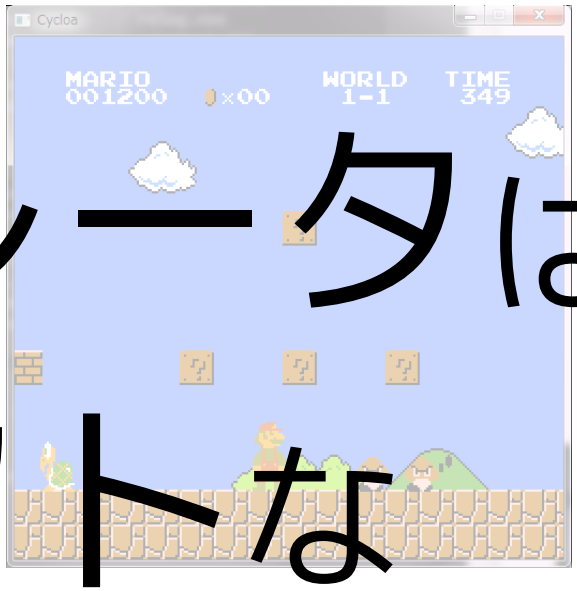
```
C28F:C5 D2     CMP $00D2 = #$AC
C291:F0 FC     BEQ $C28F
C293:60        RTS
C294:A9 00     LDA #$00
C296:8D 05 20  STA $2005 = #$00
C299:8D 05 20  STA $2005 = #$00
C29C:A9 00     LDA #$00
C29E:8D 06 20  STA $2006 = #$00
C2A1:A9 00     LDA #$00
C2A3:8D 06 20  STA $2006 = #$00
C2A6:60        RTS
C2A7:A9 00     LDA #$00
C2A9:8D 00 20  STA $2000 = #$80
C2AC:8D 01 20  STA $2001 = #$0E
C2AF:20 FD C2  JSR $C2FD
C2B2:A9 20     LDA #$20
C2B4:8D 06 20  STA $2006 = #$00
C2B7:A0 00     LDY #$00
C2B9:8C 06 20  STY $2006 = #$00
C2BC:A2 20     LDX #$20
C2BE:B1 D0     LDA ($D0),Y @ $C4F9 = #$68
C2C0:F0 28     BEQ $C2EA
C2C2:C9 FF     CMP #$FF
C2C4:F0 0D     BEQ $C2D3
C2C6:8D 07 20  STA $2007 = #$00
C2C9:C8        INY
C2CA:D0 02     BNE $C2CE
C2CC:E6 D1     INC $00D1 = #$C4
C2CE:CA        DEX
C2CF:D0 ED     BNE $C2BE
C2D1:F0 E9     BEQ $C2BC
C2D3:C8        INY
C2D4:D0 02     BNE $C2D8
C2D6:E6 D1     INC $00D1 = #$C4
C2D8:A9 20     LDA #$20
```



エミュレータはソフトなマホウ。

ハード解析“なし”で作るファミコンエミュレータ

Stage.1

CPU + Memory

```
19
20  void Processor::onHardReset()
21  {
22      //from http://wiki.nesdev.com/w/index.php/CPU_power_up_state
23      this->P = 0x24;
24      this->A = 0x0;
25      this->X = 0x0;
26      this->Y = 0x0;
27      this->S = 0xfd;
28      this->write(0x4017, 0x00);
29      this->write(0x4015, 0x00);
30      for(uint16_t i=0x4000;i<=0x400F;i++)
31          this->write(i, 0x00);
32      }
33      this->PC = (read(0xFFFC) | (read(0xFFFD) << 8));
34
35      this->NMI = false;
36      this->IRQ = false;
37      this->needStatusRewrite = false;
38      this->newStatus = 0;
39  }
40
41  void Processor::onReset()
42  {
43      //from http://wiki.nesdev.com/w/index.php/CPU_power_up_state
44      //from http://crystal.freespace.jp/pgate1/nes/nes_cpu.htm
45      consumeClock(6);
46      this->SP -= 0x03;
47      this->P |= FLAG_I;
48      this->write(0x4015, 0x0);
49      this->PC = (read(0xFFFC) | (read(0xFFFD) << 8));
50
51      this->NMI = false;
52      this->IRQ = false;
53      this->needStatusRewrite = false;
```


ψ（プサイ）の興味関心空間 http://ledyba.org/

目次

0x00 エミュレータは、ソフトなマホウ。

エミュレータって、すごく不思議なソフトウェアだと思います。

テレビに繋いで遊んでいたはずのゲームが、パソコンの画面の中で動く。

パソコンとはまるで違う”カセット”をさして、
パソコンのキーボードとまるで違うコントローラを使って、
パソコンとはずいぶん雰囲気の違うゲーム画面で遊ぶものだったのに、
エミュレータを使うと、それがなぜか、いつも使ってるパソコンの中で動く。

中学生の時の私は、このとっても不思議なソフトウェアに夢中になりました。

ゲームボーイアドバンスとパソコンを繋ぐ不思議なケーブルをアキバの怪しいお店で買い、カセットの中身とセーブデータをパソコンに転送して、それをエミュレータで動かてみると…今まで小さなゲーム機の中だけで動いていたポケモンたちが、突然私のパソコンの中でも動き出したのです。

今までゲーム機という圧倒的な壁で隔てられていたと思っていたポケモンたちが、デバッガというエミュレータの機能を使うと、0と1の生の姿として触れることができました。

ゲーム内からは決して知ることのできない彼らの隠れた個性を知ることもできましたし、彼らを増やすことも、作り変える事もできてしまいました。

今までどうしてそうなるのか不思議だったゲーム内の”物理法則”も、エミュレータのデバッガを使えば、詳しく調べることができました。ちょっと難しかったけど。

話は変わります。

私は、小さい頃、魔法少女に憧れていて、魔法少女になれたらなーって思ってました。

でも、現実には魔法はないし、残念ながら、私は男みたいです。仕方が無いので、代わりに電子の世界の魔法少女―つまり、ハッカーです！―を目指すことにしました。そんな私がエミュレータに出会ったとき、これはまさに“魔法”そのものだ！とすごく興奮しました。そんなエミュレータを作れるプログラマーたちは、私にとって憧れの”魔法使い”たちだったのです。

時は流れて、私は大学生になりました。受験も終わって、長い夏休みを手に入れる事ができました。今なら、きっと私もエミュレータが作って、そんな”魔法使い”たちに近づけるんじゃないかな？

この本は、私のそんなあこがれがたくさん詰まった本です。

0x01 必要なものを揃えなきゃ。

とまあ、めっちゃ恥ずかしいポエムはこれくらいにしておいて、具体的にエミュレータを開発するのに何が必要なのか考えて行きましょう。

エミュレータは、対象のコンピュータを模倣するプログラムです。今回はファミコンですが、ファミコン含めてゲーム機は所詮はただのコンピュータです。もうちょっとぼんやりと、一般的なコンピュータも交えながら、考えてみましょうか。

コンピュータは、何でできているでしょう。それらを全部ソフトウェアで再現することができれば、エミュレータができ上がるはずではないでしょうか…？

まずまっさきに思いつくのは、もちろんCPUです。それから、メモリ（Random Access Memory; RAM）。メモリからCPUが機械語を読みだしてプログラムを実行し、同じメモリ上の値を書き換える…というのが、コンピュータの基本的な動作です。

あと、現在のPCで言うところのHDDやSSDにあたる機構が必要です。メモリは電源を切ったら消えてしまいますから、お目当てのプログラム（今回は、ゲームですね）を読み込んだり、編集したデータ（今回なら、セーブデータですね）を保存したりするための、外部記憶装置が必要です。現在だとHDDやSSDが一般的ですが、昔は音楽テープ（！）とか、フロッピーとかを使っていました。一度書いたら書き換えることができませんが、ROM(Read Only Memory)もこのポジションにいます。他の媒体と比べてROMはデータがいきなり消えたりしにくいので、BIOSやゲームソフトのような、書き換える必要がないデータを格納する用途には適しています。ファミコンのカートリッジでも、プログラムや絵を記録するためにROMが使われています。

ここまではコンピュータの中だけの話ですが、もちろん、コンピュータ、特にゲーム機は人間からの操作に応じて動き、操作の結果起こったゲームの進行を見せてくれます。人間とコンピュータの間の架け橋になるようなパーツも、コンピュータに欠かせない一部分でしょう。

とりあえず、画面出力でしょうか。モニタに文字や画像（ファミコンなら、マリオとか！）を出力してくれなければ、何が起こっているのかわかりません。この画像出力処理はその時々のCPUにとって重い処理であることが多く、別のハードウェアに任せていたりすることが多いようです（現在でも、大体GPUがやっていますよね）。となると、CPU以外にそれらも再現しなきゃ…少し大変です。

あとは、コンピュータを操作できなければどうしようもありません。キーボードやマウス、ゲーム機のコントローラなどの、入力デバイスも必要です。ファミコンはコンピュータと違ってコントローラで操作しますが、コンピュータにはファミコンのコントローラは接続できませんから、パソコン用のUSBコントローラを、エミュレータでも利用できるようにしなきゃいけませんね。

最後に、音楽や効果音といったサウンドによっても、ゲームは私達を楽しませてくれます。サウンドの再生方法は、現在ではレコーディングした音声ファイルを再生することが多いですが、昔は直接音の波形を作るのが一般的で、しかも専用のハードウェアが行なっていることが多いようです。これもちょっと、やっかい。

さて、以上をまとめると、コンピュータを再現するために必要なものは、

- CPU
- メモリ（RAM）
- HDDとかSSD、フロッピー、カセットみたいな外部記憶装置
- ビデオ
- サウンド

- キーボードやコントローラのような入力装置

といった所。これらのパーツを、それぞれファミコンを模倣して実装して纏めあげれば、きっとエミュレータになるはず。

0x0101 資料を集めよう

というわけで、ぼんやりとは何を作ればいいのかが分かりましたから、具体的に、ファミコンはそれぞれのパーツがどのようになっているのかを調べましょう。

本来は、こういうのは半田ごてとロジックアナライザ、オシロスコープなどを駆使して解析して調べるところですが、今の私のスキルでは、まだこの"魔法"は使えそうにありません。これは将来の目標にして、今回この部分は他人の情報を頼りにして作っていくことにしました。

幸いにも、作るターゲットを「ファミコン」としたおかげで、これには困らないくらいにたくさんの資料がインターネット上に存在します。他のゲーム機では、解析無しに作るのはおそらく無理だろうと思います。

一番安直なのが、Wikipedia。さすがにどうかと思われるかもしれませんが、実は"おおまか"に調べる程度であればいろいろと載っています。また、「PPU」とか、チップの型番みたいな難しい単語（？）が色々載っているので、Googleで調べる際に使う単語のリストとして使えます。

先ほどの６つのパーツに関しては、次のような答えが得られました。

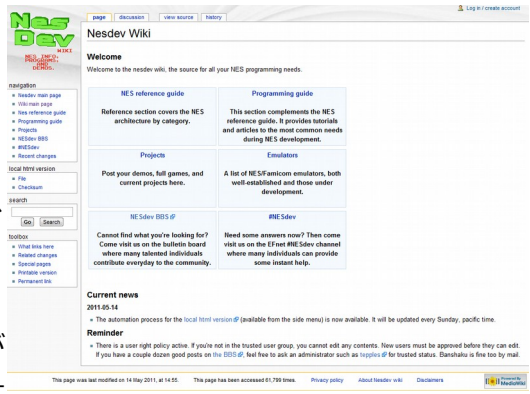
- CPU:カスタムされたMOS 6502
- メモリ：2KB(たったの2048バイト！)
- 外部記憶：カートリッジ
- ビデオ：リコー製PPU
- サウンド：APUと呼ばれる矩形波やパルス波といった波を直接出力する独自音源、CPU内蔵

それぞれについて実際に実装するときには、このキーワードたちで検索していけばよいでしょう。

ファミコンは非常に人気なゲーム機ですから、ファミコンを対象にした解析資料サイトも存在しています。それらの中から、とっても参考になったものをいくつかご紹介しましょう。開発している間は、ずっと開いていたと言っても過言ではないくらい便利なサイトたちです。

NESDEV Wiki

URL: http://wiki.nesdev.com/w/index.php/Nesdev_Wiki



NESのエミュレータ・自作ソフト開発者にとってのバイブルと言ってもいいです。CPU、ビデオ、サウンド、カートリッジの他、電極の配置のような、ハードウェアの仕様まで書いてあります。エミュレータの開発時に躓きやすいポイントや、エミュレータのテスト用プログラムが置いてあるなど、至れり尽くせりです。

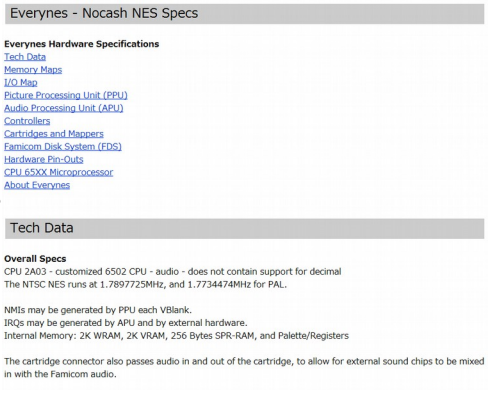
なお、このWikiは派生元で、元のNESDEV（URL：http://nesdev.parodius.com/）にあった情報を、有志がまとめて加筆訂正したものであるみたいです。元のNESDEVは玉石混淆すぎるため、あまり参照しなかったのですが、CPUの命令別の詳しい動作情報が載っているテキストはこちらにしか無いなど、元の方にも参考になる情報もありました。

本当に細かい仕様が分からずに悩んだ場合は、フォーラムで検索してみると、答えが得られる事も

あるかもしれません。（URL：http://nesdev.parodius.com/bbs/）

Everynes

URL:http://nocash.emubase.de/everynes.htm



エミュレータマニアには有名なno$シリーズの作者による解析情報です。NDS/ゲームボーイアドバンスや、MSXのエミュレータも手がけていて、それらの解析情報も公開されています。

概ね内容はNESDEVと被るのですが、たまにNESDEVと言っていることが食い違って、こちらの方が正しいと思われる場合があります（逆もあります）。また、同じであったとしても表現にNESDEVとずいぶん違う所があるため、NESDEVに書いてあることがいまいちわからない、あるいはうまく実装できない…と思ったら、こちらに来ると良いと思います。1ページに全部纏めてるせいでちょっと見づらいのが難点。

NES on FPGA

URL: http://crystal.freespace.jp/pgate1/nes/

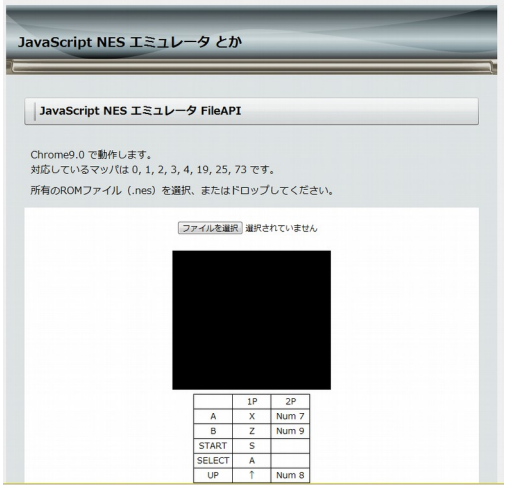
3サイト目は日本語のサイト！こちらはファミコンをコンピュータでのエミュレーションでなく、FPGAという、「"回路をプログラミング"できるチップ」を使って再現された方による、資料＆開発日誌です。単なる資料としてでなく、開発日誌が読み物としても、とっても楽しいです♪

前２つに比べるとあまり詳しくはないのですが、サウンドのところは、前２つに比べて動作が自然言語で詳しく説明されており、日本語であることも相まって非常にわかりやすいです。前２つを読む前に、とりあえずこのサイトを読んでおくのがおすすめです。

読み物として楽しい開発日誌の方も、同じポイントで躓いた場合に助けになるかもしれません。

JavaScript NES Emulator

URL:http://twoseater.my-sv.net/nesfileapi/



他人のエミュレータのソースコードを参考にするのは、ちょーっと、気が引けます？だいじょうぶ、すぐにそんなこと言ってられなくなります（笑）。自然言語だけで書かれたテキストではやっぱり細かい所がわからなかったり、サイトどうしで意見が食い違っていたり、そもそも詳しく書かれていなかったりする場合、実際にゲームが動くエミュレータは、とても頼りになる情報源です。

でも、エミュレータ同士でもある機能に関する動作が違う（けど、ソフトは動く）事があったりして、ホント訳がわからなくなってしまうのですが、ここは一つ「厳密に再現しなくても、エミュレータは動く」と好意的に解釈して行きましょう（笑。

このJavaScriptのエミュレータは、速度を第一に書かれた他のCやJavaで書かれたエミュレータに

比べて、とてもソースコードの見通しがよく、非常に読みやすいです。また、ブラウザに付属のデバッガを用いることで、かなり容易に動作を追いかけることができます。

ただし、速度を稼ぐためなのか、ビデオ処理で随分再現性を損ねている部分があるので、ご注意を。

英語必須。

さて、そろそろお気づきかと思いますが、さすがに日本語のサイトだけでは情報が足りません。とはいえ、JavadocとMSDNを原文で何とか読めて、JavaやC#のAPIが使える程度の英語力があればなんとかなります。なお、英語で検索するときは、「ファミコン」や「Famicom」ではなく、海外での名称「NES（Nintendo Entertainment System）」で検索しないとヒットしませんよ～！

また、ネット上の有象無象のゲームマニアのサイトにも、様々な詳しさや視点から書かれたテキストがたくさんあります。例えば、ファミコンで作曲をしたい人向けに書かれたウェブサイトには、前述の解説サイトたちのようなハードウェアを再現する視点からではなく、プログラマ向けのソフトウェアの立場からの説明でもなく、あくまで音源として使いたい人たちの視点から書かれた説明があります。

もちろん、もっと一般の人向けに書かれた、簡単で概念的にさらっと書いてあるようなテキストも、なんと言ったってファミコンですから、た～くさん存在します。

エミュレータは、対象のすべてのパーツを一から再現しなくてはならないため、非常に広い範囲の知識が要求されます。もしもあなたのあまり知らない分野などで、詳しい資料を直接見ても理解できなかった場合（私は、サウンドがいまいちわかりませんでした）、こういった一般向けのサイトや、別の視点からのウェブサイトを見ることで、理解の助けになることがあります。

0x0102 それ以外に「プログラム」を実装するのに必要なこと。

今までエミュレーション対象であるファミコンの側からずっと見てきましたが、エミュレータもただのプログラムです。普通のプログラムと同じように、プログラミング言語を使って開発して、普通のプログラムと同じようにビルドして、普通のプログラムと同じように起動します。そこは同じです。

では、エミュレータに適した言語や、適した開発環境は何でしょう？

…実は**そんなもの無い**ので、お好きなもの、あるいは一番慣れてるものを使ってください。C++でもPythonでもJavaでも良いですし、前述の資料に使うとしたエミュレータは、JavaScript製でした。以下の条件を一応挙げておきますが、有名な言語なら殆どの言語で満たす条件です。

- 「手続き型」言語として書ける（関数型などでない）
- 画像や音声がある程度高速（秒間60回）に出力できる
- ユーザのキーボードやコントローラからの入力を受け取ることができる

今回私はC++を使いました。この言語では、SDL[1]というゲーム開発で有名なライブラリを使うことで、音声や画像の出力、ジョイパッドからのユーザ入力を簡単に行うことができます。

CでなくC++にしたのは、CPUやビデオといったそれぞれのパーツを"クラス"とし、クラスの隠蔽機能を使うことで、パーツごとの独立性を高めて見通しをよく出来ると考えたからです。また、カセットはたくさん種類があり（例えば、同じマリオでもマリオ１とマリオ３はカセットの種類―マッパーといいます―が違います）、それらをうまく表現するにはクラスの「継承」が使えると考えたからです。

ついでに言うと、C++で書かれた既存のエミュレータたちと速度の比較をしたかったので…（笑。

1 Simple Directmedia Layer; http://www.libsdl.org/

しかし、C++特有のテンプレートのような機能はエミュレータ本体では使っていませんし、C++の標準ライブラリなども殆ど使っていません。

この本はプログラミング入門の本ではないため、あなたがC++を普通に読めることを前提としてソースコードの解説を行なっていきますが、ご了承ください。でも多分JavaかC#が読めるなら、この本の中でのC++も普通に読めると思います。

0x02 設計は計画的に。

今までは文章だけでしたが、このセクションからは、ファミコンにターゲットを絞って、プログラムを本格的に組み上げていきます。…が、コードを書くのはもう少し先です…。

ソフトウェアの組み方には概ね、具体的なパーツから作って最後にくっ付ける「ボトムアップ」タイプと、全体の構造を決めて、それには何が必要で…とパーツの立ち位置を決めてからつくっていく「トップダウン」タイプの２つのタイプがありますが、対象についてあまり詳しくない場合は、トップダウンの方が時間はかかるけど、うまくいくことが多いように感じています。

私たちは、まだあまりファミコンの事を知りませんから、今回はトップダウン式で考えていきましょう。

0x0201 司令塔のCPUの立場から、全体の設計を考えよう

0x01で、どんなパーツがあるのかについては既に検討しました。この６つ（CPU、メモリ、カセット、ビデオ、サウンド、コントローラ）を、どのようにまとめ上げていくかを考えていきましょう。

コンピュータの中心部は、やはり司令塔たるCPUでしょう。このCPUから”ファミコンの他のパーツはどのように見えるのか”という視点から、考えてみましょう。

バスとメモリーマップドIO

ファミコンのCPUは「MOS 6502」という、Apple2にも使われた超有名な伝説のCPUです。詳しくは実装する時に解説するとして[2]、このCPUは基本的に8ビットのCPUです。つまり、6502は基本的に8ビットの0～255までの数値を一度に扱えます。…いや、「しか扱えない」と言ったほうが、64 bit CPUが普及した昨今はいいのかもしれませんね。ちなみに掛け算も割り算もできないので、足し算と引き算だけでそれらを自分で計算する必要があります。

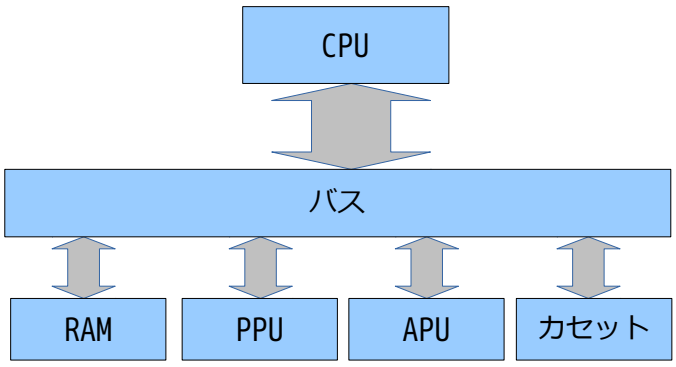


図 *1:*アドレスバス

そして、計算するときに使う値の場所を指定するためにアドレスが必要ですが、6502では16ビットです。つまり、ポインタ[3]の大きさが16bitということです。こちらも最近のCPUだと64bitですが、ファミコンはその1/4の大きさしかありません。最近のCPUは演算できるビット数とポインタのビット数がだいたい同じですが、この時代のCPUは演算するビット数とポインタのビット数が異なることはよくあったようです。

これらをすこしハードの視点に翻訳しましょう。この6502 CPUには、データを入力・出力するための8ビットのデータの通り道と、その場所（アドレス）を指し示すための16ビットのデータの通り道がつながっています。これらをそれぞれ「データバス」、「アドレスバス」と呼び、これらはだい

2 今のうちに知りたい方は、NESDEV本体（http://nesdev.parodius.com/）の6502のドキュメントや、NESon FPGAのCPUの項目を見ておく事をお勧めします。http://6502.org/なんてウェブサイトもあって、オリジナルのMOS 6502の発売当時のプログラミングガイドとかもあります。

3 ポインタは理解している、という前提で今の所書いてますが、エミュレータを作ることでポインタへの理解を深められることもあるでしょう。ポインタがいまいち分かってない人もいるかと思いますが、ここでは「ポインタ＝アドレス＝何らかの数値」ぐらいに思っておいてください。

たいセットで単に「バス」と呼びます。基本的に、6502はこのバスを通じて外部とつながります。

でも、まってください。CPUにはいろいろな部品がぶら下がっているはずです。メモリーとか、グラフィックを制御するPPUとか、サウンドを生成するAPUとか、コントローラとか、カセットとか。でもバスしか通り道が（これから紹介する割り込みを除いて）ないのに、どうやって？

その解決策が「メモリマップドIO」です。

アドレスは数値です。さっき言ったとおり16ビットなので、65536種類の場所を指定できます。

が、しかし、いわゆる「普通の変数」が格納されているRAMは2048バイトしかないので、全アドレス65536個のうち2048個だけあればRAMのどこに読み書きするのかの指定ができます。つまり、殆どあまってます。

アドレス	デバイス
0x0000-0x07ff	RAM
0x0800-0x1fff	RAMのミラーリング
0x2000-0x3fff	PPU（ビデオ）へのIOポート
0x2008-0x3fff	IOポートのミラーリング
0x4000-0x401f	APU（サウンド）その他へのIOポート
0x4020-0x5fff	特殊なカセットで使われる拡張RAM領域
0x6000-0x7fff	セーブデータ用RAM（存在すれば）
0x8000-0xffff	カートリッジ

表 *1:CPU*アドレスマップ

ファミコン（に限らずメモリマップドIOのマシン）では、この空きを有効活用します。表 1にあるとおり、RAMのアドレスは最初の方だけで、残りはPPUやAPUのような他のパーツに割り当てています。これはCPUからの視点で見ると、まるで「普通の変数」に読み書きしているように上記の他のパーツに割り当てているアドレスに読み書きすると、PPUなどの他の部品に司令を飛ばしたり、その結果を受け取ったりすることができるということです。

これを実現するために、ファミコンのハードウェアでは、それぞれのパーツに繋ぐためにCPUから出たバスはいろいろ分岐したりしています。わたしたちも、ソフトウェア的にそれを再現する必要があります。

具体的な設計としては、全体をまとめ上げるような存在のクラスを作って、そのクラスにバスの役割を担ってもらい、CPUが読み書きするデータの仲介を行ってもらいましょう。

割り込み

このメモリーマップドIOという方式では、情報をやりとりする主導権は、CPUにあります。つまり、CPUが思った時にCPUが各パーツに（特定のアドレスに書き込むことで）司令を与え、CPUが欲しいなと思ったときに各パーツから（特定のアドレスから読み込むことで）情報を貰うというスタイルです。

これはこれで便利なのですが、それぞれのパーツが自主的にCPUに情報を教えたい場合、例えばPPU（ビデオ）が、「画面が描き終わったよ」などとCPUに伝えたい場合、メモリーマップドIOだけでは不便です。もしもメモリーマップドIOだけだったら、CPUは画面が描き終わった事を知りたい場合、PPUに聞き続ける（PPUのアドレスから値を読み続ける）しかなく、CPUは他の仕事ができなくなってしまいます。これはとても不

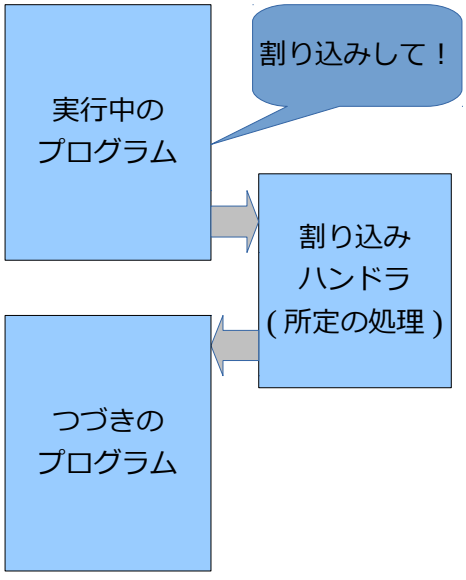


図 *2:*割り込みハンドラ

便です。

このために用意されているのが、”割り込み”（Interrupt）という機構です。Linuxに詳しい方だったら「シグナル」のようなものだと考えてください。PPUやAPUがバスとは別の電線を使って、CPUに「割り込みして！」というシグナル伝える事が出来ます。CPUがその割り込みシグナルを受け取ると、実行中のプログラムをいったん中断して「割り込みハンドラ(ISR; Interrupt Serivice Routine)」というプログラムを実行し始めます。このプログラムの中で、CPUはどうして割り込まれたのかを”とある方法”（それは、割り込みを実装するときにお話しましょう）で知り、その内容（例：画面が描き終わった）に応じた処理を行ってから、割り込みを起こしたパーツに「割り込みしたよ」とメモリマップドIOを通じて伝え（Acknowledge）、その後続きのプログラムを実行します。

メモリマップドIOとは少し違いますが、これも、同じように全体をまとめ上げるクラスに、仲介してもらいましょう。

時間

何のこっちゃ？と思われるかもしれませんが、つまり、CPUやその他のパーツが動作するのに掛かる時間のことです。実はこの部分はかなり気をつかわなければならないのです！

というのも、「PPUがとある処理をするのに掛かる時間は、CPUの＊＊命令N回分の時間だから、N回＊＊命令を実行して、その処理が終わるのを待機しよう」とやっているゲームが、**実に多い**のです！今ではCPUの命令実行時間が同じ命令でも変わる事もあり考えられませんが、昔は割と普通だったみたいです…。

CPUが司令塔ですから、このCPUが命令を実行するのに合わせて動く時計を作りましょう。そのCPU時計に合わせて、他のパーツが動いてくれるように設計するのです。これも、先述のまとめ上げるクラスにやって貰いましょう。

0x0202 クラス図を描いて、設計の最終段階！

CPUと他のパーツが一緒に動くための仕組みをいくつか見てきました。どうやら、CPUと他のパーツを協調して動作させるためのクラスを一番上に用意して、その下にCPUその他のパーツを並列で置いて、まとめ上げるクラスを仲介して動作させればうまくいきそうです。この仲介役をVirtual Machineクラスとし、各パーツはこのクラスのメンバオブジェクトとしましょう（図 3）。

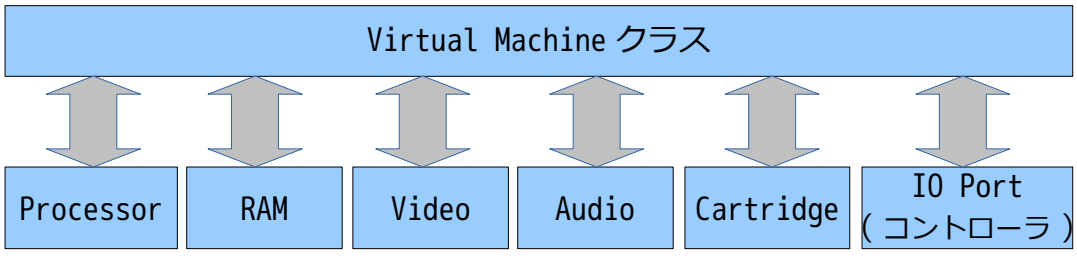


図 *3:*今回のクラス設計

各パーツの動作で、他のパーツへの干渉（バスを介した情報のやりとり、割り込みシグナルの送信、タイミングの同期）が必要になった場合、必ず上位の存在であるVirtual Machineクラスを経由させることで、各パーツの独立性を高め、デバッグしやすくすることを狙います。多少オーバーヘッドが有りそうですが、C++ですし、コンパイラの最適化がきっとうまくやってくれるでしょう。

さて、次の章からは、具体的にこれらを少しずつ実装していきます。やっとコードが！出るぞ～！

0x03 CDの中身をご紹介。

これから具体的にソースコードを見ていきます。という訳で、githubに置いてあるソースコードを参照しながら読み進めてみてください。流石に同人誌では全ソースコードを貼っていたらいくら紙面が有ってもたりないのです。。。よろしくお願いします。

ソースコード： https://github.com/ledyba/Cycloa

この本のPDF： https://github.com/ledyba/NesBook

0x0301 ファイル構成

次のようなフォルダ・ファイル構成となっております。

- docs/
  - この文章の、PDFとODFファイルが入っています。
- Cycloa/
  - エミュレータ本体のソースコードが入っています。
- bin/
  - 32bitのWindows用にビルドしたエミュレータのバイナリが入っています。

docs/に関しては直接中身を見ていただくとして、src/とbin/の中身をご説明します。

0x0302 ソースコードフォルダの構成

今回のソースコードは、Windows上ではMSYS/MINGW[4]とCMAKE[5]を用いてコンパイルすることができます（VisualStudio上でも可能なはずですが、試していません）。フォルダ構成はこんな感じです。

- cmake/ cmake用のファイル
- misc/, nacl/ Chromeで動かすとき用のビルドファイルなど
- src/

さらに、src内のソース本体は次のような構成です。

- Cycloa.cpp: main関数が入っており、ここから実行が始まります。
  - emulator/ エミュレータ本体です。
  - fairy/
    エミュレータ本体と、ビデオ・サウンドの出力やコントローラの具体的なライブラリをつなぐ、”妖精”たちが入っています。emulator/fairy/に入っている妖精クラスたちの、具象クラスです。現在はSDLのみ。

さらにemulator/以下を見ると、

- VirtualMachine.h
  - 各クラスの宣言が置いてあります。とりあえず、これを見ると良いでしょう。

4 MINGW http://www.mingw.org/
5 CMAKE http://www.cmake.org/

VirtualMachineクラスはこのヘッダの一番下に宣言されています。

- AudioChannel.h
  - オーディオの各チャンネル（矩形波、三角波 etc）クラスの定義です。
- VirtualMachine.cpp, Processor.cpp, Audio.cpp, Video.cpp
  - それぞれの実装が入っています。ここが心臓部♪　これらのクラスは、継承されません。
- Cartridge.cpp
  - カートリッジはゲームによって実装が変わるので、Cartridgeクラスの具象クラスとして実装されています。ここには、カートリッジごとで共通な処理の実装が入っています。
- file/
  - .nesファイルを読み込んで解釈するNesFileクラスの定義と実装が入っています。
- mapper/
  - ゲームごとによって異なるCartridgeクラスの具象クラスが入っています。
    しばらくはMapper0.h/cppだけ使うことになります。
- fairy/
  - ビデオやサウンドの出力、コントローラなどのユーザIOを抽象化する"妖精"たちの継承元になる、抽象クラスたちが入っています。
- exception/
  - エミュレータ実行中に起きる例外、EmulatorExceptionクラスの定義と実装が入っています。エミュレータそのものからは少し離れているので、無視しても構いません。

いきなりファイル構成を紹介されてもいまいち分からないと思いますが、大丈夫です。次章から、じっくりと解説していきます。

0x0303 ソースコードのコンパイル方法

このエミュレータは、前述でご紹介したライブラリ「SDL」を用いています。まずこのSDLをコンパイルした後に、MSYS/MINGWとCmakeを用いてビルドすることになります。

なお、コンパイラやCMakeそのものの導入は、この本はプログラミング入門本ではないので、取り上げません。ご了承下さい。とりあえず、それぞれで普通にC++のコンパイルができることを前提としています。そうでない場合は、Google等で検索して、環境構築しておいてください。

SDL ver2.0のインストール

SDLは、ゲームによく使われる有名なライブラリです。ビットマップ画像を高速に表示したり、音声を出力したり、ゲームパッドやキーボードからのユーザの入力を読み取る事ができます。

今回、このライブラリを使って、画像や音声の出力、コントローラの処理を行なっています。これらの処理は抽象化しエミュレータ本体からは独立しているので、あなたが望むなら、他のライブラリを使うように修正することも簡単に出来るでしょう。

SDLの現在の安定バージョンは1.2ですが、RC版の2.0が登場して既に数年が経過しています。機能が強化されている上に、AndroidやiPhone、iPadでも動かすことができるようになるようで、後々と

してはそれらで動かせると面白いな～と思ったので…。

SDLのsnapshotのサイト[6]からtarballを取得ができます。もしmercurialをお使いでしたら、最新のソースをmercurialを使って取得することもできます。

```
$ hg clone http://hg.libsdl.org/SDL sdl2
```

その後、ダウンロードしたフォルダに移動し、いつものように（？）、configure & make & make installでインストールすることができます。

```
$ cd sdl2
$ ./configure
$ make
$ make install
```

本体のビルド

本体はCMakeを用いてビルドします。

CDからCycloaフォルダをどこかにコピーし（CDの上にあると、CDには書き込めないのでビルドできません）、次のようなコマンドを叩いてください。

```
cmake -G "MinGW Makefiles" .
mingw32-make
```

エラーメッセージが表示されなければ、ビルド成功です。同じフォルダにCycloaSDL2.exeというファイルが出来上がっているはずです。

0x0304 エミュレータ「Cycloa（仮）」の使い方

さて、こうしてコンパイルしたエミュレータを、実際に動かしてみましょう。ROMとして、テスト用のROMを使いましょう（後々で、市販ゲームのROMを吸い出す幾つかの方法をご紹介します）。

Cycloaは最低限動けばいいよね、という基準で開発されているので、特に立派なGUIとかはありません。コマンドラインだけです。

SDLのライブラリSDL.dllと、エミュレータ本体のCycloaSDL2.exe、あとROMが有れば動かす事が出来ます。ROMの吸い出しに関しては後々という事にして、まずはネット上で公開されている、無料の自作ゲームを動作させてみましょう。

SDL.dllは、以下のディレクトリに出力されているはずです。

```
<SDLインストール先>/bin/SDL.dll
```

CycloaSDL2.exeはさきほど書いた通り、

```
Cycloa/CycloaSDL2.exe
```

にあります。

これらを同じフォルダにコピーして、次のように実行します。

```
$ CycloaSDL2.exe [ROMのファイル名]
```

あるいは、ROMをCycloa.exeにドラッグ＆ドロップすることでも実行できます。

あとはゲームを用意するだけです。色々な家庭用ゲーム機向けのフリーゲームがたくさん配布されているPD ROMS(URL：http://pdroms.de/)から、Alter Egoというゲームを DLしてみます。（URL：http://pdroms.de/files/nintendoentertainmentsystem/alter-ego）

6 http://www.libsdl.org/hg.php

DLして解凍し、ROMを指定して実行したのが図 4です。

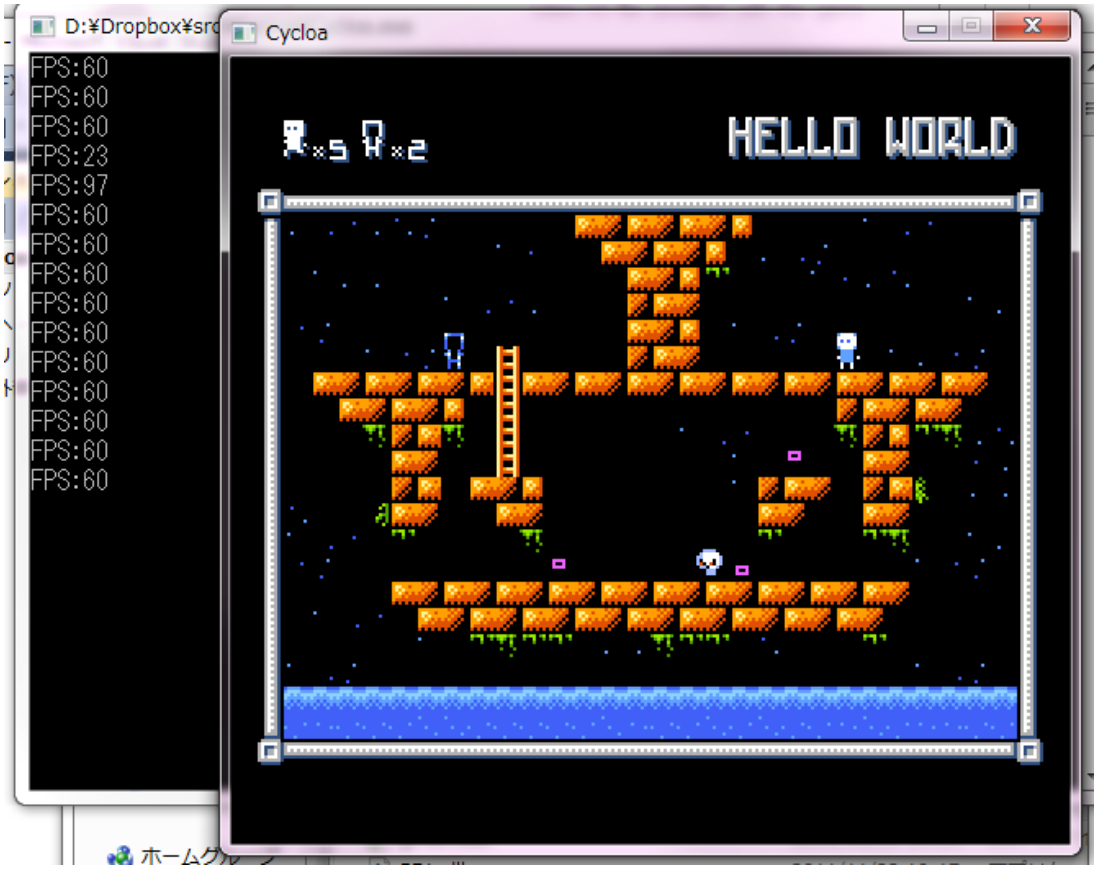


図 *4* 結構面白いんですよ、このゲーム。

ちゃんと動いてますね。操作は以下のとおりです。

- 上下左右：キーボード↑↓←→、ジョイスティック１のスティック１
- A：キーボードZ、ジョイスティック１のボタン１
- B：キーボードX、ジョイスティック１のボタン２
- スタート：キーボードA、ジョイスティック１のボタン３
- セレクト：キーボードS、ジョイスティック１のボタン４

操作キーの設定はソースに直書きで、Cycloa/src/fairysdl/SDLGamepadFairy.cppにあります。変えたい場合は、ここを直してコンパイルしなおしてください。

0x0305 ソースコードのライセンス

エミュレータのソースコードは、GPLv3（またはそれ以降）[7]とさせて頂きます。このライセンスに従う限りにおいて、あなたはこのエミュレータを自由に改造し、再配布する権利があります。もしももっと強力でCoolな”マホウ”に仕上がったら…ぜひ私にも教えてくださいね！

7 http://www.gnu.org/licenses/gpl.html

0x04 実装は丁寧に：まずは、バス。

では、ソースコードの準備も整いましたから、実際に開発していきましょう！

何度も書いている通り、コンピュータの司令塔はCPUです。これを最初に書くと見通しが良くなるので、まずは”最低限CPUを走らせること”を目標にしましょう。この部分はエミュレータの基礎であり、また比較的テストしやすい部分なので、何度も何度も丹念にテストして、バグの無いようにします。

CPUを動作させるのに最低限必要なのは、CPU本体と、RAMとカートリッジ、そしてそれらとCPUをつなぐバス部分です。これらの４つがあれば、画面も音も出ないし、入力もできないけれど、カートリッジからプログラムを読んでRAMを読み書きする、最低限の”コンピュータ”の完成になります。

0x0401 アドレスに応じたパーツへの振り分け

CPUはカートリッジやRAMから命令データを読み込み、それを順々に実行していきます。ということは、まずはそのデータを読み出せないとどうしようもありません。そこで、バスを、最低限必要な部分から作っていきましょう。

この機能は、src/emulator/VirtualMachine.h内に

- uint8 VirtualMachine#read(uint16 addr)
- void VirtualMachine#write(uint16 addr, uint8 val)

として実装してあります。速度を稼ぐためにインライン関数として実装しました。

```
inline uint8_t read(uint16_t addr) //from processor to subsystems.
{
 switch(addr & 0xE000){
  case 0x0000:
   return 0; //TODO: RAM
  case 0x8000:
  case 0xA000:
  case 0xC000:
  case 0xE000:
   return 0; //TODO: カートリッジ
  default:
   return 0; //TODO: まだたくさん残ってる
 }
}
```

以上が読み込み側です。同じように、書き込み側も作成します。読み込む場合とと書き込む場合で、同じアドレスでも宛先のパーツや機能が異なる場合があるので（後々出てきます）、関数は多少冗長になっても読み書きで違うものにします。

```
inline void write(uint16_t addr, uint8_t value) // from processor to subsystems.
{
 switch(addr & 0xE000){
  case 0x0000:
   // TODO: RAMへの書き込み
   break;
  case 0x8000:
  case 0xA000:
  case 0xC000:
  case 0xE000:
   // TODO: カートリッジ
   break;
  default:
   //TODO: まだたくさん残ってる
 }
}
```

アドレスの振り分けに使っているswitch文は、いったいどういう意味なのでしょう？アドレスマップ（表 1）の開始アドレスを、２進数にしてよく観察してみましょう（表 2）。

0xE000 = 0b1110000000000000とandを取ると、andを取った値のうち、赤字の部分と同じ桁のビットだけが残り、あとは全て0になります。これを踏まえて、各バスの開始アドレスの表を見ましょう。

各開始アドレスのうち、赤で示した上位3ビットだけを見ることで、割りと分類することができそうです。

0x0000と0x0800、0x2000と0x2008のような、ミラーリングアドレスと元のアドレスとは上位3ビットだけでは区別できませんが、後で見るとおり、実はこれらは区別する必要がありませんので、大丈夫です。

0x4000と0x4020も区別できていませんが、これは上位3ビットだけで振り分けてから、再度振り分けるほうが、全体としてはコードは綺麗になりそうです。

開始アドレス	内容
0x0000=0b0000000000000000	RAM
0x0800=0b0000100000000000	RAMのミラーリング
0x2000=0b0010000000000000	PPUへのIOポート
0x2008=0b0010000000001000	PPUへのIOポートのミラーリング
0x4000=0b0100000000000000	APU＆その他へのIOポート
0x4020=0b0100000000100000	拡張RAM
0x6000=0b0110000000000000	セーブデータ用RAM
0x8000=0b1000000000000000	カセットのデータ

表 2:開始アドレスの２進数表記

という訳で、上位3ビットだけ1にした0b1110000000000000=0xE000とアドレスをandして上位3ビットだけ取り出し、その値でswitch文で振り分けた、というわけです。

エミュレータでは、こういったビット演算が多用されます[8]。慣れておいてくださいね。

0x05 一番簡単なパーツ：RAM

つなぐ部分が出来ましたから、パーツを組み上げます。どれから作ってもよいのですが、値の読み込みと書き込みさえ出来ればよい、6つのパーツの中では最も単純なパーツであるRAMから作りました。

8 どうしてビット演算を沢山使う必要があるのかは…実は回路図を眺めるとわかります。

RAMへのアクセスはCPUの命令実行ごとに数回呼び出され、関数呼び出しのオーバヘッドが無視できないと思われるほど非常に多いので、こちらはすべてインラインで実装してみました。

src/emulator/VirtualMachine.h内に定義と実装があります。

```
class Ram
{
public:
	enum{
		WRAM_LENGTH = 2048
	};
public:
	explicit Ram(VirtualMachine& vm) : VM(vm){}
	~Ram(){}
	inline void onHardReset()
	{
		//from http://wiki.nesdev.com/w/index.php/CPU_power_up_state
		memset(wram, 0xff, WRAM_LENGTH);
		wram[0x8] = 0xf7;
		wram[0x9] = 0xef;
		wram[0xa] = 0xdf;
		wram[0xf] = 0xbf;
	}
	inline void onReset()
	{
	}
	inline uint8_t read(uint16_t addr)
	{
		return wram[addr & 0x7ff];
	}
	inline void write(uint16_t addr, uint8_t value)
	{
		wram[addr & 0x7ff] = value;
	}
protected:
private:
	VirtualMachine& VM;
	uint8_t wram[WRAM_LENGTH]; //2KB WRAM
};
```

親のVirtualMachineへの参照[9]のVMはRAMでは使っていないのでコンストラクタで引数として取る必要もないのですが、他のパーツでは必要になる[10]ため、一応デザインとして統一しておきました。

初期化はコンストラクタではなく、onHardResetと、onResetで行います。onHardResetは、「ファミコンの電源を付ける」動作に、onResetは「ファミコンのリセットボタンを押す」動作に対応します。処理を見ればわかる通り、実はこれらは似たようで違う挙動を示すようです。つまり、ハードリセット時はメモリの内容が0でクリアされ、一部アドレスに初期値が入るのですが、リセット時はメモリの内容はそのままです（だから、テニスとマリオを使った一部では有名な裏技[11]が成立します）。

9 C++にあまり馴染みの無い方へ補足すると、参照はCで言うところのポインタとほぼ同じです。
10 他のパーツに干渉する際に、親のVirtualMachineを経由させるデザインでしたね。
11 http://www.nicovideo.jp/watch/sm439824 バグ技で出る魅惑の２５６ワールドの動画

onHardResetでの挙動は、コメントに書いてある通り、NESDEV Wikiを参照して書きました。ファミコンの挙動に関する調べた知識を実際にコードとして実装する際、その参照元を必ず書いておく事をお勧めします。後々のデバッグ時になって「このコードは本当に正しいのかな？」となった時、再度その情報源を読んで確認することができるからです。

0x0501 ビット演算で割り算の余りを求めて「ミラーリング」

さて、次はこのパーツの一応の核心部であるread/writeを見て行きましょう。アドレスの値をそのまま使うのでなく、0x7ffとandした結果にアクセスしています。なぜでしょう？

0x7ffは2047、つまりメモリの大きさである2048-1です。2^{11}-1なので、二進数で表すと0b11111111111となり、すべて1になります。これとandをすると、2048で割ったあまりが出ます。

```
addr & 0x7ff == addr % 2048 //常に等しい
```

これが何故なのかは…2進数と10進数で考えれば分かります。andは、対応するビットがどちらも1なら1になる演算です。たとえば、11192と2047をandすることを考えましょう。

11192	0b10101110111000
2047	0b00011111111111
and	0b00001110111000

この結果をよく見て下さい。2047はずっと1が並んでいますから、andを取ると、2047で1が立っている下11桁の数字はすべて残り、0になっている12桁以降の数字はすべて0になります。2進数で考えると難しいかもしれませんが、10進数で考えると、どうでしょうか？23573の下３桁を残す処理は、具体的にはどのような処理でしょうか？

23573
***
253

はい、何のことはない、1000での割り算の余りです。このように考えれば、2047でのand⇔2進数で下11桁を残す処理⇔2048での割り算、というのが納得できるのではないでしょうか。

割り算は人間にとってもコンピュータにとって非常に重い処理なので、このように、できる限り高速なビット演算で代用します。実はそもそも、このようなビット演算はファミコン内の電子回路とまったく同じ方法だと言ってもよいです（ビット演算でなくて、12ビット目以降の値を無視している、と行った感じですが）。

さて、アドレスの2048での余りを取って、メモリにアクセスさせていました。

じつは、これがアドレスマップ（表 1、10ページ）に現れていた”ミラーリング”の正体なのです。

つまり、本当はRAMにはアドレス0x0000から0x7ffまででなく、0x0000から0x1fffまで割り当てられていて、この中で4回ループしている、というわけです（図 5）。

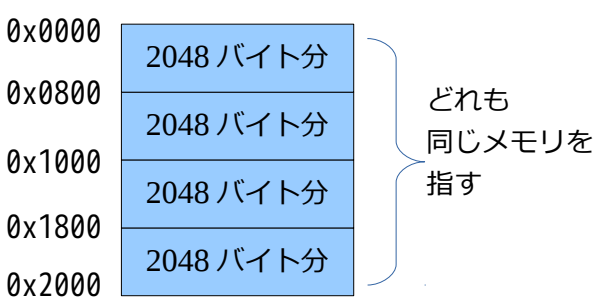


図 5:メモリのミラーリング

0x0502 最後に、バスで繋げ合わせる

というわけで、初期化と読み書き、そしてミラーリングを実装し、やっとRAMが完成しました…。というわけで、このRAMをアドレスバスとつなげて、完成としましょう。

```
inline uint8_t read(uint16_t addr) //from processor to subsystems.
{
 switch(addr & 0xE000){
  case 0x0000:
   return ram.read(addr); //追加
  case 0x8000:
  case 0xA000:
  case 0xC000:
  case 0xE000:
   return 0; //TODO: カートリッジ
  default:
   return 0; //TODO: まだたくさん残ってる
 }
}
```

```
inline void write(uint16_t addr, uint8_t value) // from processor to subsystems.
{
 switch(addr & 0xE000){
  case 0x0000:
   ram.write(addr, value); //追加
   break;
  case 0x8000:
  case 0xA000:
  case 0xC000:
  case 0xE000:
   // TODO: カートリッジ
   break;
  default:
   //TODO: まだたくさん残ってる
 }
}
```

さて、次はカートリッジです。

0x06 とりあえず仮で、動けばいいや：カートリッジ

0x03で動かしたように、カートリッジは「.NES」という拡張子のファイルを読み込みます。まずはこのファイルと、現実のカートリッジとの対応関係を考えましょう。

0x0601 ファミコンのROMファイルって、どうなってるの？

ファミコンのカセットの中には、ROMが入っています。Read Only Memoryの名が示すとおり、読み込みだけで、書き込みのできないメモリーチップです。これはいわゆる「カートリッジ」で動くゲーム機で共通する特長ですが、さらにファミコンにはこのROMに二種類あるという特徴があります。

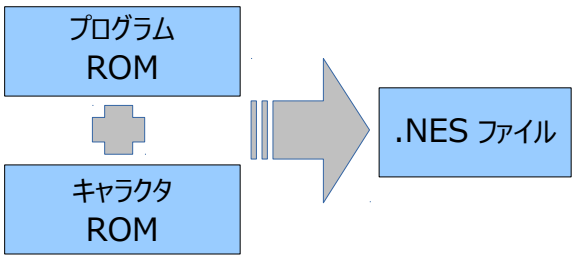


図 *6:.NES*ファイルの構成

それは「プログラムROM」と「キャラクタROM」と呼ばれています。名前が示すとおり、プログラムROMにはCPUの実行するプログラムが、キャラクタROMには、画面上に表示される画像データが、それぞれ格納されています。

パソコン上では一つのファイルである.NESファイルは、この２つのROMの中身を統合したものなのです（図 6）。

カートリッジの基板の上では、ファミコン本体とつなぐバスや物理的なチップが、それぞれのROMで独立して存在しています。メモリーマップ（表 1、10ページ）上の「カートリッジ」とあった部分は、実はプログラムROMだけで、キャラクタROMはPPU（ビデオ）がCPUとは別に持っているバスと繋がっています。PPUのバスはCPUから直接は利用できず、CPUのバスからPPUにアクセスし、PPUを経由するという間接的な方法でしか、キャラクタROMにアクセスできません。

そういえばドラクエⅢなどに搭載されている、セーブデータ用のRAMはどうなっているのでしょう？

このRAMもプログラムROMやキャラクタROMとは別のチップですが、キャラクタROMと共通のバスを利用しています。そのため、CPUからアクセスでき、アドレスマップに「セーブデータ用RAM」領域として存在しているのです。

0x0602 .NESファイルにも、フォーマットがある

ファミコンのROMが２つの独立した部分から成り立っていて、それらの異種のデータを一つのファイルに纏めている以上、.NESファイルにも所定のファイルフォーマットがあります。それが、iNES[12]フォーマットと呼ばれているフォーマットで、この業界（？）のデファクトスタンダードです。

エミュレータ界隈で生まれた野良フォーマットなので、「標準」とかは基本的にありません。自分でゲームデータをカートリッジから吸い出して、自分のエミュレータで走らせるだけなら、自分でiNESフォーマットに変わるものを開発してしまって、全然構わないのです。

でもまあ、自分で考えるのも大変ですし、PDRomsのような自作ゲームは大抵iNESフォーマットでリリースされているので、私達のエミュレータもこのフォーマットを採用してしまいましょう。

正確な仕様書のないフォーマット

このiNESフォーマット、標準化機関がもちろん標準化したりはしていません。また、オリジナルのiNESフォーマットをいろいろな人が勝手に拡張してしまっているので、「これ！」という仕様書はありません。ので、フォーマットが解説されている誰かの書いた”仕様書”を、私たちのエミュレータで必要な程度に、適当に実装しておけば、まあ大丈夫です（ぇ。

今回は、Everynes[13]の仕様書と、NESDEV Wikiのフォーマットを解説した項目[14]を適当に参照しながら実装しました。

そのパーサーとなるクラスNesFileは、カートリッジを表すクラスCartridgeとは別に、Cycloa/src/emulator/file/NesFile.h、同NesFile.cに定義・実装しました。後々ビデオやサウンドを実装するときにわかるのですが、カートリッジは単にROMデータを持っている他にもいくつか役割があります。それらをうまく抽象化するために、ファイルの内容を持っている部分と、カートリッジ本体は分けて実装することにしました。

12 http://fms.komkon.org/iNES/ このエミュレータの作者が考えたフォーマットだそうです。
13 http://nocash.emubase.de/everynes.htm#cartridgeromimagefileformats
14 http://wiki.nesdev.com/w/index.php/INES

ソースコード自体はバイナリデータをパースするだけなので、特に解説する点はありません。iNESフォーマットは、最初の10バイトがヘッダで、その後に具体的なデータが格納されているデータです。

この中に、仕様書に示してあるとおり、いくつかフラグやパラメータが記述されています。以下に、それらの値について、軽く説明します。

- マッパー番号（Mapper Number）
  ファミコンのカートリッジの中身は、ソフトによって違う事があります。例えば大容量のROMを積んでいたりする場合です。それらを区別するための番号がこちらです。
  当面は一番簡単なMapper0を使うでしょう。後で詳しく解説します。
- Four-screen VRAM Layout
- Horizontal Mirroring/Vertical Mirroring
  この２つは、PPU（ビデオ）を実装する際に詳しく検討しましょう。ビデオ用のメモリにも、前実装したRAMのようなミラーリングがあり、それを指定するフラグです。
- Battery-packed SRAM
  0x6000-0x7fffのセーブデータ領域に現れる、SRAMが存在するかどうかのフラグです。このフラグが立っている場合は、この領域が読み書きできて、電源を切ってもデータが残る、と言うことになります。が、今回のエミュレータでは、保存するのが面倒なのでエミュレータを終了させるとセーブデータも消えてしまいます。
- Trainer
  その昔、まだエミュレータの完成度が低かった頃、エミュレータで市販ゲームをうまく動作させるために、ROMへの修正が必要だった時代があり、その修正パッチをTrainerと呼んでいたようです。自分で吸い出したROMや、自作ゲームを遊ぶにはいらない項目です。元のiNESフォーマットにもあるようなので、今回の実装でも一応解釈していますが、使っていません。

他は後々に付け加えられた拡張的な部分なので、無視でいいと思います。

0x0603 マッパーって？

さて、iNESのマッパー番号で表される”マッパー”は、カートリッジの種類のことでした。なぜ、複数種類のカートリッジの基板が必要なのでしょう？普通に考えると、一種類に統一してしまえば量産が効いて安くなるし、技術的にもスッキリするのではないでしょうか？

これも、メモリーマップ（表 1、10ページ）を見ながら考えていけばわかります。プログラムROM領域は0x8000から0x7fffで、サイズは32KBです。ファミコンの時代を考えると結構多いように見えます。

たしかに、マリオやサッカーなど、初期のゲームはその程度の容量でも間に合っていたのですが、ファミコンの成熟によってソフトに必要な容量が増えてきました。例えばDQ4は512KBもあります。伝説のクソゲー「チーターマンⅡ」の乗っているAction53のROMは2MBくらいあるそうです。

32KBしか載せられないはずのメモリ空間に、どうして大容量が実現できるのか？

答えはこうです：「同じメモリアドレスが常に

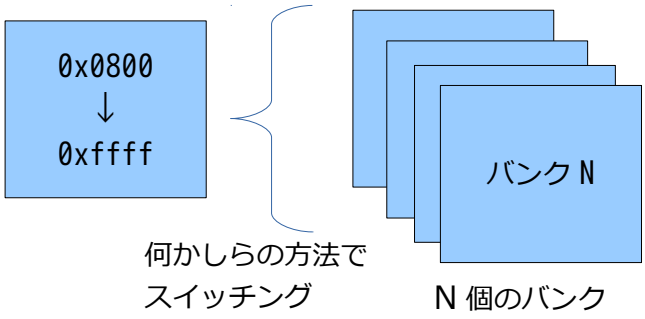


図 7:バンク切り替え

カートリッジ内の同じ場所を指している必要はない」。

メモリコントローラと呼ばれるパーツをCPUとカートリッジの間に載せて、同じメモリ・アドレスに、カートリッジの別の場所を割り当てて、必要に応じて実際にはどこにアクセスするのかを切り替えられるようにしてしまいました（エラー: 参照先が見つかりません。）。これを「バンク切り替え」と呼びます。32KBの領域をN個のバンクで切り替えれば、N×32KBものの大容量が使える、というわけです。

…が、たしかに増やせるのですが、同時に扱うことが出来るデータはやはり32KBしかありません。このバンク切替えはプログラムROMだけでなく、キャラクタROMでも使われるのですが、キャラクタROMは画面表示に使われ、グラフィックはCPUとは別に独立して動作するため、「同時」の条件がプログラムROMよりシビアになります。

このバンク切替方式も統一すれば良いのに…と思うのですが、実際には各社が様々なチップ（任天堂ですら、何種類も出しています）を開発し、ファミコンの様々なゲームに使われました。

特殊カセットは「バンク切り替え」だけのためのものではありません。実はカセット端子には音声入力・出力もあり[15]、もともとのファミコン音源を超える綺麗な音を表現するために、カセット内に独自の音源チップを積んでいる場合があります（「ラグランジュポイント」のVRC7は特に有名です！）。

さらに独自のタイミングで割り込みを発生させるための端子もあり、この割り込みを利用して、綺麗な画面効果（ラスタースクロールなど）を実現しているソフトも、結構あります。

いろいろな特殊カートリッジ（マッパー）があることが分かって頂けたと思いますが、当面は私たちはマッパー0のみを扱いましょう。このマッパーは、ROMチップをカートリッジ上にただ単に載せただけのマッパーで、プログラムROMは32KB、キャラクタROMは8KBのメモリ空間の上限までしか載せることができません。

0x0604 マッパーの抽象化は考えずに、マッパー0だけ実装しておく。

カートリッジは、思いの外沢山の機能を担っていることがわかりました。となると、あまりファミコンハードに詳しくない私達には、いきなりカートリッジの抽象化を行うのは、ちょっと荷が重いですね。まずは、マッパー0、しかも、CPUを動かすのに必要な、CPUメモリからアクセスされる部分だけを実装しましょう。

とりあえず、ヘッダでの定義はこんな感じにしてみました。

```
class Cartridge
{
      public:
            Cartridge(VirtualMachine& vm, const NesFile* nesFile);
            virtual ~Cartridge();

            /* for CPU */
            uint8_t readBankHigh(uint16_t addr);
            void writeBankHigh(uint16_t addr, uint8_t val);
            uint8_t readBankLow(uint16_t addr);
            void writeBankLow(uint16_t addr, uint8_t val);
            inline uint8_t readSram(uint16_t addr) const
            {
                  if(hasSram){
                        return this->sram[addr & 0x1fff];
```

15 http://crystal.freespace.jp/pgate1/nes/nes_cart.htm

```
                }else{
                        return 0;
                }
        }
        inline void writeSram(uint16_t addr, uint8_t value)
        {
                if(hasSram){
                        this->sram[addr & 0x1fff] = value;
                }
        }
    //TODO: もっとたくさんの関数が後々に必要になる
    protected:
        const NesFile* const nesFile;
    private:
        VirtualMachine& VM;
        bool hasSram;
        uint8_t sram[SRAM_SIZE];
        const uint16_t addrMask;
};
```

セーブ用のRAMは、ROMデータを表すNesFileとは分離して持つことにしました。もしNesFileのSRAMを持っているフラグが立っている場合、このカートリッジ内のhasSramもフラグが立ち、SRAMへの読み書きが有効になるというわけです。もっと面倒な場合は、SRAMフラグの有無にかかわらずSRAMを使用可能にしてしまっても、大丈夫だとは思います。

さて、今度は実装です。

```
Cartridge::Cartridge(VirtualMachine& vm, const NesFile* nesFile) :
    nesFile(nesFile),
    VM(vm),
    hasSram(nesFile->hasSram()),
    // 16KBなら、同じ内容が繰り返される
    addrMask(nesFile->getPrgPageCnt() > 1 ? 0x7fff : 0x3fff)
{
    if(nesFile == NULL){
        throw EmulatorException("NES FILE CAN'T BE NULL!");
    }
}
Cartridge::~Cartridge()
{
    if(this->nesFile){
        delete this->nesFile;
    }
}


/* for CPU */
uint8_t Cartridge::readBankHigh(uint16_t addr)
{
    return this->nesFile->readPrg(addr & addrMask);
}
void Cartridge::writeBankHigh(uint16_t addr, uint8_t val)
{
    //have no effect
```

```
}
uint8_t Cartridge::readBankLow(uint16_t addr)
{
      return this->nesFile->readPrg(addr & addrMask);
}
void Cartridge::writeBankLow(uint16_t addr, uint8_t val)
{
      //have no effect
}
```

プログラムROMの読み書き関数が二つに分かれていますが、これは先述のJavaScriptでのエミュレータを参考にしてのことで、0x8000-0xbfffの前半がLow、0xc000-0xffffの後半がHighの関数の範囲です。最後まで作って分かりましたが、このようにするとすんなりと書ける時もあったり無かったり、です。一つにまとめても良いと思いました。

マッパー0はプログラムROMが16KBのときと、32KBのROMの時があります。32KBの場合はアドレス領域をすべて使い切るのですが、16KBの時は、16KBの内容が二回ループします。そう！RAMのミラーリング（19ページ）と同じです。nesFile->getPrgPageCnt()は、iNESファイルで16KBごとのカウントである「ページ数」を返すので、この値を使ってaddrMaskを動的に切り替え、これでミラーリングの時のようにand演算を使って割り算を行い、「ミラーリング」させています。

カートリッジとしての機能が殆ど有りませんが、まずはこれで十分でしょう。

最後に、バスでつなげます。

```
inline uint8_t read(uint16_t addr) //from processor to subsystems.
{
 switch(addr & 0xE000){
  case 0x0000:
   return ram.read(addr);
    case 0x6000:
          return cartridge->readSram(addr); //追加
    case 0x8000:
    case 0xA000:
          return cartridge->readBankLow(addr); //追加
    case 0xC000:
    case 0xE000:
          return cartridge->readBankHigh(addr); //追加
  default:
   return 0; //TODO: まだたくさん残ってる
 }
}
inline void write(uint16_t addr, uint8_t value) // from processor to subsystems.
{
 switch(addr & 0xE000){
  case 0x0000:
   ram.write(addr, value);
   break;
    case 0x6000:
          cartridge->writeSram(addr, value); //追加
    case 0x8000:
    case 0xA000:
          cartridge->writeBankLow(addr, value); //追加
          break;
    case 0xC000:
    case 0xE000:
```

```
            cartridge->writeBankHigh(addr, value); //追加
            break;
    default:
      //TODO: まだたくさん残ってる
  }
}
```

ふ〜。CPUを動かすための舞台装置が、これで完成です。

0x07 一番重要だけど、一番素直なパーツ：CPU

というわけで、やっと司令塔であるCPUです！CPUが一番難しそうなイメージがありますが、CPUを再現するには、少数のルールを正確に実装すれば、実は単純作業に成り下がってしまいます。あんまり気負わずに、気楽にやっていきましょう。

0x0701 CPUって、何から出来てる？

すごく基本的な話から始めましょう。CPUは私たちが買うときは1つのチップとして販売されていますが、このCPUは、いったいどのようなパーツからできあがっているのでしょう？

エミュレータを作るという観点からみたCPUは、以下の二つの部分から出来ています。

- 命令解釈部
- レジスタ

命令解釈部は、その名の通り、いわゆる機械語を解釈してどういうプログラムが書かれているのかを理解する部分です。レジスタは、途中の計算結果を格納したり、現在のCPUの状態を表すために使う、せいぜい数十バイトの小さな小さなメモリです。

0x0702 CPUの基本的な動作

CPUの基本動作は、

1. プログラムカウンタという特殊レジスタの示すアドレスから、命令を表す数値を読み込む
2. 命令に応じて、CPU内のレジスタ、あるいはバスを通じてメモリを読み書き・計算する
3. 1に戻って繰り返す。割り込みがあったら所定の手続きでISR（割り込みハンドラと呼ばれるプログラム）の実行に移る

これの繰り返しだけです。もしも8文字だけでプログラムを書ける超簡易言語Brainfuck[16]をご存知なら、その規模を大きくしただけ、と言っても過言ではありません。Brainfuckのインタプリタの実装の経験がお有りなら、きっとすんなりとCPUのエミュレーションも出来るでしょう。

0x0703 ファミコンのCPU、6502のもつレジスタ

ファミコンのCPUは、MOS 6502でした。6502を表すクラス、Processorのメンバを考えるために、まずはこのCPUの持つレジスタを知りましょう。

こんな感じです[17]。

- A（8ビット）：アキュームレータ。途中の計算結果を格納する。
- X,Y（それぞれ8ビット）：インデックスレジスタ。アドレスの計算に使う。
- PC（16ビット）：現在実行中の命令のアドレスを示す。
- SP（8ビット）：
  CPUが持つスタック（0x0100->0x01ff）の、一番下のアドレスの下位8ビットを示します。

16 http://www.kmonos.net/alang/etc/brainfuck.phpや、Wikipediaなど参照。
17 http://crystal.freespace.jp/pgate1/nes/nes_cpu.htm

- P(8ビット)：
  それぞれのビットがCPUの動作状態を表すフラグです。NVBDIZCの7つのフラグがあります。

各命令の実行ごとに、これらのレジスタ、あるいはバスを通じてメモリが書きかえられていくのが、6502というCPUの基本的な動作と言うことになります。

以上を使って、基本的なProcessorクラスの定義を作ったのがこちらです。

```
class Processor
{
	public:
		explicit Processor(VirtualMachine& vm);
		~Processor();
		void run(uint16_t clockDelta);
		void onHardReset();
		void onReset();
	protected:
	private:
		inline uint8_t read(uint16_t addr); // VMの同関数を呼ぶラッパ
		inline void write(uint16_t addr, uint8_t value); // VMの同関数を呼ぶラッパ
		//定数
		enum{
			FLAG_C = 1,
			FLAG_Z = 2,
			FLAG_I = 4,
			FLAG_D = 8,
			FLAG_B = 16, //not used in NES
			FLAG_ALWAYS_SET = 32,
			FLAG_V = 64,
			FLAG_N = 128,
		};
		VirtualMachine& VM;
		uint8_t A;
		uint8_t X;
		uint8_t Y;
		uint16_t PC;
		uint8_t SP;
		uint8_t P;
};
```

ハードリセット・ソフトリセット時の処理はNESDEV WikiやNES on FPGAの各ページ[18]にあるので、この通りに実装しておきました。Processor.cppに実装があります。

ステータスレジスタBは、本来の6502では、「二進化十進表現」計算モードであるか否かを判別するフラグなのですが、ファミコンの6502ではこの機能は不要とされ、削除されています（フラグをセット・解除してモード変更は形式上できるのですが、実際にはモードは変わりません）。

0x0704 機械語の読み方

というわけで、次に機械語を読み取って実行する、命令解釈部の実装に移りましょう。6502の機械語は、１つだけ取り出すと次のような姿をしています。

18 http://wiki.nesdev.com/w/index.php/CPU_power_up_state
http://crystal.freespace.jp/pgate1/nes/nes_cpu.htm

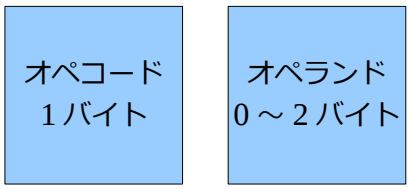


図 *8:6502*機械語

…なぜ図にしてしまったのか、後悔するほどに簡単です。各命令は、「どんな命令か」、つまり私たちの自然言語で言えば、「動詞」を表すオペコード１バイトと、「どの対象に向けての命令か」、つまり「目的語」を表すオペランド0～1つ（有る場合は1バイトか2バイトの可変長）を取ります。

抽象的で分かりづらいですね。具体的な例を見ましょう。

```
0x69 0x44
```

これを、もう少し分かりやすい、「アセンブリ言語」で表すと、こんな感じ。

```
ADC #$44
```

この2バイトの最初の1バイト目「0x69」でこの命令が「ADC命令」である事を表します。ADC命令は、オペランドの表す値を、プロセッサのAレジスタに足す、という命令です。この場合は、機械語に直接書かれた「0x44」をそのままAレジスタに足す、と言うことになります。

最初の１バイト目0x69が「ADC命令」を表すと書きましたが、この0x69というバイトは、この例のように、機械語に直接書かれた値（「イミディエイト」と表現します）を足すADC命令を表す場合の数値で、同じADC命令でも、最初の1バイト目の数値は違う場合があります。その場合、Aレジスタに足される値が何なのかが変化します。次の例を見ましょう。

```
0x7D 0x00 0x06
```

これを同様にアセンブリ言語で表記すると、

```
ADC $0600,X
```

これはこんな意味です（よく分からなかったらこの行は飛ばしてください）：「二バイト目と三バイト目を”リトルエンディアン”、つまり２バイト目を下位8ビット、３バイト目を上位8ビットとする16ビットの値として解釈し（＝$0600）」、「それにプロセッサのXレジスタを足した値のアドレス（例えばX＝0x78なら、0x0678）を使ってバスから**値を読み出し**」、**「その値をAレジスタに足す」**。

前の例に比べると非常に複雑になりましたが、「どこからか持ってきた値を」「Aレジスタに足す」という点では、共通しています。違いは、その値を機械語の値を直接使うのか、機械語からの値を足し引きして持ってきたものを使うかどうか、です。この「どこから持ってくるかの違い」を、**アドレッシングモード**と呼びます。同じADC命令でも0x69と0x7Dと1バイト目が違うのは、このアドレッシングモードの違いを表しているのです。

0x0705 6502のアドレッシングモード

さて、このようなアドレッシングモード、私たちの実装する6502には、「オペランドなし」も含めると11個あります。

一応列挙するとこんな感じ。詳しく解説してもしょうがない（本当に書かれてるとおりにするだけ）ので、NES on FPGAのCPUの項目[19]に解説は譲ります。

- オペランドなし：cli（iフラグクリア）など、オペランドを指定する必要のない命令

19 http://crystal.freespace.jp/pgate1/nes/nes_cpu.htm

- Immediate、イミディエイト：機械語に書かれたオペランドの8ビット値そのまま
- Zero Page、ゼロページ：上位8ビット0x00、下位8ビットが機械語に書かれたオペランドの8ビット値であるアドレスを使う（つまり、範囲は0x0000->0x00ffになります）。6502の超有名機能。
- Absolute、絶対番地：オペランドの示す16ビットのアドレスを使う
- Zero Page X、Zero Page Y：
  先述のゼロページでのアドレスに、XレジスタかYレジスタを足してつかう。
- Absolute X、Absolute Y：
  先述の絶対番地に、XレジスタかYレジスタを足して使う。
- Relative: 分岐命令で使う特殊アドレッシングです。
- Indirect X：ゼロページXのアドレスから値を連続で2バイト、リトルエンディアンで読み込み、その2バイトをアドレスとして使います。
- Indirect Y：ゼロページの値から2バイト、リトルエンディアンで読み込み、その2バイトにYレジスタを足したものを使います。

いまいち分からない？大丈夫です！先述のNES on FPGAに自然言語での説明が、そしてNESDEVドキュメント[20]には実例を交えて詳しく解説されたものがあります。これを読んだままをソースコードに表現すれば、大丈夫です。

0x0706 オペコードとオペランドを、分離して実装する

さて、同じ「足し算」の命令でも、アドレッシングモードによって、動作が変わることが分かりました。私たちのエミュレータでも、同じように、「命令」と「アドレッシングモード」の部分を分けて実装することで、同じようなコードが連続するのを避け、見通しよく実装しましょう。

各命令は、アドレッシングモードによって得た16ビットのアドレスを取って動作する、一種の関数であると見なす事ができます[21]。この関数としての命令の動作について、疑似言語を用いた非常に詳しい解説が、NESDEVのドキュメントにあります[22]。これを見れば、各命令についてほぼ迷いなく実装することができます。これはProcessor#ADC(const uint16_t addr)などをご覧ください。

アドレッシングモードは、今度はPCレジスタから命令を読み取って加工したアドレスを返す、関数とみなせます。こちらはuint16_t Processor#addrImmediate()などに実装しました。

さて。あとは**これらを組み合わせて、各命令を解釈＆実行する**だけです！

0x0707 オペコード分岐255個をswitch文で。ぶっちゃけ、根性。

では、命令の具体的な解釈は、どう実装すれば良いのでしょう？

各命令の最初の1バイト目のオペコードによって、命令の種類と使用するアドレッシングモードが区別されているとお話しました。一番てっとり早いのは、これを全部そのまま、switch文で振り分けて

---

20 http://nesdev.parodius.com/opcodes.txt 日本語
アドレッシングモードはわかりやすいけど、その他はあまり参考にならないです。

21 イミディエイトはどうするのかって？
イミディエイトの値自身が置かれているアドレスを指すアドレッシングモードだと解釈しましょう。

22 http://nesdev.parodius.com/6502.txt

しまうことです。デバッグも簡単ですし、数は多いですが**気合いでなんとか**なります。

先述のドキュメントの中盤に、この対応があるので、これをエディタなどで加工しつつ、255個のswitch文に書き換え、他の処理を加えたのが、こちら。

```
void Processor::run(uint16_t clockDelta)
{
   this->P |= FLAG_ALWAYS_SET;

   const uint8_t opcode = this->read(this->PC);
   #define CPUTRACE // 命令のトレースログを出力
   #ifdef CPUTRACE
   char flag[9];
   flag[0] = (this->P & FLAG_N) ? 'N' : 'n';
   flag[1] = (this->P & FLAG_V) ? 'V' : 'v';
   flag[2] = (this->P & FLAG_ALWAYS_SET) ? 'U' : 'u';
   flag[3] = (this->P & FLAG_B) ? 'B' : 'b';
   flag[4] = (this->P & FLAG_D) ? 'D' : 'd';
   flag[5] = (this->P & FLAG_I) ? 'I' : 'i';
   flag[6] = (this->P & FLAG_Z) ? 'Z' : 'z';
   flag[7] = (this->P & FLAG_C) ? 'C' : 'c';
   flag[8] = '\0';
   printf("%04x op:%02x a:%02x x:%02x y:%02x sp:%02x p:%s IRQ?:%s NMI?:%s\n", this->PC, opcode, this-
>A, this->X, this->Y, this->SP, flag, IRQ ? "on" : "off", NMI ? "on" : "off");
   fflush(stdout);
   #endif
   this->PC++;

   switch(opcode){
      case 0x00: // BRK
         this->BRK();
         break;
      case 0x01: // ORA - (Indirect,X)
         this->ORA(addrIndirectX());
         break;
      //case 0x02: // Future Expansion
      //case 0x03: // Future Expansion
      //case 0x04: // Future Expansion
      case 0x05: // ORA - Zero Page
         this->ORA(addrZeroPage());
         break;
            /* 中略 */
      default:
         throw EmulatorException("[FIXME] Invalid opcode!");
   }
   consumeClock(CycleTable[opcode]);
}
```

割り込みは複雑なので、まだ実装していません。

フラグPは必ずセットされる仕様だそうなので、一応セットしておきました。無くてもたぶん動きます。その次に命令の実行ログを出力していて、これは、次の項でテストに使います。

プログラムカウンタの進め方

命令を正確に実行するには、１命令を実行するごとに、正確にその命令分、プログラムカウンタを追加する必要があります。おさらいすると、オペコードが１バイトで、オペランドが0〜2バイトでしたから、命令ごとに1〜3バイト分加算する必要があります。すべての命令でオペコードとして必ず１バイト使う分は、共通で加算しておきます。デバッグメッセージ出力の次の行です。

残りのオペランドの１〜２バイトは、アドレッシングモードだけに依存して変わるので、各アドレッシングモードの関数内で加算します。例えば、インダイレクトXはこのような感じ。

```
inline uint16_t Processor::addrIndirectX()
{
     uint8_t idx = read(this->PC) + this->X;
     this->PC++;
     uint16_t addr = read(idx);
     idx++;
     addr = addr | (read(idx) << 8);
     return addr;
}
```

インダイレクトXは、オペコードの次の１バイトとXレジスタを組み合わせて値のアドレスを計算するので、オペランドの１バイト分を二行目で加算しています。１行目でread(this->PC++)とすると行を分ける必要もなく、これはこれでスッキリするのですが、X++と++Xの区別はたまに間違えるので、今回はやりません。

クロック数＝実行時間を正確に記録しよう

最後のconsumeClock()が気になりますね。各命令や各アドレッシングモードで掛かる命令クロック数をカウントして、親のVirtual Machineに伝える関数です。これも各命令ごとに異なります。加算されるクロック数はアドレッシングモードだけでなく、オペコードごとに異なるので、すべてのオペコードについてクロック数を計算してあるテーブルを用いて、加算しています。

基本的には同じオペコードなら同じクロック数なのですが、たまに時間が異なります。その分はクロック数の延長が発生する部分で個別に足してます。例えば、IndirectYを見てみましょう。

```
inline uint16_t Processor::addrAbsoluteIdxY()
{
  uint16_t orig = read(this->PC);
  this->PC++;
  orig = orig | (read(this->PC) << 8);
  this->PC++;
  const uint16_t addr = orig + this->Y;
  if(((addr ^ orig) & 0x0100) != 0){
    consumeClock(1);
  }
  return addr;
}
```

インダイレクトYでは、PCに続く２バイトをアドレスとして解釈し、さらにそのアドレスにYレジスタを足した結果が、最終的なアドレスとなります。このYを足した時に、アドレスの最初の１バイトが増える”繰り上がり”が発生していると、内部回路の都合で実行時間が１クロックだけ増えてしまいます。これを、「(addr ^ orig) & 0x0100) != 0」という部分で検出しています。

^演算子は”xor” と呼ばれるもので、ビット同士を比較して異なっていたビットだけが1になります。

さて、もし、繰り上がりが起こっていた場合、Yレジスタは1バイト、0から255までの値を取りますので、アドレスの上位１バイトは最高でも1しか増えません。例えば、0x10ffに0xffを足しても0x11feとなり、１しか増えていないと確認できます。

上位１バイトが高々１しか増えないということは、繰り上がりが発生していた場合、上位１バイトの一番下の1ビットは必ず書き換わります。

```
0x10ff = 00010000 11111111
0x  ff =          11111111
0x11fe = 00010001 11111110
```

この変化を調べるために、まず足す前と足した後をxorすることで、変化したビットを調べます。

さらに調べたいビットだけに1を設定した0x0100とandを取ると、調べたいビットが変化していたときだけ、0以外の値になり、これで繰り上がりで上位１バイトの一番下の1ビットが変化したかを調べられる…というわけです。

それをまとめたのが、この「(addr ^ orig) & 0x0100) != 0」という条件式です。

0x0708 ネット上のテストROMを使って、テストしよう。

さて、このCPUエミュレータを延々テストしましょう。NESDEVには、なんとCPUのエミュレータをテストするための専用のROMが配布されています[23]！流石は有名なファミコンです。

このROMを0xC000から実行して（onHardResetあたりに、PCを0xC000で初期化するコードを一時的に書きましょう）、その動作ログを、同ページからDLできる正しい動作ログと見比べて、少しでも違いがあればバグだよ、というわけです。**…でも、その動作ログ、結構間違ってます。**他のエミュレータと多数決とった結果と違うし、CPUの仕様書と比べても変な挙動なところが散見されます。

というわけで、github.com/ledyba/NesBook/doc/nestest_trace.logに、FCUEXというエミュレータから私が取った（たぶん）正しい動作ログがあるので、これを使ってください。

これと、先述の #ifdef CPUTRACE以下に書いたデバッグコードを有効にしてコンパイルして実行するとコンソール画面に延々と実行されている命令が出力されるので、見比べましょう。

コマンドラインで、次のように実行すると、その動作ログをファイルに保存する事ができます。

```
$ [エミュレータ] nestest.nes > [動作ログ]
```

さて、この動作ログをnestest_trace.logと1行1行見比べてデバッグします。普通にやると、とても大変そうです…。

でも大丈夫。Notepad++[24]などにある、画面を2分割して別のテキストを開きつつ、二つのテキストのスクロールを同期するモードを使って見比べましょう。どちらの動作ログも、1命令実行1行、なので、同じ行には同じアドレスの同じ命令が表示されるはずです。違ったら、バグなので直しましょう。ただ、本当に１行１行見ていくのはとても骨が折れる作業で、数万行の行のどこから食い違っているのかを探すのは本当に大変です。

そこで簡単なコツとして、「二分探索」があります。動作ログの１万行目を見て、正しい結果と異なっていた場合、0行目から1万行目のどこかで違っているはずですが、気が遠くなりますね。こんなときは、少しずつ範囲を絞って行きましょう。具体的には、こんな感じ：

1. 0行目と1万行目の間の、5000行目が間違っているかどうか、調べます。

23 http://wiki.nesdev.com/w/index.php/Emulator_tests nestestを今回は使います

24 http://notepad-plus-plus.org/

ここでは合っていたので、多分この後だと見当がつきます。

2. さらに、5000行目と1万行目の間の、7500行目を調べます。
   ここでは間違っていたので、5000行目と7500行目の間のどこかから間違っています。

3. 5000と7500の間にある、6700行目くらいを調べます。
   ここでは合っていたので、この後から7500行目のどこかにあります。

4. 1000行の間のどこかまで絞れたので、ここからは1行ずつ調べてしまいましょう。

このようにすれば、比較的短時間で間違った場所を特定できるので、おすすめです。

ここで動作が違っている場所が直接的にはバグの原因なので、その命令を調べてみましょう。ただし、いくつか（ひょっとすると数千も）前の命令でのバグの影響が、たまたまその命令で表面化していることもあるので、表面化しているところの命令の実行にバグが無さそうに見えたら「表面化したときの命令が**どの状態（たち）に依存しているか**」を調べて、**どの状態が何だったら期待通りに動いていたか**を考え、**その状態はどこで書き換わるか**を追いかけて行くとデバッグ出来なくはないですが、言うほど簡単ではないです。

このCPU実行ログを見比べててデバッグする方法は、他のモジュールをデバッグする際にも使った、基本的（？）な方法なので、ぜひやってみてください。デバッグが完了すれば、これで「最低限のコンピュータ」が完成したことになります。

0xFF To be continued...

この本は数年前から書き続けているのですが、今回からいままでのを分割してStageごとに本を分けてシリーズ物にすることにしました。ページ多すぎてもう一冊に出来なくなってしまった…。

C87ではわかりづらそうな説明の修正などがメインになっています。新しいのを書きたいのですが、きりょくがたりない！でも新しいことを書くのもわかりやすくするのも大事なはず。。。

Stage.1では、導入をした後にCPUとメモリをつくりました。これで最低限の「コンピュータ」はできたわけですが、まだ画像も音も出ないし操作もできません。

Stage.2では、画像を出力するPPUを作成します！

この本のPDFは、githubでも公開しています。

https://github.com/ledyba/NesBook

エミュレータはソフトなマホウ。

～ハード解析なしで作るファミコンエミュレータ～

written by: ψ（プサイ）

Blog: 「ψ（プサイ）の興味関心空間」http://ledyba.org/

Mail: psi@ledyba.org

Twitter: @tikal