PC + 2) -> PCH (Ref: 4.0.2) (Ref: 9.8.1)

N Z C I D V
_ _ _ _ _ _

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Absolute | JMP Oper | 4C | 3 | 3 |
| Indirect | JMP (Oper) | 6C | 3 | 5 |

JSR — JSR Jump to new location saving return address — JSR

Operation: PC + 2 toS, (PC + 1) -> PCL
(PC + 2) -> PCH

N Z C I D V
_ _ _ _ _ _

(Ref: 8.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Absolute | JSR Oper | 20 | 3 | 6 |

LDA — LDA Load accumulator with memory — LDA

Operation: M -> A

N Z C I D V
/ / _ _ _ _

(Ref: 2.1.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | LDA #Oper | A9 | 2 | 2 |
| Zero Page | LDA Oper | A5 | 2 | 3 |
| Zero Page,X | LDA Oper,X | B5 | 2 | 4 |
| Absolute | LDA Oper | AD | 3 | 4 |
| Absolute,X | LDA Oper,X | BD | 3 | 4* |
| Absolute,Y | LDA Oper,Y | B9 | 3 | 4* |

| (Indirect,X) | LDA (Oper,X) | A1 | 2 | 6 |
|---|---|---|---|---|
| (Indirect),Y | LDA (Oper),Y | B1 | 2 | 5* |

* Add 1 if page boundary is crossed.

LDX LDX Load index X with memory LDX

Operation: M -> X

N Z C I D V
/ / _ _ _ _

(Ref: 7.0)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | LDX #Oper | A2 | 2 | 2 |
| Zero Page | LDX Oper | A6 | 2 | 3 |
| Zero Page,Y | LDX Oper,Y | B6 | 2 | 4 |
| Absolute | LDX Oper | AE | 3 | 4 |
| Absolute,Y | LDX Oper,Y | BE | 3 | 4* |

* Add 1 when page boundary is crossed.

LDY LDY Load index Y with memory LDY

N Z C I D V
Operation: M -> Y / / _ _ _ _

(Ref: 7.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | LDY #Oper | A0 | 2 | 2 |
| Zero Page | LDY Oper | A4 | 2 | 3 |
| Zero Page,X | LDY Oper,X | B4 | 2 | 4 |
| Absolute | LDY Oper | AC | 3 | 4 |
| Absolute,X | LDY Oper,X | BC | 3 | 4* |

* Add 1 when page boundary is crossed.

LSR LSR Shift right one bit (memory or accumulator) LSR

+-+-+-+-+-+-+-+-+
Operation: 0 -> |7|6|5|4|3|2|1|0| -> C N Z C I D V
+-+-+-+-+-+-+-+-+ 0 / / _ _ _

(Ref: 10.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Accumulator | LSR A | 4A | 1 | 2 |
| Zero Page | LSR Oper | 46 | 2 | 5 |
| Zero Page,X | LSR Oper,X | 56 | 2 | 6 |
| Absolute | LSR Oper | 4E | 3 | 6 |
| Absolute,X | LSR Oper,X | 5E | 3 | 7 |

NOP NOP No operation NOP

Operation: No Operation (2 cycles)

N Z C I D V
_ _ _ _ _ _

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | NOP | EA | 1 | 2 |

ORA — ORA "OR" memory with accumulator — ORA

Operation: A V M -> A

N Z C I D V
/ / _ _ _ _

(Ref: 2.2.3.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | ORA #Oper | 09 | 2 | 2 |
| Zero Page | ORA Oper | 05 | 2 | 3 |
| Zero Page,X | ORA Oper,X | 15 | 2 | 4 |
| Absolute | ORA Oper | 0D | 3 | 4 |
| Absolute,X | ORA Oper,X | 10 | 3 | 4* |
| Absolute,Y | ORA Oper,Y | 19 | 3 | 4* |
| (Indirect,X) | ORA (Oper,X) | 01 | 2 | 6 |
| (Indirect),Y | ORA (Oper),Y | 11 | 2 | 5 |

* Add 1 on page crossing

PHA — PHA Push accumulator on stack — PHA

Operation: A toS

N Z C I D V
_ _ _ _ _ _

(Ref: 8.5)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | PHA | 48 | 1 | 3 |

PHP — PHP Push processor status on stack — PHP

Operation: P toS

N Z C I D V
_ _ _ _ _ _

(Ref: 8.11)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | PHP | 08 | 1 | 3 |

PLA — PLA Pull accumulator from stack — PLA

Operation: A fromS

N Z C I D V

```
                                                          _ _ _ _ _ _
                              (Ref: 8.6)
+---------------+----------------------+---------+---------+----------+
| Addressing Mode| Assembly Language Form| OP CODE |No. Bytes|No. Cycles|
+---------------+----------------------+---------+---------+----------+
|  Implied      |   PLA                |    68   |    1    |    4     |
+---------------+----------------------+---------+---------+----------+
```

PLP PLP Pull processor status from stack PLA

```
Operation:  P fromS                                    N Z C I D V
                                                        From Stack
                              (Ref: 8.12)
+---------------+----------------------+---------+---------+----------+
| Addressing Mode| Assembly Language Form| OP CODE |No. Bytes|No. Cycles|
+---------------+----------------------+---------+---------+----------+
|  Implied      |   PLP                |    28   |    1    |    4     |
+---------------+----------------------+---------+---------+----------+
```

ROL ROL Rotate one bit left (memory or accumulator) ROL

```
             +------------------------------+
             |         M or A               |
             |   +-+-+-+-+-+-+-+-+    +-+   |
Operation:   +-< |7|6|5|4|3|2|1|0| <- |C| <-+         N Z C I D V
             +-+-+-+-+-+-+-+-+    +-+             / / / _ _ _
                              (Ref: 10.3)
+---------------+----------------------+---------+---------+----------+
| Addressing Mode| Assembly Language Form| OP CODE |No. Bytes|No. Cycles|
+---------------+----------------------+---------+---------+----------+
|  Accumulator  |   ROL A              |    2A   |    1    |    2     |
|  Zero Page    |   ROL Oper           |    26   |    2    |    5     |
|  Zero Page,X  |   ROL Oper,X         |    36   |    2    |    6     |
|  Absolute     |   ROL Oper           |    2E   |    3    |    6     |
|  Absolute,X   |   ROL Oper,X         |    3E   |    3    |    7     |
+---------------+----------------------+---------+---------+----------+
```

ROR ROR Rotate one bit right (memory or accumulator) ROR

```
             +------------------------------+
             |                              |
             |   +-+    +-+-+-+-+-+-+-+-+   |
Operation:   +-> |C| -> |7|6|5|4|3|2|1|0| >-+         N Z C I D V
             +-+    +-+-+-+-+-+-+-+-+             / / / _ _ _
                              (Ref: 10.4)
+---------------+----------------------+---------+---------+----------+
| Addressing Mode| Assembly Language Form| OP CODE |No. Bytes|No. Cycles|
+---------------+----------------------+---------+---------+----------+
|  Accumulator  |   ROR A              |    6A   |    1    |    2     |
|  Zero Page    |   ROR Oper           |    66   |    2    |    5     |
|  Zero Page,X  |   ROR Oper,X         |    76   |    2    |    6     |
|  Absolute     |   ROR Oper           |    6E   |    3    |    6     |
|  Absolute,X   |   ROR Oper,X         |    7E   |    3    |    7     |
+---------------+----------------------+---------+---------+----------+
```

Note: ROR instruction is available on MCS650X microprocessors after June, 1976.

RTI RTI Return from interrupt RTI

N Z C I D V

Operation: P fromS PC fromS From Stack

(Ref: 9.6)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | RTI | 4D | 1 | 6 |

RTS RTS Return from subroutine RTS

N Z C I D V

Operation: PC fromS, PC + 1 -> PC _ _ _ _ _ _

(Ref: 8.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | RTS | 60 | 1 | 6 |

SBC SBC Subtract memory from accumulator with borrow SBC

Operation: A - M - C -> A N Z C I D V

/ / / _ _ /

Note:C = Borrow (Ref: 2.2.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Immediate | SBC #Oper | E9 | 2 | 2 |
| Zero Page | SBC Oper | E5 | 2 | 3 |
| Zero Page,X | SBC Oper,X | F5 | 2 | 4 |
| Absolute | SBC Oper | ED | 3 | 4 |
| Absolute,X | SBC Oper,X | FD | 3 | 4* |
| Absolute,Y | SBC Oper,Y | F9 | 3 | 4* |
| (Indirect,X) | SBC (Oper,X) | E1 | 2 | 6 |
| (Indirect),Y | SBC (Oper),Y | F1 | 2 | 5 |

* Add 1 when page boundary is crossed.

SEC SEC Set carry flag SEC

Operation: 1 -> C N Z C I D V

_ _ 1 _ _ _

(Ref: 3.0.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | SEC | 38 | 1 | 2 |

SED SED Set decimal mode SED

Operation: 1 -> D

N Z C I D V
_ _ _ _ 1 _

(Ref: 3.3.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | SED | F8 | 1 | 2 |

SEI SEI Set interrupt disable status SED

Operation: 1 -> I

N Z C I D V
_ _ _ 1 _ _

(Ref: 3.2.1)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | SEI | 78 | 1 | 2 |

STA STA Store accumulator in memory STA

Operation: A -> M

N Z C I D V
_ _ _ _ _ _

(Ref: 2.1.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Zero Page | STA Oper | 85 | 2 | 3 |
| Zero Page,X | STA Oper,X | 95 | 2 | 4 |
| Absolute | STA Oper | 8D | 3 | 4 |
| Absolute,X | STA Oper,X | 9D | 3 | 5 |
| Absolute,Y | STA Oper, Y | 99 | 3 | 5 |
| (Indirect,X) | STA (Oper,X) | 81 | 2 | 6 |
| (Indirect),Y | STA (Oper),Y | 91 | 2 | 6 |

STX STX Store index X in memory STX

Operation: X -> M

N Z C I D V
_ _ _ _ _ _

(Ref: 7.2)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Zero Page | STX Oper | 86 | 2 | 3 |
| Zero Page,Y | STX Oper,Y | 96 | 2 | 4 |
| Absolute | STX Oper | 8E | 3 | 4 |

STY STY Store index Y in memory STY

Operation: Y -> M

N Z C I D V
_ _ _ _ _ _

(Ref: 7.3)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Zero Page | STY Oper | 84 | 2 | 3 |
| Zero Page,X | STY Oper,X | 94 | 2 | 4 |
| Absolute | STY Oper | 8C | 3 | 4 |

TAX TAX Transfer accumulator to index X TAX

Operation: A -> X

N Z C I D V
/ / _ _ _ _

(Ref: 7.11)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | TAX | AA | 1 | 2 |

TAY TAY Transfer accumulator to index Y TAY

Operation: A -> Y

N Z C I D V
/ / _ _ _ _

(Ref: 7.13)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | TAY | A8 | 1 | 2 |

TSX TSX Transfer stack pointer to index X TSX

Operation: S -> X

N Z C I D V
/ / _ _ _ _

(Ref: 8.9)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | TSX | BA | 1 | 2 |

TXA TXA Transfer index X to accumulator TXA

N Z C I D V
Operation: X -> A / / _ _ _ _

(Ref: 7.12)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | TXA | 8A | 1 | 2 |

TXS TXS Transfer index X to stack pointer TXS

Operation: X -> S

N Z C I D V
_ _ _ _ _ _

(Ref: 8.8)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | TXS | 9A | 1 | 2 |

TYA TYA Transfer index Y to accumulator TYA

Operation: Y -> A

N Z C I D V
/ / _ _ _ _

(Ref: 7.14)

| Addressing Mode | Assembly Language Form | OP CODE | No. Bytes | No. Cycles |
|---|---|---|---|---|
| Implied | TYA | 98 | 1 | 2 |

INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES
(in clock cycles)

| | ADC | AND | ASL | BCC | BCS | BEQ | BIT | BMI | BNE | BPL | BRK | BVC | BVS | CLC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Accumulator | . | . | 2 | . | . | . | . | . | . | . | . | . | . | . |
| Immediate | 2 | 2 | | . | . | . | . | . | . | . | . | . | . | . |
| Zero Page | 3 | 3 | 5 | . | . | . | 3 | . | . | . | . | . | . | . |
| Zero Page,X | 4 | 4 | 6 | . | . | . | . | . | . | . | . | . | . | . |
| Zero Page,Y | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| Absolute | 4 | 4 | 6 | . | . | . | 4 | . | . | . | . | . | . | . |
| Absolute,X | 4* | 4* | 7 | . | . | . | . | . | . | . | . | . | . | . |
| Absolute,Y | 4* | 4* | . | . | . | . | . | . | . | . | . | . | . | . |
| Implied | . | . | . | . | . | . | . | . | . | . | . | . | . | 2 |
| Relative | . | . | . | 2** | 2** | 2** | . | 2** | 2** | 2** | 7 | 2** | 2** | . |
| (Indirect,X) | 6 | 6 | . | . | . | . | . | . | . | . | . | . | . | . |
| (Indirect),Y | 5* | 5* | . | . | . | . | . | . | . | . | . | . | . | . |
| Abs. Indirect | . | . | . | . | . | . | . | . | . | . | . | . | . | . |

| | CLD | CLI | CLV | CMP | CPX | CPY | DEC | DEX | DEY | EOR | INC | INX | INY | JMP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Accumulator | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| Immediate | . | . | . | 2 | 2 | 2 | . | . | . | 2 | . | . | . | . |
| Zero Page | . | . | . | 3 | 3 | 3 | 5 | . | . | 3 | 5 | . | . | . |
| Zero Page,X | . | . | . | 4 | . | . | 6 | . | . | 4 | 6 | . | . | . |
| Zero Page,Y | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| Absolute | . | . | . | 4 | 4 | 4 | 6 | . | . | 4 | 6 | . | . | 3 |
| Absolute,X | . | . | . | 4* | . | . | 7 | . | . | 4* | 7 | . | . | . |
| Absolute,Y | . | . | . | 4* | . | . | . | . | . | 4* | . | . | . | . |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Implied | 2 | 2 | 2 | . | . | . | . | 2 | 2 | . | . | 2 | 2 | . |
| Relative | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| (Indirect,X) | . | . | . | 6 | . | . | . | . | . | 6 | . | . | . | . |
| (Indirect),Y | . | . | . | 5* | . | . | . | . | . | 5* | . | . | . | . |
| Abs. Indirect | . | . | . | . | . | . | . | . | . | . | . | . | . | 5 |

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES
(in clock cycles)

| | JSR | LDA | LDX | LDY | LSR | NOP | ORA | PHA | PHP | PLA | PLP | ROL | ROR | RTI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Accumulator | . | . | . | . | 2 | . | . | . | . | . | . | 2 | 2 | . |
| Immediate | . | 2 | 2 | 2 | . | . | 2 | . | . | . | . | . | . | . |
| Zero Page | . | 3 | 3 | 3 | 5 | . | 3 | . | . | . | . | 5 | 5 | . |
| Zero Page,X | . | 4 | . | 4 | 6 | . | 4 | . | . | . | . | 6 | 6 | . |
| Zero Page,Y | . | . | 4 | . | . | . | . | . | . | . | . | . | . | . |
| Absolute | 6 | 4 | 4 | 4 | 6 | . | 4 | . | . | . | . | 6 | 6 | . |
| Absolute,X | . | 4* | . | 4* | 7 | . | 4* | . | . | . | . | 7 | 7 | . |
| Absolute,Y | . | 4* | 4* | . | . | . | 4* | . | . | . | . | . | . | . |
| Implied | . | . | . | . | . | 2 | . | 3 | 3 | 4 | 4 | . | . | 6 |
| Relative | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| (Indirect,X) | . | 6 | . | . | . | . | 6 | . | . | . | . | . | . | . |
| (Indirect),Y | . | 5* | . | . | . | . | 5* | . | . | . | . | . | . | . |
| Abs. Indirect | . | . | . | . | . | . | . | . | . | . | . | . | . | . |

| | RTS | SBC | SEC | SED | SEI | STA | STX | STY | TAX | TAY | TSX | TXA | TXS | TYA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Accumulator | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| Immediate | . | 2 | . | . | . | . | . | . | . | . | . | . | . | . |
| Zero Page | . | 3 | . | . | . | 3 | 3 | 3 | . | . | . | . | . | . |
| Zero Page,X | . | 4 | . | . | . | 4 | . | 4 | . | . | . | . | . | . |
| Zero Page,Y | . | . | . | . | . | . | 4 | . | . | . | . | . | . | . |
| Absolute | . | 4 | . | . | . | 4 | 4 | 4 | . | . | . | . | . | . |
| Absolute,X | . | 4* | . | . | . | 5 | . | . | . | . | . | . | . | . |
| Absolute,Y | . | 4* | . | . | . | 5 | . | . | . | . | . | . | . | . |
| Implied | 6 | . | 2 | 2 | 2 | . | . | . | 2 | 2 | 2 | 2 | 2 | 2 |
| Relative | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| (Indirect,X) | . | 6 | . | . | . | 6 | . | . | . | . | . | . | . | . |
| (Indirect),Y | . | 5* | . | . | . | 6 | . | . | . | . | . | . | . | . |
| Abs. Indirect | . | . | . | . | . | . | . | . | . | . | . | . | . | . |

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

```
01 - ORA - (Indirect,X)          21 - AND - (Indirect,X)
02 - Future Expansion            22 - Future Expansion
03 - Future Expansion            23 - Future Expansion
04 - Future Expansion            24 - BIT - Zero Page
05 - ORA - Zero Page             25 - AND - Zero Page
06 - ASL - Zero Page             26 - ROL - Zero Page
07 - Future Expansion            27 - Future Expansion
08 - PHP                         28 - PLP
09 - ORA - Immediate             29 - AND - Immediate
0A - ASL - Accumulator           2A - ROL - Accumulator
0B - Future Expansion            2B - Future Expansion
0C - Future Expansion            2C - BIT - Absolute
0D - ORA - Absolute              2D - AND - Absolute
0E - ASL - Absolute              2E - ROL - Absolute
0F - Future Expansion            2F - Future Expansion
10 - BPL                         30 - BMI
11 - ORA - (Indirect),Y          31 - AND - (Indirect),Y
12 - Future Expansion            32 - Future Expansion
13 - Future Expansion            33 - Future Expansion
14 - Future Expansion            34 - Future Expansion
15 - ORA - Zero Page,X           35 - AND - Zero Page,X
16 - ASL - Zero Page,X           36 - ROL - Zero Page,X
17 - Future Expansion            37 - Future Expansion
18 - CLC                         38 - SEC
19 - ORA - Absolute,Y            39 - AND - Absolute,Y
1A - Future Expansion            3A - Future Expansion
1B - Future Expansion            3B - Future Expansion
1C - Future Expansion            3C - Future Expansion
1D - ORA - Absolute,X            3D - AND - Absolute,X
1E - ASL - Absolute,X            3E - ROL - Absolute,X
1F - Future Expansion            3F - Future Expansion

40 - RTI                         60 - RTS
41 - EOR - (Indirect,X)          61 - ADC - (Indirect,X)
42 - Future Expansion            62 - Future Expansion
43 - Future Expansion            63 - Future Expansion
44 - Future Expansion            64 - Future Expansion
45 - EOR - Zero Page             65 - ADC - Zero Page
46 - LSR - Zero Page             66 - ROR - Zero Page
47 - Future Expansion            67 - Future Expansion
48 - PHA                         68 - PLA
49 - EOR - Immediate             69 - ADC - Immediate
4A - LSR - Accumulator           6A - ROR - Accumulator
4B - Future Expansion            6B - Future Expansion
4C - JMP - Absolute              6C - JMP - Indirect
4D - EOR - Absolute              6D - ADC - Absolute
4E - LSR - Absolute              6E - ROR - Absolute
4F - Future Expansion            6F - Future Expansion
50 - BVC                         70 - BVS
51 - EOR - (Indirect),Y          71 - ADC - (Indirect),Y
52 - Future Expansion            72 - Future Expansion
53 - Future Expansion            73 - Future Expansion
54 - Future Expansion            74 - Future Expansion
55 - EOR - Zero Page,X           75 - ADC - Zero Page,X
56 - LSR - Zero Page,X           76 - ROR - Zero Page,X
57 - Future Expansion            77 - Future Expansion
58 - CLI                         78 - SEI
```

```
59 - EOR - Absolute,Y          79 - ADC - Absolute,Y
5A - Future Expansion          7A - Future Expansion
5B - Future Expansion          7B - Future Expansion
5C - Future Expansion          7C - Future Expansion
5D - EOR - Absolute,X          7D - ADC - Absolute,X
5E - LSR - Absolute,X          7E - ROR - Absolute,X
5F - Future Expansion          7F - Future Expansion

80 - Future Expansion          A0 - LDY - Immediate
81 - STA - (Indirect,X)        A1 - LDA - (Indirect,X)
82 - Future Expansion          A2 - LDX - Immediate
83 - Future Expansion          A3 - Future Expansion
84 - STY - Zero Page           A4 - LDY - Zero Page
85 - STA - Zero Page           A5 - LDA - Zero Page
86 - STX - Zero Page           A6 - LDX - Zero Page
87 - Future Expansion          A7 - Future Expansion
88 - DEY                       A8 - TAY
89 - Future Expansion          A9 - LDA - Immediate
8A - TXA                       AA - TAX
8B - Future Expansion          AB - Future Expansion
8C - STY - Absolute            AC - LDY - Absolute
8D - STA - Absolute            AD - LDA - Absolute
8E - STX - Absolute            AE - LDX - Absolute
8F - Future Expansion          AF - Future Expansion
90 - BCC                       B0 - BCS
91 - STA - (Indirect),Y        B1 - LDA - (Indirect),Y
92 - Future Expansion          B2 - Future Expansion
93 - Future Expansion          B3 - Future Expansion
94 - STY - Zero Page,X         B4 - LDY - Zero Page,X
95 - STA - Zero Page,X         B5 - LDA - Zero Page,X
96 - STX - Zero Page,Y         B6 - LDX - Zero Page,Y
97 - Future Expansion          B7 - Future Expansion
98 - TYA                       B8 - CLV
99 - STA - Absolute,Y          B9 - LDA - Absolute,Y
9A - TXS                       BA - TSX
9B - Future Expansion          BB - Future Expansion
9C - Future Expansion          BC - LDY - Absolute,X
9D - STA - Absolute,X          BD - LDA - Absolute,X
9E - Future Expansion          BE - LDX - Absolute,Y
9F - Future Expansion          BF - Future Expansion

C0 - Cpy - Immediate           E0 - CPX - Immediate
C1 - CMP - (Indirect,X)        E1 - SBC - (Indirect,X)
C2 - Future Expansion          E2 - Future Expansion
C3 - Future Expansion          E3 - Future Expansion
C4 - CPY - Zero Page           E4 - CPX - Zero Page
C5 - CMP - Zero Page           E5 - SBC - Zero Page
C6 - DEC - Zero Page           E6 - INC - Zero Page
C7 - Future Expansion          E7 - Future Expansion
C8 - INY                       E8 - INX
C9 - CMP - Immediate           E9 - SBC - Immediate
CA - DEX                       EA - NOP
CB - Future Expansion          EB - Future Expansion
CC - CPY - Absolute            EC - CPX - Absolute
CD - CMP - Absolute            ED - SBC - Absolute
CE - DEC - Absolute            EE - INC - Absolute
CF - Future Expansion          EF - Future Expansion
```

```
   D0 - BNE                        F0 - BEQ
   D1 - CMP   (Indirect@,Y         F1 - SBC - (Indirect),Y
   D2 - Future Expansion           F2 - Future Expansion
   D3 - Future Expansion           F3 - Future Expansion
   D4 - Future Expansion           F4 - Future Expansion
   D5 - CMP - Zero Page,X          F5 - SBC - Zero Page,X
   D6 - DEC - Zero Page,X          F6 - INC - Zero Page,X
   D7 - Future Expansion           F7 - Future Expansion
   D8 - CLD                        F8 - SED
   D9 - CMP - Absolute,Y           F9 - SBC - Absolute,Y
   DA - Future Expansion           FA - Future Expansion
   DB - Future Expansion           FB - Future Expansion
   DC - Future Expansion           FC - Future Expansion
   DD - CMP - Absolute,X           FD - SBC - Absolute,X
   DE - DEC - Absolute,X           FE - INC - Absolute,X
   DF - Future Expansion           FF - Future Expansion
```

INSTRUCTION OPERATION

The following code has been taken from VICE for the purposes of showing how each instruction operates. No particular addressing mode is used since we only wish to see the operation of the instruction itself.

```
src : the byte of data that is being addressed.
SET_SIGN : sets\resets the sign flag depending on bit 7.
SET_ZERO : sets\resets the zero flag depending on whether the result
           is zero or not.
SET_CARRY(condition) : if the condition has a non-zero value then the
           carry flag is set, else it is reset.
SET_OVERFLOW(condition) : if the condition is true then the overflow
           flag is set, else it is reset.
SET_INTERRUPT :  }
SET_BREAK :      }  As for SET_CARRY and SET_OVERFLOW.
SET_DECIMAL :    }
REL_ADDR(PC, src) : returns the relative address obtained by adding
           the displacement src to the PC.
SET_SR : set the Program Status Register to the value given.
GET_SR : get the value of the Program Status Register.
PULL : Pull a byte off the stack.
PUSH : Push a byte onto the stack.
LOAD : Get a byte from the memory address.
STORE : Store a byte in a memory address.
IF_CARRY, IF_OVERFLOW, IF_SIGN, IF_ZERO etc : Returns true if the
           relevant flag is set, otherwise returns false.
clk : the number of cycles an instruction takes. This is shown below
           in situations where the number of cycles changes depending
           on the result of the instruction (eg. Branching instructions).

AC = Accumulator
XR = X register
YR = Y register
PC = Program Counter
SP = Stack Pointer
```

```
/* ADC */
```

```
    unsigned int temp = src + AC + (IF_CARRY() ? 1 : 0);
    SET_ZERO(temp & 0xff);     /* This is not valid in decimal mode */
    if (IF_DECIMAL()) {
        if (((AC & 0xf) + (src & 0xf) + (IF_CARRY() ? 1 : 0)) > 9) temp += 6;
     SET_SIGN(temp);
     SET_OVERFLOW(!((AC ^ src) & 0x80) && ((AC ^ temp) & 0x80));
     if (temp > 0x99) temp += 96;
     SET_CARRY(temp > 0x99);
    } else {
     SET_SIGN(temp);
     SET_OVERFLOW(!((AC ^ src) & 0x80) && ((AC ^ temp) & 0x80));
     SET_CARRY(temp > 0xff);
    }
    AC = ((BYTE) temp);

/* AND */
    src &= AC;
    SET_SIGN(src);
    SET_ZERO(src);
    AC = src;

/* ASL */
    SET_CARRY(src & 0x80);
    src <<= 1;
    src &= 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    STORE src in memory or accumulator depending on addressing mode.

/* BCC */
    if (!IF_CARRY()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* BCS */
    if (IF_CARRY()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* BEQ */
    if (IF_ZERO()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* BIT */
    SET_SIGN(src);
    SET_OVERFLOW(0x40 & src);     /* Copy bit 6 to OVERFLOW flag. */
    SET_ZERO(src & AC);

/* BMI */
    if (IF_SIGN()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }
```

```
/* BNE */
    if (!IF_ZERO()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* BPL */
    if (!IF_SIGN()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* BRK */
    PC++;
    PUSH((PC >> 8) & 0xff);     /* Push return address onto the stack. */
    PUSH(PC & 0xff);
    SET_BREAK((1));             /* Set BFlag before pushing */
    PUSH(SR);
    SET_INTERRUPT((1));
    PC = (LOAD(0xFFFE) | (LOAD(0xFFFF) << 8));

/* BVC */
    if (!IF_OVERFLOW()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* BVS */
    if (IF_OVERFLOW()) {
     clk += ((PC & 0xFF00) != (REL_ADDR(PC, src) & 0xFF00) ? 2 : 1);
     PC = REL_ADDR(PC, src);
    }

/* CLC */
    SET_CARRY((0));

/* CLD */
    SET_DECIMAL((0));

/* CLI */
    SET_INTERRUPT((0));

/* CLV */
    SET_OVERFLOW((0));

/* CMP */
    src = AC - src;
    SET_CARRY(src < 0x100);
    SET_SIGN(src);
    SET_ZERO(src &= 0xff);

/* CPX */
    src = XR - src;
    SET_CARRY(src < 0x100);
    SET_SIGN(src);
    SET_ZERO(src &= 0xff);
```

```
/* CPY */
    src = YR - src;
    SET_CARRY(src < 0x100);
    SET_SIGN(src);
    SET_ZERO(src &= 0xff);

/* DEC */
    src = (src - 1) & 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    STORE(address, (src));

/* DEX */
    unsigned src = XR;
    src = (src - 1) & 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    XR = (src);

/* DEY */
    unsigned src = YR;
    src = (src - 1) & 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    YR = (src);

/* EOR */
    src ^= AC;
    SET_SIGN(src);
    SET_ZERO(src);
    AC = src;

/* INC */
    src = (src + 1) & 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    STORE(address, (src));

/* INX */
    unsigned src = XR;
    src = (src + 1) & 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    XR = (src);

/* INY */
    unsigned src = YR;
    src = (src + 1) & 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    YR = (src);

/* JMP */
    PC = (src);

/* JSR */
```

```
    PC--;
    PUSH((PC >> 8) & 0xff);     /* Push return address onto the stack. */
    PUSH(PC & 0xff);
    PC = (src);

/* LDA */
    SET_SIGN(src);
    SET_ZERO(src);
    AC = (src);

/* LDX */
    SET_SIGN(src);
    SET_ZERO(src);
    XR = (src);

/* LDY */
    SET_SIGN(src);
    SET_ZERO(src);
    YR = (src);

/* LSR */
    SET_CARRY(src & 0x01);
    src >>= 1;
    SET_SIGN(src);
    SET_ZERO(src);
    STORE src in memory or accumulator depending on addressing mode.

/* NOP */
    Nothing.

/* ORA */
    src |= AC;
    SET_SIGN(src);
    SET_ZERO(src);
    AC = src;

/* PHA */
    src = AC;
    PUSH(src);

/* PHP */
    src = GET_SR;
    PUSH(src);

/* PLA */
    src = PULL();
    SET_SIGN(src);     /* Change sign and zero flag accordingly. */
    SET_ZERO(src);

/* PLP */
    src = PULL();
    SET_SR((src));

/* ROL */
    src <<= 1;
    if (IF_CARRY()) src |= 0x1;
    SET_CARRY(src > 0xff);
```

```
    src &= 0xff;
    SET_SIGN(src);
    SET_ZERO(src);
    STORE src in memory or accumulator depending on addressing mode.

/* ROR */
    if (IF_CARRY()) src |= 0x100;
    SET_CARRY(src & 0x01);
    src >>= 1;
    SET_SIGN(src);
    SET_ZERO(src);
    STORE src in memory or accumulator depending on addressing mode.

/* RTI */
    src = PULL();
    SET_SR(src);
    src = PULL();
    src |= (PULL() << 8);     /* Load return address from stack. */
    PC = (src);

/* RTS */
    src = PULL();
    src += ((PULL()) << 8) + 1;     /* Load return address from stack and add 1. */
    PC = (src);

/* SBC */
    unsigned int temp = AC - src - (IF_CARRY() ? 0 : 1);
    SET_SIGN(temp);
    SET_ZERO(temp & 0xff);     /* Sign and Zero are invalid in decimal mode */
    SET_OVERFLOW(((AC ^ temp) & 0x80) && ((AC ^ src) & 0x80));
    if (IF_DECIMAL()) {
     if ( ((AC & 0xf) - (IF_CARRY() ? 0 : 1)) < (src & 0xf)) /* EP */ temp -= 6;
     if (temp > 0x99) temp -= 0x60;
    }
    SET_CARRY(temp < 0x100);
    AC = (temp & 0xff);

/* SEC */
    SET_CARRY((1));

/* SED */
    SET_DECIMAL((1));

/* SEI */
    SET_INTERRUPT((1));

/* STA */
    STORE(address, (src));

/* STX */
    STORE(address, (src));

/* STY */
    STORE(address, (src));

/* TAX */
    unsigned src = AC;
```

```
    SET_SIGN(src);
    SET_ZERO(src);
    XR = (src);

/* TAY */
    unsigned src = AC;
    SET_SIGN(src);
    SET_ZERO(src);
    YR = (src);

/* TSX */
    unsigned src = SP;
    SET_SIGN(src);
    SET_ZERO(src);
    XR = (src);

/* TXA */
    unsigned src = XR;
    SET_SIGN(src);
    SET_ZERO(src);
    AC = (src);

/* TXS */
    unsigned src = XR;
    SP = (src);

/* TYA */
    unsigned src = YR;
    SET_SIGN(src);
    SET_ZERO(src);
    AC = (src);
```

```
                ..................
 ..............: СТАНДАРТНЫЕ КОДЫ :..............
:                                                :\
:  [a?] Другой вариант  [p?] Пиратский           :\
:  [b?] Плохой дамп     [t?] С трейнером         :\
:  [f?] Исправленный    [T-] Старый перевод      :\
:  [o?] Излишний дамп   [T+] Обновлённый перевод:\
:  [h?] Хак             (-) Неизвестный год      :\
:  [!p] Нерассмотренный [!] Утверждённый         :\
:       дамп                хороший дамп         :\
: (M#)  Многоязычный (# - число языков)          :\
: (###) Контрольная     (??k) Размер ROM'a       :\
:       сумма           (Unl) Нелицензированный  :\
:................................................:\
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

                ...............
 ..............: ОСОБЫЕ КОДЫ :..............
:                                              :\
: .-----Gameboy-----.  .----Super Nintendo----. :\
: [  [C] Color      ]  [ (BS) ROM'ы BS        ] :\
: [  [S] Super      ]  [ (ST) Sufami Turbo    ] :\
: [ [BF] Bung Fix   ]  [ (NP) Nintendo Power  ] :\
: `-----------------'  `----------------------' :\
:                      .--------Atari---------. :\
: .-----Genesis-----.  [ (PAL) Европейская    ] :\
: [ (1) Япония      ]  [       версия         ] :\
: [ (1) Япония      ]  `----------------------' :\
: [ (4) США         ]  .---------GBA----------. :\
: [ (5) Только NTSC ]  [ [hI??] Хак вступления] :\
: [ [R-] Страны     ]  [ [f_?] Фикс EEPROMV124] :\
: [ (8) Только PAL  ]  `----------------------' :\
: [ (B) Не США      ]                          :\
: [ [c] Контр. сумма]  .--------Coleco--------. :\
: [ [x] Неверная    ]  [ (Adam) Версия ADAM   ] :\
: [     контр. сумма]  `----------------------' :\
: `-----------------'                          :\
:                      .--------NES/FC--------. :\
: .--NeoGeo Pocket--.  [ (PC10) PlayChoice-10 ] :\
: [ [M] Только моно ]  [   (VS) Versus        ] :\
: `-----------------'  [ [hFFE] FFE Copier fmt] :\
:                      `----------------------' :\
:................................................:\
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

                ..............
 ..............: КОДЫ СТРАН :..............
:                                              :\
:  (1) Япония и Корея    (4) США и Бразилия NTSC:\
:  (A) Австралия         (J) Япония             :\
:  (B) Не США (Genesis)  (K) Корея              :\
:  (C) Китай            (NL) Нидерланды         :\
:  (E) Европа           (PD) Всеобщее достояние :\
:  (F) Франция           (S) Испания            :\
: (FC) Канада (франц.)  (Sw) Швеция             :\
```

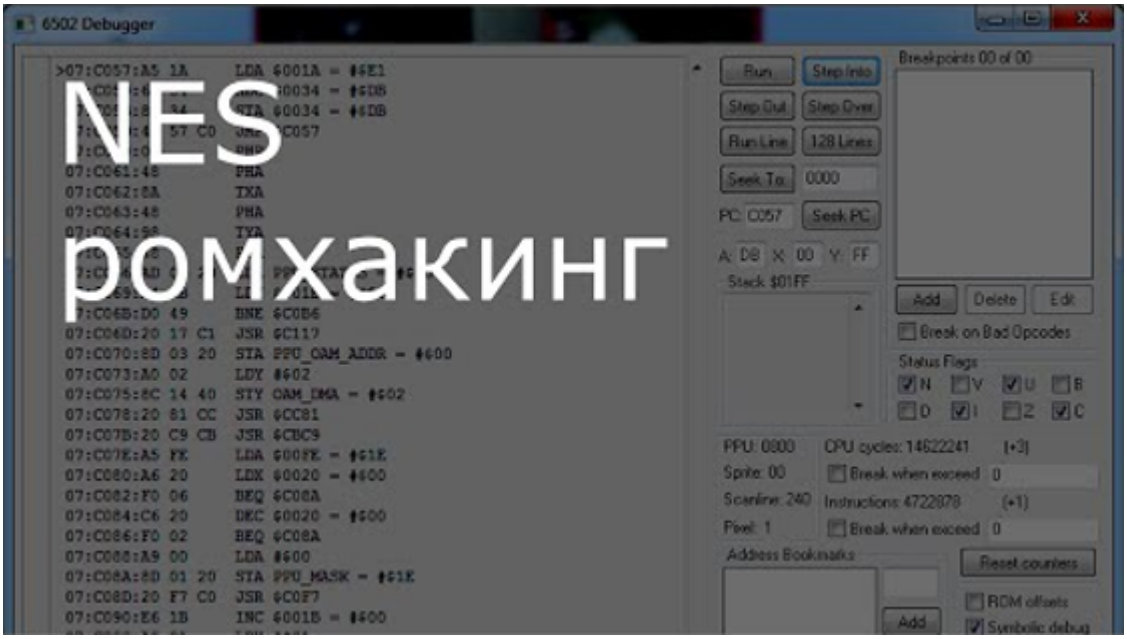
```
: (FN) Финляндия          (U) США                 :\
:  (G) Германия          (UK) Англия              :\
: (GR) Греция           (Unk) Страна неизвестна   :\
: (HK) Гонконг            (I) Италия               :\
:  (D) Голландия        (Unl) Нелицензировано     :\
:.................................................:\
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

          ..............................
 .........: ЗАМЕЧАНИЯ К СТАНДАРТНЫМ КОДАМ :.........
:                                                  :\
: [a] Это просто альтернативная версия ROM'а.      :\
:     Многие игры переиздавались для исправ-       :\
:     ления ошибок или чтобы убрать коды           :\
:     Game Genie (да, Нинтендо его ненавидит).     :\
:             -------------------                  :\
: [b] Дамп часто получается плохим у старых игр,   :\
:     из-за неисправного дампера или плохого       :\
:     соединения. Ещё один частый источник         :\
:     ROM'ов [b] - неудачная закачка их по FTP.    :\
:             -------------------                  :\
: [f] Игра, исправленная так, что она стала        :\
:     лучше работать на приставке и эмуляторе.     :\
:             -------------------                  :\
: [h] Что-то в этом ROM'е отличается от ориги-     :\
:     нала. Зачастую у взломанного ROM'а всего     :\
:     лишь изменён заголовок или добавлена         :\
:     возможность игры в прочих регионах. В ос-    :\
:     тальных случаях это может быть заставка      :\
:     группы, снявшей ROM, или какой-нибудь чит,   :\
:     или забавный хак.                            :\
:             -------------------                  :\
: [o] В излишне снятом образе ROM содержится       :\
:     больше данных, чем есть на картридже.        :\
:     Дополнительная информация не означает ни-    :\
:     чего и удалена из действительного образа.    :\
:             -------------------                  :\
: [t] Трейнер - это особый код, задействующийся    :\
:     до запуска игры. Он позволяет получить       :\
:     доступ к читам из меню.                      :\
:             -------------------                  :\
: [!] Утверждённый хороший дамп. Восславим         :\
:     Господа за него!                             :\
:................................................:\
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

              ............................
 ..........: ЗАМЕЧАНИЯ К ОСОБЫМ КОДАМ :..........
:                                                  :\
:                  **** SNES ****                  :\
: (BS) Эти японские ROM'ы распространялись по      :\
:      японской системе спутников, известной       :\
:      как Broadcast Satellaview. Они передава-    :\
:      лись во время показа телесериала, каким-    :\
:      то образом имеющего отношение к игре.       :\
:      В такие игры можно было играть только       :\
:      в течение часа, пока шла серия, и многие    :\
```

```
:      были рассчитаны по времени так, что      :\
:      игрались лишь в определённые периоды.    :\
:             ------------------                :\
: (ST) Устройство Sufami Turbo позволяло вста-  :\
:      вить в SNES два картриджа размером с     :\
:      картриджи для GameBoy. Некоторые картри- :\
:      джи, соединившись, давали новую игру,    :\
:      наподобие Sonic & Knuckles от Sega.      :\
:             ------------------                :\
: (NP) Журнал Nintendo Power распространял игры :\
:      среди своих подписчиков. Большинство     :\
:      таких ROM'ов - японские, так как такое   :\
:      практиковалось в основном в Японии.      :\
:             ------------------                :\
:                                               :\
:               **** Genesis ****               :\
:  (1) Картриджи с таким кодом играются как на  :\
:      японских, так и на корейских приставках. :\
:             ------------------                :\
:  (4) Технически этот код - то же самое, что и :\
:      (U) code, но он имеет более новый формат :\
:      заголовка и означает, что картридж будет :\
:      играться на приставках формата NTSC      :\
:      США и Бразилии                           :\
:             ------------------                :\
:  (B) Такой код региона указывает, что картридж:\
:      запустится на любой приставке не из США. :\
:             ------------------                :\
:  [c] Этот код обозначает картридж с известны- :\
:      ми несовпадениями Контрольной суммы.     :\
:             ------------------                :\
:                                               :\
:               **** GameBoy ****               :\
: [BF] Компания Bung выпустила программируемый  :\
:      картридж, совместимый с GameBoy, который :\
:      мог содержать любые нужные данные. Тем   :\
:      не менее, многие игры не работают на кар-:\
:      триджах Bung v1.0, и их нужно 'править'. :\
:             ------------------                :\
:                                               :\
:               **** Nintendo ****              :\
: PC10 PlayChoice-10 - аркадный автомат, проиг- :\
:      рывавший точные копии игр NES. На авто-  :\
:      матах был выбор из 10 игр, и они рабо-   :\
:      тали по 3 минуты за 25 центов.           :\
:             ------------------                :\
:                                               :\
:   VS Платформа Versus system основывалась на  :\
:      оборудовании, схожем с автоматами PC10,  :\
:      и лишь позволяла играть друг против      :\
:      друга.                                   :\
:................................................:\
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

## Ученик

Статьи уровня ученик.

## Ссылки на статьи




Редактирование титульника (lancuster) **https://vk.cc/61OIlw**

Разбираемся в тайловом редакторе YY-CHR (lancuster) **https://vk.cc/61OlSm**

Оптимизация графики в 8 bit (2bpp) (lancuster) **https://vk.cc/61OmBx**

Русифицирование РОМов (Базовый РОМ хакинг) (limnique) **https://vk.cc/61OmIp**

Перевод приставочных игр (ALLiGaToR) **https://vk.cc/61Oq6p**

NES изнутри (CaH4e3) **https://vk.cc/61OrQg**

Несколько статей с Шедевра, в основном про перевод игр **https://vk.cc/61On1Y**

Несколько статей про перевод игр и графику **https://vk.cc/61OquJ**

Несколько статей от Griever **https://vk.cc/61OrlX**

Архитектура Денди **https://vk.cc/1PT0RF**

Разное **https://vk.cc/61Oqxb**

Разное **https://vk.cc/61OrrU**

Программы для перевода **https://vk.cc/61Ordk**

# Адепт

Статьи уровня адепт.

# Расширение рома (Ti)

Как расширить ром и переключать банки.

Из чего состоит ром:
из заголовка iNes в 16 байт нужного для эмуляторов, из PRG-rom и CHR-rom(если есть).

В заголовке указано: номер маппера, количество банков prg и chr.

Узнаём эти параметры с помощью эмулятора FCEUX 2.2.2.
После открытия рома: help->message log.

Например ром Chip and Dale.
Нужная нам информация:
PRG ROM: 8 x 16KiB
CHR ROM: 16 x 8KiB
Mapper name: MMC1

Значит имеем 8x16=128 Кб PRG (часть с кодом и данным) и 16x8=128 Кб CHR (часть с графикой - тайлами).
Итого получается 131072+131072+16 = 262160 байт Ром.

Расширить CHR-ROM: (мапперы MMC1, MMC3)
Нам понадобится программа WinHex.
1) Количество chr-банков указано в роме по адресу (offset) 000005.
Его нужно удвоить, то есть меняем 10->20; 20->40, 08->10 (числа указаны в hex)
Максимальный размер CHR для MMC1=128 Кб, для MMC3=256 Кб.
*Ром из примера расширить нельзя, так как уже максимум. Так что надо искать этот же ром переделанный под MMC3.
Большое кол-во таких переделок я видел на сайте raregame.ru ; также можно в good и non-good сетах поискать.

Тогда пока используем другой ром для примера - Ninja Cats.
Параметры такие же, только маппер mmc3:
PRG ROM: 8 x 16KiB
CHR ROM: 16 x 8KiB
Mapper name: MMC3

Меняем используя WinHex: 000005: 10->20
Мы указали вдвое большее количество банков.

2) Теперь доклеим их к рому, увеличив размер файла.
Поскольку увеличивали с 128 до 256 Кб, надо доклеить 128 Кб.
Переходим в WinHex в концу файла (CTRL+END); Нажимаем Edit-paste zero bytes.
Append to the end of file? Нажимаем - yes.
Но значение надо в байтах указывать, 128кб это 131072 байта, указываем нажимаем OK и сохраняем.

После этого перезапускаем ром для проверки:
Должно получиться: CHR ROM: 32 x 8KiB
А размер рома - 384 Кб.

Расширить PRG-ROM: (мапперы MMC1, MMC3, UNROM)
1) Всё почти также как и при расширении CHR, только offset = 000004.
Максимальные размеры PRG для MMC1=256 Кб; для MMC3=512 Кб; для UNROM=256 Кб, но можно и более в зависимости от эмулятора.

Меняем используя WinHex:  000005: 08->10 (Указали вдвое больше количество банков).
2) Теперь надо добавить prg-банков.
Тут вначале разберемся как разбивается PRG-ROM на банки:
Если у нас 128 кб (8 по 16кб), то в общем случае это выглядит так:

--------- rom: 0x10     Cpu: $8000-$BFFF
BANK0
--------- rom: 0x4010     Cpu: $8000-$BFFF
BANK1
--------- rom: 0x8010     Cpu: $8000-$BFFF
BANK2
--------- rom: 0xC010     Cpu: $8000-$BFFF
BANK3
--------- rom: 0x10010     Cpu: $8000-$BFFF
BANK4
--------- rom: 0x14010     Cpu: $8000-$BFFF
BANK5
--------- rom: 0x18010     Cpu: $8000-$BFFF
BANK6
--------- rom: 0x1c010     Cpu: $C000-$FFFF
BANK7
--------- rom: 0x20010    конец PRG рома  (Далее пойдет Chr часть рома (если есть))

Банки 0-6 называются подключаемыми, а 7 - постоянным.
(В нём находится большинство кода, и код подключающий банки 0-6 к адресам $8000-$BFFF).

Последний банк находится всегда в конце PRG-ROM'а, поэтому расширение рома производится перед ним:
В WinHex идём к адресу (position-go to offset) 1c010.
Нажимаем Edit-paste zero bytes. Поскольку увеличивали с 128 до 256 Кб, опять же вставляем ещё 128кб (131072 байта).
В примере chip n dale - получим 384 Кб (так как prg мы там не расширяли).
В ninja cats - получим 512 Кб (и prg и chr увеличили в 2раза).

После этого перезапускаем ром для проверки:
PRG ROM:   16 x 16KiB
CHR ROM:   32 x  8KiB

Результат после расширения с 128 до 256 Кб PRG:

--------- rom: 0x10     Cpu: $8000-$BFFF
BANK0
--------- rom: 0x4010     Cpu: $8000-$BFFF
BANK1
--------- rom: 0x8010     Cpu: $8000-$BFFF
BANK2
--------- rom: 0xC010     Cpu: $8000-$BFFF
BANK3
--------- rom: 0x10010     Cpu: $8000-$BFFF
BANK4
--------- rom: 0x14010     Cpu: $8000-$BFFF
BANK5
--------- rom: 0x18010     Cpu: $8000-$BFFF
BANK6
--------- rom: 0x1c010     Cpu: $8000-$BFFF
BANK7                      ; новый пустой банк

--------- rom: 0x20010  Cpu: $8000-$BFFF
BANK8                 ; новый пустой банк
--------- rom: 0x24010  Cpu: $8000-$BFFF
BANK9                 ; новый пустой банк
--------- rom: 0x28010  Cpu: $8000-$BFFF
BANKA                 ; новый пустой банк
--------- rom: 0x2C010  Cpu: $8000-$BFFF
BANKB                 ; новый пустой банк
--------- rom: 0x30010  Cpu: $8000-$BFFF
BANKC                 ; новый пустой банк
--------- rom: 0x34010  Cpu: $8000-$BFFF
BANKD                 ; новый пустой банк
--------- rom: 0x38010  Cpu: $8000-$BFFF
BANKE                 ; новый пустой банк
--------- rom: 0x3c010  Cpu: $C000-$FFFF
BANKF                 ; банк #7 переместился в конец PRG, и теперь он #$0F
--------- rom: 0x40010  конец PRG рома

Далее идёт Chr часть рома (если есть).

Никак вмешательств в код рома не требуется, поскольку оригинальные банки остались на своих местах, а единственный банк который 'сдвинулся' внутри рома - непереключаемый.

Соответственно если надо будет расширять с 256Кб PRG до 512Кб PRG, то адрес вставки будет 0x3c010, а размер - 262144 байта.

Переключение банков:
В большинстве игр уже предусмотрена процедура подключения банков, надо только её найти.

UNROM (только prg)
Чтобы переключить банк на этом маппере надо записать его номер в ром-память. ($8000-$FFFF)
Пример:
LDA #4
STA $8000
Подключили банк #04.

Но эти мапперы требуют, чтобы записываемое число совпало с числом в роме (для избежания т.н. bus conflicts).
Поэтому код выглядит обычно вот так, пример duck tales 2:

```
; =============== S U B    R O U T    I N E =======================================

; A=bankID

set_bank:                 ; ...
        CMP     current_bank
        BEQ     locret_5400_D39B
        STA     current_bank
; End of function set_bank


; =============== S U B    R O U T    I N E =======================================

; A=bank ID
```

```
bank_set_nmi:                   ; ...
        TAY
        STA     bank_tbls,Y

locret_5400_D39B:               ; ...
        RTS
; End of function bank_set_nmi

; ---------------------------------------------------------------------------
bank_tbls:      .BYTE 0, 1, 2, 3, 4, 5,     6, 7 ; ...
```

Указывается номер банка в A. Далее вызывается функция его подключения:

```
 LDA     #4
 JSR     set_bank
```

В ней A копируется в Y, и далее вписывается в таблицу. Таблица содержит номера 0,1,2,3 и т.д., таким образом номер записываемого числа (банка) совпадет с числом в роме.
При расширении рома для корректной работы на всех эмуляторах и железе эту таблицу также надо дописать (При 16 банках - до 0F). Возможно придётся переписать её в другом месте, если место после занято.
Также в этой функции банк дополнительно записывается в current_bank. (текущий номер банка).
Это необходимо для сохранения банка во время обработки кода в nmi. (но некоторые игры содержат заранее
пронумерованные банки по адресам $BFFF или $8000, возможно эти номера придется прописать для добавленных банков).

Исходя из кода выше найти эту процедуру включения банка легко - вбив в winhex в поиске 00010203040506.
(не забываем что может встретиться подобное, но не быть процедурой подключения банка, просто совпадание, так что
проверяем если более 1 раза найдется).

Далее если в новом банке будет именно код, может быть прыжок туда:

```
 LDA    #7
 JSR     set_bank
 JSR    $8000     ; начало банка 7, куда будем добавлять новый код.

     ; может быть также необходимо восстановить старый банк,который был до этого.
 LDA    #4
 JSR    set_bank
```

MMC1 - PRG:
Этот маппер имеет не очень удобную процедуру подключения банка. Но найти её будет не слишком сложно.
Номер банка записывается не разом, а побитно 5 раз.  (Запись в адреса рома $E000-$FFFF)

Вот простейший пример:

```
        LDA     #3     ; номер банка
        STA     $FFF0     ; записываем
        LSR     A
        STA     $FFF0     ; записываем
        LSR     A
        STA     $FFF0     ; записываем
```

```
        LSR     A
        STA     $FFF0     ; записываем
        LSR     A
        STA     $FFF0     ; записываем
```

А Вот пример процедуры смены банка из игры Chip n Dale 2:

```
; =============== S U B     R O U T     I N E =======================================


change_prg_bank:                ; ...
        PHP
        CMP     current_bank
        BEQ     loc_5400_C2E3
        STA     current_bank
        INC     bank_changing_flag
        STA     $EFFF
        LSR     A
        STA     $EFFF
        LSR     A
        STA     $EFFF
        LSR     A
        STA     $EFFF
        LSR     A
        STA     $EFFF
        DEC     bank_changing_flag

loc_5400_C2E3:                  ; ...
        PLP
        RTS
; End of function change_prg_bank
```

То есть чтобы включить нужный добавленный PRG-банк, указываем его номер, и вызываем готовую функцию
подключения банка:

```
 LDA     #7     ; номер-банка-7       ; hex - A9 07
 JSR     $C298     ; change_prg_bank ; hex - 20 98 C2
```

MMC1 - CHR:

Подключение CHR похоже на PRG, только Адреса рома куда записывать номера - $B000-$BFFF, и $D000-$DFFF.

Так что в случае MMC1 всё сводится к нахождению готовых процедур и изучения их работы.

Пример опять же Chip n Dale2, функция указывает chr-банки. Номера в X и Y.

```
; =============== S U B     R O U T     I N E =======================================


change_chr_bank:                ; ...
        INC     bank_changing_flag
        TXA
        STA     $BFFF
        LSR     A
        STA     $BFFF
        LSR     A
```

```
        STA     $BFFF
        LSR     A
        STA     $BFFF
        LSR     A
        STA     $BFFF
        TYA
        STA     $DFFF
        LSR     A
        STA     $DFFF
        LSR     A
        STA     $DFFF
        LSR     A
        STA     $DFFF
        LSR     A
        STA     $DFFF
        DEC     bank_changing_flag
        RTS
; End of function change_chr_bank
```

Но если посмотреть код дальше, то номера X и Y для этой функции считываются из адресов RAM 1f и 20.
То есть в случае с CHR никакого вмешательства скорее всего не потребуется.
Достаточно будет указать только новый номер в том месте, где задавались номера блоков графики, но уже
из новых добавленных банков, а игра всё сделает за вас.

MMC3 - PRG и CHR.
В этом маппере вначале указываем что меняем, а затем его номер.
В $8000 - записываем что менять (регистр), $8001 - номер банка (данные).
Регистры 0-1 - переключают 2 блока графики фона по 4кб каждый.

```
LDA #0          ; меняем блок графики фона - верхний кусок в 4 Кб.
STA $8000
LDA #34     ; номер - 34  (при 256Кб всего 128 номеров по 2кб).
STA $8001

LDA #1          ; меняем блок графики фона - нижний кусок в 4 Кб.
STA $8000
LDA #36     ;
STA $8001
```

Регистры 2-5 - переключают 4 блока графики спрайтов по 2кб каждый.

```
LDA #2          ; меняем блок графики спрайтов - первый кусок в 2кб.
STA $8000
LDA #44     ; номер
STA $8001

LDA #3          ; меняем блок графики спрайтов - второй кусок в 2кб.
STA $8000
LDA #49     ; номер
STA $8001

LDA #4          ; меняем блок графики спрайтов - третий кусок в 2кб.
STA $8000
LDA #15     ; номер
```

```
 STA $8001

 LDA #5         ; меняем блок графики спрайтов - четвёртый кусок в 2кб.
 STA $8000
 LDA #33     ; номер
 STA $8001
```

Регистры 6-7 - переключают 2 банка PRG.
То есть у MMC3 на самом деле 2 подключаемых банка - один $8000-$9FFF; и второй $A000-$BFFF.
Поэтому номер PRG указывает не для 16Кб блоков а для 8Кб блоков.
(7-ой для MMC1/UNROM, станет парой $0E и $0F для mmc3)

```
 LDA #6         ; меняем prg-bank #1  (cpu: $8000-$9FFF )
 STA $8000
 LDA #$0E     ; номер
 STA $8001

 LDA #7         ; меняем prg-bank #2 (cpu: $A000-$BFFF )
 STA $8000
 LDA #$0F     ; номер
 STA $8001
```

Также в связи с этим есть такой момент, что один и тот же банк может быть подключен как в $8000-$9FFF, так и
$A000-$BFFF.
Многие игры этим пользуются, что может усложнять хакинг.
Также могут использовать другие режимы и положение банков в роме.(вначале $A000-$BFFF, а потом $8000-$9FFF)
Могут и режим спрайтов/фона поменять (4 сменных куска фона, и 2 спрайтов) - но такое редко.

Другое дело что бывают ромы где фискированным банком является $8000-$BFFF;  а $C000-$FFFF - переключаемым.

Из-за этого некоторые игры могут потребовать дополнительные корректировки при расширении (пример TMNT3,TMNT-Tournament Fighters)
(Для начала попробовать не добавлять нулевые байты, а продублировать целиком prg-банк с copy-paste, либо
скопировать какие-то отдельные банки на место новых добавленных)

Использовать процедуры смены CHR-банка, опять же врядли понадобится,
а вот примеры для переключения PRG из игры Ninja Cats:

```
; =============== S U B     R O U T     I N E =======================================


set_4k_prg_bank:              ; set 2 banks
        STX     prg1_id
        LDY     #6
        STY     last_bank_register
        STY     $8000
        STX     $8001
        INX
        STX     prg2_id
        INY             ; 6->7
        STY     last_bank_register
        STY     $8000
```

```
        STX     $8001
        RTS
; End of function set_4k_prg_bank


; =============== S U B     R O U T     I N E =======================================


set_2k_prg_bank:                ; ...
        STX     prg1_id
        LDY     #6
        STY     last_bank_register
        STY     $8000
        STX     $8001
        RTS
; End of function set_2k_prg_bank

; =============== S U B     R O U T     I N E =======================================


set_2k_prg_bank2:                   ; ...
        LDX     #7
        STX     last_bank_register
        STX     $8000
        STA     $8001
        STA     prg2_id
        RTS
; End of function set_2k_prg_bank2
```

Так что опять же всё сводится к поиску уже готовых процедур смены банков в роме.
В этом варианте , чтобы их найти смотрим в роме записи подобного вида:

```
STA     $8000  ; hex - $8D 00 $80
STA     $8001  ; hex - $8D 01 $80
STX     $8000  ; hex - $8E 00 $80
STX     $8001  ; hex - $8E 01 $80
STY     $8000  ; hex - $8C 00 $80
STY     $8001  ; hex - $8C 01 $80
```

Более подробную информацию можно прочитать на http://wiki.nesdev.com/w/index.php/MMC3

Если пригодилось ждите новых статей:
1) как быстро найти звуковой движок в роме / как открыть ром в IDA.
2) как использовать symbolic debug в fceux debugger, автоматически полученный из asm6_sonder.

Если ничего не получилось, отписывайтесь что именно не ясно.

Автор статьи - Ti (c) 2014.

## Эксперт

Статьи уровня эксперт.

# Мастер

Статьи уровня мастер.

## Контакты


Ромхакер и автор учебника **BZK**
https://vk.com/id234423097

Тема на форуме
**http://www.emu-land.net/forum/index.php?topic=76665**