ファミコンゲームプログラミング

サークル 2P
https://2p.netlify.app

改版履歴

rev.3	2021/07/16	改版履歴ページを追加 誤字脱字を修正 目次のリンク漏れを修正 動作環境を更新 参考文献を更新 いくつか補足説明を追加
rev.2	2019/10/08	動作環境修正
rev.1	2019/10/01	初版

はじめに

本書をご購入いただきありがとうございます。本書はファミコンのゲーム開発をやってみたい方、またはそこまでいかなくてもどんなものか興味のある方に向けて基礎から若干の応用までを説明しています。具体的には2進数とはなんぞやからラスタースクロールのやり方くらいまでを説明します。

開発はファミコンのエミュレータで動作するROMファイル(.nesファイル)を作るところまでを目的としています。物理カセットの作り方は分野が異なるので本書では触れませんが、実機で動かす方法については少し紹介します。

本書を読むにあたって必要な前提知識はあまりありませんが、CPUやメモリなどのコンピュータの基本知識や、プログラミングの経験がまったくないと少々つらいかもしれないです。

動作環境

WindowsまたはmacOSが必要です。
macOSは10.14までならwineを使ってWindows系のツールを実行できます。
macOSの11以降は仮想化ソフトのVMwareが無償で提供されているのでVMwareにWindowsをインストールして使うことができます。Windowsはマイクロソフトのページでイメージが無償公開されています。ライセンス認証をしなくてもファミコン開発で困ることはないです。

著者について

81年生まれのファミコン世代です。子供の頃はゲームが作りたくて大学も情報工学科に入ったのに結局作ってないダメな人です。そんな自分にとってもこの本が新たな一歩となれば良いなと思います。

目次

改版履歴.....2
はじめに.....3
1 章　コンピュータの数値表現.....5
2 章　ファミコンのハードウェア.....7
3 章　開発環境の準備.....13
4 章　ファミコンゲームプログラミング.....17
5 章　プログラミングのいろは.....25
6 章　パレット.....30
7 章　スプライト.....35
8 章　ゲームコントローラーによる入力.....42
9 章　BG (背景).....45
10 章　ゲームレイアウト.....53
11 章　8 ビットを越える計算.....58
12 章　スコアの表示.....67
13 章　種々のテーブル.....75
14 章　PPU 詳細.....81
15 章　サウンドエンジン.....91
16 章　CNROM (マッパー 3).....97
17 章　ラスタースクロール.....101
ボーナス 1　横スクロール.....105
ボーナス 2　MMC1.....115
ボーナス 3　デモゲーム解説.....121
ボーナス 4　開発よもやま話.....141
参考文献.....144
おわりに.....145

1章　コンピュータの数値表現

この章ではコンピュータで数値を扱う場合に避けては通れない2進数と16進数の説明をします。とりあえず最初は10進数以外にも2つの数値表現方法があるのだなという程度の理解で大丈夫です。読み進めてわからなかったらまたここに戻ってみてください。

10進数

2進数の話をする前にまず10進数の話をします。10進数というのは普段生活で使っている数です。10ごとに位が進むので10進数と言います。0から始まり9の次は10です。さらに11、12、、と続き19の次は20です。99の次は100です。馴染み深いですね。

普段は意識しないと思いますが、この10進数は下記のように分解できます。例えば254の場合、

254 =（2 ＊ 10の2乗）+（5 ＊ 10の1乗）+（4 ＊ 10の0乗）

なんでこんな話をするかというと以降の説明が円滑になるからです。

2進数

ファミコンに限らずコンピュータの世界では数値は2進数で表されます。2ごとに位が進むので2進数と言います。0から始まり1の次は10です。10の次は11でその次は100です。さらに
101、110、111、1000と進みます。

2進数の数値を人間がどうやって解釈すれば良いかというと、上でやったような分解がそのまま適用できます。例えば1011という2進数は下記のようになります。

（1 ＊ 2の3乗）+（0 ＊ 2の2乗）+（1 ＊ 2の1乗）+（1 ＊ 2の0乗）

これは10進数だと11ですね。少し慣れが必要ですが、人間はこのようにして2進数を10進数に解釈することができます。

この2進数ひと桁あたりをコンピュータの世界では1ビットと呼びます。例えば0101は4ビットの数値です。また8ビットのことを1バイトという単位で呼びます。

ファミコンは8ビットマシンなどとも呼ばれますが、扱う数のほとんどが8ビットだからです。8ビットの2進数の最大値は11111111で、これは10進数にすると255です。ドラクエなどで馴染みのある最大値ですね。

なお1バイトの2進数(例:10011011)は左端の桁から順にビット7、ビット6、ビット5というふうに呼び、右端はビット0となります。よく出てくるので覚えておいてください。また、コンピュータの世界では1ではなく0から数えるのがなかば常識ですので、併せて覚えておいてください。

16進数

最後に16進数です。もうお察しかと思いますが、16ごとに位が進むので16進数と言います。0からはじまり9の次はAです。さらにB、C、D、E、Fと続き、Fの次は10です。1Fの次は20で、FFの次は100です。

16進数を書くときはそれが16進数だとわかりやすいように数値の前に0xをつける習わしがあります。xは16進数を意味する英単語hexadecimalの略です。

16進数の数値を人間が解釈するにはまた同じような計算をします。例えば0x26という16進数は下記のようになります。

（2 * 16の1乗）+（6 * 16の0乗）

これは10進数で38ですね。これもやってれば慣れます。

コンピュータの内部では2進数を使うと言ったのになぜ16進数を覚える必要があるのかという話ですが、16進数は2進数の数値をまとめて表現するのに都合が良いのです。

2進数は8ビットともなると数値が8つも並ぶので見辛いです。16進数は1桁で2進数4桁をまとめて表現できます。これだと8ビット(1バイト)も2桁となり見た目がスッキリします。いくつか例を挙げておきます。

00000000（2進数）= 0x00（16進数）
00100101（2進数）= 0x25（16進数）
10001100（2進数）= 0x8B（16進数）
11111111（2進数）= 0xFF（16進数）

ファミコンプログラミングにおける表記方法

すぐ上で16進数は一般的に0xをつけて表記すると説明したばかりですが、実はファミコンプログラミングでは0xではなく$を使います。同様に2進数には%を使います。10進数には何もつけません。いくつか例を挙げます。

  0（10進数）= %00000000（2進数）= $00（16進数）
 64（10進数）= %01000000（2進数）= $40（16進数）
128（10進数）= %10000000（2進数）= $80（16進数）
255（10進数）= %11111111（2進数）= $FF（16進数）

本書でも今後はこの表記を使っていきます。

コンピュータの容量単位

コンピュータではメモリなどの容量を表現するのに以下の単位をよく使います。

 1B（1バイト）　　：8ビットのことです
1KB（1キロバイト）：1024Bのことです
1MB（1メガバイト）：1024KBのことです
1GB（1ギガバイト）：1024MBのことです

ファミコンの時代にはカセットのROM容量はバイトではなくてビットで表記されていました。1バイトは8ビットなのでなかなかハッタリの効いた数字になります。ネオジオの100メガショックは100メガビットショックです。バイトにすると12.5MBですね。

2章　ファミコンのハードウェア

この章ではファミコンのハードウェアについて説明します。

ファミコンのハードウェア構成

ファミコン本体はだいたい以下のハードウェアで構成されています。

CPU (central processing unit)

メインプロセッサです。MOS 6502というアップル社の古いコンピュータ(Apple II)に採用されていた8ビットCPUをカスタムしたものです。なので資料を探す場合は6502で検索するとだいたいの場合はこと足ります。

APU (audio processing unit)

音を出すためのサウンドチップです。物理的にはCPUの中に統合されています。

RAM

CPUに付属するメモリです。様々な値を保存するのに使います。容量は2KBです。パソコンがメインメモリ16GBとか搭載している現代では考えられない容量の小ささですね。本書で単にRAMまたはCPU RAMと記述してある場合はこれのことを指します。

PPU (picture processing unit)

画面に絵を出すためのグラフィックチップです。これはCPUとは物理的に独立しています。

PPU RAM (VRAM)

PPU付属のメモリです。BGの描画情報とパレットを保存します。容量は2KBです。これも現代では考えられない容量の小ささですね。1画面あたり1KB利用するので、これは2画面分の容量になります。

OAM (object attribute memory)

PPU付属のメモリです。スプライトの描画情報を保存します。容量は256バイトです。スプライト1つあたり4バイトで管理するので最大64スプライトを管理できます。

カートリッジのハードウェア構成

ファミコンカセットはだいたい以下のハードウェアで構成されています。

PRG

プログラムメモリです。ゲームのコードやサウンドデータなどを格納しているチップです。

CHR

キャラクタメモリです。グラフィックデータを格納しているチップです。

WRAM (work RAM)

拡張 RAM です。通電されていればリセットボタンを押してもデータは残ります。いわゆるバッテリーバックアップに使われます。ゲームによってついてたりついてなかったりします。

CPU 概要

CPU アドレス空間

ファミコンは CPU としてカスタムした 6502 を搭載しています。この CPU は 16 ビットのアドレスバスを持ち、$0000 から$FFFF の 64KB 範囲のアドレスにアクセス可能です。

CPU アドレスは様々なハードウェアに対応しています。このアドレスとハードウェアの対応をマッピングと呼び、マッピングをまとめたものをアドレス空間と呼びます。

CPU のアドレス空間は以下のようになっています。

$0000 - $07FF : 2KB の CPU RAM

$2000 - $2007 : PPU との IO(入出力)に利用
$4000 - $4017 : APU との IO、コントローラーとの IO

$6000 - $7FFF : 8KB のカートリッジ WRAM
$8000 - $FFF9 : 32KB のカートリッジ PRG
$FFFA - $FFFF : 割り込みハンドラのアドレス（NMI、RESET、IRQ）

例えば RAM にアクセスするにはアドレス$0000 から$07FF の範囲にアクセスし、1 コンの入力情報を知りたければアドレス$4016 にアクセスする、といった具合です。それぞれの詳細は各章で説明しますのでここで暗記する必要はありません。

マッパーとバンクスイッチ

CPU アドレス空間のマッピング対応を見ると、カートリッジ PRG にアクセスするためのアドレスの範囲は 32KB となっています。ビットにすると 256K ビットになります。1 バイトは 8 ビットでしたね。

初期のゲームはこれでも足りていました。しかし中期以降は 4M ビット以上のゲームを見たかと思います。星のカービィは 6M ビットですし、最大容量で有名なメタルスレイダーグローリーは 8M ビットです。全然足りませんね。そういったソフトはどうやっていたのでしょうか。

実はカートリッジには色々種類があって、これはマッパーなどと呼ばれます。マッパーの中には大容量 ROM を扱うためのバンクスイッチという機能を持つものが存在します。

バンクスイッチはこの 32KB のアドレス範囲にマッピングする PRG の範囲を切り替える機能です。例えば 64KB の PRG を持つゲームで、前半の 32KB を使うか、後半の 32KB を使うか、というのを切り替えることができます。この機能で 32KB 以上のゲームに対応します。切り替え方法はマッパーによって異なります。

マッパーには NROM だとか MMC3 だとかそれぞれ名前があったりしますが、番号で表されることもあります。例えば CNROM はマッパー 3 といった具合です。

どのタイトルがどのマッパーだったかというのは、下記のサイトにリストがあります。

http://tuxnes.sourceforge.net/nesmapper.txt

PPU 概要

PPU は画面にグラフィックの描画を行うチップです。CPU とは独立して非同期に動作します。PPU は内部に独自のメモリ(PPU RAM、OAM)を持っており、カラーパレット、BG(背景)、スプライトの情報を保持します。

PPU はテレビのスキャンライン単位で描画します。1 ラインには 8 つのスプライトまで表示でき、これを超えると無視されます(表示されない)。くにお君のサッカーなどでよくキャラが点滅してたと思いますが、あれはこの制限によるものです。横にスプライトが多く並んだときは一部が消えて、移動によってずれると表示するを繰り返すので点滅して見えます。

すべてのラインを画面に描画すると、次の画面の描画まで VBlank と呼ばれる空白の時間に入ります。通常はこの VBlank の間に次に描画する画面を用意する必要があります。VBlank 以外の時間、つまり画面の描画中に画面の更新処理をすると表示がおかしくなったりします。

画面解像度は横 256x 縦 240 ピクセルです。しかし NTSC のテレビでは上下の 8 ピクセルはカットされてしまうので、256x224 ピクセルになります。テレビによってはさらに追加で最大 8 ピクセル程度カットするものがあるらしいので、重要な情報はその領域に描画しないようにしましょう。

PPU アドレス空間

PPU にも CPU と同じくアドレス空間があり、以下のような対応になっています。

$0000 - $0FFF : 4KB の CHR パターンテーブル 0
$1000 - $1FFF : 4KB の CHR パターンテーブル 1

$2000 - $23BF : 960 バイトのネームテーブル 0
$23C0 - $23FF : 64 バイトの属性テーブル 0

$2400 - $27BF : 960 バイトのネームテーブル 1
$27C0 - $27FF : 64 バイトの属性テーブル 1

$2800 - $2BBF : 960 バイトのネームテーブル 2
$2BC0 - $2BFF : 64 バイトの属性テーブル 2

$2C00 - $2FBF : 960 バイトのネームテーブル 3
$2FC0 - $2FFF : 64 バイトの属性テーブル 3

$3F00 - $3F0F : 16 バイトの BG パレット
$3F10 - $3F1F : 16 バイトのスプライトパレット

パターンテーブルというのはカートリッジの CHR です。ネームテーブルと属性テーブルは PPU RAM(VRAM)になります。パレットも PPU RAM(VRAM)になります。

グラフィックシステム

ここではファミコンのグラフィック関連の要素について説明します。詳細はまたそれぞれの章で説明しますので、ここでは概要だけを覚えてもらえば良いです。

タイル

グラフィックの最小単位です。1タイルは横8x縦8ピクセルのドット絵です。1ドットあたり2ビットのカラーインデックスという情報を持ち、1タイルは16バイトです。

カラーインデックスは色そのものの情報ではなくて、パレット内の指定になります。カートリッジのCHRはこのタイルデータの集合体です。

BG（背景）

背景のグラフィックです。BGは1画面あたり横32x縦30のタイルで構成されます。PPU RAMは2画面分のBGの情報を保存できます。スクロールもできます。

スプライト

画面内の好きな位置に配置できるグラフィックです。1スプライトあたり1タイルです。スプライトは位置情報などを保持したタイルとも言えます。PPUはスプライトの情報をOAMという256バイトのメモリに保持し、最大で同時に64スプライトまで扱えます。

1キャラに1スプライトでは小さすぎるので大抵は複数のスプライトを使って1キャラを表現します。例えばちびマリオは16x16ピクセルですが、これはスプライト4つで表現されています。

パターンテーブル

カートリッジのCHRはPPUアドレスの$0000から$1FFFの8KB範囲にマッピングされます。この範囲をパターンテーブルと呼びます。画面に表示するタイルはこの範囲に存在する必要があります。

パターンテーブルはアドレス$0000-$0FFFと$1000-$1FFFで2つに分けて使います。片方にBG用CHRデータを、もう片方にスプライト用CHRデータをマッピングします。CHRデータはタイルの集合です。

CPUアドレス空間のカートリッジPRG領域の32KBもそうでしたが、グラフィックデータの領域が8KB(64Kビット)というのはどうも小さい気がしますね。PRGと同じく大容量のゲームではマッパーがCHRのバンクスイッチを提供していました。

パレット

パレットは色の情報を保持します。タイルは色そのものの情報は実は持っておらず、代わりに 1 ドットごとにカラーインデックスという番号を保持しています。パレット内でこの番号に対応する色をつけて初めて色のついたグラフィックとなります。

つまり同じタイルでもパレットの内容によって色が代わります。パレットはゲーム中に変更することができます。ロックマンとかが分かりやすいですね。ロックマンでは武器を変更するとアイテムの色が変わりますが、あれは同じタイルでパレットだけを変更しています。

ネームテーブル

BG を構成する 32x30 のタイルの番号を羅列したものです。BG 用のパターンテーブルにある何番目のタイルかという番号で埋まっています。ネームテーブル 1 つで 1 画面分の BG が表現されます。

PPU のアドレス空間を見るとネームテーブルが 4 つありますが、2 つはミラー(コピー)です。ファミコンは 2 画面分の PPU RAM しか持ちません。

属性テーブル

BG に使うパレットの指定を羅列したものです。ネームテーブルとセットで使います。

3章　開発環境の準備

ファミコンソフトの開発には最低でも3つのソフトが必要です。アセンブラとエミュレータとテキストエディタです。

アセンブラはソースコードをROMファイル(.nesファイル)に翻訳するものです。エミュレータはアセンブラが生成したROMファイルを読み込んでゲームを動作させます。また、この翻訳のことを専門用語ではアセンブルと呼びます。

ソースコードは人間が読める形でゲームのプログラムを記述したテキストファイルです。本書で主に説明するのがまさにこのソースコードの書き方となります。ソースコードの編集にはテキストエディタを用います。

本章ではこのアセンブラとエミュレータの準備と、アセンブラを使う上で必要となるコマンドラインの知識について説明します。

ファミコン開発に利用できるアセンブラ

ファミコン開発用のアセンブラはいくつかあるのですが、本書ではnesasmというアセンブラを使って説明していきます。

nesasmは以下から入手できます。

Mac用
https://github.com/camsaul/nesasm/archive/master.zip

Windows用
http://www.nespowerpak.com/nesasm/NESASM3.zip

Macの場合はダウンロード後に実行ファイルをビルドする必要があります。面倒だと思うので本書に添付しているサンプルにはあらかじめMac用とWindows用の実行ファイルを含めてあります。

コマンドライン

nesasmはいわゆるコマンドラインで実行するプログラムです。Macの場合はターミナル、Windowsの場合はコマンドプロンプトを起動してそこから実行します。以下ではコマンドラインで使う簡単なコマンドとnesasmの実行方法までを説明します。

コマンドラインの起動方法

コマンドラインの起動方法は以下の通りです。

Macの場合
Finderを開き、アプリケーション => ユーティリティ => ターミナルと辿って起動

Windowsの場合
検索ボックス(Windows 10の場合は画面左下)でコマンドプロンプトと検索して起動

コマンドラインで最低限知っておきたいコマンド

コマンドラインではクリックなどではなく、文字通りコマンドを打ち込んでプログラムを実行します。ここではコマンドラインで最低限知っておいて欲しいコマンドを紹介します。てきとうに叩いてみて感覚をつかんでください。

カレントディレクトリの表示

Mac の場合 : pwd
Windows の場合 : dir

カレントディレクトリというのは現在地のことです。コマンドプロンプト実行後は通常自分がログインしているユーザのホームディレクトリにいると思います。上記のコマンドを叩くと現在地がパスとして表示されるはずです。

hoge ユーザとしてログインしている場合、ホームディレクトリのパスは、
Mac なら/Users/hoge、Windows なら￥Users￥hoge あたりになると思います。

カレントディレクトリにあるファイルの表示

Mac の場合 : ls -l
Windows の場合 : dir

カレントディレクトリにあるファイルは上記のコマンドで確認できます。

ディレクトリの移動

Mac の場合 : cd /Users/admin/Desktop
Windows の場合 : cd ￥Users￥admin￥Desktop

cd コマンドはパラメータとして移動先をパスで与えます。上記は admin ユーザのデスクトップに移動する場合の例です。

ファイルのコピー

Mac の場合 : cp nesasm /Users/admin/Desktop
Windows の場合 : copy NESASM3.exe ￥Users￥admin￥Desktop

cp または copy コマンドはパラメータを 2 つとります。コピーするファイルと、コピー先のディレクトリをそれぞれパスで指定します。上記はカレントディレクトリにある nesasm ファイルを admin ユーザのデスクトップにコピーしています。

上記のコマンドを覚えておけばとりあえず OK です。

nesasm の実行

nesasm の実行方法は簡単です。nesasm の実行ファイルがあるパスに移動して実行するだけです。以下は本書の zip ファイルを hoge ユーザのデスクトップに解凍した場合の例です。

Mac の場合
```
cd /Users/hoge/Desktop/fcprogramming/src/04/
./nesasm
```

Windows の場合
```
cd ￥Users￥hoge￥Desktop￥fcprogramming￥src￥04￥
NESASM3.exe
```

上記を実行すると以下のような出力がされると思います。

```
NES Assembler (v3.1)

nesasm [-options] [-? (for help)] infile

-s/S   : show segment usage
-l #   : listing file output level (0-3)
-m     : force macro expansion in listing
-raw   : prevent adding a ROM header
-srec  : create a Motorola S-record file
infile : file to be assembled
```

これは nesasm コマンドの使い方に関するメッセージです。このメッセージが示すように実際に使う場合はパラメータとしてソースコードを指定する必要があります。

アセンブラについてはここまでできれば OK です。

ファミコン開発に利用するエミュレータ

実行環境となるエミュレータについてですが、動かすだけならなんでも良いです。ただ開発の場合はデバッグ機能がついていると便利です。本書では fceux というエミュレータを利用していますが、最近では Mesen というエミュレータの人気が強いようです。

fceux は以下の URL より入手できます。

http://www.fceux.com/web/home.html

fceux は Windows 用のグラフィカルなツールなので Windows の場合はダブルクリックで起動できます。Mac の場合は 10.14 までなら wine というソフトを使って実行できます。

wine のインストール方法は”wine mac インストール”とかで検索してください。インストールできたならコマンドラインで以下のように叩けば起動できます。cd の移動先は fceux を保存したディレクトリを指定してください。

```
cd /Users/hoge/Desktop/fceux-2.2.3-win32/
wine fceux.exe
```

ファミコン開発に利用するテキストエディタ

ソースコードの編集に使うテキストエディタですが、特にこれを使えというのはありません。普段使い慣れているエディタを使用してください。筆者はSublime Textというエディタを愛用していますが、昨今はVisual Studio Codeが人気のようです。

4章　ファミコンゲームプログラミング

人間が読めるレベルでプログラムが記述されたテキストファイルをソースコードと呼びます。ソースコードを nesasm に翻訳させて ROM ファイルを生成します。

ソースコードは CPU(6502)用のアセンブラ言語で記述します。アセンブラ言語とは CPU 固有のプログラミング言語です。本章ではこのソースコードの記述方法について説明します。

nesasm 向けのソースコードに出てくる要素

まずはソースコード内にどのような要素が登場するかを先に説明しておきます。

ディレクティブ

nesasm に対しての指示です。翻訳するときに参照されます。.org とか.rsset とかドットではじまります。インデント(字下げ)して記述します。

ラベル

人間がソースコードの管理をしやすくするためのものです。例えば初期化のコードを記述する場合、インデントせず行頭に init:などと書きます。ラベル名は任意ですが重複してはいけません。ラベル名のあとにコロンをつけます。

ラベルは nesasm によってアドレスに翻訳されるので、アドレスを指定する箇所で利用できます。

オペコード(命令)

CPU への命令です。lda とか sta とか jmp とか色々あります。インデントして記述します。6502 の命令はおよそ 56 種類ですが、よく使うのはそのうちの一部です。

オペランド

オペコードの引数です。引数というのはパラメータのことです。例えば lda #$FF とあったら lda がオペコードで#$FF がオペランドです。

$で始まる値は 16 進数、%で始まる値は 2 進数を表し、なにもつかない場合は 10 進数となります。さらに#で始まる値は即値と言ってそのまま数値として解釈されます。#がつかないとアドレスとして扱います。

コメント

ソースコードを人間が読みやすくするためのものです。セミコロンで始まる部分はコメントとなります。ラベルと違って翻訳時には無視されます。

ソースコードの例

ソースコードは上記で説明した要素の集合です。一部を抜き出すと以下のようになっています。

```
    .org $8000 ; アドレス$8000 から下記のコードを配置
loop:
    lda #$FF   ; A レジスタに#$FF をロード
    jmp loop   ; loop ラベルに飛ぶ。jmp 命令はオペランドにアドレスをとる。
```

上記の処理は適当に書いたのでなんとも意味不明ですが一応説明すると、lda #$FF を実行したら loop ラベルに飛ぶというもので、いわゆる無限ループです。

loop というラベルは nesasm が翻訳すると$8000 というアドレスになるので、この処理は CPU アドレスの$8000 にロードされます。

インデント(字下げ)は揃っていれば半角スペース何文字でも良いのですが、本書では上記のように一貫して半角スペース 4 文字で字下げしています。

レジスタ

上記の説明ですでに名前が少し出てますが、ファミコンプログラミングにおいてはレジスタというものを理解する必要があります。

レジスタというのはCPU付属の小さな記憶装置です。6502はいくつかレジスタを持ちますが、まず知っておいて欲しいのは４つの８ビットレジスタです。それぞれのレジスタは８ビットの値をストア(書き込み)したり、ロード(読み込み)したりできます。それぞれ説明していきます。

Aレジスタ (Accumulator)

Aはアキュムレータの略です。名前の通り計算などに使います。ほとんどの処理で使われます。

使用例

```
lda #$3A  ; $3A(即値)をAレジスタにロードする
clc
adc #$10  ; Aレジスタの値に$10(即値)を加算する。Aレジスタの値は$4Aになる。
sta $0000 ; Aレジスタの値($4A)をアドレス$0000にストアする
```

ロードは読み込み、ストアは書き込みという意味です。よく使います。

Xレジスタ (Index register X)

ループ処理などでインデックスとして使われるレジスタです。このレジスタはAレジスタと違って計算には使えません。

Yレジスタ (Index register Y)

ループ処理などでインデックスとして使われるレジスタです。このレジスタはAレジスタと違って計算には使えません。Xレジスタと基本的には同じですが、Xレジスタしか使えない、Yレジスタしか使えないというケースが若干あります。

ステータスレジスタ

最後に実行した命令の結果をフラグとして保持しています。計算結果がゼロであったかどうか、などです。色々な命令がこの状態を参照して動作します。

nesasmのソースコードの構造

nesasmで開発する場合のソースコードの構造はおよそ以下のようになります。

iNESヘッダ

ファミコンのROMファイルを作るにあたって、そのROMに関する情報(メタデータ)を記述する必要があります。これはiNESヘッダといって、ディレクティブで記述します。

以下がiNESヘッダを記述するディレクティブになります。ソースコードの一番最初に記述します。

```
.inesprg 1   ; PRGのサイズ（16KB単位）
.ineschr 1   ; CHRのサイズ（8KB単位）
.inesmap 0   ; マッパー番号
.inesmir 0   ; BGのミラーリングを横にするか縦にするか
```

バンク

nesasmはPRGコードとCHRデータを8KB単位でバンクとして管理します。上記の例のようにiNESヘッダでPRGを16KBと指定した場合、nesasmのソースコード中での記述は2バンク使うことになります。上記のROMはさらにCHRが8KBあるので合計3つのバンクを持つことになります。

バンクは下記のように.bankディレクティブで記述します。

```
.bank 0
.org $C000
; 以下8KBのPRGコードを記述

.bank 1
.org $E000
; 以下8KBのPRGコードを記述

.bank 2
.org $0000
; 以下8KBのCHRデータを記述
```

上記の例を見ると.bankディレクティブのあとに.orgディレクティブがついていますね。.orgディレクティブは、そのディレクティブ以下に記述するコードをそのアドレスから配置するという指定です。.orgはバンク中に好きなだけ記述できます。

上記ではバンク0のPRGコードはアドレス$C000からの8KB区画に、バンク1のPRGコードはアドレス$E000からの8KB区画に配置されます。2章でCPUのアドレス空間を紹介したと思いますが、このアドレス範囲($8000 - $FFF9)はカートリッジROMをマップする範囲でしたね。

バンク2はCHRデータを配置するバンクになります。nesasmではPRGのバンクに続いてCHRのバンクを記述します。ここの.orgディレクティブの指定アドレスはPPUのアドレスになります。2章でPPUのアドレス空間も紹介しましたが、PPUアドレス$0000-$1FFFの先頭8KBはパターンテーブルでしたね。

外部ファイルの取り込み

ところでCHRデータというのはファミコン専用の画像データなわけですが、通常は専用のドットエディタなどのツールを使って作ります。つまりnesasmのソースコードとは別のファイルとして管理するのが普通です。

そうした外部のバイナリファイルをソースコード内に取り込みたい場合は.incbinというディレクティブを使います。例えば以下のようにして外部ファイルを取り込めます。

```
.bank 2
.org $0000
.incbin “mygraphics.chr”
```

また、CHRデータだけでなく、ソースコードも機能別にファイルを分割して管理したいという場合があります。そういう場合は.includeというテキストファイル用のディレクティブもあります。

テキストとバイナリ

上でさらっとバイナリファイルとか言いましたが、バイナリファイルというのは基本的には人間ではなくソフトが管理するデータのことです。人間がメモ帳などで自然に読めるテキストファイル以外はバイナリファイルという認識でだいたいOKです。上記の画像データなどがそうですね。

より具体的には、バイナリファイルというのは$00から$FFまでの1バイトの値が連続して記録されているファイルのことです。nesasmでは上記の.incbinディレクティブで取り込む以外に、ソースコード内に直接バイナリデータを記述する方法があります。

.byteディレクティブと.wordディレクティブというのがそのうち出てきますが、それがソースコード内に直接バイナリを記述するためのディレクティブです。出てきたらまた説明します。

割り込み

ファミコンのCPUは3種類の割り込みを受けます。割り込みというのは、ハードウェアが特定の状況になったときにCPUに通知する信号のことです。

CPUは割り込みを受信したら、現在実行中の処理を中断して、割り込みハンドラ(ベクター)というそれぞれの割り込みに対応した処理を実行します。この処理は自分で記述して登録する必要があります。

まず3つの割り込みについて紹介します。

RESET

ファミコンの起動時またはリセットボタンが押された場合に受ける割り込みです。ゲームの初期化処理などはRESETの割り込みハンドラで行います。

NMI

PPUが1画面描画したあとのVBlankの開始時に受ける割り込みです。NTSCの場合1秒に60回の画面描画となるので、このNMIの割り込みハンドラでVBlank内に処理を完結できれば、そのゲームは60fpsの動作になります。

IRQ

その他の割り込みです。いくつかのマッパーなどが発生させます。本書では扱いません。

割り込みハンドラの書き方と登録方法は以下のようになります。

```
RESET:
    ; 以下ゲーム開始時の初期化処理を記述
    rti ; 割り込みハンドラ終了の命令

NMI:
    ; 以下VBlank中に行う処理を記述
    rti ; 割り込みハンドラ終了の命令

IRQ:
    ; 以下IRQに応じた処理を記述
    rti ; 割り込みハンドラ終了の命令

    ; それぞれの割り込みハンドラを登録
    .bank 1
    .org $FFFA
    .word NMI
    .word RESET
    .word IRQ
```

2章のアドレス空間を見ると、アドレス$FFFAから$FFFFの6バイトは割り込みハンドラのアドレスとなっていましたね。

.wordというディレクティブは2バイトのデータの記述です。ここではラベルを書いていますが、ラベルはnesasmが翻訳するときにアドレスになります。アドレスは16ビット(2バイト)でした。

最初のゲーム

長々と説明してきましたが、ここで実際にごく簡単なソースコードを眺めてみましょう。

```
    .inesprg 1 ; 16KB の PRG バンク 1 個
    .ineschr 1 ; 8KB の CHR バンク 1 個
    .inesmap 0 ; マッパー 0
    .inesmir 0 ; 水平ミラーリング

    .bank 0
    .org $C000
RESET:
    sei
    cld

    lda #%10000000 ; A レジスタに値をロード(2 進数の即値)
    sta $2001      ; PPU のコンフィグアドレスに A レジスタの値を書き込み
.forever:
    jmp .forever    ; 無限ループ

NMI:
    rti

IRQ:
    rti

    .bank 1
    .org $FFFA
    .word NMI
    .word RESET
    .word IRQ

    .bank 2
    .org $0000
```

上記のソースコードはゲーム的には特に何もしませんが、nesasm を使ってファミコンの ROM ファイルを生成できます。生成した ROM ファイルはもちろんエミュレータで起動することができます。

nesasm はコマンドラインから以下を実行します。

Mac の場合（hoge ユーザのデスクトップに解凍していた場合）

```
cd /Users/hoge/Desktop/fcprogramming/src/04/
./nesasm 04.asm
```

Windows の場合（hoge ユーザのデスクトップに解凍していた場合）

```
cd ￥Users￥hoge￥Desktop￥fcprogramming￥src￥04￥
NESASM3.exe 04.asm
```

04.nes というファイルが生成されると思いますが、これが ROM ファイルになります。同時に 04.fns というファイルができるかと思いますが、これはラベルと翻訳後のアドレスの対応を出力したもので、開発時にゲームのデバッグなどで役に立ちます。

本書を解凍したディレクトリの下の src/04/ディレクトリに 04.asm として上記のソースコードを置いています。nesasm で生成した 04.nes ファイルも置いているので動作だけ見たい場合はエミュレータに 04.nes を読み込ませてください。

ゲームは青い画面が出るだけです。ここがスタートラインになります。

5章　プログラミングのいろは

前章で見たソースコードはほぼ最小限の雛形になります。ゲーム開発ではこの雛形に必要な処理を記述していきます。本章ではプログラミングをしていくにあたってよく使う要素を説明します。

よく使う命令

プログラミングとは命令を書き連ねていく作業です。CPUの命令はだいたい英文のフレーズから頭文字をとったような名前をしています。例えばldaはload data to A registerとかの略です多分。ここではとてもよく使う命令をいくつか紹介します。

lda

Aレジスタに指定した値をロードします。オペランドの指定はいくつかあります。

```
lda #14   ; 14(10進数)を値としてAレジスタにロード
lda $0001 ; アドレス$0001(RAMの領域)に格納されている値をAレジスタにロード
```

sta

Aレジスタの内容を特定のアドレスにストアします。オペランドはアドレスです。

```
sta $0002 ; アドレス$0002(RAM)にAレジスタの値のストア
```

inc、dec

incは指定したアドレスの値を+1します。decは逆に指定したアドレスの値を-1します。

```
inc $0003 ; アドレス$0003(RAM)の値を+1
dec $0004 ; アドレス$0004(RAM)の値を-1
```

cmp

Aレジスタの値を指定した値と比較します。比較した結果はステータスレジスタに反映されます。以下のように使います。

```
lda $0001 ; アドレス$0001(RAM)の値をAレジスタにロード
cmp #10   ; Aレジスタの値を10(10進数)と比較
```

ここまで紹介した命令はほぼAレジスタ関連ですが、XおよびYレジスタにも同等の命令が存在します。lda相当はldx、 ldyとなり、staはstx、styとなり、incはinx、inyとなり、cmpはcpx、cpyとなります。

次に紹介する命令はレジスタ間の値のコピーです。命令はどれもｔの文字で始まりますがこれは転送を意味するtransferの略です。

tax、tay

taxはAレジスタの値をXレジスタにコピーします。tayでYレジスタにコピーします。

txa

Xレジスタの値をAレジスタにコピーします。

tya

Yレジスタの値をAレジスタにコピーします。

次に紹介する命令は条件による分岐などに使う命令です。

jmp

指定したアドレスに飛びます。処理を飛ばしたい場合などに使います。ソースコード上では通常は具体的なアドレスの代わりにラベルを指定します。ここで指定するアドレスは絶対アドレスと呼ばれるもので、どこにでもジャンプできます。

```
jmp somewhere
```

beq、bne、bcs、bcc

cmp命令などの結果により指定したアドレスに飛びます。例えばbeqはcmpの結果が等しかった場合に指定したアドレスに飛びます。これも通常はラベルを指定します。

頭文字のｂは枝分かれを意味する英単語branchからきています。ここで指定するアドレスは相対アドレスと呼ばれるやつで、ジャンプできる距離には限界があります。

以下のようにして使います。

```
lda $0001     ; アドレス$0001(RAM)から値をAレジスタにロード
cmp #20       ; Aレジスタの値と20(10進数)を比較
beq somewhere ; 比較結果が等しければラベルsomewhereにジャンプ
```

bneはbeqの逆です。cmpの結果が等しくなかった場合に飛びます。

以上を知っていればだいたいOKです。これ以外の命令が出てくるときはなるべく都度説明しますが、知らない命令が出てきたらインターネットで検索すると良いです。その際のキーワードとしては6502という単語を添えて例えば”6502 lda”などと検索すると良いです。

変数

変数とはRAMのアドレスに名前をつけたものです。プラグラミングにおいては絶対に出てくる要素で、値を保存するのに使います。例えばゲームのスコアやキャラクターの位置情報、入力されたボタンの情報などを保存します。

RAMに値を保存するにはsta命令などを使いますが、アドレス指定だと下記のように書く必要がありました。

```
sta $0004 ; アドレス$0004にAレジスタの値を保存
```

この書き方だと、どこかにメモってないとアドレス$0004に何の値を保存するのかというのがわかりにくいと思います。そこで変数を使えば下記のように記述できます。ここでは$0004はキャラクタのHPを保存するのに使うとします。

```
sta hp ; Aレジスタの値をhp変数に保存
```

だいぶ分かりやすくなりましたね。変数はアドレスに名前をつけたものなのでアドレスを指定できる命令で使えます。例えばこんな感じで扱えます。

```
lda hp ; hp変数の値をAレジスタにロード
inc hp ; hp変数の値を+1
```

使い方に関しては以上ですが、変数を使う前にはまずどのアドレスにどういう名前をつけるかという、いわゆる変数宣言をする必要があります。

nesasmでは.rssetと.rsという2つのディレクティブを使って変数宣言を行います。まず.rssetディレクティブで変数の開始アドレスを設定し、.rsディレクティブでRAM領域を確保して名前をつけます。以下にサンプルコードを記載します。

```
    .rsset $0000 ; アドレス$0000からRAMを確保していく
score .rs 1 ; スコアを保存する変数として1バイト確保
posX .rs 1  ; X座標を保存する変数として1バイト確保
posY .rs 1  ; Y座標を保存する変数として1バイト確保
scores .rs 4 ; 各ステージごとのスコアを保存する変数として4バイト確保
level .rs 1 ; レベルを保存する変数として1バイト確保

    .rsset $0020 ; アドレス$0020からRAMを確保していく
enemyPosX .rs 1 ; 敵キャラのX座標を保存する変数として1バイト確保
enemyPosY .rs 1 ; 敵キャラのY座標を保存する変数として1バイト確保
```

上記のように.rssetはソースコード中に複数記述できます。この例ではscore変数は$0000になります。posXは$0001、posYは$0002、scoresは$0003ですが、ここで4バイト確保しているので、次に宣言されているlevelのアドレスは$0007になります。

その下で再び.rssetディレクティブで開始アドレスを指定しなおしているので、enemyPosXは$0020になり、enemyPosYは$0021になります。

定数

定数とは値に名前をつけたものです。定数は以下のように記述します。

```
MAXHP = $40       ; 最大HPを表し、その値は$40とする。
MAXBULLETS = $20 ; 一度に出せるショット数を表し、その値は$20とする。
INITX = $10       ; X座標の初期値を表し、その値は$10とする。
```

例えば上記の定数を使ってX座標を初期化したい場合は以下のように記述します。

```
lda #INITX ; 定数INITXの値($10)をAレジスタにロード
sta posX   ; posX変数にAレジスタの値($10)をストア
```

定数を使えば直接値を書くよりソースコードが読みやすくなるかと思います。また、値に変更がある場合、定数を定義している部分の数値だけをいじれば全体の変更ができるので便利です。

サブルーチン

サブルーチンとは特定のコードの塊に名前をつけたものです。関数とも呼ばれます。以下のように記述します。ラベルとrts命令でセットです。

```
; 変数を初期化するサブルーチンの例
initVars:
    lda #0
    sta posX
    sta posY
    sta score
    lda #40
    sta hp
    rts ; サブルーチン終了の命令
```

サブルーチンを呼び出すにはjsr命令を使います。

```
    jsr initVars ; 変数の初期化処理をコール
```

サブルーチンを使うメリットはいくつかあります。

・複数箇所で同じ処理を実行したい場合に同じ処理を書かなくて良い

・サブルーチンだけを修正すれば、利用箇所全体に修正が適用できる

・通常は特定の処理単位で記述するので、修正などがしやすい

・ソースコードの見通しがよくなる

ローカルラベル

これまでラベルとしてたのは実はグローバルラベルと言って、ソースコード内どこでも使えました。しかしグローバルラベルは重複してはいけないというルールがあり、例えば複数のサブルーチン内で似たようなラベル名(done とか)を使いたい場合には面倒です。

そこでサブルーチン内ではローカルラベルというラベルが使えます。ローカルラベルはサブルーチン内で重複しなければ良いので、他のサブルーチン内に同じ名前のローカルラベルがあっても OK です。

ローカルラベルの記述は以下のようになります。ドットではじまりインデントしません。

```
; レベルを上げるサブルーチン
levelUp:
    ; レベルが 4 だったらローカルラベル.done へ飛ぶ
    lda level
    cmp #4
    beq .done

    ; レベル 4 じゃなかったらレベルを+1 する
    inc level

.done ; ローカルラベル
    rts
```

プログラミングの基礎的な話はここでいったん完了です。次章からはより具体的なファミコンの仕様を紹介していきます。

6章　パレット

ファミコンのグラフィックは大きく分けてスプライトとBG(背景)の2つがあります。これらはCHRにあるタイルにより構成されますが、タイルは実は色そのものの情報を持っていません。

タイルは8x8ピクセルのドット絵ですが、それぞれのドットは0から3までのカラーインデックスという番号を持ちます。ここに具体的な色情報を与えるのがパレットの役割です。

本章ではパレットの構造から色の設定方法までを説明します。

パレットとは

パレットとは、スプライトまたはBGを構成するタイルに適用される色のデータです。パレットが適用されて初めて色のついたグラフィックが表示されます。

パレットはスプライト用とBG用の2つがPPU RAM(VRAM)に用意されています。それぞれのパレットはさらに4つのサブパレットに分割されます。各サブパレットは共通の背景色と3色の合計4色を持ちます。実際にはこのサブパレットを指定する形で使います。

色の値は1色あたり1バイトで$00から$3Fまでの値が指定できます。色と値の対応は下記の表を参照してください。

https://en.wikipedia.org/wiki/List_of_video_game_console_palettes#NES

サブパレット

2章で見たPPUのアドレス空間では、パレットは以下のPPUアドレスに対応していました。

$3F00 - $3F0F : BG用のパレット。16バイト。
$3F10 - $3F1F : スプライト用のパレット。16バイト。

これは以下のようにさらにサブパレットに分割されます。スプライトもBGも実際に指定するのはこのサブパレットの番号となります。

BG用サブパレット（4個）

$3F00 - $3F03 : BG用サブパレット0
$3F04 - $3F07 : BG用サブパレット1
$3F08 - $3F0B : BG用サブパレット2
$3F0C - $3F0F : BG用サブパレット3

スプライト用サブパレット（4個）

$3F10 - $3F13 : スプライト用サブパレット0
$3F14 - $3F17 : スプライト用サブパレット1
$3F18 - $3F1B : スプライト用サブパレット2
$3F1C - $3F1F : スプライト用サブパレット3

タイルは各ドットに0から3までの値を持つと言いましたが、これはサブパレット内のオフセットです。また各サブパレットは4色持ちますが、0番は共通の背景色になります。

PPUアドレス$3F00と$3F10が背景色の指定で、物理的にはPPU RAMの同じ場所に対応しています。なので$3F10に書き込めば$3F00の値も変わるし、逆も同様です。

つまり8x8ピクセルのタイル1つあたりは実際には3色までしか選べないということです。

パレットの設定

パレットに対応するPPUアドレス範囲に色の値を書き込んでパレットを設定します。PPUのアドレス空間に書き込む場合は、まず書き込むPPUアドレスをCPUアドレス$2006に書き込んで指定する必要があります。

例えばスプライト用のパレットを設定したい場合、$3F10というPPUアドレスをアドレス$2006に書き込みます。具体的には以下のようなコードになります。

```
; PPUステータスをリセット（お約束）
lda $2002

; まずアドレスの上位バイト$3Fを書き込む
lda #$3F
sta $2006

; 次にアドレスの下位バイト$10を書き込む
lda #$10
sta $2006
```

2バイトのアドレス$3F10を指定したいわけですが、1バイト(8ビット)ずつしか扱えないため2度に分割して$2006に値を書き込んでいます。この$2006に対する現在の書き込み順序は$2002をロードすることでリセットできます。

PPUアドレスを指定したら、CPUアドレスの$2007に1バイトずつ色の値を書き込んでいきます。

```
; PPUアドレス$3F10に値$32を書き込み
lda #$32
sta $2007

; PPUアドレス$3F11に値$10を書き込み
lda #$10
sta $2007

; PPUアドレス$3F12に値$2Aを書き込み
lda #$2A
sta $2007
```

アドレス$2007に書き込むたびに次に書き込むPPUアドレスが+1されます。16バイト書き込むにはCPUアドレス$2007に16回書き込みを行います。

ループ処理

わざわざ16回分こんなコードを書くのかというと普通はやりません。普通はパレットデータをどこかに定義しておいて、それを頭から順次読み込んで$2007に書き込むという処理を書きます。これにはループ処理(繰り返し処理)を使います。

まずは以下のようにパレットデータをソースコードのどこかに定義します。

```
spritePalette1:
    .byte $0F, $1C, $15, $14
    .byte $0F, $02, $38, $3C
    .byte $0F, $2C, $16, $24
    .byte $0F, $04, $32, $1D
```

.byteディレクティブは1バイトのデータの記述です。上記は16色分で16バイト記述しています。色の値は全部てきとうです。

上記で定義したデータをループ処理で$2007に書き込んでいきます。コードは以下のようになります。ここではサブルーチンとしました。

```
loadSpritePalette:
    ; PPUステータスをリセット（お約束）
    lda $2002
    ; まずアドレス$3F10の上位バイト$3Fを書き込む
    lda #$3F
    sta $2006
    ; 次にアドレス$3F10の下位バイト$10を書き込む
    lda #$10
    sta $2006

    ; Xレジスタを0で初期化。ループ処理のカウンタとして使う。
    ldx #0
.load
    ; spritePalette1 + x のアドレスの値をロードして、$2007にストア
    lda spritePalette1, x
    sta $2007

    ; Xレジスタの値をインクリメント(+1)
    inx
    ; Xレジスタの値を16(10進数)と比較
    cpx #16
    ; Xレジスタがまだ16じゃなければ.loadローカルラベルに戻る
    bne .load
    ; Xレジスタが16だったら終了
    rts
```

処理の内容はコメントに書いてある通りです。ループ処理はこんな感じで書きます。この短い処理に出てきた命令は頻出なので軽く覚えておきましょう。ループ内のldaの書き方だけ初見になると思うので説明します。

まずこのように書いた場合ですが、

```
lda spritePalette1
```

ラベルは翻訳時にアドレスになるんでしたね。なのでこれは spritePalette1 のアドレスの値をロードします。上記の例だと$0F ですね。

次にこのように書いた場合ですが、

```
lda spritePalette1, x
```

これは”spritePalette1 + X レジスタの値”のアドレスの値をロードします。ここで X レジスタはインデックスなどとも呼びます。X レジスタはループに入る前に ldx #0 とゼロで初期化しており、ループ処理の中で inx 命令で 16 になるまで毎回+1 しています。

つまり、ループ回数によって以下のようにロードする値を変更しています。

```
 1回目: spritePalette1 + 0  (値は$0F)
 2回目: spritePalette1 + 1  (値は$1C)
 3回目: spritePalette1 + 2  (値は$15)
        中略
 7回目: spritePalette1 + 6  (値は$38)
 8回目: spritePalette1 + 7  (値は$3C)
        中略
15回目: spritePallete1 + 14 (値は$32)
16回目: spritePalette1 + 15 (値は$1D)
```

こういう書き方は lda だけではなく sta 命令などでもできます。また、インデックスとなるレジスタは X レジスタだけではなく、Y レジスタも使うことができます。インデックスの値が 1 ではなく 0 から始まるのはプログラミングあるあるです。

7章　スプライト

ファミコンのグラフィックにはスプライトとBGの2種類があります。またこれらを構成する最小単位は8x8ピクセルのタイルです。BGはタイルを格子状に並べたものですが、スプライトは画面の好きな位置に表示することができます。

スプライト1つはタイル1つで表現されます。8x8ピクセルより大きなサイズのマリオなどのキャラクターは複数のスプライトから構築されています。PPUはOAMという256バイトのスプライト用メモリを持っていて、スプライトを64個まで同時に扱えます。

本章ではスプライトの表示方法を説明します。

CHRデータの作り方

スプライトを表示するにはまずタイルデータ(CHR)が必要です。2章で説明した通り、CHRは8x8ピクセルのタイルの集合です。これらのタイルはいわゆるドット打ちをして作っていくわけですが、通常は専用のツールを使います。ここではわりとメジャーなyychrを紹介します。

どうもジオシティーズで公開していたらしく、サイトが消えてしまっているので、インターネットアーカイブからダウンロードしてください。比較的新しい.net版がありますが、移植されてない機能があったりするのでC++版をオススメします。

https://web.archive.org/web/20190330001841/http://www.geocities.jp/yy_6502/yychr/0yychr.html

これもWindows用のGUIツールなのでMacユーザはwine経由で起動するか(10.14以下の場合)、VMware上のWindowsで実行してください。

使い方はわりと直感的にわかると思います。これで作れるのはCHRデータとパレットデータです。パレットはパレットセット(.dat)というのを保存してください。

yychrはファミコン以外にも使えるみたいで、保存したパレットのサイズをみると256バイトありますが、これの先頭16バイトを読み込めば良いです。

パターンテーブル

PPU アドレスの$0000 から$1FFF の 8KB をパターンテーブルと呼びます。画面に表示する CHR はここにマッピングする必要があります。

ツールで作った CHR データをマッピングするには以下のように.incbin ディレクティブを使って組み込む必要があります。

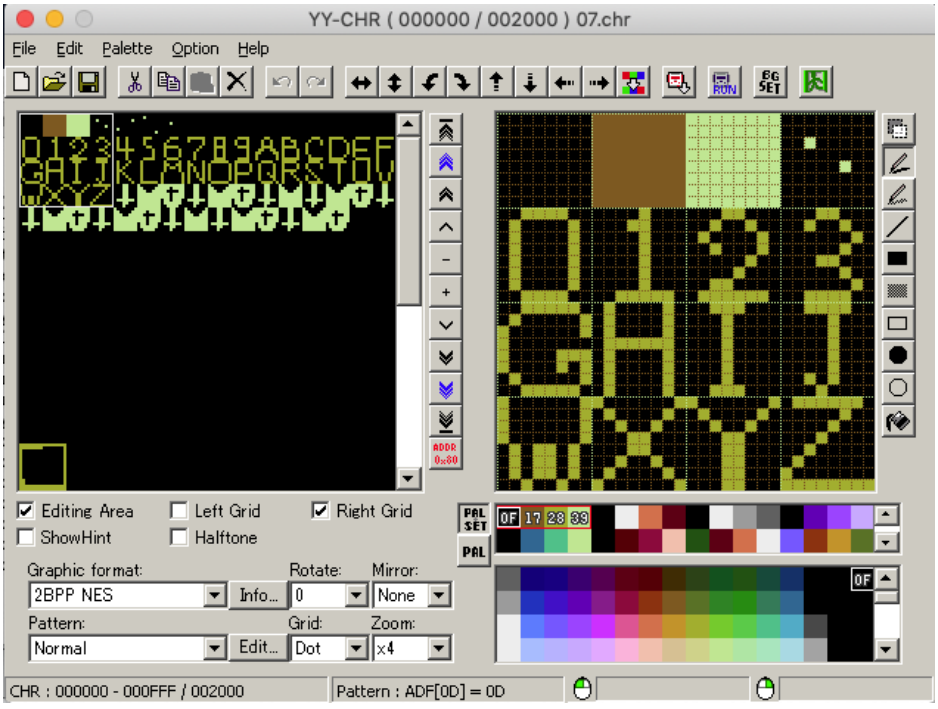
```
.bank 2
.org $0000
.incbin "07.chr"  ; ドットエディタで作ったファイルを組み込み
```

パターンテーブルはアドレス$0000-$0FFF と$1000-$1FFF で 4KB ごとに 2 つに分けて使います。片方に BG 用 CHR データを、もう片方にスプライト用 CHR データをマッピングします。

本書では一貫して$0000-$0FFF の範囲に BG の CHR を、$1000-$1FFF の範囲にスプライトの CHR を指定していきます。

スプライト、BG はそれぞれのパターンテーブルから使うタイルを番号で指定します。番号は 0 から 255 です。CHR を表示したり編集したりするツールでは大抵 1 行あたりタイルを 16 個並べていて、左上が 0 番、右下が 255 番となります。

以下は yychr の表示です。左側が 4KB の CHR データです。ここでは例えば数字の 0 はタイル番号 16 番、1 が 17 番、2 が 18 番です。そしてアルファベットの F が 31 番で、G が 32 番です。



スプライトDMA

スプライトを表示するもっとも簡単で早い方法はスプライトDMAになります。これはCPU RAMの256バイト範囲をPPUのOAMへコピーする機能です。

あらかじめCPU RAMにスプライトの情報を書き込んでおき、それを一気にOAMへ転送します。RAMの範囲は使える範囲ならどこを使っても良いのですが、$0200-$02FFがよく使われるようです。

スプライトはタイルに座標と属性を持たせたものです。これを管理するのにスプライト1つあたりRAMを4バイト使います。4バイトの内訳は先頭から順番に以下の通りです。

1. Y座標
   解像度は256x240なので0($00)から239($EF)までが有効な値です。0が画面の最上部です。
   $EFより大きい値は画面外になります。

2. スプライトに使うタイルのタイル番号
   タイルはスプライト用のパターンテーブルから指定します。

3. スプライトの表示属性
   属性は以下の指定ができます。ビット単位で指定します。

   ビット7　：スプライト上下反転（1で反転）
   ビット6　：スプライト左右反転（1で反転）
   ビット5　：0でBGの手前に表示、1でBGの裏に表示
   ビット4-2：----
   ビット1-0：スプライト用パレットのサブパレットの指定
   　　%00(10進数で0)から%11(10進数で3)まで指定可能

4. X座標
   0が画面左端で249($F9)を超えると画面外になります。

RAMに書き込んだスプライトをPPUに転送するにはCPUアドレスの$2003と$4014にそのアドレスを書き込みます。今回はスプライトDMA用にアドレス$0200から$02FFを使うとします。

```
; $0200の下位バイトを指定
lda #$00
sta $2003
; $0200の上位バイトを指定して転送開始
lda #$02
sta $4014
```

これだけで64個のスプライトがすべてPPUのOAMに転送されます。この処理はVBlank時にNMIの割り込みハンドラで実行します。

スプライトの有効化

スプライトを有効にするにはPPUのコンフィグポートに書き込みを行います。PPUのコンフィグポートはCPUアドレスの$2000と$2001です。下記のような設定が可能です。

$2000の設定

```
ビット7   : VBlank時にNMI割り込みを発生させる（1: on）
ビット6   : ----
ビット5   : スプライトのサイズ（0: 8x8, 1: 8x16）
ビット4   : BG用パターンテーブルのPPUアドレス        （0: $0000, 1: $1000）
ビット3   : スプライト用パターンテーブルのPPUアドレス（0: $0000, 1: $1000）
ビット2   : PPU RAMの自動アドレスインクリメント（0: +1, 1: +32）
ビット1-0 : ベースネームテーブルのアドレス
            (%00: $2000, %01: $2400, %10: $2800, %11: $2C00)
```

$2001の設定

```
ビット7    : 青色を強化
ビット6    : 緑色を強化
ビット5    : 赤色を強化
ビット4    : スプライト描画を有効にする
ビット3    : BG描画を有効にする
ビット2    : スプライトクリッピング（0: 左端8ピクセルカット, 1: クリッピングOFF）
ビット1    : BGクリッピング        （0: 左端8ピクセルカット, 1: クリッピングOFF）
ビット0    : グレイスケール（0: ノーマルカラー）
```

上記のように設定は色々ありますが、スプライトを有効にするにはアドレス$2001のビット4のフラグを立てます。

スプライト表示のコード例

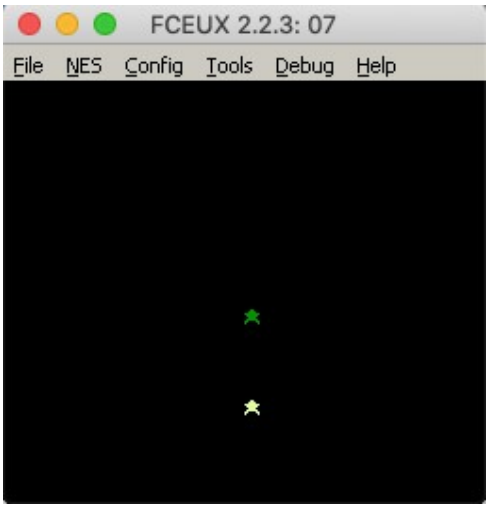
スプライトを画面に表示するコードは以下のようになります。

```
RESET:
    ; 画面真ん中あたりに表示する

    ; Y 座標は$80
    lda #$80
    sta $0200

    ; タイル 0 番を表示する。
    lda #$00
    sta $0201

    ; 属性の指定。サブパレットは%00。画像の反転などはしない。
    lda #$00
    sta $0202

    ; X 座標は$80
    lda #$80
    sta $0203

    ; 以下 PPU のコントロールポートの設定
    ; NMI を発生させる。スプライトは PPU アドレス$1000 からロードする
    lda #%10001000
    sta $2000

    ; スプライトを有効にする。
    lda #%00010000
    sta $2001
    ; 無限ループに入って、あとは NMI ハンドラで処理する。
.loop
    jmp .loop


NMI:
    ; スプライト DMA 転送（$0200-$02FF から PPU の OAM に転送）
    lda #$00
    sta $2003
    lda #$02
    sta $4014
    rti
```

スプライトを表示するだけのゲーム

本書を解凍したディレクトリの下の src/07/ディレクトリに 07.asm として本章の内容をまとめたソースコードを置いています。nesasm で生成した 07.nes ファイルも置いているので動作だけ見たい場合はエミュレータに 07.nes を読み込ませてください。

このゲームは 2 つのスプライトを画面に表示します。1 つは画面の中央あたりで、もう 1 つはその少し下に表示します。



表示するスプライトのタイル番号は同じですが、属性でサブパレットの指定を変えているので違う色が適用されています。

このゲームが RESET のハンドラでやっていることは以下の通りです。

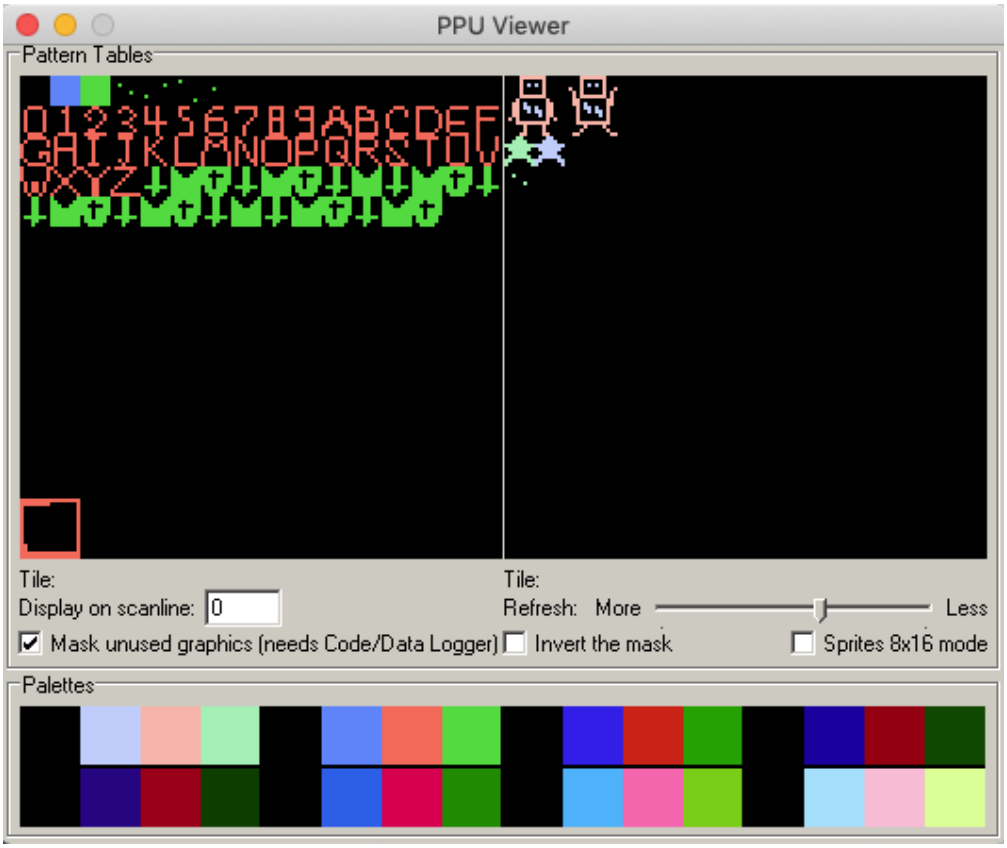
1. 初期化
2. BG 用パレットの設定
3. スプライト用パレットの設定
4. 1 つ目のスプライトを設定
5. 2 つ目のスプライトを設定
6. スプライトを有効化

また、NMI のハンドラでやっていることはスプライト DMA だけです。

初期化のコードは当面はお約束だと思っててください。

PPU Viewer

上記ゲームのバンク 2(.bank 2)でパターンテーブルにマッピングしている CHR(07.chr)ですが、エミュレータに fceux を使っていれば Debug => PPU Viewer で内容を見ることができます。PPU Viewer は同時にパレットの設定内容も確認できます。



左側はパターンテーブルの前半(PPU アドレス$0000-$0FFF)で、右側が後半(PPU アドレス$1000-$1FFF)です。スプライトのパターンテーブルは以下のコードで後半を使うように設定しています。

```
; 以下 PPU のコントロールポートの設定
; NMI を発生させる。スプライトは PPU アドレス$1000 からロード
lda #%10001000
sta $2000
```

タイル番号は各パターンテーブルの左上が 0 で右に行くと 1 ずつ増えていき、右端にくると次の行の左端になります。右下のタイル番号は 255($FF)になります。今回は$20 を指定していますが、これは上の画像だと最初の星のタイルになります。

8章　ゲームコントローラーによる入力

ファミコンのコントローラーと言えば1コン2コンですが、この章ではそれぞれの入力を読み取る方法を説明します。

コントローラーによる入力の読み取り

コントローラーの入力値を調べるにはCPUアドレスの$4016および$4017を読み込みます。それぞれが1コン、2コンに対応していて、ロードするごとに各ボタンのオンオフ状態を取得できます。1Pのコントローラーの入力を読み込むコードは以下の通りです。

```
    ; 読み込む前の準備（お約束）
    lda #$01
    sta $4016
    lda #$00
    sta $4016

    ; 以下1Pのコントローラーの入力を読み込み
    lda $4016  ; Aボタン
    lda $4016  ; Bボタン
    lda $4016  ; セレクトボタン
    lda $4016  ; スタートボタン
    lda $4016  ; 上（十字キー）
    lda $4016  ; 下（十字キー）
    lda $4016  ; 左（十字キー）
    lda $4016  ; 右（十字キー）
```

Aレジスタに読み込んだ値のビット0(右端)が1であればそのボタンが押されています。これを調べるにはand命令を使います。and命令とは、お互いに1の場合に計算結果が1となる命令です。それをビット単位で行います。

例えば以下のような演算をすると、

```
    lda %10011110
    and %00110101
```

結果は%00010100となります。この計算結果はAレジスタに格納されます。読み込んだ値のビット0を調べるコードは以下のようになります。

```
    lda $4016
    and %00000001
    ; and命令の結果が0(ボタン押されてない)ならば飛ぶ
    beq nextCheck1
    ; ここに上記で調べたボタンが押されていたときの処理を書く
nextCheck1:
    ; 同じように調べる
```

beq等のブランチ系命令はcmp命令以外にもand命令でも使えます。and命令の結果が0であればbeq命令で指定したアドレスへジャンプします。

上記のようにして8ボタン分を8回調べても良いのですが、コードが若干冗長になりますね。ループ処理を使って処理をまとめることができそうです。

よりベターなコントローラー入力読み込み

変数とループ処理を利用した少し頭の良いコントローラー入力の取得方法を紹介します。コードは次のようになります。ここではサブルーチンとして実装しています。

```
readOneCon:
    ; 以下はお約束
    lda #$01
    sta $4016
    lda #$00
    sta $4016

    ; 8回入力を読み込むのでカウンタ(Xレジスタ)を8に初期化
    ldx #$08
.loop
    ; ボタンの状態をロードしてビット0をCarryフラグにセット
    lda $4016
    lsr a

    ; buttons変数を左に1ビットシフトする
    ; buttons変数のビット0にはCarryフラグがセットされる
    rol buttons

    ; 8回処理するまで.loopラベルにジャンプ
    dex
    bne .loop
    rts
```

lsr命令(logical shift right)は指定したレジスタの値を1ビット右にシフトします。このとき、はみ出たビット0はステータスレジスタのCarryフラグにセットされます。またビット7には0が補充されます。

rol命令(rotate left)は指定したアドレスの値を1ビット左にシフトします。このとき、ビット0にはCarryフラグが補充されます。

lda $4016でロードした値のビット0にはそのボタンのオンオフ状態が入っています。そのビット0をlsr命令でCarryフラグにセットし、rol命令でbuttons変数のビット0にセットしています。

上記のreadOneConサブルーチンを呼び出せば、結果としてbuttons変数には8つボタンの入力状態が保存されます。調べたボタンの順にビット7からビット0までボタンの状態が格納されます。

最初に調べたAボタンがビット7、次に調べたBボタンがビット6、最後に調べた右キーがビット0に格納されています。例えばBボタンの入力をチェックしたければ、以下のようにビット6を調べれば良いことになります。

```
    ; readOneCon関数を呼び出し
    jsr readOneCon

    ; buttons変数をロードしてビット6を調べる
    lda buttons
    and %01000000
    beq notPressed
    ; ここにBボタンが押されてたときの処理
notPressed:
```

コントローラーを使ったゲーム

本書を解凍したディレクトリの下のsrc/08/ディレクトリに08.asmとして本章の内容をまとめたソースコードを置いています。nesasmで生成した08.nesファイルも置いているので動作だけ見たい場合はエミュレータに08.nesを読み込ませてください。

内容はほぼ7章のゲームのソースコードと同じですが、表示するスプライトが1つになっています。ただしコントローラーの読み取り処理が追加されています。

コントローラーの読み取り処理はreadOneConサブルーチンに記述しています。NMIハンドラで毎フレーム呼び出し、Aボタンが押されていたらスプライトに指定するサブパレットの番号を変更します。また、十字キーが入力されていたらスプライトを移動します。

サブパレットの番号はsubPalNo変数で管理していて、3の次は0に戻るようにしています。この処理はchangeSubPaletteサブルーチンに記述しています。Aボタンを押し続けると星の色が4色でループしていく様子が見れると思います。

毎フレームAボタンが押されているかどうかを見ているのでちょい押しでも連打っぽくなってしまうのは仕様です。本当はAボタンを1回押したら1回サブパレットを変更する、としたいところですが、それをやると若干コードが複雑になるのでここではやっていません。