5 提出形式 A1　　6 表現 B1
「結果と考察」のアウトラインがない

実験レポート

氏名・学籍番号	H28i17 小谷 将太郎
実験テーマ名・実験日時	AVR 実験(5)(6) アーキテクチャとアセンブラ, 2018/5/29, 6/19
実験結果に影響する 機材番号・環境等の条件	演習室の PC PC-3F12 バイナリエディタ, objdump

1.実験目的

1実験目的 A1　　(コメント) 白いボックスで塗りつぶしてテキストを上書きする場合, 再編集できない形式
(ボックスをこちらで編集できない形式) にしてから提出してください.

組込機器, PC などの制御対象によって CPU は異なる設計(アーキテクチャ)がなされている. アーキテクチャ には多くの種類が存在するため, 特定の CPU に限らないような, 汎用的なアーキテクチャの知識は多様 な環境においてプラムを制御する上で重要な知識である。本実験では、CPU アーキテクチャやアセンブリ言語に共通する一般的な機能を理解することを目的とする。具体的には、共通の C 言語ソースから生成された各種の CPU のアーキテク向けの実行バイナリの解析を通じて, アーキテクチャの違い, アセンブリ言語の仕様の違いを把握する。また、X86 実行バイナリに対する静的解析を通じて, x86_64 アセンブリ言語の基本的および応用的な利用方法を学習する。

2原理 A1

2.原理・理論

■ 一般的な CPU アーキテクチャに備わっている仕組み, 機能

今日ではノイマン型アーキテクチャが基本的なアーキテクチャの一つとされている。これはプログラム内蔵方式 とは記憶領域に保存された命令をハードウェアが順に実行することで, 高い汎用性(メモリ上のプログラムを変えれば動作が変わる)を実現する方法である。
主に以下の 3 つの部位から構成される.
1. 演算機 :命令を実行する
2. メモリアクセス管理+アドレス系レジスタ :どの命令を実行するか, どのデータを処理するか指定・管理する
3. 制御ユニット+制御系レジスタ :システムやハードを制御・管理する

■ 本実験で対象とするアーキテクチャ

(1) AVR
組込用途のマイクロコントローラ. 命令セットは 16bit 固定長. 代表的な採用例は Arduino.
(2) x86
x86 は Intel 社(広義には AMD 社製品も含む)が採用するアーキテクチャであり多くの PC 向け CPU で採用される. x86 という名称は初期のプロセッサの型番が 8086, 80186 のように~86 であったことに由来している. x86 を更に分類すると 16bitCPU の 8086 系統, 32bit CPU の Intel80386(i386)系統, 64bit CPU の x86_64 (x64)系統に分類できる.
(3) PowerPC
Power アーキテクチャと呼ばれる系統には組込系, サーバ系など用途に応じていくつか派生がある.PowerPC はデスクトップ PC 向けの製品であり古い mac(macintosh)などで採用されていた. Power アーキテクチャの派生系は Wii, WiiU 等のゲーム機, ルーターや, DeepBlue (IBM のチェス専用スパコン)などで採用されており, Playstation3, セル Regza(TV)では Cell アーキテクチャと呼ばれる Power アーキテクチャ系 CPU が採用された.
(4) MIPS
RISC の元祖. 現在は組込系の製品で主に採用される. 製品の例としてはルーター等の通信機器, レコーダー系の映像機器,Nintendo 64, Playstation, Playstation2, Playstation Portable など.
(5) ARM
省電力・それなりに高機能な組込系で主に使用される.様を加えたチップを生産できる点が特徴.ARM 社とライセンス契約を結ぶことで ARM のコア部分に独自仕スマートフォンやタブレットでの採用例は Tegra, Snapdragon など.他にも RaspberryPi や Nintendo DS, Playstation Vita, Nintendo Switch なども ARM 系の CPU が採用される.

■ AVR の実行バイナリの解析

今回の実験の条件設定：C 言語のソースコードと逆アセンブル結果を使ってどのように実験を進めるか

本実験では, 同じソースコードから生成された各アーキテクチャごとのアセンブリコードを読み, その動作理解および関連する CPU の機構についての調査・考察を行う.

関数の戻り値

アセンブリ言語において r24, r25 といった「r+数値」の記法はレジスタを表現するためによく用いられており,

・ C 言語の int 型は規格上, 最低 16bit の桁数を保証する必要がある

・ AVR のレジスタは 8bit であり, レジスタ 1 つでは 16bit を表現できない

といった理由のため, AVR の 8bit レジスタを 2 つ合わせて 16bit の領域として扱っている.よって int や short のような 16bit の数値を return する場合, r25(上位桁), r24(下位桁)に値をセットすることが返り値の設定である。また、long 型のような 16bit より桁数の大きな値を return する場合, ビット数が増えるごとに r25, r24, r23, r22 とレジスタ番号が小さいレジスタを連続して使用する

関数の引数処理

関数の引数が int 型の場合, 第 1 引数から順にレジスタのペア(r24,r25), (r22,r23), (r20r, 21),...へと格納される。引数処理においては, 第 1 引数を格納するレジスタ(r24,r25)は返り値を格納するレジスタ(r24,r25)としての役割を兼ねる

アドレス指定

C 言語の構造体のメモリ上への配置は

・ 連続したアドレス上に確保される場合が多い

・ 一つ目のメンバ変数(この場合 int a)のアドレスが構造体のアドレス

という特徴がある。

また、アドレスの指定方法には直接参照(直接アドレッシング), 間接参照(間接アドレッシング)が存在し, AVR では間接アドレッシングが採用される。AVR では(R26,R27), (R28,R29), (R30, R31)に X レジスタ, Y レジスタ, Z レジスタという別名がついており, アドレス指定(アドレッシング)の機能を持つ。AVR の Z レジスタは「Z+1」のように「アドレスの番地+差分」の形でアドレスを指定できる。

関数呼び出し

AVR の関数呼び出しは

・関数呼び出し対象の関数が格納されているアドレスへ rcall でジャンプする

・機械語では「rcall k (-2048~+2047)」で呼び出し先の関数へジャンプしている

・逆アセンブル結果のアセンブリコードでは, アドレスの指定は PC(rcall 実行前より値が進んでいる)を基準に「.アドレスの差分」と表記され, もとの機械語とは厳密に対応しない.

スタック処理

関数の呼び出しなどで特定のアドレスへジャンプする場合, 処理中のデータを破壊しないようにするために以下のようなデータの退避が必要になる.

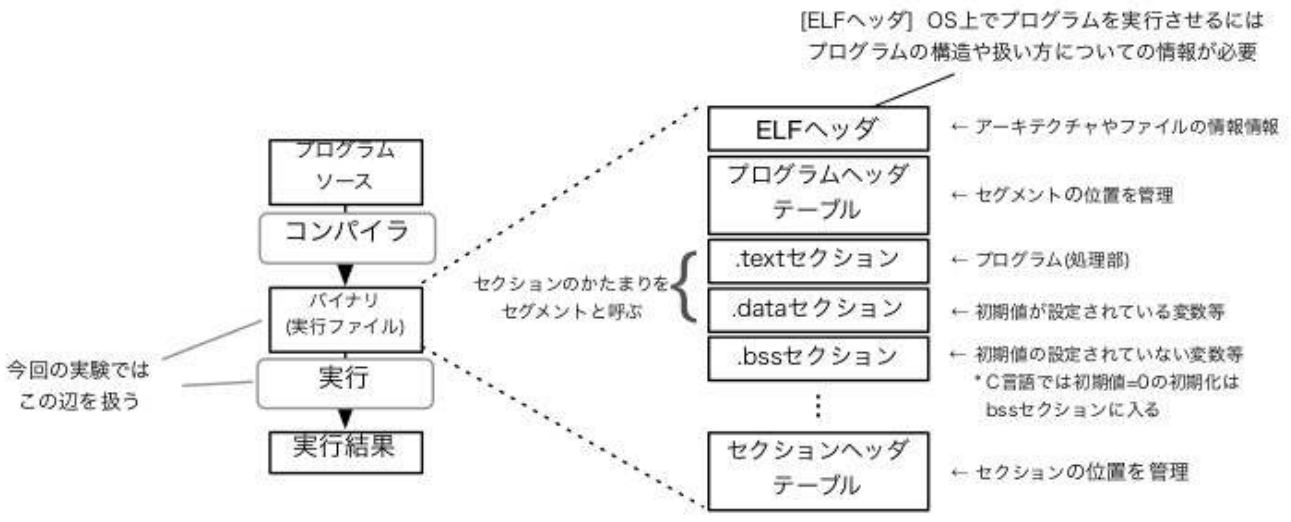
・ ジャンプから復帰するアドレスの保存

・ 処理途中のレジスタ内容の一時保存(データの退避)

また, 関数呼び出しから復帰する場合にはこれらの処理と逆の処理, つまり退避したデータを戻す処理が必要となる。

■ x86 実行バイナリの静的解析

x86 実行バイナリの構造



（図 1）x86 実行バイナリの構造

ELF フォーマット

プログラミングではソースコードをバイナリ(機械語)に変換するが, このバイナリを OS 上で実行するためにはライブラリやプログラムの構造についての情報が必要になる. ELF(Excutable and Linkable Format)は主に Linux や BSD の派生 OS で採用されるプログラムのフォーマットである.

objdump の機能

プログラムの動作を解析する場合に, 機械語をアセンブリ言語に変換する操作を逆アセンブル(disassemble)と呼ぶ. objdump(オブジェクトダンプ)はバイナリを機械語として解釈し, 各アドレスに保存されている値, およびその逆アセンブル結果である命令を表示する.

x86 の条件分岐命令

x86 系のアセンブリ言語においてほとんどの条件分岐命令は「j+条件」のニーモニックで記載される。jne, je...他の分岐命令については下記を参照する。: Steve Friedl s Unixwiz.net Tech Tips Intel x86 JUMP quick reference,http://unixwiz.net/techtips/x86-jumps.html

バイナリエディタの機能

バイナリエディタ(hex エディタ)はバイナリを表示・編集可能なツールである。

3手順 A1

3.手順

3-1. 前提条件と実験機材の準備

逆アセンブル結果の解析には配布ファイルを使用する。64bit の Linux 環境(x86-64)の実行環境で実験を行う。

3-2. 実験手順

- **実験 1**
  [手順 2.1] 指導書中の Z レジスタの例では, AVR では間接アドレス指定が可能であることを確認した. AVR では直接アドレス指定が可能かどうか調査せよ.
  [手順 2.2] X レジスタ, Y レジスタも Z レジスタと同様に「アドレス+差分」の形でアドレスを指定可能であるか調査せよ.
  [手順 3] C 言語ソース中の condition(),loop()の 2 つの関数を逆アセンブルして得られるアセンブリコードを配布ファイル avr.d 中から見つけ出し, それぞれについて動作を説明しなさい
- **実験 2**
  [手順 1.1] 配布ファイルの i386.d 中から, 命令が可変長であることが確認できる箇所を示しなさい.[手順 1.2] x86 が持つレジスタについて調査し, 主要なレジスタについて簡単に説明せよ
  [手順 1.3] AX レジスタに対する EAX レジスタのように頭文字に E が付くのはどのような意味をもつか述べよ
  [手順 2]C 言語ソース中の以下の関数を逆アセンブルして得られるアセンブリコードを配布ファイル i386.d 中から見つけ出し動作を説明しなさい.
- **実験 3**
  [手順 1] [PowerPC のアーキテクチャ] PowerPC では演算結果の符号等を管理する特殊レジスタとして条件レジスタ(CR, Condi-tion Register)が採用されている. PowerPC の条件レジスタについて調査し, powerpc.d 中から条件レジスタが使用されていると思われる箇所を一つ示し CR の動作を簡単に説明せよ.
  [手順 2] [MIPS のアーキテクチャ 1] MIPS ではサブルーチン等の呼び出しから復帰する際の命令アドレスを保存する特殊レジスタとしてリンクレジスタ(mips.d 中の ra)が採用されている. MIPS のリンクレジスタについて調査し, mips.d 中からリンクレジスタが使用されていると思われる箇所を一つ示しリンクレジスタの動作を簡単に説明せよ**.**
  [手順 3][MIPS のアーキテクチャ 2] MIPS では CPU の処理効率を向上させる仕組みとしてパイプラインが採用されている. パイプライン, 遅延スロット, 分岐遅延スロットの仕組みについて調査し, mips.d 中から分岐遅延の動作が確認できる箇所を一つ挙げ, パイプラインの動作を簡単に説明せよ.
- **実験 4**
  [手順 1][プログラムの静的解析 1] 配布した「4-1.out」はメッセージをポップアップウインドウに表示する Linux 用実行バイナリである。「ポップアップ表示されるメッセージをバイナリエディタを使用して改変しなさい」
  [手順 2][プログラムの静的解析 2] 配布した「4-2.out」はパスワード入力に応じてメッセージが変化する Linux 用実行バイナリである.§2 を参考に, パスワード文字列が何であるか解析しなさい.
  ◦
  [手順 3] [プログラムの改変] 「4-2.out」を, 「パスワード入力の内容に関わらず, 常に正しいパスワードを入力した時の動作を行うように」バイナリエディタで改造しなさい. レポートには, どのように改造したか, および, 動作確認結果を記載すること.

4.結果と考察

4-1. 実際に得られた結果

今回の実験では結果の観測というよりは、逆アセンブリ結果を読んだり、各アーキテクチャの調査、プログラムの静的解析など考察に相当するものがほとんどであったため、4-2 考察にまとめた。

4-2. 考察~~（感想ではないので注意）~~

実験 1 AVR のアーキテクチャとアセンブリ

- **[手順 2.1] 指導書中の Z レジスタの例では, AVR では間接アドレス指定が可能であることを確認した. AVR では直接アドレス指定が可能かどうか調査せよ.**
  AVR 命令仕様書の 14 ページを参照すると、直接アドレス指定が可能であることが分かる。

- **[手順 2.2] X,Y,Z レジスタ 指導暑中の例では, Z レジスタは Z+1, Z+2 の様に「アドレス+差分(オフセット,ディスプレースメントとも呼ぶ)」の形でアドレスを指定可能であった. X レジスタ, Y レジスタも Z レジスタと同様に「アドレス+差分」の形でアドレスを指定可能であるか調査せよ**
  Y レジスタは変位付きデータ関節アドレス指定が可能であるが、X レジスタは不可能である。

- **[手順 3] [アセンブリコードの読解] C 言語ソース中の以下の 2 つの関数を逆アセンブルして得られるアセンブリコードを配布ファイル avr.d 中から見つけ出し, それぞれについて動作を説明しなさい**
  まず、condition について説明する。引数の a が r24, r25、b が r22, r23 である。 *cp r24, r22* で r24 と r22 を比較、*cpc r25, r23* で前比較のキャリーを含めた比較を行う。比較した結果、不一致だった場合、Z=0 となり直後の BRNE 命令で fe159c まで分岐する。一致した場合、分岐せずに命令を進め、*ldi r22, 0x01, ldi r23, 0x00* が行われる。ここで b に 1 が加算される。その後、6 行目以下の処理では返り値のレジスタである r25, r24 に b のレジスタの r, 23、r 22 を代入し、adiw 命令で r24 に即値 1 を加算し、ret で返却している。

```
1.  fe1592:     86 17     cp      r24, r22
2.   fe1594:    97 07     cpc     r25, r23
3.   fe1596:    11 f4     brne    .+4       ; 0xfe159c <condition+0xa>
4.   fe1598:    61 e0     ldi     r22, 0x01         ; 1
5.   fe159a:    70 e0     ldi     r23, 0x00         ; 0
6.   fe159c:    97 2f     mov     r25, r23
7.   fe159e:    86 2f     mov     r24, r22
8.   fe15a0:    01 96     adiw    r24, 0x01         ; 1
9.    e15a2:    08 95     ret
```

(表 1)Condition のアセンブリ言語のソースコード

次に、loop について説明する。1,2 行目で引数の値 r24, r25 を r20,r21 に代入する。3,4 行目で r24, r25 を定数 0 に設定する。5,6 行目で r18, r19 に r24, r25 を代入する。ここまでの状態で各レジスタの値はいかのようになる。r20, r21 が引数の値、r24, r25 が 0、r18, r19 が 0 となる。cp r24, r20 と cpc r25, r21 で r25, r24 と r21, r20 の比較を行う。ここで BRGE 命令が行われる。R25>=r21 だった場合は、0xfe15c4 に分岐する。R25>=R21 となっていない場合は、直後の命令に進む。add r24, r18、adc r25, r19 で r24, r25 に r18、r19 が加算される。subi r18, 0xFF, sbci r19, 0xFF で r18, r19 からキャリーを含めて 0xFF が減算される。0xFF は 2 の補数で-1 なので減算とすると+1 となり、(r18, r19) に 1 が加算される。続いて(r18, r19)と(r20, r21)が比較され、(r18, r19) < (r20, r21)となっていれば、0xfe15b6 に分岐する。その後、(r24, r25)に(r18, r19)を代入、(24, r25)に 1 を加算するという処理を (r18, r19) < (r20, r21)が不成立となるまで繰り返し行われる。その後、ret で(r24, r25)が返却される。

```
1.  00fe15a4 <loop>:
2.    fe15a4:   48 2f       mov     r20, r24
3.    fe15a6:   59 2f       mov     r21, r25
4.    fe15a8:   80 e0       ldi     r24, 0x00         ; 0
```

```
5.    fe15aa:   90 e0              ldi     r25, 0x00          ; 0
6.    fe15ac:   28 2f              mov     r18, r24
7.    fe15ae:   39 2f              mov     r19, r25
8.    fe15b0:   84 17              cp      r24, r20
9.    fe15b2:   95 07              cpc     r25, r21
10.   fe15b4:   3c f4      brge    .+14    ; 0xfe15c4 <loop+0x20>
11.   fe15b6:   82 0f              add     r24, r18
12.   fe15b8:   93 1f              adc     r25, r19
13.   fe15ba:   2f 5f      subi    r18, 0xFF; 255
14.   fe15bc:   3f 4f      sbci    r19, 0xFF; 255
15.   fe15be:   24 17              cp      r18, r20
16.   fe15c0:   35 07              cpc     r19, r21
17.   fe15c2:   cc f3      brlt    .-14    ; 0xfe15b6 <loop+0x12>
18.   fe15c4:   08 95              ret
```

(表 2)loop のアセンブリ言語のソースコード

(コメント) 授業中では触れていませんが，「表のキャプションは上側」につけてください

実験 2 x86(i386)のアーキテクチャとアセンブリ

■ **[手順 1.1] 配布ファイルの i386.d 中から，命令が可変長であることが確認できる箇所を示しなさい．**

```
1.  00fe1495 <store_long>:
2.     fe1495:   8b 44 24 04                  mov    0x4(%esp),%eax
3.     fe1499:   c7 00 44 33 22 11            movl   $0x11223344,(%eax)
4.     fe149f:   c3                    ret
```

2 行目、 3 行目、 4 行目での命令の長さはそれぞれ違うため、可変長であることが分かる。

■ **[手順 1.2] x86 が持つレジスタについて調査し，主要なレジスタについて簡単に説明せよ**

x86 は 8 個汎用レジスタ、6 個のセグメントレジスタ、1 個のフラグレジスタを持っている。

汎用レジスタには算術演算結果が格納されるアキュムレータレジスタ、算術演算操作と I/O 操作に使用されるデータレジスタなどがある。セグメントレジスタとは配列のようなメモリ領域の先頭を示すポインタである。フラグレジスタにはキャリーフラグ、0 フラグ、符号フラグなど各種フラグを格納している。

■ **[手順 1.3] AX レジスタに対する EAX レジスタのように頭文字に E が付くのはどのような意味をもつか述べよ**

インテル社の 8080CPU では汎用レジスタを'a', 'b', 'c'と名付けていた。これを拡張した 8086CPU では汎用レジスタは'ax', 'bx', 'cx'となった。x は extend の意味である。その後、80386CPU で 32 ビットになり、レジスタは'eax', 'ebx', 'cbx'となった。この e も extend の意味である。そのため頭文字には e がついている。

- **C 言語ソース中の以下の関数を逆アセンブルして得られるアセンブリコードを配布ファイル i386.d 中から見つけ出し動作を説明しなさい.**

  null():

  返却命令である ret を実行する。

  return_zero():

  eax レジスタに 0x0 を代入して返却する。

  return_one():

  eax レジスタに 0x1 を代入して返却する。

  return_int_size():

  eax レジスタに 0x4 を代入して返却する。

  return_short():

  eax レジスタに 0x7788 を代入して返却する。

  return_long():

  eax レジスタに 0x778899aa を代入して返却する。

  return_arg1():

  esp はスタックポインタである。0x4(%esp)で esp レジスタに+4 したアドレスの値となる。ここのスタック上の値を mov 命令で eax レジスタに代入し返却する。

  return_arg2():

  0x4(%esp)が第一引数、0x8(%esp)が第二引数にあたる。第二引数の値を mov 命令で eax レジスタに代入し返却する。

  add():

  まず、第二引数の値を eax レジスタに代入し、次に第一引数の値を eax レジスタに加算する。

  add3():

  第二引数の値を eax レジスタに代入、第一引数の値を eax レジスタに加算、第三引数の値を eax レジスタに加算する。

  load():

  第一引数の値を eax レジスタに代入し、レジスタ間接によって、eax レジスタの指す先を eax レジスタにロードしている。

  store():

  第一引数の値を eax に代入し、0xFF という即値を eax レジスタに代入している。

  member():

  第一引数の値を eax に代入し、eax レジスタに+4 したアドレスの値に 0x01 を代入している。そして、eax レジスタに+8 したアドレスの値を eax に代入し返却している。

  condition():

  第二引数の値を eax に代入し、cmp 命令で第一引数と eax レジスタの値を比較する。jne 命令により一致しなかったら inc 命令までジャンプし eax をインクリメントしそのまま返却する。一致した場合はジャ

ンプせずに eax に 1 を代入して、eax をインクリメントし返却する。

loop():
第一引数の値を ecx レジスタに代入、0x0 を eax レジスタに代入、0x0 を edx レジスタに代入する。
ecx レジスタの値と eax レジスタの値を比較するが、これは edx と eax が等価なためである。

実験 3 その他のアーキテクチャ（PowerPC, MIPS, ARM, x86-64)

- **[手順 1] [PowerPC のアーキテクチャ] PowerPC では演算結果の符号等を管理する特殊レジスタとして条件レジスタ(CR, Condi-tion Register)が採用されている. PowerPC の条件レジスタについて調査し, powerpc.d 中から条件レジスタが使用されていると思われる箇所を一つ示し CR の動作を簡単に説明せよ.**

  CR レジスタは各条件によってビットが変化する。condition 内の cr7 が条件レジスタである。cmpw で r3 と r4 を比較し、cr7 を更新している。

ビット	フラグ	意味
0	負	結果が負の時にセット
1	正	結果が正（かつゼロでない）ときにセット
2	ゼロ	結果がゼロの時にセット
3	サマリオーバーフロー	命令完了時の XER のコピー

(表 3 PowerPC の条件レジスタ)

- **[手順 2] [MIPS のアーキテクチャ 1] MIPS ではサブルーチン等の呼び出しから復帰する際の命令アドレスを保存する特殊レジスタとしてリンクレジスタ (mips.d 中の ra)が採用されている. MIPS のリンクレジスタについて調査し, mips.d 中からリンクレジスタが使用されていると思われる箇所を一つ示しリンクレジスタの動作を簡単に説明せよ**

  リンクレジスタには「関数を呼び出したアドレスの次のアドレス」の値が関数を呼び出した時に（自動的に) 代入されている。 ra の実体は汎用レジスタの r31 である。mips.d の 8 行目の jr ra などすべての関数で使われている。

- **[手順 3][MIPS のアーキテクチャ 2] MIPS では CPU の処理効率を向上させる仕組みとしてパイプラインが採用されている. パイプライン, 遅延スロット, 分岐遅延スロットの仕組みについて調査し, mips.d 中から分岐遅延の動作が確認できる箇所を一つ挙げ, パイプラインの動作を簡単に説明せよ.**

  パイプライン:

  命令フェッチ、命令デコード、実行などの異なる工程を並列に動作させる。数個の命令が並列で動作しているようにみせかけ 1 命令あたりの消費クロック数を少なくする。

  遅延スロット:

  直前の命令が効力を発揮する前に実行される命令のスロット。

  分岐遅延スロット:

遅延スロットを用いて分岐前に実行する命令のスロット。

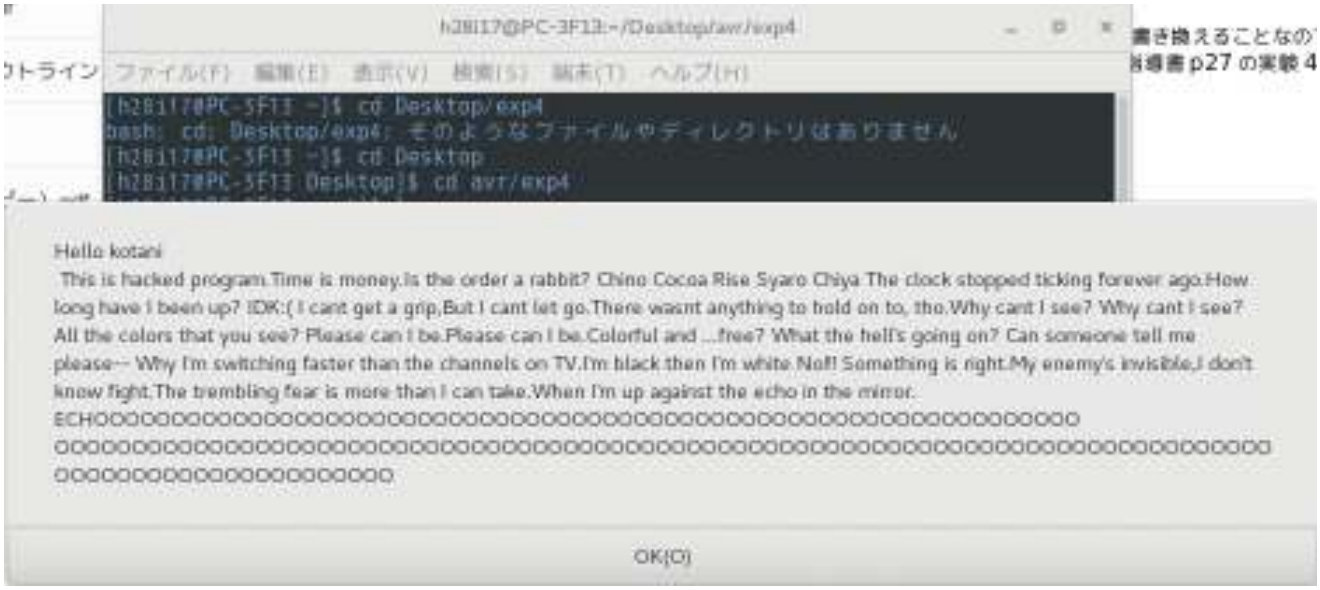
mips の分岐命令は全て 1 命令だけ遅延する。つまり、分岐命令の次の命令が遅延スロットとなり実行される。

```
1. 00fe1408 <return_zero>:
2.     fe1408:    03e00008            jr          ra
3.     fe140c:    00001021            move        v0,zero
```

分岐遅延として、「jr ra」が実行される前に、[move v0, zero」が実行される

実験 4 x86_64 用プログラムの静的解析

- **[手順 1][プログラムの静的解析 1] 配布した「4-1.out」はメッセージをポップアップウインドウに表示する Linux 用実行バイナリである。「ポップアップ表示されるメッセージをバイナリエディタを使用して改変しなさい**
  まず、実験環境で動作させるために権限を与える。
  chmod u+x 4-2.out
  次に、strings コマンドを用いて、Hello World のオフセット値を調べる。オフセット値とはファイルの先頭から目的のバイトまでの位置のことである。端末を開いて以下のように入力する。
  $ strings 4-1.out -t x | grep Hello
  a20 Hello World!
  出力から Hello World のオフセット値が@0xA20 であることが分かる。
  次にバイナリエディタを開く。今回の実験では Okteta を用いる。Okteta の「ファイル」から 4-1.out を開く。「編集」から「オフセットへ移動」を選択する。16 進数になっていることを確認し、先に調べて置いた Hello World のオフセット値(0xA20)を入力しエンターキーを押す。すると、カーソルが 0xA20 に移動する.
  目的は Hello, World!を書き換えることなので、好きな文字に書き換える。
  保存ボタンを押し、実験指導書 p27 の実験 4 プログラムの実行を参考に実行し、変更した通りに出力されれば成功である。



(図 2 改変した 4-1.out の実行結果)

- **[手順 2][プログラムの静的解析 2] 配布した「4-2.out」はパスワード入力に応じてメッセージが変化する Linux 用実行バイナリである.§2 を参考に, パスワード文字列が何であるか解析しなさい.**
  strings コマンドを用いて解析する。
  $ strings 4-2.out
  /lib64/ld-linux-x86-64.so.2
  u1D C
  ~省略~
  Enter password:
  tgbyhn
  3I Experiments!
  Incorrect password
  ;*3$"
  ~~~~
  strings コマンドの出力からパスワードは” tgbyhn”と推定できる。実際にプログラムを実行し、推定したパスワードを入力すると、3I Experiments!と出力されることから、推定したパスワードは正しかったといえる。
- **[手順 3] [プログラムの改変]「4-2.out」を,「パスワード入力の内容に関わらず, 常に正しいパスワードを入力した時の動作を行うように」バイナリエディタで改造しなさい. レポートには, どのように改造したか, および, 動作確認結果を記載すること.**
  まず、実験環境で動作させるために権限を与える。
  chmod u+x 4-2.out
  次に、objdump で 4-2.out を逆アセンブルする。

```
$objdump -D 4-2.out
000000000040062d <main>:
1.   40062d:  55                 push   %rbp
2.   40062e:  48 89 e5           mov    %rsp,%rbp
3.   400631:  48 83 ec 50        sub    $0x50,%rsp
4.   400635:  89 7d bc           mov    %edi,-0x44(%rbp)
5.   400638:  48 89 75 b0        mov    %rsi,-0x50(%rbp)
6.   40063c:  bf 30 07 40 00     mov    $0x400730,%edi
7.   400641:  b8 00 00 00 00           mov    $0x0,%eax
8.   400646:  e8 a5 fe ff ff     callq  4004f0 <printf@plt>
9.   40064b:  48 8d 45 c0        lea    -0x40(%rbp),%rax
10.  40064f:  48 89 c6           mov    %rax,%rsi
11.  400652:  bf 40 07 40 00     mov    $0x400740,%edi
12.  400657:  b8 00 00 00 00           mov    $0x0,%eax
13.  40065c:  e8 cf fe ff ff     callq  400530 <__isoc99_scanf@plt>
14.  400661:  48 8d 45 c0        lea    -0x40(%rbp),%rax
15.  400665:  be 43 07 40 00           mov    $0x400743,%esi
16.  40066a:  48 89 c7           mov    %rax,%rdi
17.  40066d:  e8 9e fe ff ff     callq  400510 <strcmp@plt>
18.  400672:  85 c0              test   %eax,%eax
19.  400674:  75 11              jne    400676 <main+0x49>
```
  ~~~~

```
20.  400676:  bf 4a 07 40 00      mov    $0x40074a,%edi
21.  40067b:  b8 00 00 00 00              mov    $0x0,%eax
22.  400680:  e8 6b fe ff ff      callq  4004f0 <printf@plt>
23.  400685:  eb 0a               jmp    400691 <main+0x64>
24.  400687:  bf 5d 07 40 00      mov    $0x40075d,%edi
25.  40068c:  e8 4f fe ff ff      callq  4004e0 <puts@plt>
26.  400691:  b8 00 00 00 00              mov    $0x0,%eax
27.  400696:  c9                  leaveq
28.  400697:  c3                  retq
29.  400698:  0f 1f 84 00 00 00 00         nopl   0x0(%rax,%rax,1)
30.  40069f:  00
```

パスワード入力を求めるプログラムでは、パスワード入力が正しいか判定する処理が必要である。このことを考慮すると、プログラム中では条件分岐処理が行われていると予想できる。x86 系のアセンブリ言語においてほとんどの条件分岐命令は「j+条件」のニーモニックで記載され、上記のアセンブル結果を見ると,19 行目の jne が条件に応じて分岐する処理であるとす遅く出来、条件分岐の挙動を改編できれば、パスワードを入力をスキップできる。
次に, バイナリエディタ(hex エディタ)と呼ばれるバイナリを表示・編集可能なツールによりこの条件分岐命令を変更し, パスワード入力処理を回避できるようにプログラムを改造する. 具体的には以下の手順でバイナリエディタを開き, 4-2.out を編集する.
1. バイナリエディタを起動し, バイナリエディタでコンパイル済みの a.out を開く
2.C 言語のパスワードチェック処理(プログラム 3,11 行目)と対応する処理を逆アセンブル結果から探す. C 言語のソース中で strcmp が呼ばれていることに着目すると, 逆アセンブル結果の 17 行目直後の分岐命令が目的の条件分岐である可能性が高い.
3.逆アセンブル結果の 19 行目の jne 命令と対応する箇所を修正する. 手順としては実行バイナリ中の当該命令をバイナリエディタで修正すればよい. バイナリエディタで jne 命令と対応するバイト列「75 11」をバイナリエディタで検索する。検索結果のうち,「75 11 bf 4a 07 40 00」のように jne 命令の前後と一致する箇所を特定する。
4. jne 命令と一致する箇所を特定したら「75 11」を「75 00」へと変更し上書き保存する
「jne ○」という命令は条件が成立するとき(not equal であるとき)アドレス〇へとジャンプする命令である. このバイナリに対する逆アセンブル結果は「75 11」であるが, 11 はアドレスの絶対指定としては値が小さい ため, 現在のアドレスからの差分を表していると推測できる. この性質を応用すれば, 先程の方法とは別のアプローチで岐命令を無効化でき, 具体的には「75 11」を「75 00」へと変更すると, ジャンプ先のアドレスが「現在のアドレス+0=ジャンプしない」となり, プログラムの動作は下記のようにパスワード入力に関わらず正常なパスワードを入力したときの処理が実行される.

A.その他補足情報

A-1. 参考文献

1. AVR 命令一式手引書[https://avr.jp/user/DS/PDF/AVRinst.pdf]
   AVR の命令の使用を確認するために参考にした。
2. X86 アセンブラ/x86 アーキテクチャ[https://ja.wikibooks.org/wiki/X86%E3%82%A2%E3%82%BB%E3%83%B3%E3%83%96%E3%83%A9/x86%E3%82%A2%E3%83%BC%E3%82%AD%E3%83%86%E3%82%AF%E3%83%81%E3%83%A3]
   x86 のレジスタを調査するために参考にした。
3. PowerPC のレジスタ[https://www.mztn.org/ppcasm/ppcasm02.html]
   PowerPC の条件レジスタを調査する為に参考にした。
4. PowerPC 分岐プロセッサーでのプログラミング
   [https://www.ibm.com/developerworks/jp/linux/library/l-powasm3.html]
   PowerPC の条件レジスタを調査する為に参考にした。
5. MIPS アーキテクチャ[https://ja.wikipedia.org/wiki/MIPS%E3%82%A2%E3%83%BC%E3%82%AD%E3%83%86%E3%82%AF%E3%83%81%E3%83%A3]
   MIPS についての調査をするために参考にした。
6. MIPS のレジスタ[http://www.swlab.cs.okayama-u.ac.jp/~nom/lect/p3/what-is-register.html]
   MIPS についての調査をするために参考にした。
7. 遅延スロット[https://ja.wikipedia.org/wiki/%E9%81%85%E5%BB%B6%E3%82%B9%E3%83%AD%E3%83%83%E3%83%88]
   MIPS の遅延スロットについて調査する為に参考にした。
8. 遅延スロット[https://www.wdic.org/w/SCI/%E9%81%85%E5%BB%B6%E3%82%B9%E3%83%AD%E3%83%83%E3%83%88]
   MIPS の遅延スロットについて調査する為に参考にした。
9. パイプライン構造[http://www.comp.tmu.ac.jp/morbier/work/gradcpu2007/cpu2007lectureno3.pdf]
   MIPS のパイプライン構造について調査する為に参考にした。