1. 仮想コンピュータ

ここでは、コンピュータとプログラムのしくみを理解するために用意した、Java アプレット版「仮想コンピュータ」について説明します。

（1） 仮想コンピュータの仕様

データ長	8 ビット
メモリ容量	16 バイト
レジスタ	アキュムレータ （8 ビット） 命令レジスタ （8 ビット） プログラムカウンタ （4 ビット） インデックスレジスタ （4 ビット） フラグレジスタ （2 ビット ： キャリー、ゼロフラグ）
制御方式	逐次制御。命令語はすべて 1 語長。プログラムは 0 番地開始固定。
クロック周波数	約 1/16 ～ 約 64Hz 可変。
命令サイクル	2 サイクル（Fetch & Excute）

（2） 仮想コンピュータの使い方

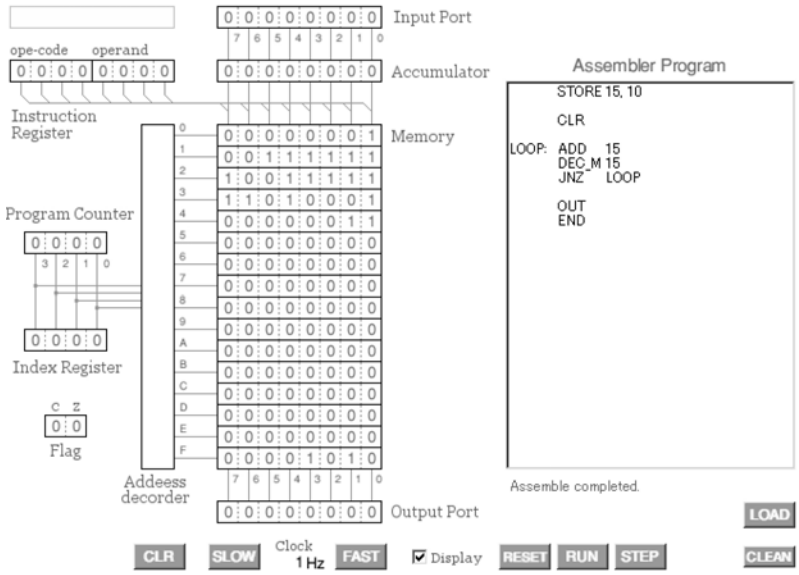

図 1 仮想コンピュータ

1. Assembler Program 欄にプログラムを書き、“LOAD” ボタンをクリックします。プログラムにエラーがなければ欄の下に “Assemble complated.” と表示され、メモリにはマシン語に翻訳された命令コードが書き込まれて表示されます。
2. “STEP” ボタンをクリックすると、命令が命令レジスタに読み込まれ（fetch）、もう一度クリックすると実行（excute）されます。
3. 実行されている命令は右上の窓に表示されます（ソースプログラムの表記にかかわらず、オペランドはすべて 16 進数で表示されます）。
4. “RUN” ボタンをクリックすると、連続的に実行されます。
5. “FAST” または “SLOW” ボタンをクリックすると実行速度が変わります。1/16～64Hz のクロック周波数が表

示されていますが、これは目安の数字で、パソコンの処理速度や次項の動作表示の有無などによって実行速度が変わります。パソコンの処理や表示が終わって、1.024 秒待機する状態を 1Hz、0.512 秒の場合を 2Hz… と表示しています。

6. “Display” がチェックされていると、参照されるメモリやレジスタ、データの流れなどが表示されます。チェックを外すと表示されなくなります。
7. メモリや各レジスタ、入出力ポートの値は、マウスでクリックすることによって直接編集できます。
8. “RESET” ボタンをクリックすると、プログラムカウンタがリセットされて 0 になります。
9. “CLR” ボタンをクリックすると、入力ポートを除くすべてのメモリ、レジスタ、ポートが 0 になります。
10. “Display” をチェックして動作表示中、画面をスクロールしたり他のソフトのウィンドウに隠れたりすると、動作表示が乱れることがあります。赤い線が消えずに見にくくなった場合は、“CLEAN” ボタンをクリックすると消えます。ただし、そのまま動作表示を続けると再び乱れます。動作表示中は極力スクロールしたりしないようにして下さい。

（3） アセンブラ仕様

1. 命令はすべて 1 語長（8 ビット）。
   上位 4 ビットは命令コード、下位 4 ビットはオペランド（ただし、オペランドがない命令は、8 ビット全てが命令コードとなります)。
2. 使用できる文字は英数字と記号 “: ; , % $ ♯”、空白文字（space）とタブ（tab）です。‘;’ の後にコメントを書くことができます。
3. インデックスアドレッシング（IX）を指定すると、オペランドは “0000” になります。
   オペランド “0000” は、分岐命令以外はインデックスアドレッシングモードを意味します。このため、原則として 0 番地のデータは参照できません。（0 番地のメモリにアクセスするには、インデックスレジスタを “0000” にした上で、インデックスアドレッシングを使用します。）
4. オペランドは 10 進数、16 進数、2 進数のいずれでも記述できます。2 進数は 4 桁で表記します（0000～1111)。ソースプログラム上で明示する場合は、識別子として “%”、“$”、“♯” をつけることができます。（例： 13、D、1101、%13、$D、♯1101 はすべて同じ。）
5. 疑似命令 STORE のアドレスとデータは、識別子を省略すると 10 進数と見なされます。
   10 進数であることを明示する、あるいは 16 進数、2 進数を使用する場合は、識別子 “%”、“$”、“♯” を使用しなければなりません。STORE 命令での 2 進数の桁数は任意です。
6. 疑似命令 STORE の番地とデータの間は ‘,’（comma）で区切ります。
7. 分岐命令の飛び先番地を指定するために任意のラベルが使用できます。
   末尾に ‘:’ をつけるとラベルとして認識されます。

   ```
   例：   LOOP:   READ   13
                  JEZ    LOOP
   ```

   これで、READ した（13 番地の）データが 0 であれば READ を繰り返すプログラムになります。
8. ラベル、命令語、オペランドの間には、任意の数の空白文字（space)、タブ（tab）が使用できます。

（4） 命令セット

命　　令	命令語	書　　式	命令コード	動　　作	Z	C
1. 入出力	IN	IN	0000 0010	In port → Acc	↕	•
	OUT	OUT	0000 0011	Acc → Out port	↕	•
2. アキュム	CLR	CLR	0000 0001	0 → Acc	1	•
レータ操作	INC	INC	0000 0100	Acc + 1 → Acc	↕	•
	DEC	DEC	0000 0101	Acc − 1 → Acc	↕	•
	SFT_L	SFT_L	0000 1000	$C \leftarrow {}_7[Acc]_0 \leftarrow 0$	↕	↕
	SFT_R	SFT_R	0000 1001	$0 \rightarrow {}_7[Acc]_0 \rightarrow C$	↕	↕
3. インデックス	SET_X	SET_X adr	1010 xxxx	adr → Ix	↕	•
レジスタ操作	INC_X	INC_X	0000 0110	Ix + 1 → Ix	↕	•
	DEC_X	DEC_X	0000 0111	Ix − 1 → Ix	↕	•
4. リード・	READ	READ adr	0001 xxxx	M → Acc	↕	•
ライト	WRITE	WRITE adr	0010 xxxx	Acc → M	↕	•
5. メモリ操作	INC_M	INC_M adr	1000 xxxx	M + 1 → M	↕	•
	DEC_M	DEC_M adr	1001 xxxx	M − 1 → M	↕	•
6. 算術演算	ADD	ADD adr	0011 xxxx	Acc + M → Acc	↕	↕
	SUB	SUB adr	0100 xxxx	Acc − M → Acc	↕	↕
7. 比較演算	CMP	CMP adr	0101 xxxx	Acc − M	↕	↕
8. 論理演算	AND	AND adr	0110 xxxx	Acc ・ M → Acc	↕	•
	OR	OR adr	0111 xxxx	Acc + M → Acc	↕	•
	NOT	NOT	0000 1010	$\overline{Acc} \rightarrow Acc$	↕	•
9. 分岐	JMP	JMP adr	1011 xxxx	adr → PC	•	•
10. 条件分岐	JEZ	JEZ adr	1100 xxxx	(Z=1) adr → PC	•	•
	JNZ	JNZ adr	1101 xxxx	(Z=0) adr → PC	•	•
	JCS	JCS adr	1110 xxxx	(C=1) adr → PC	•	•
	JCC	JCC adr	1111 xxxx	(C=0) adr → PC	•	•
11. その他	NOP	NOP	0000 0000	no operation	•	•
12. 疑似命令	STORE	STORE adr, n		n → M		
	END	END		Stop.		

（5） 命令

1. 入出力命令

IN（input）〔IN〕　（Input port → Acc）[1]

入力ポートのデータをアキュムレータに読み込みます。

OUT（output）〔OUT〕　（Acc → Output port）

アキュムレータのデータを出力ポートに出力します。

2. アキュムレータ操作命令

CLR（clear）〔CLR〕　（0 → Acc）

アキュムレータをクリアします。

INC（increment）〔INC〕　（Acc + 1 → Acc）

[1] **IN** は命令語、（input）はフルスペル、〔IN〕は書式、（Input port → Acc）は動作内容を表しています。

アキュムレータの値に 1 を加えます。

DEC（decrement）〔DEC〕　（Acc − 1 → Acc）

アキュムレータの値を 1 減少させます。

SFT_L（shift left）〔SFT_L〕　（C ← [$_7$ Acc $_0$] ← 0）

アキュムレータのデータを左に 1 ビットシフトさせます。最下位ビットには 0 が入り、最上位ビットのデータはキャリーフラグに入ります。

SFT_R（shift right）〔SFT_R〕　（0 → [$_7$ Acc $_0$] → C）

アキュムレータのデータを右に 1 ビットシフトさせます。最上位ビットには 0 が入り、最下位ビットのデータはキャリーフラグに入ります。

3. インデックスレジスタ操作命令

SET_X（set index register）〔SET_X　dd〕　（dd → Ix）

インデックスレジスタに数値（dd）を代入します。例えば、“SET_X　10”でインデックスレジスタは ‘10’（2 進数では ‘1010’）になります。“SET_X　1010”、“SET_X　A”でも同じです。

INC_X（increment index register）〔INC_X〕　（Ix + 1 → Ix）

インデックスレジスタの値に 1 を加えます。

DEC_X（decrement index register）〔DEC_X〕　（Ix − 1 → Ix）

インデックスレジスタの値を 1 減少させます。

4. リード・ライト命令

READ（read）〔READ　dd，READ　IX〕　（M → Acc）

メモリのデータをアキュムレータに読み込みます。例えば、“READ　%10”で 10 番地のメモリのデータを、“READ　IX”ではインデックスレジスタにセットされている値の番地のメモリのデータが読み込まれます。

WRITE（write）〔WRITE　dd，WRITE　IX〕　（Acc → M）

アキュムレータのデータをメモリに書き込みます。例えば、“WRITE　10”では 10 番地に、“WRITE　IX”ではインデックスレジスタにセットされている値の番地のメモリにデータが書き込まれます。

5. メモリ操作命令

INC_M（increment memory）〔INC_M　dd，INC_M　IX〕　（M + 1 → M）

メモリの値に 1 を加えます。例えば、“INC_M　$A”では 10 番地、“INC_M　IX”ではインデックスレジスタにセットされている値の番地のメモリの値に 1 が加えられます。

DEC_M（decrement memory）〔DEC_M　dd，DEC_M　IX〕　（M − 1 → M）

メモリの値から 1 を引きます。例えば、“DEC_M　A”では 10 番地、“DEC_M　IX”ではインデックスレジスタにセットされている値の番地のメモリの値から 1 が引かれます。

6. 算術演算命令

ADD（add）〔ADD　dd，ADD　IX〕　（Acc + M → Acc）

アキュムレータの値とメモリのデータを加算し、その結果をアキュムレータに書き込みます。例えば、“ADD　♯1010”では 10 番地、“ADD　IX”ではインデックスレジスタにセットされている値の番地のメモリの値と、アキュムレータの値が加算されます。

SUB（subtract）〔SUB　dd，SUB　IX〕　（Acc − M → Acc）

アキュムレータの値からメモリのデータを減算し、その結果をアキュムレータに書き込みます。例えば、“SUB 1010”では10番地、“SUB　IX”ではインデックスレジスタにセットされている値の番地のメモリの値が、アキュムレータの値から減算され、結果がアキュムレータに残ります。

7. 比較演算命令

CMP（compare）〔`CMP　dd`，`CMP　IX`〕　（Acc − M）

アキュムレータの値からメモリのデータを減算しますが、アキュムレータは更新されません。演算結果によってZ、Cフラグが変化します。

8. 論理演算命令

AND（and）〔`AND　dd`，`AND　IX`〕　（Acc・M → Acc）

アキュムレータの値とメモリのデータの論理積がアキュムレータに書き込まれます。例えば、“AND　%9”ではアキュムレータと9番地、“AND　IX”ではアキュムレータとインデックスレジスタにセットされている値の番地のメモリとの論理積になります。

OR（or）〔`OR　dd`，`OR　IX`〕　（Acc＋M → Acc）

アキュムレータの値とメモリのデータの論理和がアキュムレータに書き込まれます。例えば、“OR　$9”ではアキュムレータと9番地、“OR　IX”ではアキュムレータとインデックスレジスタにセットされている値の番地のメモリとの論理和になります。

NOT（not）〔`NOT`〕　（$\overline{\text{Acc}}$ → Acc）

アキュムレータの値の論理否定がアキュムレータに書き込まれます。例えば、アキュムレータの値が“10001110”のときにNOT命令を実行すると、“01110001”になります。

9. 分岐命令

JMP（jump）〔`JMP　dd`〕　（dd → PC）

無条件にプログラムカウンタの値を変更します。例えば、“JMP　9”命令を実行すると、プログラムカウンタは“1001”になり、次には9番地の命令を実行します。

10. 条件分岐命令

JEZ（jump on equal zero）〔`JEZ　dd`〕　（Z=1: dd → PC）

処理の結果が0の場合（Z=1）にプログラムカウンタの値が`dd`になり、分岐します。処理結果が0でない場合（Z=0）は次の命令を実行します。

JNZ（jump on not equal zero）〔`JNZ　dd`〕　（Z=0: dd → PC）

処理の結果が0でない場合（Z=0）にプログラムカウンタの値が`dd`になり、分岐します。処理結果が0の場合（Z=1）は次の命令を実行します。

JCS（jump on carry set）〔`JCS　dd`〕　（C=1: dd → PC）

処理によってキャリーフラグが1になった場合（加算の結果が255を越えた、減算・比較の結果がマイナスになった、あるいはシフト命令によって、など）にプログラムカウンタの値が`dd`になり、分岐します。キャリーフラグが1でない場合（C=0）は次の命令を実行します。

JCC（jump on carry clear）〔`JCC　dd`〕　（C=0: dd → PC）

キャリーフラグが0の場合にプログラムカウンタの値が`dd`になり、分岐します。キャリーフラグが0でない場合（C=1）は次の命令を実行します。

11. その他

NOP（no opration）〔NOP〕　（no operation）

コンピュータは何もしません。プログラムカウンタだけがインクリメントされて、次の命令に進みます。命令コードは “00000000” ですから、“CLR” ボタンをクリックしてメモリが全てクリアされた直後は、すべて NOP 命令が書き込まれていることになります。

12. 疑似命令

STORE（store）〔STORE　dd, n〕　（n → M）

疑似命令はコンピュータの命令ではなく、アセンブラプログラムがアセンブル時に実行する命令です。従って、メモリにプログラムが書き込まれることはありません。STORE は、指定した番地dd に値n を書き込みます。例えば、“STORE　10, 100” は、“LOAD” ボタンをクリックしたときに、メモリ 10 番地に値 100、2 進数で “01100100” が書き込まれます。

STORE 命令で指定する番地や値は、識別子をつけなければ 10 進数と見なされます。16 進数や 2 進数を使用する場合は、識別子 “$”、“♯” が必要です。10 進数であることを明示する場合は識別子 “%” を使用します。

END（end）〔END〕　（Stop）

プログラムの終了を表す疑似命令です。プログラムの最後に必ず記入しなければなりません。

（6）　プログラム例

（a）　加算（123 + 100）

```
        STORE   15, 123
        STORE   14, 100

        READ    15          ; 123 → Acc
        ADD     14          ; Acc + 100 → Acc

        OUT                 ; Acc → output port
        END
```

メモリ 15 番地に 123、14 番地に 100 をストアしておきます。アキュムレータは先ず 15 番地のデータ（123）を読み込み、次に 14 番地のデータ（100）を加算し、出力ポートに出力してプログラムが終了します。実行すると出力ポートは “11011111” になります。これは 10 進数にすると 223 です。

（b）　10 までの整数の加算

```
        STORE   15, 10      ; 10 → M(15)

        CLR                 ; 0 → Acc

LOOP:   ADD     15          ; M(15) + Acc → Acc
        DEC_M   15          ; M(15) − 1 → M(15)
        JNZ     LOOP        ; if M(15) = 0,  jump to LOOP

        OUT                 ; Acc → output port
        END
```

メモリ 15 番地に 10 をストアし、アキュムレータはクリアして 0 にしておきます。アキュムレータに 15 番地のデータ 10 を加算し（0 + 10）、次に 15 番地のデータをデクリメントします。15 番地のデータは 9 になり、0 ではありませ

んから、プログラムはLOOPというラベルのあるアドレスにジャンプ（分岐）します。再びアキュムレータに15番地のデータを加えて（10 + 9）、アキュムレータは19になります。また15番地のデータをデクリメントし…ということを0になるまで繰り返すと、アキュムレータには10までの整数の和が残りますから、これを出力ポートに出力して終了します。「10までの整数の加算」と書きましたが、実際には10 + 9 + 8 +…+ 2 + 1という計算をしています。

（c） 最大値を出力する

```
        STORE   15, 207
        STORE   14, 222
        STORE   13, 94
        STORE   12, 178
        STORE   11, 144

        SET_X   11          ; 11 → IX
L0:     READ    IX          ; M (IX) → Acc
        OUT                 ; Acc → output port

L1:     INC_X               ; IX + 1 → IX
        JEZ     L2          ; if IX=0,  jump to L2
        CMP     IX          ; Acc − M (IX)
        JCC     L1          ; if Acc < M (IX),  jump to L1

        JMP     L0          ; jump to L1

L2:     END
```

長いプログラムに見えますが、実際のプログラムは“SET_X”からです（このコンピュータにはメモリが16バイトしかありませんから、そもそも「長いプログラム」を書けるはずがありません！）。

11〜15番地に納められているデータの中から、最も大きい値を見つけて出力しますが、こういう問題はコンピュータが得意とするところです。一連のデータを処理する場合は、インデックスレジスタを使います。

まずインデックスレジスタにデータがある最初の番地、11をセットします。これで“READ　X”命令は、インデックスレジスタのデータ（11）の番地にあるデータ（144）を読み込みます。とりあえずこれを出力ポートに出力しておいて、次にインデックスレジスタをインクリメントします。この結果、インデックスレジスタが0になったらL2にジャンプしてプログラムは終了しますが、今は12ですから、次に“CMP　X”命令を実行します。

“CMP”命令はアキュムレータからメモリのデータを引きますが、アキュムレータの値は変化しません。もしアキュムレータの値（今は144）よりメモリのデータ（この場合は12番地の178）の方が大きければ、キャリーフラグが1になります。JCC命令は、キャリーフラグが0ならL1に分岐します。1なら次の“JMP　L0”命令によってL0にジャンプします。今はキャリーフラグは1ですから、分岐先のREAD命令で比較の結果大きかった方のデータがアキュムレータに読み込まれます。これを出力ポートに出力して、同じ処理をインデックスレジスタが15になるまで繰り返し、次に0になれば修了です。出力ポートはデータの大小によって何度か書き換えられますが、プログラムが終了したときには最大値が書き込まれています。

（d） かけ算（21 × 11）

```
        STORE   15, 21
        STORE   14, 11
        STORE   13, 0

L0:     READ    14          ; M(14) → Acc
        JEZ     L2          ; if Acc = 0,  jump to L2
        SFT_R               ; shift right
```

```
        WRITE   14      ; Acc → M(14)
        JCC     L1      ; if Carry = 0,  jump to L1

        READ    13      ; M(13) → Acc
        ADD     15      ; Acc + M(15) → Acc
        WRITE   13      ; Acc → M(13)

L1:     READ    15      ; M(15) → Acc
        SFT_L           ; shift left
        WRITE   15      ; Acc → M(15)
        JMP     L0      ; jump to L0

L2:     END
```

掛け算のプログラムです。コンピュータですから 2 進数で計算しますが、このプログラムを理解するために、2 進数での掛け算はどうすればいいのかを考えてみます。

といっても、次のように、ふだん行っている 10 進数の掛け算と比べてみるとまったく同じですから、むしろ九九の計算がないだけ、2 進数の方が簡単です。

```
   123          10101
 × 203        × 1011
-------      --------
   369          10101
   000         10101
  246         00000
-------      10101
 24969      --------
            11100111
```

プログラムは、これとまったく同じことをコンピュータに行わせています。この例では被乗数 21 を 15 番地に、乗数 11 を 14 番地にストアします。13 番地には積を累算するため、あらかじめクリアしておきます。

まず、乗数（14 番地 : 1011）を右にシフトして、最下位ビット（キャリーフラグに入る）が 1 であれば被乗数（15 番地 : 10101）を 13 番地に加えて（最下位ビットが 0 なら加えずに）、被乗数を左にシフトします。

これを繰り返して、右にシフトされた乗数が 0 になれば計算終了です。

プログラムを実際に仮想コンピュータに入力して、掛け算が進行していく様子を確認して下さい。なお、このプログラムもメモリ不足のため、結果は出力できずにメモリの中にあるだけです（もっとも、出力ポートも 1 バイトですから、出力するのも難しいですが）。

（e）　車のうっかりミス防止

これは「論理回路（用語解説）」のページの「車のうっかりミス防止回路」と同じ働きをするプログラムです。入力ポートの 0 ビット目にはドアのセンサからの信号が、1 ビット目にはエンジン、2 ビット目はキー、3 ビット目にはライトの信号が接続されています。それぞれの動きと 0, 1 の対応は同ページの表の通りです。また、出力ポートの 0 ビット目はブザーに接続されていて、これが 1 になればブザーが鳴るようになっているものとします。

```
        STORE   15, ♯00000011

LOOP:   IN              ; input port → Acc
        SFT_R           ; shift right
        JCC     LOOP    ; if Carry = 0,  jump to LOOP

        SFT_R           ; shift right
        JCS     LOOP    ; if Carry = 1,  jump to LOOP
```

```
AND     15      ; Acc・M(15) → Acc
JEZ     LOOP    ; if Acc = 0,  jump to LOOP

CLR             ; 0 → Acc
INC             ; Acc + 1 → Acc
OUT             ; Acc → output port

END
```

このプログラムの考え方は、論理回路で構成したものとは少し異なっています。

入力ポートの 0 ビット目にはドアの信号が接続されていますから、ドアが開かない限り（1 でなければ)、ブザーをならす必要はありません。入力ポートのデータをアキュムレータに読み込んで右にシフトし、キャリーフラグが 0 であれば LOOP に分岐して入力命令を繰り返します。

ドアが開いたときに、エンジンがどうなっているかをチェックするためにもう一度右にシフトしてキャリーフラグを検査します。エンジンが動いていればブザーは鳴らさなくていいので、この場合も LOOP に分岐します。

ドアが開いてエンジンが止まったとき、キーやライトがどうなっているかを調べるために、15 番地のデータとの論理積（AND）を取ります。アキュムレータのデータはこのときには 2 度シフトされていますから、2 ビット目、3 ビット目に接続されているキーとライトのデータは 0、1 ビット目に移動しています。従って、これと 15 番地の ♯00000011 との論理積をとると、キーとライトのデータがどちらも 0 であればアキュムレータはゼロになって、やはりブザーは鳴らさなくていいので LOOP に分岐します。

キーとライトのデータのうち、少なくともひとつが 1 であれば、♯00000011 との論理積をとるとアキュムレータの値はゼロになりません。この場合はブザーを鳴らせばいいので、出力ポートの 0 ビット目を 1 にするために先ずアキュムレータをクリアして、その後インクリメントして出力ポートに出力します。

このように、コンピュータの入力ポートにセンサの信号、出力ポートにはブザーの他ライトやモーターなどのアクチュエータ（actuator）を接続して、さまざまの機械、機器を自由に制御することができます。