コンピュータシステムの理論と実装

4章 機械語

アジェンダ

- 一般的な機械語・ハードウェアについて
- Hack機械語・Hackコンピュータについて
- 課題

機械語とは

- 対象とするハードウェア が読み込み、解釈し、実行することができる
  - 算術演算や論理演算、メモリからのデータの読み込みや保存
    レジスタ間のデータ移動、ブーリアン値の条件テスト、など
- 対して 高水準言語 とは
  - 汎用性や機能性に主眼を置いて基本的な設計が行われる
- Hack機械語
  - Hack コンピュータ を対象とした機械語

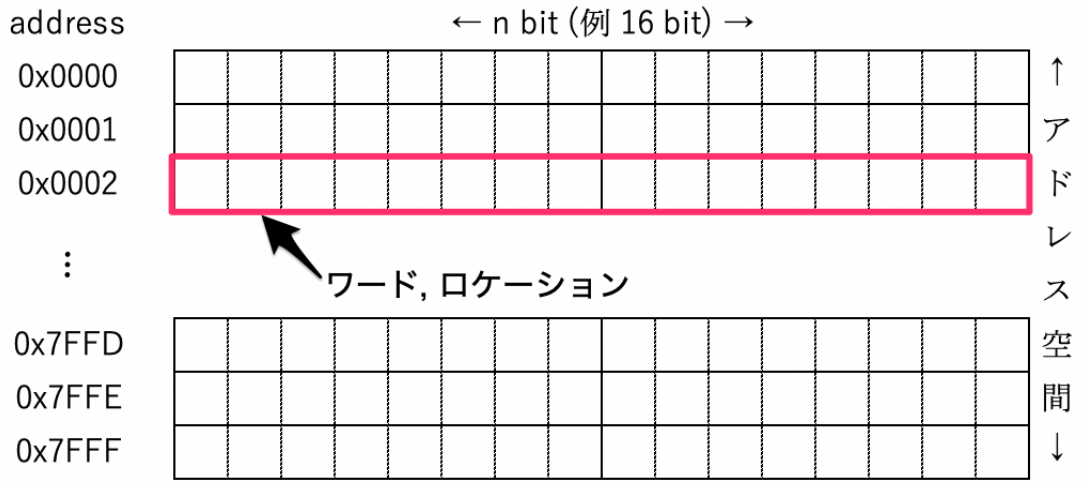
ハードウェアとは

- 機械語は仕様に従いハードウェアを制御する
- 本章ではハードウェアを抽象化し３つの要素のみを考える
  - メモリ
  - プロセッサ
  - レジスタ
  - 詳細は次章

- 機械語は プロセッサ と レジスタ を用いて メモリ を操作する

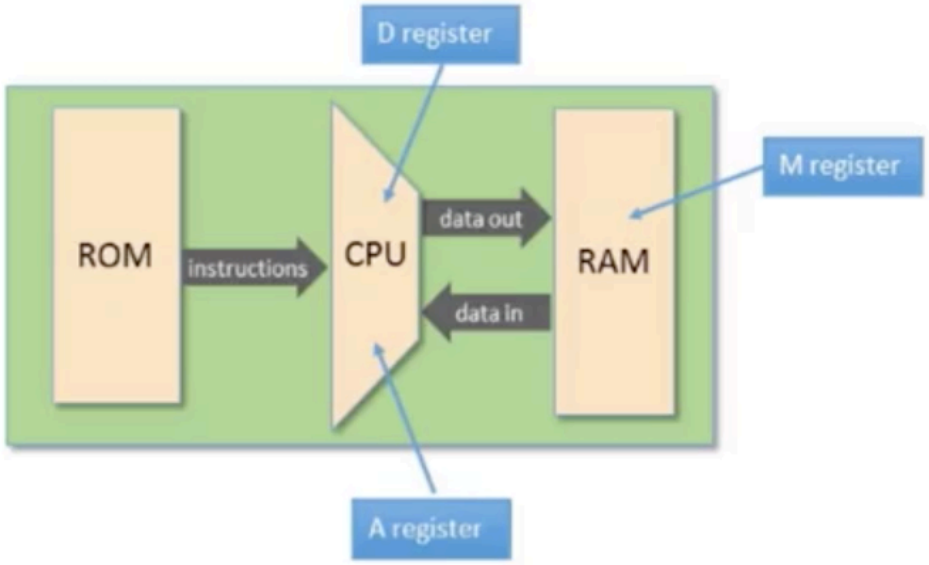
ハードウェア - メモリ

- データや命令を保存するハードウェアデバイス
- あるビット幅のセルが連続して並んでいる
  - Hackコンピュータは 16ビット幅
- 各セルを ワード と呼ぶ
- 各セルはそれぞれアドレスで指定できる
- Memory[addr], RAM[addr], M[addr]



ハードウェア - CPU

- プロセッサ (中央演算装置)
- 基本的な 命令セット を実行する
  - 算術演算,論理演算,メモリアクセス演算,制御演算...
- 演算の対象となるデータは レジスタ や メモリ から取り出される
- 演算の結果も同様に レジスタ や メモリ に格納する



ハードウェア - レジスタ

- ひとつだけの値を保持することができる
- プロセッサにいくつか備え付けられている
  - プロセッサに極めて近いためアクセスが高速
  - メモリへのアクセスはレジスタと比べれば遅い

機械語

- 機械語
  - 16 ビットコンピュータでの例

```
1010 0001 0010 0011      // バイナリコード
```

  - バイナリコードは人には理解が難しい
    - ニーモニック - mnemonic (tr.記憶を助ける,記憶術の)
    - コードと対応した記号や英単語で表す

```
1010 0001 0010 0011 → ADD R1,R2,R3
```

- アセンブリ言語
  - 記号による抽象化を プログラムを書く ために用いる
- アセンブラ
  - アセンブリから機械語へ 変換 するプログラム

機械語

- メモリアクセスとアドレッシングの種類
  - 直接アドレッシング - direct addressing
    - アドレスを直接指定する. もしくはシンボルで特定のアドレスを参照する

```
LOAD R1, 67     // R1 ← RAM[67]
                // foo が参照する値が 67 の時
LOAD R1, foo    // R1 ← RAM[67]
```

  - イミディエイトアドレッシング - immediate addressing
    - 命令コード中の値をそのまま読み込む. 定数を読み込む

```
LOADI R1, 67     // R1 ← 67
```

  - 関節アドレッシング - indirect addressing
    - ポインタを扱うために用いられる
      配列の要素にアクセスするために ベースアドレスにインデックスを加えるような処理

```
// x = foo[i]
ADD R1, foo, i    // R1 ← foo (base addr) + i (idx)
LOAD R2, R1       // R2 ← RAM[R1]
STR R2, x         // x ← R2
```

機械語

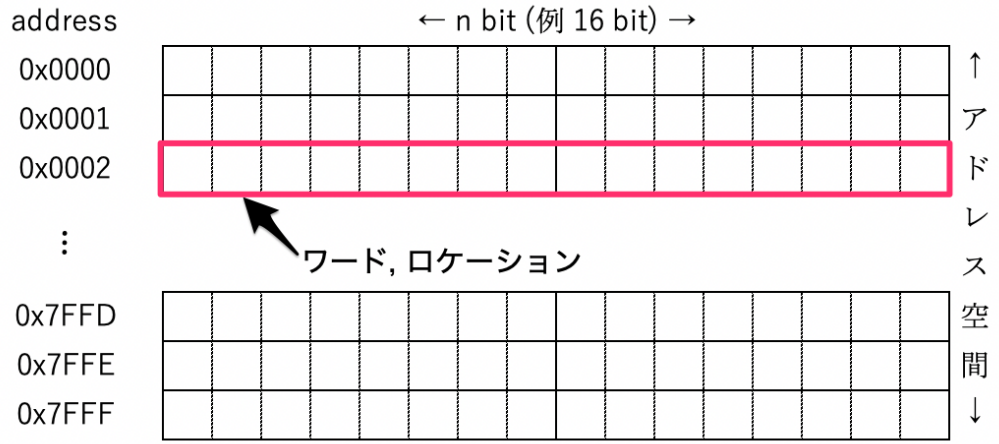
- 分岐命令
  - プログラムを頭から順に実行できない時に指定した位置に移動する
  - 反復, 条件分岐, 関数呼び出し

```
// 高水準
while (R1 >= 0) {
    // foo
}

// bar
```

```
// 機械語
beginWhile:
    JNG R1, endWhile
    // foo
    JMP beginWhile
endWhile:
    // bar
```

Hackコンピュータ

- メモリ
  - 二種類のメモリ空間
    - 命令メモリ
    - データメモリ
  - 16ビット幅, 15ビットのアドレス空間
- レジスタ
  - 二種類のレジスタ
    - D レジスタ
      - データレジスタ：データ値 だけを保持する
    - Aレジスタ
      - データレジスタ と アドレスレジスタの役割
      - 状況に応じて データ値 と アドレス として解釈される



Hack言語

- Hack機械語には アドレス命令 と 計算命令 の２つの命令がある
- メモリ操作を伴う操作は２つの命令がセットになる
  - 操作を行いたいアドレスを指定する アドレス命令
  - 実際の操作を指示する 計算命令

Hack言語 - A命令

- Aレジスタに15ビットの値を設定するために用いる

```
@value                  // A ← value
```

- 用途
  - 定数を代入する

```
@21
D = A                   // D ← 21
```

  - メモリ位置の指定

```
@21
D = M                   // D ← RAM[21]
```

  - 命令メモリの位置の指定

```
@21
JMP                     // goto 21
```

Hack言語 - C命令

- 何を計算するか：comp
- 計算した結果をどこに格納するか：dest
- 次に何をするか：jump

```
dest = comp; jump
```

- とりあえず、どんな計算ができて、どこに格納できて、どんな条件で移動できるかだけ見て

d1	d2	d3	ニーモニック	保存先(計算された値を格納する場所)
0	0	0	null	値はどこにも格納されない
0	0	1	M	Memory[A] (メモリ中のアドレスがAの場所)
0	1	0	D	Dレジスタ
0	1	1	MD	Memory[A]とDレジスタ
1	0	0	A	Aレジスタ
1	0	1	AM	AレジスタとMemory[A]
1	1	0	AD	AレジスタとDレジスタ
1	1	1	AMD	AレジスタとMemory[A]とDレジスタ

(a=0のとき) *comp*ニーモニック	c1	c2	c3	c4	c5	c6	(a=1のとき) *comp*ニーモニック
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	ニーモニック	効果
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

Hack言語 - シンボル

- アセンブラのコマンドは、定数 もしくは シンボル を用いてメモリ位置 (アドレス) を参照でき
- 定義済みシンボル
  - RAMアドレスの特別なものは予め定義されている
- ラベルシンボル
  - ユーザーが定義でき, goto コマンドの行き先として用いる

```
(FOO)           // FOO と宣言する
 ⋮
@FOO            // FOO への参照
0; JMP
```

- 変数シンボル
  - ユーザーが定義でき, 変数として扱う

```
@R0
D=M
@foo
M=D                 // foo=R0
```

Hack言語 - 定義済みシンボル

シンボル	値
R0	0
R1	1
…	…
R15	15
SCREEN	16384
KBD	24576

シンボル	値
SP	0
LCL	1
ARG	2
THIS	3

- R0,R1,…, R15：仮想レジスタ
  - プログラミングを単純化するために予め定義されている
- SCREEN, KBD：入出力ポインタ
  - スクリーンとキーボードのメモリマップのベースアドレスを示す
- SP, LCL, ARG, THIS：定義済みポインタ
  - 7章, 8章のバーチャルマシンで使う

Hack言語 - 例

- 1 + … + 100 の和を求める

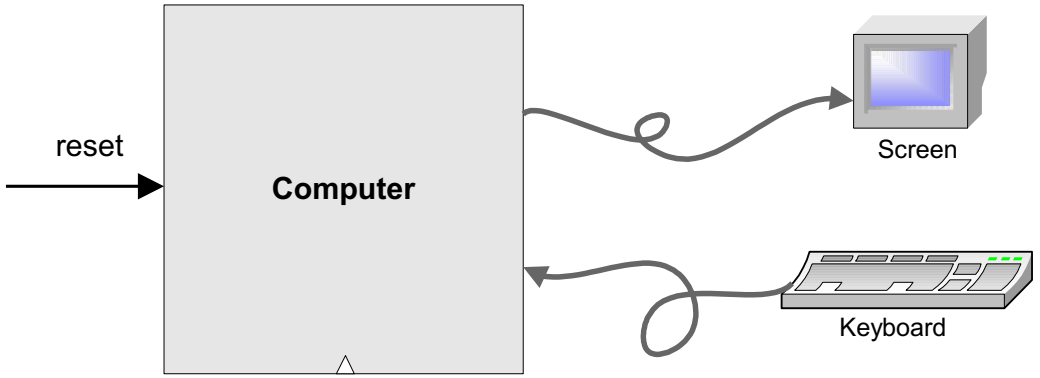
```
int i = 100;
int sum = 0;

while (i > 0) {
    sum += i;
    i--;
}
```

```
        @100
        D=A         // 定数 100 を代入
        @i
        M=D         // 変数 i に 100 を代入
        @sum
        M=0         // 変数 sum を 0 で初期化
(LOOP)
        @i
        D=M         // D = i
        @sum
        M=M+D       // sum += i
        @i
        MD=M-1      // i--
        @LOOP
        D;JGT       // i > 0 なら LOOP へ
(END)
        @END
        0;JMP       // 無限ループ
```

@FOO	A命令
D=C;J	C命令
(FOO)	ラベルシンボル

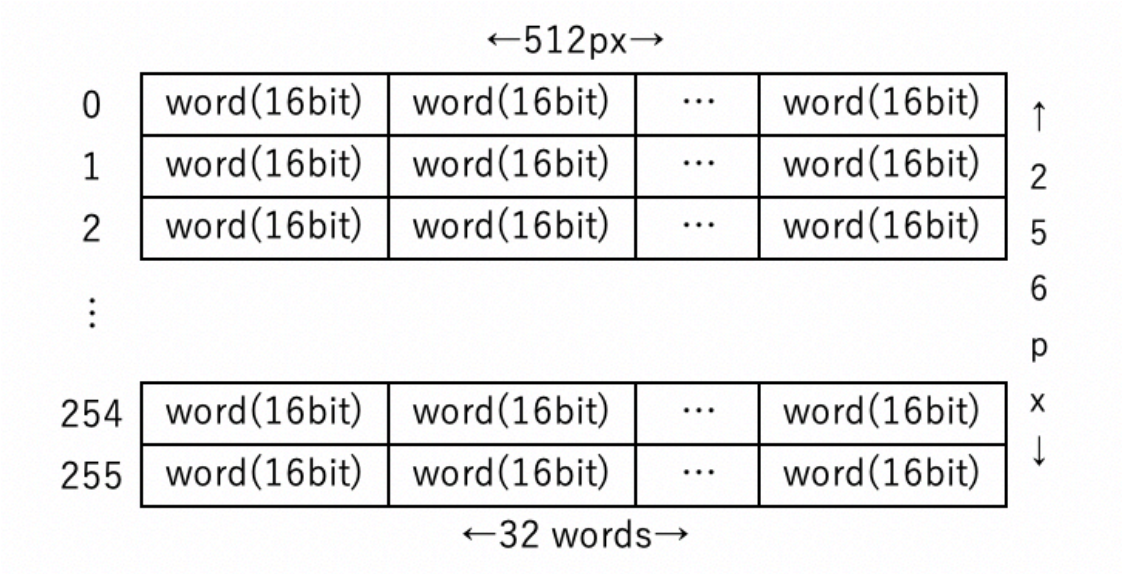
Hack言語 - 入出力

- Hackコンピュータは スクリーン と キーボード を繋ぐことができる
- メモリマップを介して入出力をおこなう



スクリーン

- 512 x 256 ピクセル, 白黒
- RAM[0x4000 (16384)] から 8k のメモリマップで表す
  - 8k = (512 / 16) * 256
- 各ビットが 1 = 黒, 0 = 白



キーボード

- RAM[0x6000 (24576)] のメモリマップで表す
- キーが押下されると16ビットのASCIIコードが現れる
  - 押下されていない時は 0
  - Hack キーボードの特別なキーコードもある

押されたキー	コード	押されたキー	コード
newline	128	end	135
backspace	129	pageup	136
leftarrow	130	pagedown	137
uparrow	131	insert	138
rightarrow	132	delete	139
downarrow	133	esc	140
home	134	f1–f12	141–152

課題 - Mult.asm

- 乗算プログラム (Mult.asm)
  - R2 = R0 * R1
  - ただし R0 ≧ 0, R1 ≧ 0, R0 * R1 < 32768 (0x8000) と想定して良い

- 高水準言語バージョン (実装案)

```
int r0, r1;   // input
int r2 = 0;   // sum
while (r1 > 0) {
    r2 += r0;
    r1--;
}
```

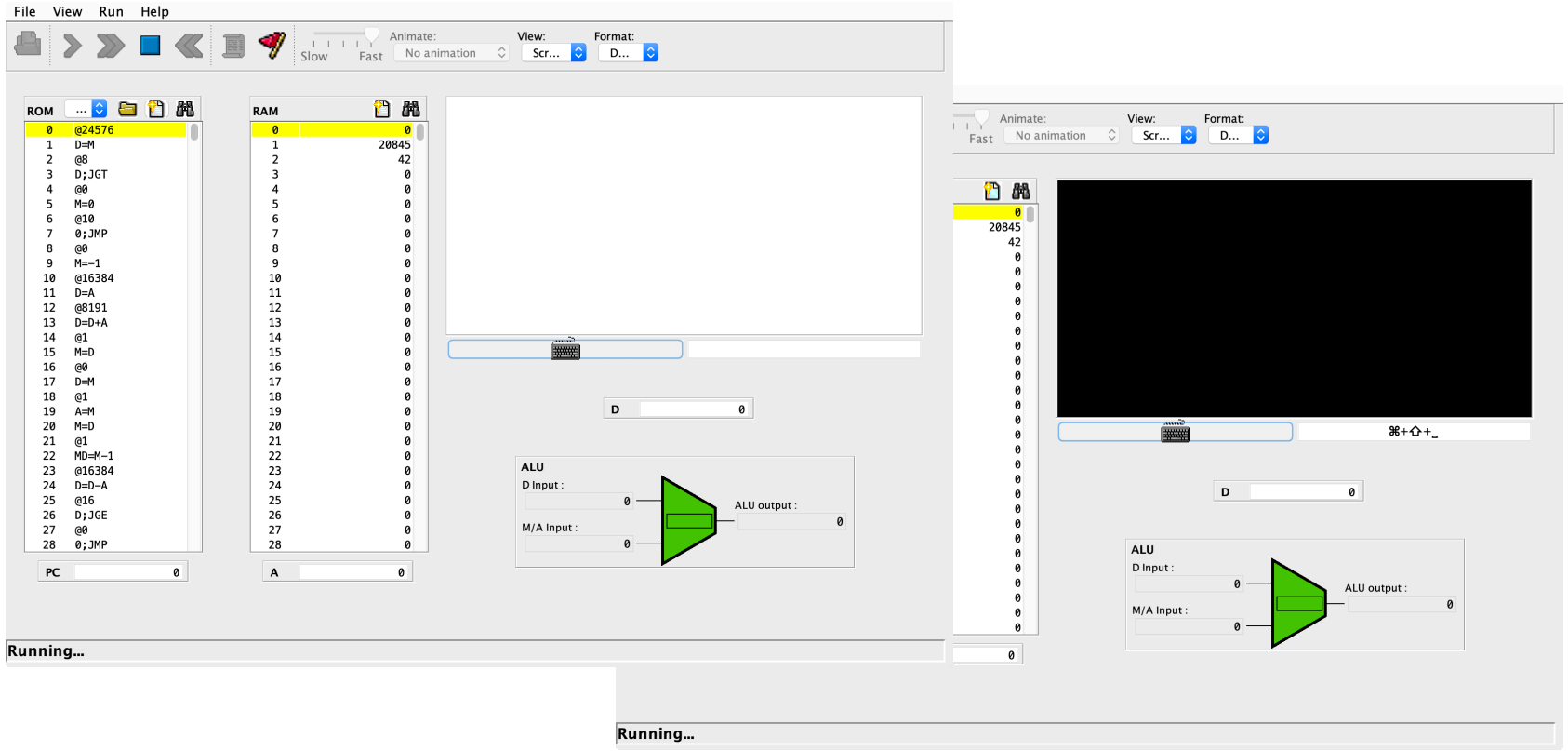
課題 – Mult.asm (実装案)

```
int r0, r1;  // input
int r2 = 0;  // sum
while (r1 > 0) {
    r2 += r0;
    r1--;
}
```

```
@R2
M=0          // 結果を格納する R2 を 0 で初期化

@R1
D=M
@END
D;JLE        // R1 が 0 以下の場合は END へジャンプ

(LOOP)
    @R0
    D=M
    @R2
    M=M+D    // R2 += R0

    @R1
    MD=M-1   // R1 をデクリメント

    @LOOP
    D;JGT    // R1 > 0 の時, Loop にジャンプ

(END)
    @END
    0;JMP    // 無限ループは Hack プログラムを "終了させる"
```

課題 - Fill.asm

- 入出力操作プログラム (Fill.asm)
  - キーボードが押下されたとき、スクリーンの全てのピクセルを黒で描画する
    - キーボードを押している間は真っ黒
    - キーボードを押していない時は真っ白



課題 - Fill.asm

- キーボード入力

```
@KBD
D=M      // キーボード入力
@FOO
D;JGT    // キーが入力されている場合 FOO へ
@BAR
D;JEQ    // 入力されていない場合 BAR へ
```

- スクリーン出力

```
@SCREEN
M=1     // SCREEN[0] のワードの1ビットを黒く描画する
@SCREEN
A=A+1 // SCREEN[1] を指定する
M=1     // 指定されたワードの1ビットを黒く描画する
```

課題 - Fill.asm (実装案)

```
(LOOP)
    // キーボード入力
    @KBD
    D=M

    // 押下されている場合 ON へジャンプ
    @ON
    D;JGT

    // 押下されていない場合 R0 に 0(0b000000000000) をセット
    @R0
    M=0

    // FILL へジャンプ
    @FILL
    0;JMP

(ON)
    // R0 に -1(0b111111111111) をセット
    @R0
    M=-1


// 右へ続く
```

```
(FILL)
    // 描写開始位置 (スクリーン右下) を初期化し R1 へ
    @SCREEN
    D=A
    @8191     // 8192 = (512 / 16) * 256 - 1
    D=D+A
    @R1
    M=D

    (FILLLOOP)
        // 描画する値
        @R0
        D=M

        // R1 に描画
        @R1
        A=M   // Aレジスタに描画位置のアドレスをセット
        M=D

        // 描画位置のアドレスをデクリメント
        @R1
        MD=M-1

        // スクリーン左上に到達するまで FILLLOOP へジャンプ
        @SCREEN
        D=D-A
        @FILLLOOP
        D;JGE

@LOOP
0;JMP
```

おわり

- お疲れ様でした。
- 次章でいよいよハードウェア編終了です。