KUECHIP 3 シミュレータマニュアル

著者

2017 年 10 月 28 日
Ver. 0.8

目次

第 1 章

セットアップ

1.1　セットアップ

KUECHIP 3 シミュレータの実行に必要なセットアップに関する記述.

第 2 章

命令セット

KUECHIP 3 で実行出来る命令に関する記述.

2.1　アセンブラ文法

表 2.1　KUECHIP 3 のアドレスモード

ACC	アキュムレータ
IX	インデックスレジスタ
SP	スタックポインタレジスタ
d	即値アドレス
[d]	絶対アドレス
[IX+d]	インデックス修飾アドレス
[SP+d]	スタックポインタ修飾アドレス

表 2.2　KUECHIP 3 論理アドレスモード

ea	全てのアドレスモード
reg	レジスタアドレスモード
imm	即値アドレスモード
ma	メモリ参照アドレスモード

表 2.3　アドレスモード対応表

	ACC	IX	d	[d]	[IX+d]	[SP+d]
{ea}	○	○	○	○	○	○
{reg}	○	○	×	×	×	×
{imm}	×	×	○	×	×	×
{ma}	×	×	×	○	○	○

表 2.4: KUECHIP 3 命令表

略記号	アドレスモード	命令機能
HLT		Halt
NOP		No Operation
IN		INput
OUT		OUTput
SCF		Set Carry Flag
RCF		Reset Carry Flag
LD	{reg}, {ea}	LoaD
	IX, SP	
	SP, {imm}	
	SP, IX	
ST	{reg}, {ma}	STore
ADD	{reg}, {ea}	ADD
	SP, {imm}	ADD stack
ADC	{reg}, {ea}	ADd with Carry
SUB	{reg}, {ea}	SUBtract
	SP, {imm}	SUBtract stack
SBC	{reg}, {ea}	SuBtract with Carry
CMP	{reg}, {ea}	CoMPare
AND	{reg}, {ea}	AND
OR	{reg}, {ea}	OR
EOR	{reg}, {ea}	Exclusive OR
S*sm*	{reg}	Shift
R*sm*	{reg}	Rotate
RA		Right Arithmetically
LA		Left Arithmetically
RL		Right Logically
LL		Left Logically
B*cc*	{imm}	Branch
A		Always
VF		on oVerFlow
NZ		on Not Zero
Z		on Zero
ZP		on Zero or Positive
N		on Negative
P		on Positive
ZN		on Zero or Negative
NI		on No Input

次ページに続く

略記号	アドレスモード	命令機能
B*cc*	{imm}	Branch
NO		on No Output
NC		on No Carry
C		on Carry
GE		on Greater than or Equal
LT		on Less Than
GT		on Greater Than
LE		on Less than or Equal
INC		INCrement
DEC		DECrement
PSH	{reg}	PuSH
POP	{reg}	POP
CAL		CALl
RET		RETurn

表 2.5 は, KUE-CHIP3 アセンブリのアセンブル時に使用する命令の一覧であり, KUE-CHIP3 および KUE-CHIP3 シミュレータには対応していない.

表 2.5 擬似命令表

略記号	命令機能	詳細
EQU	EQUal	定数をラベル名で定義する
LOC	LOCation	データをセットするアドレスを指定する
DAT	DATa	LOC でセットしたアドレスにデータをセットする

2.2　命令セット

表 2.6　命令セット一覧

略記号	命令コード (1 語目)	B' (2 語目)	命令機能の概略	
NOP	0 0 0 0 0 - - -	×	No Operation	
HLT	0 0 0 0 1 - - -	×	HaLT	停止
	0 1 0 1 - - - -	×		未使用 (HLT)
OUT	0 0 0 1 0 - - -	×	OUTput	(ACC) → OBUF
IN	0 0 0 1 1 - - -	×	INput	(IBUF) → ACC
RCF	0 0 1 0 0 - - -	×	Reset CF	0 → CF
SCF	0 0 1 0 1 - - -	×	Set CF	1 → CF
Bcc	0 0 1 1 cc	◎	Reset CF	条件が成立すれば B' → PC
Ssm	0 1 0 0 A 0 sm	×	Shift sm	(A) → shift, rotate → A
Rsm	0 1 0 0 A 1 sm	×	Rotate sm	はみ出したビット→ CF
LD	0 1 1 0 A B	○	LoaD	(B) → A
	0 0 0 0 0 0 0 1	×		(SP) → IX
	0 0 0 0 0 0 1 0	◎		(d) → SP
	0 0 0 0 0 0 1 1	×		(IX) → SP
ST	0 1 1 1 A B	◎	STore	(A) → B
SBC	1 0 0 0 A B	○	SuB with Carry	(A) − (B) − CF → A
ADC	1 0 0 1 A B	○	ADd with Carry	(A) + (B) + CF → A
SUB	1 0 1 0 A B	○	SUBtract	(A) − (B) → A
	0 0 0 0 0 1 1 1	◎		(SP) − d → SP
ADD	1 0 1 1 A B	○	ADD	(A) + (B) → A
	0 0 0 0 0 1 1 0	◎		(SP) + d → SP
EOR	1 1 0 0 A B	○	Exclusive OR	(A) $\oplus$ (B) → A
OR	1 1 0 1 A B	○	OR	(A) $\vee$ (B) → A
AND	1 1 1 0 A B	○	AND	(A) $\wedge$ (B) → A
CMP	1 1 1 1 A B	○	CoMPare	(A) − (B)
INC	0 0 0 0 0 1 0 0	×	INCrement	(SP + 2) → SP
DEC	0 0 0 0 0 1 0 1	×	DECrement	(SP−2) → SP
PSH	0 0 0 0 1 B	◎	PuSH	(B) → Mem
POP	0 0 0 0 1 0 1 A	◎	POP	(Mem) → B'
CAL	0 0 0 0 1 1 0 0	◎	CALl	サブルーチンの先頭 (B') を呼び出す
RET	0 0 0 0 1 1 0 1	×	RETurn	呼び出し元のルーチンへ戻る

表 2.7 cc: Condition Code

A	0	0	0	0	Always	常に成立
VF	1	0	0	0	on oVerFlow	桁あふれ $VF = 1$
NZ	0	0	0	1	on Not Zero	$\neq 0$ $ZF = 0$
Z	1	0	0	1	on Zero	$= 0$ $ZF = 1$
ZP	0	0	1	0	on Zero or Positive	≥ 0 $NF = 0$
N	1	0	1	0	on Negative	< 0 $NF = 1$
P	0	0	1	1	on Positive	> 0 $(NF \vee ZF) = 0$
ZN	1	0	1	1	on Zero or Negative	≤ 0 $(NF \vee ZF) = 1$
NI	0	1	0	0	on No Input	$IBUF\_FLG\_IN = 0$
NO	1	1	0	0	on No Output	$OBUF\_FLG\_IN = 1$
NC	0	1	0	1	on No Carry	$CF = 0$
C	1	1	0	1	on Carry	$CF = 1$
GE	0	1	1	0	on Greater than or Equal	≥ 0 $(VF \oplus NF) = 0$
LT	1	1	1	0	on Less Than	< 0 $(VF \oplus NF) = 1$
GT	0	1	1	1	on Greater Than	> 0 $((VF \oplus NF) \vee ZF) = 0$
LE	1	1	1	1	on Less than or Equal	≤ 0 $((VF \oplus NF) \vee ZF) = 1$

表 2.8 sm: Shift Mode

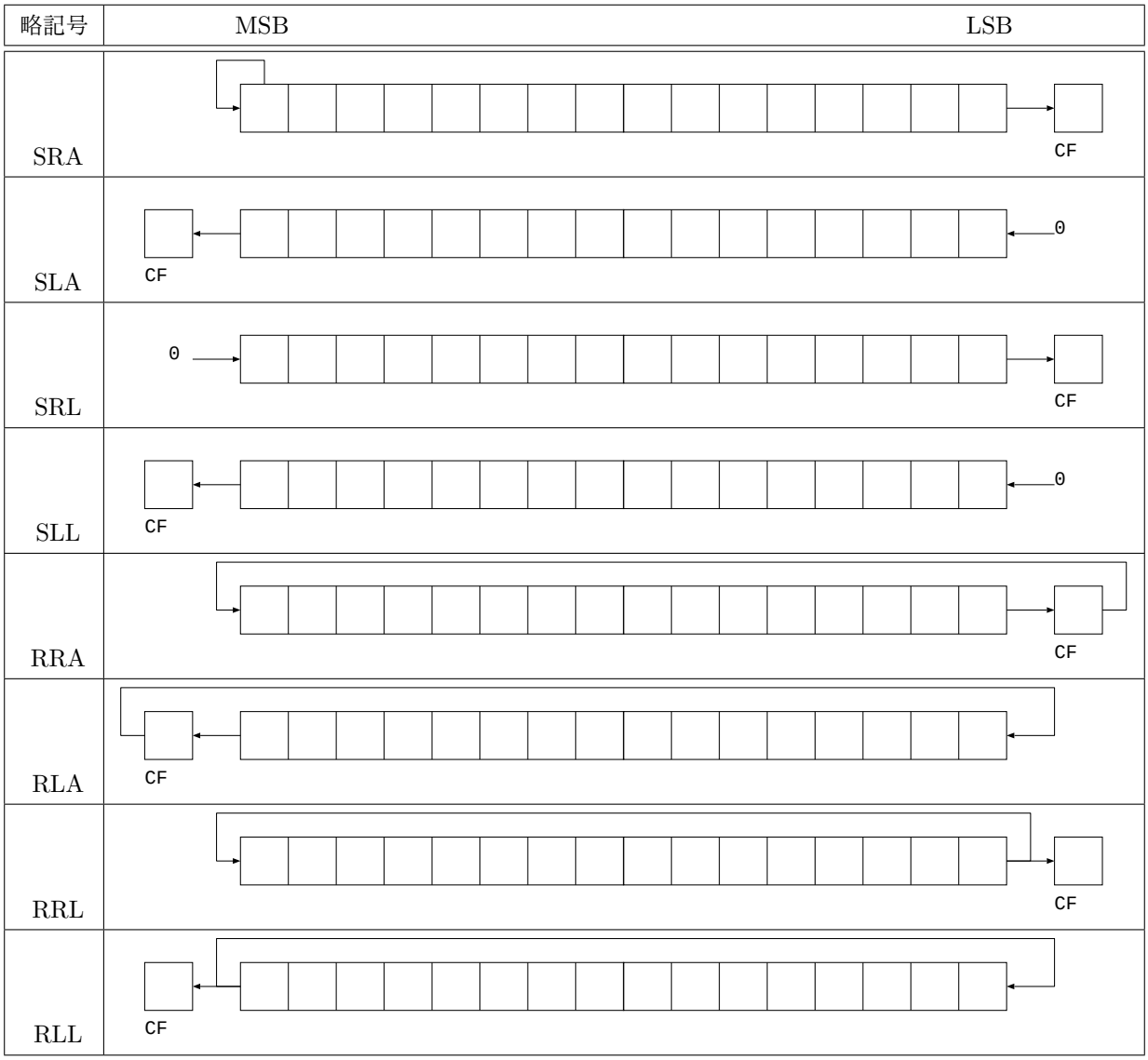
RA	0	0	Right Arithmetically
LA	0	1	Left Arithmetically
RL	1	0	Right Logically
LL	1	1	Left Logically

A = 0 : ACC
A = 1 : IX

B' (2 語目)
×: 不要
○: 不要 or 必要
◎: 必要

B = 000: ACC
B = 001: IX
B = 01-: d (即値アドレス)
B = 100: [d] (絶対アドレス)
B = 110: [IX+d] (インデックス修飾アドレス)
B = 011: [SP+d] (スタックポインタ修飾アドレス)

2.3　Shift / Rotate 命令の機能



2.4 LoaD / STore 命令の機能

KUE-CHIP3 は 16 ビットアドレスであり, 偶数番地のメモリアクセスを行う. 奇数番地のメモリアクセスがあった場合, その番地から 1 を引いた番地へのアクセスを行う.

メモリアクセスを要求する番地	実際にアクセスする番地
0x0000	0x0000
0x0001	
0x0002	0x0002
0x0003	
0x0004	0x0004
0x0005	
0x0006	0x0006
0x0007	
0x0008	0x0008
0x0009	
0x000A	0x000A
0x000B	
0x000C	0x000C
0x000D	
0x000E	0x000E
0x000F	

2.5　スタックに関連する機能

本シュミレータ上では, スタックポインタ (SP) が指す先にはデータが格納されている. 図 2.1 のようにデータが格納されているとし, スタックは下 (0xFFFF) から上 (0x0000) 方向に伸びていく.

PSH 命令は, SP を移動してからデータの書き込みを行う. POP 命令では, データの読み込み後に SP を移動する.

CAL 命令は, CAL 命令の次命令の PC をメモリ (スタック) に記憶し, サブルーチンを呼び出す. RET 命令は, SP が指す場所に格納されている戻り先アドレスを取得し, そのアドレスへ戻る.

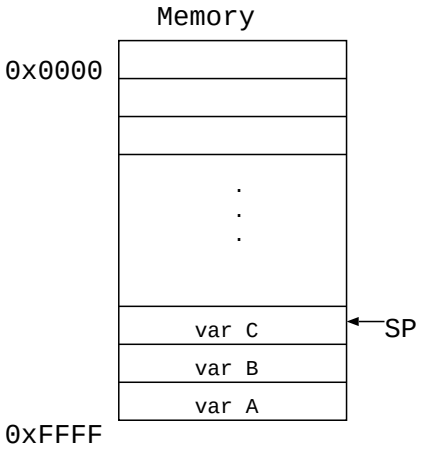


図 2.1　メモリのイメージ図

2.6 フラグ機能

表 2.9: KUECHIP 3 命令表

CF : Carry Flag　　VF : oVerflow Flag　　NF : Negative Flag　　ZF : Zero Flag

略記号	命令機能の概略	実行への影響 †				実行後の影響 ‡			
		CF	VF	NF	ZF	CF	VF	NF	ZF
NOP	No Operation	-	-	-	-	-	-	-	-
HLT	Halt	-	-	-	-	-	-	-	-
OUT	OUTput	-	-	-	-	-	-	-	-
IN	INput	-	-	-	-	-	-	-	-
RCF	Reset Carry Flag	-	-	-	-	0	-	-	-
SCF	Set Carry Flag	-	-	-	-	1	-	-	-
SRA	Shift Right Arithmetically	-	-	-	-	b0	0	N	Z
SLA	Shift Left Arithmetically	-	-	-	-	b7	V	N	Z
SRL	Shift Right Logically	-	-	-	-	b0	0	N	Z
SLL	Shift Left Logically	-	-	-	-	b7	0	N	Z
RRA	Rotate Right Arithmetically	b7	-	-	-	b0	0	N	Z
RLA	Rotate Left Arithmetically	b0	-	-	-	b7	V	N	Z
RRL	Rotate Right Logically	-	-	-	-	b0	0	N	Z
RLL	Rotate Left Logically	-	-	-	-	b7	0	N	Z
LD	LoaD	-	-	-	-	-	-	-	-
ST	STore	-	-	-	-	-	-	-	-
SBC	SuBtract with Carry	c	-	-	-	C	V	N	Z
ADC	ADd with Carry	c	-	-	-	C	V	N	Z
SUB	SUBtract	-	-	-	-	-	V	N	Z
ADD	ADD	-	-	-	-	-	V	N	Z
EOR	Exclusive OR	-	-	-	-	-	0	N	Z
OR	OR	-	-	-	-	-	0	N	Z
AND	AND	-	-	-	-	-	0	N	Z
CMP	CoMPare	-	-	-	-	-	V	N	Z
BA	Branch Always	-	-	-	-	-	-	-	-
BVF	Branch on oVerFlow	-	VF	-	-	-	-	-	-
BNZ	Branch on Not Zero	-	-	-	$\overline{ZF}$	-	-	-	-
BZ	Branch on Zero	-	-	-	ZF	-	-	-	-
BZP	Branch on Zero or Positive	-	-	$\overline{NF}$	-	-	-	-	-
BN	Branch on Negative	-	-	NF	-	-	-	-	-
BP	Branch on Positive	-	-	$\overline{NF \vee ZF}$		-	-	-	-
BZN	Branch on Zero or Negative	-	-	$NF \vee ZF$		-	-	-	-

次ページに続く

略記号	命令機能の概略	実行への影響 †				実行後の影響 ‡			
		CF	VF	NF	ZF	CF	VF	NF	ZF
BNI	Branch on No Input	-	-	-	-	-	-	-	-
BNO	Branch on No Output	-	-	-	-	-	-	-	-
BNC	Branch on No Carry	$\overline{CF}$	-	-	-	-	-	-	-
BC	Branch on Carry	CF	-	-	-	-	-	-	-
BGE	Branch on Greater than or Equal	-	$\overline{VF \oplus NF}$		-	-	-	-	-
BLT	Branch on Less Than	-	$VF \oplus NF$		-	-	-	-	-
BGT	Branch on Greater Than	-	$\overline{(VF \oplus NF) \vee ZF}$			-	-	-	-
BLE	Branch on Less than or Equal	-	$(VF \oplus NF) \vee ZF$			-	-	-	-
INC	INCrement	-	-	-	-	-	-	-	-
DEC	DECrement	-	-	-	-	-	-	-	-
PSH	PuSH	-	-	-	-	-	-	-	-
POP	POP	-	-	-	-	-	-	-	-
CAL	CALl	-	-	-	-	-	-	-	-
RET	RETurn	-	-	-	-	-	-	-	-

† 実行への影響

c	:	Referred to as carry/borrow input.
b7	:	Substituted into bit 7 of the operand A.
b0	:	Substituted into bit 0 of the operand A.
Exp.	:	Condition which causes branch.
-	:	No effect.

‡ 実行後の状態

C	:	Set to 1 iff carry/borrow occur.
V	:	Set to 1 iff overflow occur.
N	:	Set to 1 iff bit 7 of the result is 1.
Z	:	Set to 1 iff all bits of the result are 0.
b7	:	Set to 1 iff bit 7 of the operand A was 1.
b0	:	Set to 1 iff bit 0 of the operand A was 1.
0	:	Set to 0.
1	:	Set to 1.
-	:	Not modified.

2.7 命令コード早見表

表 2.10 ACC に対する代入および算術命令

	ACC	IX	d	[SP+d]	[d]	[IX+d]	機能	
LD ACC,	60	61	62	63	64	66	ACC にデータを格納.	A → ACC
ST ACC,	-	-	-	73	74	76	ACC のデータを格納.	ACC → A
SBC ACC,	80	81	82	83	84	86	桁借りを考慮した減算.	ACC - B - CF → ACC
ADC ACC,	90	91	92	93	94	96	桁上げを考慮した加算.	ACC + B + CF → ACC
SUB ACC,	A0	A1	A2	A3	A4	A6	減算.	ACC - B → ACC
ADD ACC,	B0	B1	B2	B3	B4	B6	加算.	ACC + B → ACC
EOR ACC,	C0	C1	C2	C3	C4	C6	排他的論理和.	ACC $\oplus$ B → ACC
OR ACC,	D0	D1	D2	D3	D4	D6	論理和.	ACC $\vee$ B → ACC
AND ACC,	E0	E1	E2	E3	E4	E6	論理積.	ACC $\wedge$ B → ACC
CMP ACC,	F0	F1	F2	F3	F4	F6	ACC と比較.	ACC - B

表 2.11 IX に対する代入および算術命令

	ACC	IX	d	[SP+d]	[d]	[IX+d]	機能	
LD IX,	68	69	6A	6B	6C	6E	IX にデータを格納.	A → IX
ST IX,	-	-	-	7B	7C	7E	IX のデータを格納.	IX → A
SBC IX,	88	89	8A	8B	8C	8E	桁借りを考慮した減算.	IX - B - CF → IX
ADC IX,	98	99	9A	9B	9C	9E	桁上げを考慮した加算.	IX + B + CF → IX
SUB IX,	A8	A9	AA	AB	AC	AE	減算.	IX - B → IX
ADD IX,	B8	B9	BA	BB	BC	BE	加算.	IX + B → IX
EOR IX,	C8	C9	CA	CB	CC	CE	排他的論理和.	IX $\oplus$ B → IX
OR IX,	D8	D9	DA	DB	DC	DE	論理和.	IX $\vee$ B → IX
AND IX,	E8	E9	EA	EB	EC	EE	論理積.	IX $\wedge$ B → IX
CMP IX,	F8	F9	FA	FB	FC	FE	IX と比較.	IX - B

表 2.12　スタックに関する代入および算術命令

LD IX, SP	01	IX にデータを格納.	SP → IX
LD SP, d	02	SP にデータを格納.	d → SP
LD SP, IX	03	SP にデータを格納.	IX → SP
INC	04	SP をインクリメント.	SP + 2 → SP
DEC	05	SP をデクリメント.	SP - 2 → SP
ADD SP, d	06	加算.	SP + d → SP
SUB SP, d	07	減算.	SP + d → SP
PSH ACC	08	ACC をプッシュ.	ACC → Mem
PSH IX	09	IX をプッシュ.	IX → Mem
POP ACC	0A	取り出した値を ACC に格納	Mem → ACC
POP IX	0B	取り出した値を IX に格納	Mem → IX
CAL	0C	サブルーチンの先頭を呼び出す.	
RET	0D	呼び出し元のルーチンへ戻る.	

表 2.13　分岐命令

名前	機械語	機能
BA	30	必ず分岐
BVF	38	桁あふれがあったら分岐
BNZ	31	0 以外だったら分岐
BZP	32	0 だったら分岐
BP	33	0 以上だったら分岐
BNI	34	Input が無かったら分岐
BNC	35	桁上げが無かったら分岐
BGE	36	$A \geq 0$ または $A = 0$ なら分岐
BGT	37	$A > 0$ または $A = 0$ なら分岐
BZ	39	0 だったら分岐
BN	3A	0 未満だったら分岐
BNZ	3B	0 以外だったら分岐
BNO	3C	Output が無かったら分岐
BC	3D	桁上げがあったら分岐
BLT	3E	$A < 0$ か EOR の結果 1 なら分岐
BLE	3F	$A \leq 0$ か EOR の結果 1 なら分岐

表 2.14　シフト命令

	ACC	IX	機能
SRA	40	48	Shift Right Arithmetically
SLA	41	49	Shift Left Arithmetically
SRL	42	4A	Shift Right Logically
SLL	43	4B	Shift Left Logically
RRA	44	4C	Rotate Right Arithmetically
RLA	45	4D	Rotate Left Arithmetically
RRL	46	4E	Rotate Right Logically
RLL	47	4F	Rotate Left Logically

表 2.15　制御命令

NOP	00	No Operation
HLT	0F	終了命令
OUT	10	出力
IN	1F	入力
RCF	20	CF をリセット
SCF	2F	CF をセット

2.8 命令実行フェーズ

Instruction \ Phase	P0	P1	P2	P3	P4
HLT	(PC) → MAR (PC+2) → PC	(Mem) → IR	HLT		
LD IX, SP			SP → IX		
LD SP, d			(PC) → MAR (PC+2) → PC	(Mem) → SP	
INC			+2 → (SP) → ALU → SP		
DEC			−2 → (SP) → ALU → SP		
ADD SP, d SUB SP, d			(PC) → MAR (PC+2) → PC	(SP) → (Mem) → ALU → SP	
PSH ACC (IX)			(SP−2) → MAR (SP−2) → SP	(ACC(IX)) → Mem	
POP ACC (IX)			(SP) → MAR (SP+2) → SP	Mem → ACC(IX)	
CAL			(SP−2) → MAR (SP−2) → SP	(PC+2) → Mem (PC) → MAR	(Mem) → PC
RET			(SP) → MAR (SP+2) → SP	(Mem) → PC	
NOP			No Operation		
OUT			(ACC) → OBUF	0 → OBUF_WE	
IN			(IBUF) → ACC 0 → IBUF_RE	0 → IBUF_FLG_CLR	

Phase / Instruction		P0	P1	P2	P3	P4
Bcc		(PC) → MAR (PC+2) → PC	(Mem) → IR	(PC) → MAR (PC+2) → PC	STATUS CHECK (Mem) → PC (if condition)	
Ssm Rsm				TCF SET SHIFT A	NF, ZF, VF, CF Set	
LD	ACC IX			(A) → B		
	d				(Mem) → A	
	[d]			(PC) → MAR	(Mem) → MAR	
	[IX+d] [SP+d]			(PC+2) → PC	(IX(SP)) → (Mem) → ALU → MAR	(MEM) → A
ST	[d]			(PC) → MAR	(Mem) → MAR	
	[IX+d] [SP+d]			(PC+2) → PC	(IX(SP)) → (Mem) → ALU → MAR	(A) → (Mem)
SBC ADC SUB ADD	ACC IX			(A) → (B) → ALU → A [(CF)] → NF, ZF, VF [, CF] SET		
EOR OR AND CMP	d			(PC) → MAR (PC+2) → PC	(A) → (B) → ALU → A [(CF)] → NF, ZF, VF [, CF] SET	
	[d]				(Mem) → MAR	(A) → (B) → ALU → A [(CF)] → NF, ZF, VF [, CF] SET
	[IX+d] [SP+d]				(IX(SP)) → (Mem) → ALU → MAR	

第 3 章

プログラムの入力と実行

3.1 プログラムの入力と実行

この章では KUE-CHIP3 シミュレータを円滑に扱えるよう, あるプログラムを入力し実行に至るまでの操作方法を, 非常に小さなプログラムを例にとり具体的に示す.

3.2 例題

下に示すプログラムは KUE-CHIP3 において, ACC × IX → ACC を計算し, メモリにストアする非常に簡単なプログラムである.

```
ADDRS   DATA        OPECODE
0000:   00C0        #  EOR ACC, ACC
0002:   00C9        #  EOR IX, IX
0004:   00BA 0010   #  ADD IX, 10H
0008:   00B2 0003   #  ADD ACC, 03H
000C:   00AA 0001   #  SUB IX, 1
0010:   0031 0008   #  BNZ 08H
0014:   0074 0020   #  ST ACC, [20H]
0018:   000F        #  HLT
```

3.3 例題の入力

前節で挙げた例題プログラムの入力方法を具体的に述べる.

1. [inst] の下のテキストボックスに例題を入力する. (図 3.1)
2. [setmemory] ボタンを押し, [memory] の下のテキストボックスにメモリがセットされたことを確認する. (図 3.2)

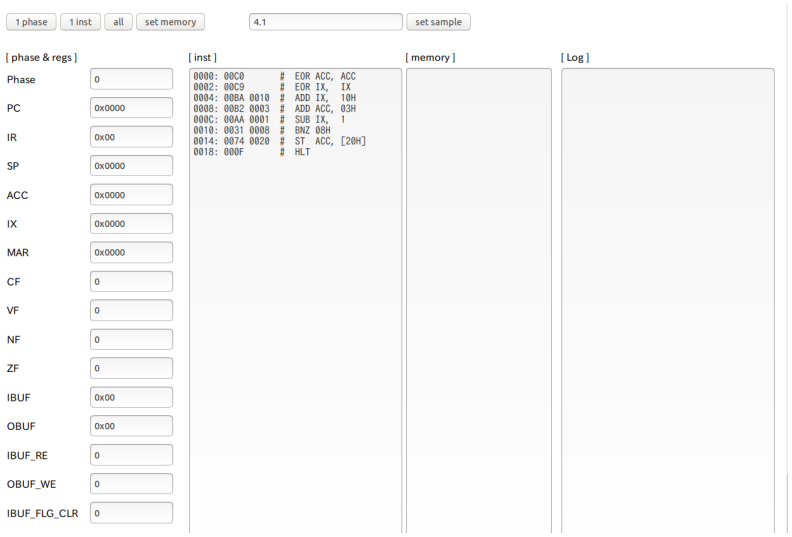


図 3.1　例題を入力した図

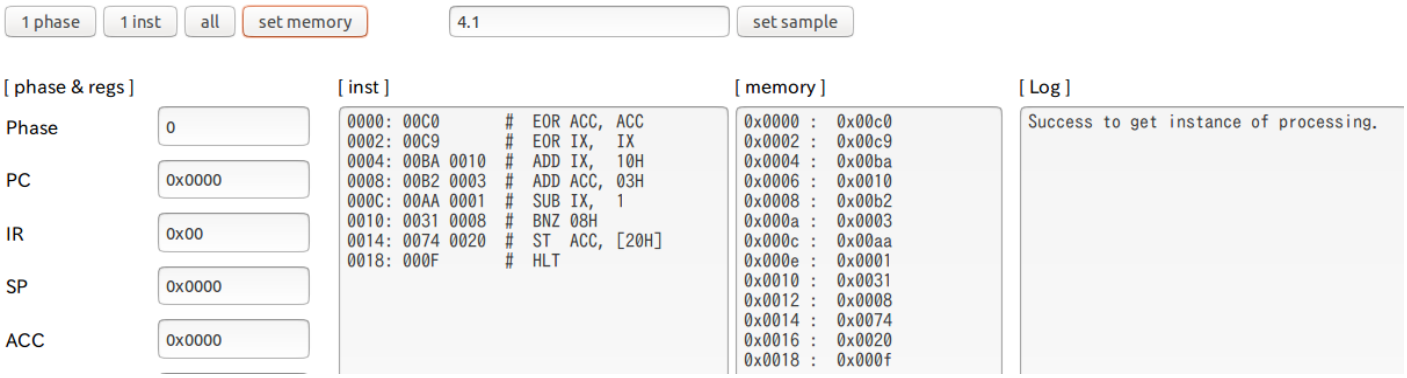


図 3.2　例題をメモリにセットした図

3.4　例題の実行

前節で入力した例題プログラムで $3 \times 10 = 30$ を計算してみる．KUECHIP3 シミュレータでは，1 フェイズ実行，1 命令実行，全命令実行の 3 種の実行モードがある．

3.4.1　1 フェイズ実行モード

1 フェイズ実行モードでは，命令フェイズを 1 つずつ実行する実行モードである．本シミュレータでは [1 phase] ボタンを押すことで実行する．

1. [1 phase] ボタンを押す．(図 3.3) このとき，本シミュレータは `00C0` (EOR ACC, ACC) 命令の P0 (フェイズ 0) を実行した結果を出力する．Phase は実行し終えたフェイズを出力するため 0, MAR が 0x0000, PC が 0x0002 となる．
2. もう一度 [1 phase] ボタンを押す．(図 3.4) このとき，本シミュレータは `00C0` (EOR ACC, ACC) 命令の P1 を実行した結果を出力するため，Phase に 1, IR にメモリの MAR 番地 (0x0000) を格納した結果

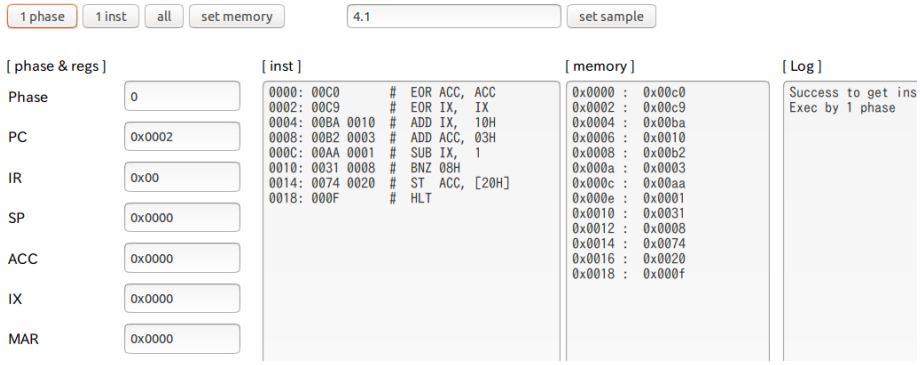


図 3.3 [1 phase] ボタンを押した図

(0x00c0) を出力する.

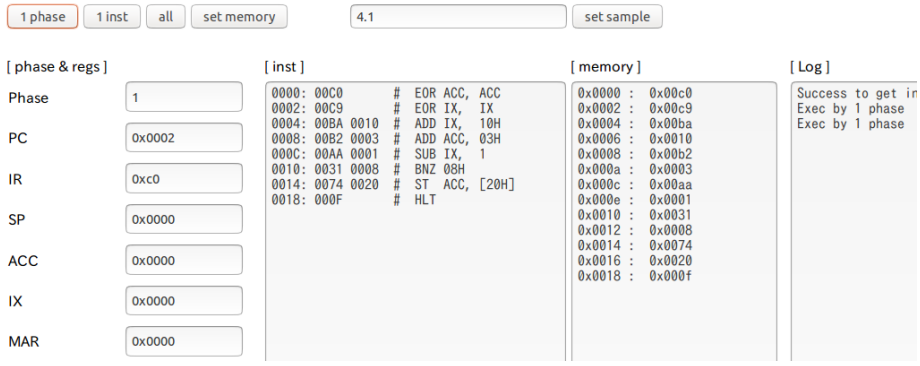


図 3.4 [1 phase] ボタンを二度押した図

3. 実行を繰り返し, `00AA` (SUB IX, 1) 命令を終え IX = 0 になったとき, Phase が 3, ZF (ゼロフラグ) が 1 (True) と出力される. (図 3.5)
4. さらに実行を繰り返し, `0074` (ST ACC, [20H]) 命令を終えた時, メモリの 0x0020 番地に ACC の値がストアされる. (図 3.6)
5. さらに実行を繰り返し, `000F` (HLT) 命令を実行したときシミュレーションを終了する.

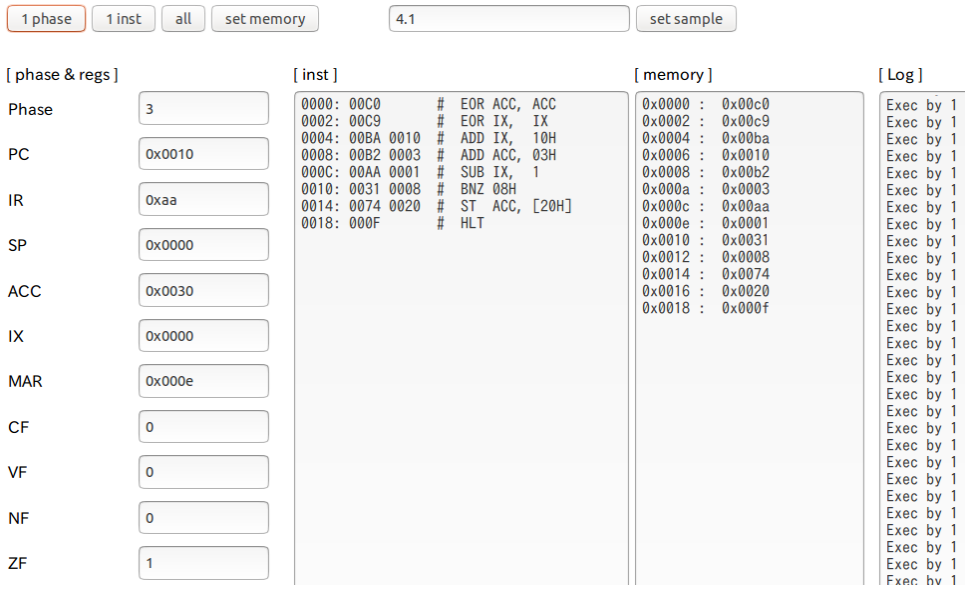


図 3.5　IX = 0 となったときの図

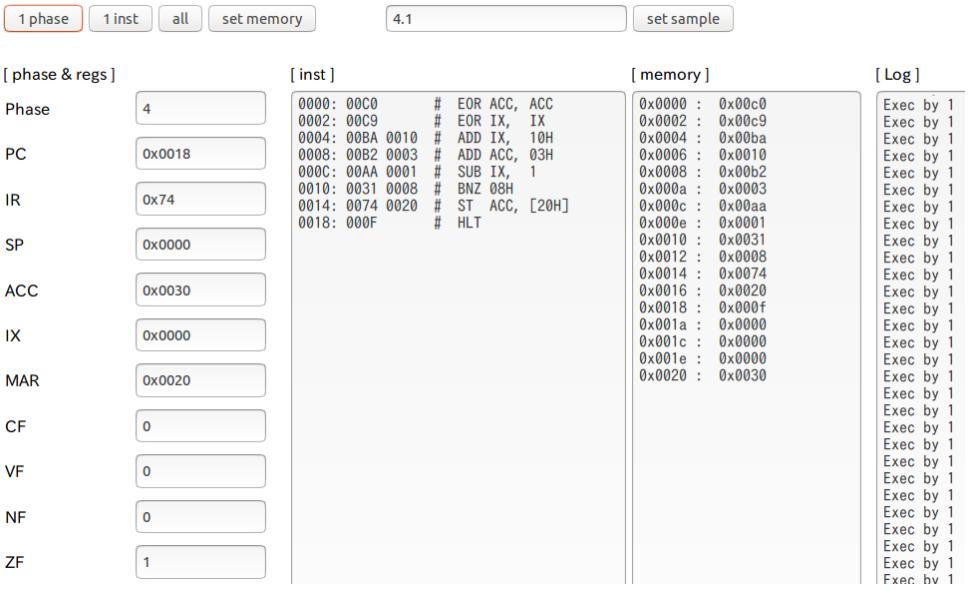


図 3.6　データがメモリにストアされた図

3.4.2 1 命令実行モード

1 命令実行モードでは, 1 命令ごとに実行する実行モードである. 本シュミレータでは [1 inst] ボタンを押すことで実行する.

1. [1 inst] ボタンを押す. (図 3.7) このとき, `00C0` (EOR ACC, ACC) 命令を実行した結果を出力するため, Phase, PC, IR, ACC, MAR, ZF の値がそれぞれ更新される.

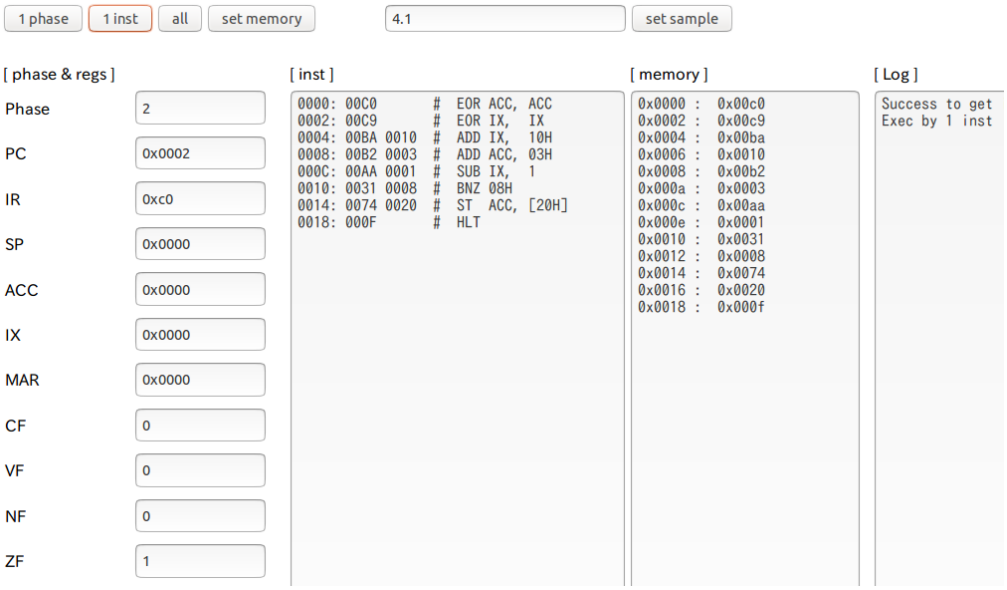


図 3.7 [1 inst] ボタンを一回押した図

2. もう一度 [1 inst] ボタンを押す. (図 3.8) このとき, `00C9` (EOR IX, IX) 命令を実行した結果を出力するため, Phase, PC, IR, IX, MAR, ZF の値がそれぞれ更新される.
3. 実行を繰り返し, `000F` (HLT) 命令を実行し終えたときシミュレーションを終了する.

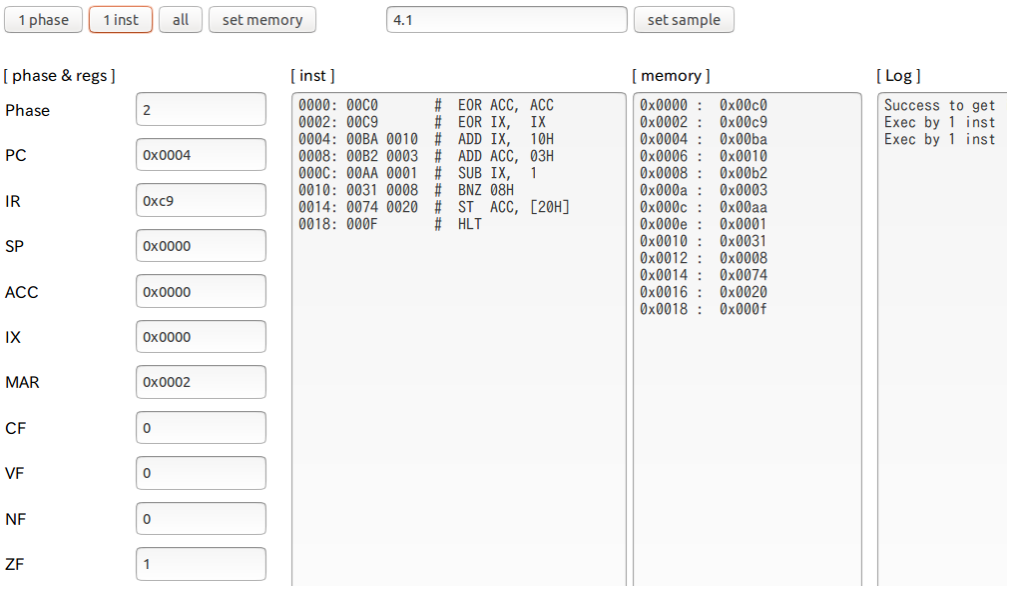


図 3.8　[1 inst] ボタンを二回押した図

3.4.3 全命令実行モード

全命令実行モードでは, すべての命令を一度に実行するモードである. 本シュミレータでは [all] ボタンを押すことで実行する.

1. [all] ボタンを押すと, すべての命令を実行し終えた結果を出力する. (図 3.9)
   このとき, 極端に大きなプログラムや無限ループを持つプログラムを全命令実行モードで実行するとシミュレータが停止するおそれがある.

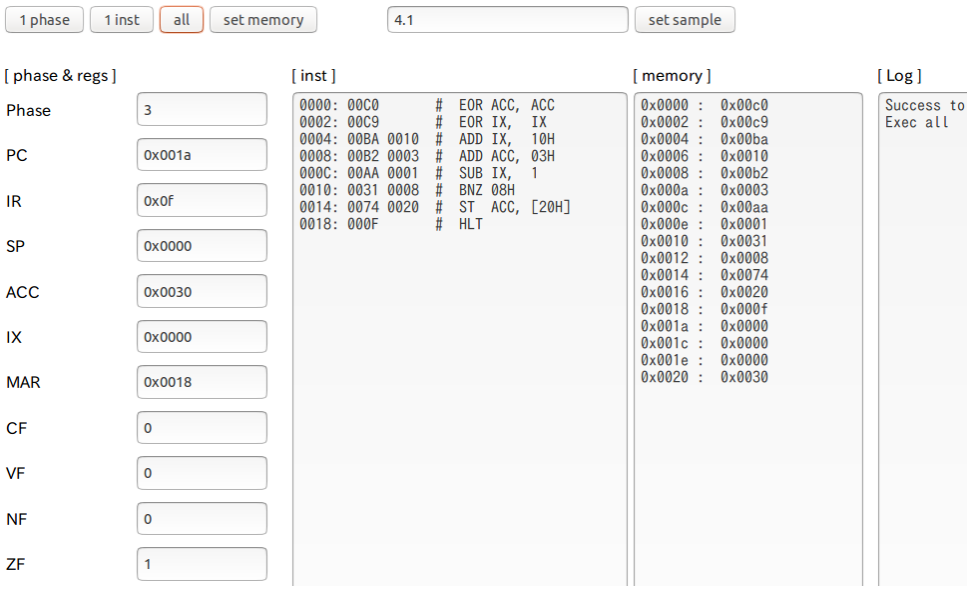


図 3.9 [all] ボタンを押した図

ここでは, 非常に簡単なプログラムで操作法を例示した. より複雑なプログラムを動かすのも同様の手順である.

第 4 章

サンプルプログラム

この章では, KUECHIP3 のサンプルプログラムを示す. 各プログラムには, 簡単な説明と入出力の例をつけておいた.

4.1　1 から N までの和

説明

1 から N (80H 番地) までの和を求め, SUM (82H 番地) に格納する. なお, 和は符号無し 1 バイトの範囲である.

入出力例

入力　`0080: 000A`　(1 から 10 までの和を求める)

出力　`0082: 0037`　(結果: 55)

コード

KUE-CHIP3 シミュレータの上部に 4.1 を入力し, [set sample] ボタンを押すとコードが入力される. (図 4.1)

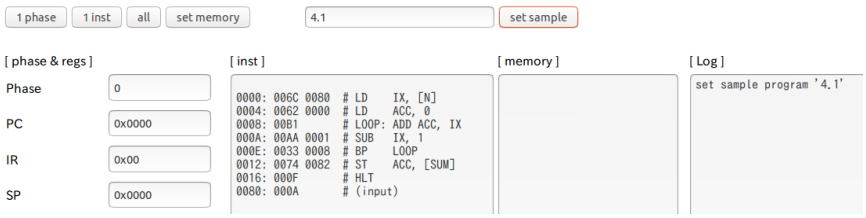


図 4.1　[set sample] ボタンを押してコードをセットした図

アセンブラによって生成された下記コードを直接入力することも可能である.

```
                # N:    EQU 80H
                # SUM:  EQU 82H
                #       LOC 80H
0080: 000A      #       DAT AH
0000: 006C 0080 #       LD IX, [N]
0004: 0062 0000 #       LD ACC, 0
0008: 00B1      # LOOP: ADD ACC, IX
000A: 00AA 0001 #       SUB IX, 1
000E: 0033 0008 #       BP LOOP
0012: 0074 0082 #       ST ACC, [SUM]
0016: 000F      #       HLT
```

4.2 多倍長の加算

説明

インデックス修飾と, キャリーフラグを使用することによって, DATA1 (80H 番地) と DATA2 (90H 番地) からの, N (C0H 番地) バイトの 2 つの数を小さい番地側を MSB として加算し, ANS (A0H 番地) に格納する.

入出力例

入力 `0080: 0000 0001` (データ 1)
`0090: 1FFF FFFF` (データ 2)
`00C0: 0004` (語長)
出力 `00A0: 2000 0000` (結果)

コード

```
                    # N:     EQU C0H
                    # DATA1: EQU 80H
                    # DATA2: EQU 90H
                    # ANS:   EQU A0H
                    #        LOC 80H
0080: 0000          #        DAT 0H
0082: 0001          #        DAT 1H
                    #        LOC 90H
0090: 1FFF          #        DAT 1FFFH
0092: FFFF          #        DAT FFFFH
                    #        LOC C0H
00C0: 0004          #        DAT 4H
0000: 006C 00C0     #        LD IX, [N]
0004: 0020          #        RCF
0006: 0066 007E     # LOOP:  LD ACC, [IX+DATA1-2]
000A: 0096 008E     #        ADC ACC, [IX+DATA2-2]
000E: 0076 009E     #        ST ACC, [IX+ANS-2]
0012: 00AA 0002     #        SUB IX, 2
0016: 0033 0006     #        BP LOOP
001A: 000F          #        HLT
```

4.3　ユークリッドの互除法による最大公約数

説明

ユークリッドの互除法で, A (80H 番地), B (82H 番地) の最大公約数を求め, GCD (84H 番地) に格納する. なお, A, B とも, 127 以下の正数とする.

入出力例

入力　0080: 0060　(A)
　　0082: 0040　(B)
出力　0084: 0020　(最大公約数)

コード

```
                    # A:    EQU 80H
                    # B:    EQU 82H
                    # GCD:  EQU 84H
                    #       LOC 80H
0080: 0060          #       DAT 60H
0082: 0040          #       DAT 40H
0000: 0064 0080     #       LD  ACC, [A]
0004: 006C 0082     #       LD  IX, [B]
0008: 00A1          # LOOP: SUB ACC, IX
000A: 0032 0008     #       BZP LOOP
000E: 00B1          #       ADD ACC, IX
0010: 00C8          #       EOR IX, ACC
0012: 00C1          #       EOR ACC, IX
0014: 00C8          #       EOR IX, ACC
0016: 0031 0008     #       BNZ LOOP
001A: 0074 0084     #       ST ACC, [GCD]
001E: 000F          #       HLT
```

4.4 バブルソートによる整列

説明

バブルソートによって, DATA (0100H 番地) から始まる, N (0080H 番地) 個のデータを昇順に整列する. なお, ソートの完了の判定に, キャリーフラグを利用している.

入出力例

```
入力  0080: 0008                                    (ソートするデータの長さ)
      0100: 1000 FFFF 0040 808E C0A0 D008 7FFF CD00  (ソートするデータ)
出力  0100: 808E C0A0 CD00 D008 FFFF 0040 1000 7FFF  (結果)
```

コード

```
                  # DATA:  EQU 0100H
                  # N:     EQU 80H
                  # WORK1: EQU 90H
                  # WORK2: EQU 92H
                  #        LOC 80H
0080: 0008        #        DAT 08H
                  #        LOC 100H
0100: 1000        #        DAT 1000H
0102: FFFF        #        DAT FFFFH
0104: 0040        #        DAT 0040H
0106: 808E        #        DAT 808EH
0108: C0A0        #        DAT C0A0H
010A: D008        #        DAT D008H
010C: 7FFF        #        DAT 7FFFH
010E: CD00        #        DAT CD00H
0000: 006C 0080   #        LD IX, [N]
0004: 00B9        #        ADD IX, IX
0006: 00AA 0002   #        SUB IX, 2
000A: 007C 0090   #        ST IX, [WORK1]
000E: 00C9        # LP1:   EOR IX, IX
0010: 0020        #        RCF
0012: 0066 0100   # LP2:   LD ACC, [IX+DATA]
0016: 00F6 0102   #        CMP ACC, [IX+DATA+2]
001A: 003F 0034   #        BLE SKIP
001E: 0074 0092   #        ST ACC, [WORK2]
0022: 0066 0102   #        LD ACC, [IX+DATA+2]
0026: 0076 0100   #        ST ACC, [IX+DATA]
002A: 0064 0092   #        LD ACC, [WORK2]
```

```
002E: 0076 0102 #        ST ACC, [IX+DATA+2]
0032: 0028      #        SCF
0034: 00BA 0002 # SKIP:  ADD IX, 2
0038: 00FC 0090 #        CMP IX, [WORK1]
003C: 0031 0012 #        BNZ LP2
0040: 0035 0054 #        BNC FIN
0044: 006C 0090 #        LD IX, [WORK1]
0048: 00AA 0002 #        SUB IX, 2
004C: 007C 0090 #        ST IX, [WORK1]
0050: 0031 000E #        BNZ LP1
0054: 000F      # FIN:   HLT
```

4.5 CRC の計算

説明

シフト／ローテート命令や, 論理演算命令を使用して, メモリ上の DATA (100H 番地) 番地から始まる N (120 番地) バイトのデータの CRC コードを計算する. ただし, データの最大長は 256 バイト (未確認) である. データが 256 バイトの場合は, バイト数を 0 とする. なお, 生成多項式として, CCITT X.25 規格の $x^{16} + x^{12} + x^5 + 1$ を用いた.

入出力例

```
入力  100: 62FF 75C0 75C1 C97D  (DATA)
      120: 0008                 (N)
出力  80: B08A                  (結果)
```

コード

```
                  # DATA: EQU 100H
                  #       LOC 100H
0100: 0062        #       DAT 62H
                  #       LOC 102H
0102: 00FF        #       DAT FFH
                  #       LOC 104H
0104: 0075        #       DAT 75H
                  #       LOC 106H
0106: 00C0        #       DAT C0H
                  #       LOC 108H
0108: 0075        #       DAT 75H
                  #       LOC 10AH
010A: 00C1        #       DAT C1H
                  #       LOC 10CH
010C: 00C9        #       DAT C9H
                  #       LOC 10EH
010E: 007D        #       DAT 7DH
                  # N:    EQU 120H
                  #       LOC 120H
0120: 0008        #       DAT 08H
                  # CRC:  EQU 80H
                  # WORK: EQU F0H
                  # TMP:  EQU F2H
0000: 0062 FFFF   #       LD ACC, FFFFH
0004: 0074 0080   #       ST ACC, [CRC]
0008: 00C9        #       EOR IX, IX
```

```
000A: 007C 00F0 #       ST IX, [WORK]
000E: 007C 00F2 # LP1:  ST IX, [TMP]
0012: 00B9      #       ADD IX, IX
0014: 0066 0100 #       LD ACC, [IX+DATA]
0018: 006C 00F2 #       LD IX, [TMP]
001C: 0043      #       SLL ACC
001E: 0043      #       SLL ACC
0020: 0043      #       SLL ACC
0022: 0043      #       SLL ACC
0024: 0043      #       SLL ACC
0026: 0043      #       SLL ACC
0028: 0043      #       SLL ACC
002A: 0043      #       SLL ACC
002C: 00C4 0080 #       EOR ACC, [CRC]
0030: 0074 0080 #       ST ACC, [CRC]
0034: 006A 0010 #       LD IX, 16
0038: 0064 0080 # LP2:  LD ACC, [CRC]
003C: 0043      #       SLL ACC
003E: 0074 0080 #       ST ACC, [CRC]
0042: 0035 004E #       BNC SKIP
0046: 00C4 0078 #       EOR ACC, [POLY]
004A: 0074 0080 #       ST ACC, [CRC]
004E: 00AA 0002 # SKIP: SUB IX, 2
0052: 0033 0038 #       BP LP2
0056: 006C 00F0 #       LD IX, [WORK]
005A: 00BA 0001 #       ADD IX, 1
005E: 00FC 0120 #       CMP IX  [N]
0062: 007C 00F0 #       ST IX, [WORK]
0066: 0031 000E #       BNZ LP1
006A: 0064 0080 #       LD ACC, [CRC]
006E: 00C2 FFFF #       EOR ACC, FFFFH
0072: 0074 0080 #       ST ACC, [CRC]
0076: 000F      #       HLT
0078: 1021      # POLY: PROG 1021H
```

4.6 符号無し 1 バイトの乗算

説明

DATA1 (80H 番地) と DATA2 (82H 番地) の積を求め, ANS (60H 番地) からの 4 バイトに格納する. なお, 乗数, 被乗数, 積は, 全て符号無しの数とする.

入出力例

入力	`0080: 0555`	(乗数)
	`0082: 0FFF`	(被乗数)
出力	`0060: 0055 4AAB`	(結果)

コード

```
                  # DATA1: EQU 80H
                  # DATA2: EQU 82H
                  # ANS:   EQU 60H
                  # WORK:  EQU 90H
                  #        LOC 80H
0080: 0555        #        DAT 555H
                  #        LOC 82H
0082: 0FFF        #        DAT FFFH
0000: 00C0        #        EOR ACC, ACC
0002: 0074 0060   #        ST ACC, [ANS]
0006: 0074 0062   #        ST ACC, [ANS+2]
000A: 0074 0090   #        ST ACC, [WORK]
000E: 0064 0082   #        LD ACC, [DATA2]
0012: 0074 0092   #        ST ACC, [WORK+2]
0016: 006C 0080   #        LD IX, [DATA1]
001A: 004A        # LOOP:  SRL IX
001C: 0035 003A   #        BNC      SKIP
0020: 0020        #        RCF
0022: 0064 0062   #        LD ACC, [ANS+2]
0026: 0094 0092   #        ADC ACC, [WORK+2]
002A: 0074 0062   #        ST ACC, [ANS+2]
002E: 0064 0060   #        LD ACC, [ANS]
0032: 0094 0090   #        ADC ACC, [WORK]
0036: 0074 0060   #        ST ACC, [ANS]
003A: 00FA 0000   # SKIP:  CMP IX, 0
003E: 0039 005A   #        BZ FIN
0042: 0064 0092   #        LD ACC, [WORK+2]
0046: 0041        #        SLA ACC
0048: 0074 0092   #        ST ACC, [WORK+2]
004C: 0064 0090   #        LD ACC, [WORK]
0050: 0045        #        RLA ACC
0052: 0074 0090   #        ST ACC, [WORK]
0056: 0030 001A   #        BA LOOP
005A: 000F        # FIN:   HLT
```

4.7 符号無し 16 バイトの乗算

説明

DATA1 (A0H 番地) からの 8 バイトと, DATA2 (B0H 番地) からの 8 バイトを小さい番地側を MSB として積を求め, RESULT (E0H 番地) からの 16 バイトに格納する. なお, 乗数, 被乗数, 積は, 全て符号無しの数とする.

入出力例

入力	`00A0: 0000 0000 8888 8888`	(乗数)
	`00B0: 0000 0000 2222 2222`	(被乗数)
出力	`00E0: 0000 0000 0000 0000 1234 5678 7654 3210`	(結果)

コード

```
                  # DATA1:  EQU A0H
                  #         LOC A0H
00A0: 0000        #         DAT 0000H
                  #         LOC A2H
00A2: 0000        #         DAT 0000H
                  #         LOC A4H
00A4: 8888        #         DAT 8888H
                  #         LOC A6H
00A6: 8888        #         DAT 8888H
                  # DATA2:  EQU B0H
                  #         LOC B0H
00B0: 0000        #         DAT 0000H
                  #         LOC B2H
00B2: 0000        #         DAT 0000H
                  #         LOC B4H
00B4: 2222        #         DAT 2222H
                  #         LOC B6H
00B6: 2222        #         DAT 2222H
                  # COUNT:  EQU C0H
                  # WORK:   EQU D0H
                  # RESULT: EQU E0H
0000: 0062 0040 #         LD ACC, 64
0004: 0074 00C0 #         ST ACC, [COUNT]
0008: 006A 0008 #         LD IX, 8
000C: 00C0        # LP1:    EOR ACC, ACC
000E: 0076 00CE #         ST ACC, [IX+WORK-2]
0012: 0076 00DE #         ST ACC, [IX+RESULT-2]
0016: 0076 00E6 #         ST ACC, [IX+RESULT+8-2]
001A: 0066 00AE #         LD ACC, [IX+DATA2-2]
001E: 0076 00D6 #         ST ACC, [IX+WORK+8-2]
0022: 00AA 0002 #         SUB IX, 2
0026: 0031 000C #         BNZ LP1
002A: 0064 00A6 # LP2:    LD ACC, [DATA1+6]
002E: 0042        #         SRL ACC
0030: 006A FFF8 #         LD IX, FFF8H
0034: 0066 00A8 # LP3:    LD ACC, [IX+DATA1+8]
0038: 0044        #         RRA ACC
```

```
003A: 0076 00A8 #         ST ACC, [IX+DATA1+8]
003E: 00BA 0002 #         ADD IX, 2
0042: 0031 0034 #         BNZ LP3
0046: 0035 0064 #         BNC LP5
004A: 006A 0010 #         LD IX, 16
004E: 0020      #         RCF
0050: 0066 00DE # LP4:    LD ACC, [IX+RESULT-2]
0054: 0096 00CE #         ADC ACC, [IX+WORK-2]
0058: 0076 00DE #         ST ACC, [IX+RESULT-2]
005C: 00AA 0002 #         SUB IX, 2
0060: 0031 0050 #         BNZ LP4
0064: 006A 0010 # LP5:    LD IX, 16
0068: 0020      #         RCF
006A: 0066 00CE # LP6:    LD ACC, [IX+WORK-2]
006E: 0045      #         RLA ACC
0070: 0076 00CE #         ST ACC, [IX+WORK-2]
0074: 00AA 0002 #         SUB IX, 2
0078: 0031 006A #         BNZ LP6
007C: 0064 00C0 #         LD ACC, [COUNT]
0080: 00A2 0002 #         SUB ACC, 2
0084: 0074 00C0 #         ST ACC, [COUNT]
0088: 0031 002A #         BNZ LP2
008C: 000F      #         HLT
```

4.8　符号無し 2 バイトの除算

説明

DVD (80H 番地) を, DVS (82H 番地) で割り, 商を QOT (84H 番地) に, 余りを RMD (86H 番地) に格納する. なお, 被除数, 除数, 商, 余りは, 全て符号無しの 1 バイトの数とする.

入出力例

入力　`0080: F0F0`　(被除数)
　　`0082: 3F3F`　(除数)
出力　`0084: 0003`　(結果: 商)
　　`0086: 3333`　(結果: 余り)

コード

```
                  # DVD:  EQU 80H
                  # DVS:  EQU 82H
                  # QOT:  EQU 84H
                  # RMD:  EQU 86H
                  # WORK: EQU 90H
                  #       LOC 80H
0080: F0F0        #       DAT F0F0H
                  #       LOC 82H
0082: 3F3F        #       DAT 3F3FH
0000: 00C0        #       EOR ACC, ACC
0002: 0074 0084   #       ST ACC, [QOT]
0006: 0074 0086   #       ST ACC, [RMD]
000A: 0064 0080   #       LD ACC, [DVD]
000E: 0074 0090   #       ST ACC, [WORK]
0012: 006A 0010   #       LD IX, 10H
0016: 0064 0090   # LOOP: LD ACC, [WORK]
001A: 0041        #       SLA ACC
001C: 0074 0090   #       ST ACC, [WORK]
0020: 0064 0086   #       LD ACC, [RMD]
0024: 0045        #       RLA ACC
0026: 0020        #       RCF
0028: 0084 0082   #       SBC ACC, [DVS]
002C: 003D 0036   #       BC SP1
0030: 0028        #       SCF
0032: 0030 003C   #       BA SP2
0036: 00B4 0082   # SP1:  ADD ACC, [DVS]
003A: 0020        #       RCF
003C: 0074 0086   # SP2:  ST ACC, [RMD]
0040: 0064 0084   #       LD ACC, [QOT]
0044: 0045        #       RLA ACC
0046: 0074 0084   #       ST ACC, [QOT]
004A: 00AA 0001   #       SUB IX, 1
004E: 0033 0016   #       BP LOOP
0052: 000F        #       HLT
```

4.9　マーチングによるメモリテスト

説明

KUE-CHIP 3 の内部メモリ, またはボード上の外部メモリが正常かどうかをテストする.

出力

エラーが検出されない間, OBUF の LED が点滅を続ける. クロック周波数が 1 MHs のとき, 毎秒 2 回程度点滅する.

コード

```
                # PROG: EQU 0
                # CA:   EQU PROG
0000: 00C0      #       EOR ACC, ACC
0002: 006A 0056 #       LD IX, TEST
0006: 0076 0000 # LP0:  ST ACC, [IX]
000A: 00BA 0002 #       ADD IX, 2
000E: 00FA 0000 #       CMP IX, PROG
0012: 0031 0006 #       BNZ LP0
0016: 00C9      #       EOR IX, IX
0018: 006A 0056 # LP2:  LD IX, TEST
001C: 00F6 0000 # LP3:  CMP ACC, [IX]
0020: 0031 0054 #       BNZ ERR
0024: 00C2 8000 #       EOR ACC, 8000H
0028: 0047      #       RLL ACC
002A: 0076 0000 #       ST ACC, [IX]
002E: 0039 003A #       BZ NX3
0032: 00F2 FFFF #       CMP ACC, FFFFH
0036: 0031 001C #       BNZ LP3
003A: 00C2 FFFF # NX3:  EOR ACC, FFFFH
003E: 00BA 0002 #       ADD IX, 2
0042: 00FA 0000 #       CMP IX, PROG
0046: 0031 001C #       BNZ LP3
004A: 00C2 FFFF #       EOR ACC, FFFFH
004E: 0010      #       OUT
0050: 0030 0018 #       BA LP2
0054: 000F      # ERR:  HLT
                # TEST: EQU CA
```

4.10　サブルーチンコールを用いた階乗計算

説明

メモリ上の INPUT 番地のデータの階乗の値を計算する．計算にはサブルーチンコールを利用する．FACT は再帰呼出しにより階乗の値を求める関数であり，MUL は積の値を求める関数である．

入出力例

出力　`80: 13b0`　(結果: 5040)

コード

```
                  # INPUT: EQU 50H
                  #        LOC 50H
0050: 0007        #        DAT 7H
                  # RES:   EQU 60H
0000: 0002 0100   #        LD  SP  100H
0004: 0062 0001   #        LD  ACC 1
0008: 006C 0050   #        LD  IX  [INPUT]
000C: 000C 0016   #        CAL FACT
0010: 0074 0060   #        ST  ACC [RES]
0014: 000F        #        HLT
0016: 000C 0028   # FACT:    CAL MUL
001A: 00AA 0001   #        SUB IX  1
001E: 003B 0026   #        BZN FACT_END
0022: 000C 0016   #        CAL FACT
0026: 000D        # FACT_END:       RET
0028: 0008        # MUL:     PSH ACC
002A: 0009        #        PSH IX
002C: 0062 0000   #        LD  ACC 0
0030: 00FA 0000   # MUL_BEGIN:       CMP IX  0
0034: 003B 0044   #        BZN MUL_END
0038: 00B3 0002   #        ADD ACC [SP+2]
003C: 00AA 0001   #        SUB IX  1
0040: 0030 0030   #        BA  MUL_BEGIN
0044: 000B        # MUL_END:       POP IX
0046: 0004        #        INC SP
0048: 000D        #        RET
```

第 5 章

アセンブラの使い方

5.1　アセンブラの使い方

この章では, KUE-CHIP3 アセンブラを扱えるよう, KUE-CHIP3 アセンブリコードからバイナリコードを生成する方法を示す.

5.2　入力ファイル

KUE-CHIP3 アセンブラの入力ファイルは KUE-CHIP3 アセンブリコードである. 下に示すプログラムは KUE-CHIP3 において, ACC × IX → ACC を計算し, メモリにストアする非常に簡単なプログラムである. このプログラムを `input_asm.txt` として保存する. [space] の部分はスペース/タブに変換, もしくは削除する.

```
[space] EOR ACC, ACC
[space] EOR IX, IX
[space] ADD IX, 10H
[space] ADD ACC, 03H
[space] SUB IX, 1
[space] BNZ 08H
[space] ST ACC, [20H]
[space] HLT

[space] END
```

図 5.1　`input_asm.txt`

5.3　アセンブリの実行

前節のアセンブリファイルからバイナリファイルを生成してみる.

1. 端末 (Cygwin terminal etc) を立ち上げ, `input_asm.txt` があるディレクトリまで移動する.
2. `$ perl assembler.pl input_asm.txt` を実行する.

assembler.pl が異なるディレクトリにある場合, 相対パスまたは絶対パスに変更する.

3. ディレクトリの中に, bin_input_asm.txt が生成されていることを確認する.

```
0000: 00C0       #    EOR ACC, ACC
0002: 00C9       #    EOR IX, IX
0004: 00BA 0010  #    ADD IX, 10H
0008: 00B2 0003  #    ADD ACC, 03H
000C: 00AA 0001  #    SUB IX, 1
0010: 0031 0008  #    BNZ 08H
0014: 0074 0020  #    ST  ACC, [20H]
0018: 000F       #    HLT
```

図 5.2　bin_input_asm.txt

5.4　注意点

KUE-CHIP3 アセンブラ使用時にバイナリファイルが生成されないときの注意事項を下記にまとめる.

- 命令の前にスペース/タブが入っているか.
- 命令, オペランド間にスペース/タブ区切りがあるか.
- アセンブラの最後に END 命令があるか.
- 16 進数の後に H がついているか.
- [IX+WORK+4] といったコードを [IX + WORK + 4] というように間にスペースを入れていないか.