CPU

1　実習の目的

単純な構成の 8 ビット・コンピュータを搭載した教育用 CPU ボードを用いて、CPU の基本的な動作を理解する。また、各マシン命令の機能、アセンブリ言語とマシン語の関係、および、アドレス方式（アドレッシングモード）などについても、実習を通して理解を深める。

本実習のスケジュールを以下に示す。

第 1 週 教育用 CPU 用のプログラムを作成し、実際に教育用 CPU ボード上でそのプログラムを実行することにより、CPU の動作、各マシン命令の機能、アセンブリ言語とマシン語の関係、などを理解する。

第 2〜4 週 パソコン上で、教育用 CPU ボードの機能シミュレータを開発する。これにより、CPU の各マシン命令の機能やアドレス方式、2 進数とその演算、などについてさらに理解を深めるとともに、デジタルシステムのシミュレーション技術を習得する。

第 4 週の実習時間の後半までに、教育用 CPU ボードのシミュレータを完成し、そのシミュレータの動作についての内覧会を開催する。

第 5 週 教育用 CPU 用のより高度なプログラムを作成し、プログラミング言語とマシン命令機能との関係、データ通信の基本機能、などについて理解を深める。

2　教育用 CPU の仕様

本実習で使用する教育用 CPU は、以下のような特徴を持つコンピュータである。

- 1 語 (word) の長さは 8 ビットである。
- CPU 内に 2 つの**レジスタ**：ACC（アキュムレータ）と IX（インデックスレジスタ）を持ち、データやアドレスを一時的に記憶する目的で使用することができる。
- 主記憶装置として、チップ上に 512（$= 2^9$）バイトの内部メモリを装備している。このうち、前半の 256（$= 2^8$）バイト（アドレス： *000*H〜*0FF*H [1]）を**プログラム領域**、後半の 256 バイト（アドレス：*100*H〜*1FF*H）を**データ領域**として使用する。データ領域にプログラムを格納しても、実行することはできない。一方、プログラム領域は、データの格納用にも用いることができる。

2.1　データ形式

教育用 CPU のマシン命令で扱うデータは 8 ビット（1 語）の**固定小数点数**のみである。ただし、**多倍長**（2 語長=16 ビット、3 語長=24 ビット、など）のデータに対して加減算を行う場合に必要となる機能を備えたマシン命令が用意されているので、これらを用いてプログラムを作成すれば、数の表現に 8 ビットでは足りない大きな数を扱うことができる。

数を 8 ビットの**符号なし整数**として表現する場合、数の大きさをそのまま正の 2 進数として表現すればよい。従って、$00000000_{(2)}$ から $11111111_{(2)}$、すなわち 0 から $255(= 2^8 - 1)$ までの範囲の整数を表現できる。

$$\begin{array}{rrcr} & 00001011 & \cdots & 11 \\ +) & 11111110 & \cdots & -2 \\ \hline & \not{1}00001001 & \cdots & 9 \end{array}$$

図 1: 2 の補数表現での $11_{(10)} - 2_{(10)}$ の計算

符号付き整数の場合は、負の数を 2 の補数を用いて表現する。ある数 A に対する 2 の補数 $\overline{A}(= 2^8 - A)$ は、A の各桁の 0 と 1 を反転し、最下位ビットに 1 を加えることで得られる。最上位ビットを符号ビットとし、符号ビットが 1 の場合は負の数を、符号ビットが 0 の場合は非負の数を表現するものと定める。従って、負の領域では $10000000_{(2)}$ から $11111111_{(2)}$、すなわち -128 から -1 までの範囲を、また、非負の領域では $00000000_{(2)}$ から $01111111_{(2)}$、すなわち 0 から 127 までの範囲を表現できる。

このような表現方法は、もともと、コンピュータ内部で減算を加算として処理する目的で考案された。例えば、図 1 に示すように、$11 - 2$ の計算は $11 + (-2)$ として計算する。11 と -2 の表現を、符号ビットも含めて普通の 2 進数とみなして加算し、**最上位桁から桁上げが生じても、これを無視**すれば計算結果として正しく 9 が得られる。

2.2　命令セット

教育用 CPU の**命令セット**を表 1 および表 2 に示す。表中の OP コードやオペランド指定は、アセンブリ言語での記述形式に従っている。表 1 は教育用 CPU のマシン命令を命令機能で分類して一覧表にしたものであり、特に、オペランド指定部分については表 2 にまとめている。表

[1] “*xx*H” のように末尾に ‘H’ をつけた数値は **16 進数**を表す。

表 1: マシン命令一覧

分類	略記号	オペランド指定	命令機能
動作制御	NOP	−	No OPeration
	HLT	−	HaLT
入出力制御	OUT	−	OUTput
	IN	−	INput
フラグ制御	RCF	−	Reset Carry Flag
	SCF	−	Set Carry Flag
データ移動	LD	{reg},{any}	LoaD
	ST	{reg},{mem}	STore
算術演算	ADD	{reg},{any}	ADD
	ADC	{reg},{any}	ADd with Carry flag
	SUB	{reg},{any}	SUBtract
	SBC	{reg},{any}	SuBtract with Carry flag
	CMP	{reg},{any}	CoMPare
論理演算	AND	{reg},{any}	AND
	OR	{reg},{any}	OR
	EOR	{reg},{any}	Exclusive OR
シフト演算	S*sm*†	{reg}	Shift....
†(Shift Mode)	R*sm*†	{reg}	Rotate....
	RA		Right Arithmetic
	LA		Left Arithmetic
	RL		Right Logical
	LL		Left Logical
分岐	B*bc*‡	{imm}	Branch....
‡(Branch Condition)	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
	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
	JAL	{imm}	Jump And Link
	JR		Jump Register

注) JAL 命令と JR 命令は教育用 CPU ボードには実装されていない。

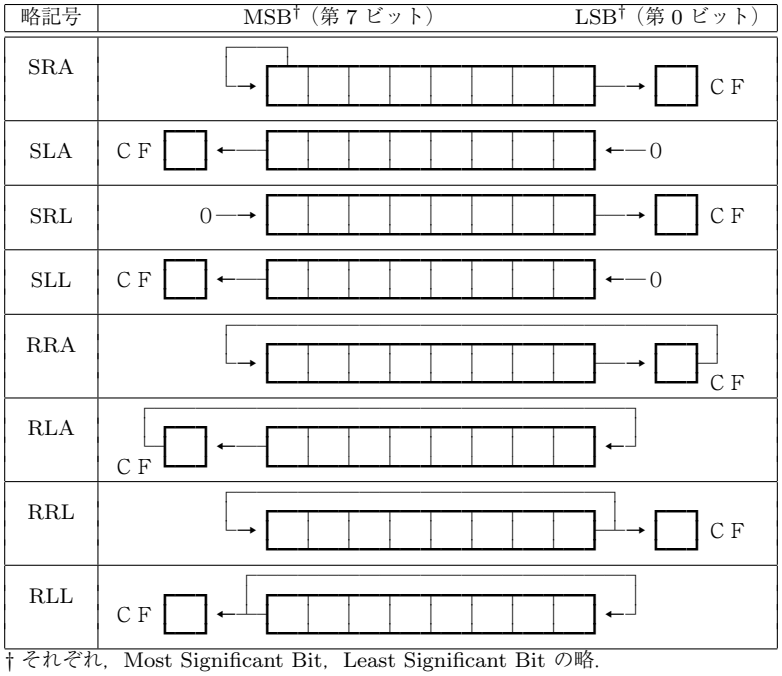
表 2: 教育用 CPU のアドレス方式

アドレス方式	記号	（指定されるオペランド）	指定記号	
レジスタ指定	ACC	（アキュムレータ）	{reg}	{any}
	IX	（インデックスレジスタ）		
即値アドレス	d	（命令語中の値）	{imm}	
絶対アドレス	[d]	（プログラム領域）	{mem}	
	(d)	（データ領域）		
インデックス修飾アドレス	[IX+d]	（プログラム領域）		
	(IX+d)	（データ領域）		

2 に示すように命令の**アドレス方式（ドレッシングモード）**は全部で 4 種類あり、命令の種類によって使用できるものと使用できないものがある。

教育用 CPU のマシン命令のうち、Shift 命令や Rotate 命令によってオペランドのデータがどのように処理されるかを表 3 に示す。なお、SLA 命令と SLL 命令ではオペ

表 3: Shift ／ Rotate 命令の機能



† それぞれ，Most Significant Bit，Least Significant Bit の略.

ランドのデータに対する操作が全く同じであるが、実際には実行後のフラグの状態が異なる（表 6 参照）ので注意すること。

また、表 1 の命令の中で、JAL 命令と JR 命令は、それぞれ、**サブルーチン**の呼び出しとリターンの機能を実現するための命令であるが、本実習に用いる教育用 CPU ボードには実装されていない。したがって、教育用 CPU ボードで実行するプログラムではこれらの命令を使用することはできないので注意すること [2]。なお、サブルーチンとは、メインルーチンに対応する術語であり、プログラム中の 1 つ以上の場所で必要になるたびに繰り返し呼び出して使用することができる部分的プログラム（プログラム部品）を指す。一般に、値を返すものを関数（function）、値を返さないものを手続き（procedure）と呼んで区別するが、後者の手続きのことをサブルーチンと呼ぶ場合もあれば、関数と手続きの両者をまとめてサブルーチンと呼ぶ場合もある。

2.3 命令語コード

教育用 CPU の 1 つのマシン命令は、8 ビットまたは 16 ビットで表現される。各命令がどのようにエンコード（コード化）されるかを表 4 にまとめて示す。

表 4 を早見表の形にまとめたものが表 5 である。この早見表は、各命令のアドレス方式ごとに、対応する命令語コードを素早く知るためのものである。パソコンなどでアセンブラが利用できる環境では必要ないが、ハンドアセンブルする際には便利である。

2.4 フラグ機能

教育用 CPU の種々の状態を表すための**フラグ**として、CF（桁上げ／借り）・VF（桁あふれ）・NF（負値）・ZF（零値）の 4 つがある。これらのフラグの機能について表 6 にまとめて示す。

各フラグは、ADD 命令などの演算系の命令を実行することによって set/reset され（「実行後の状態」欄）、特定の命令でその値を使用する（「実行への影響」欄）。例えば、条件分岐命令などでは、フラグの状態によってプログラム実行の流れを制御することができる。一般的なプログラミング言語における if 文や for 文、while 文などのループの終了判定の機能を実現するのに必須の機能である。

演算の結果、数値が表現可能な範囲を越えることを**オーバフロー（桁あふれ）**という。符号なし整数の場合は、加算などで最上位桁から**キャリー（桁上げ）**が生じればオーバフローが発生したと判断してよい。しかし、符号付き整数の加減算の場合は、図 1 に示した例のように、そもそも最上位桁からのキャリーを無視することで計算する方式を採用している。そのため、オーバフローを検出するためには専用の機構を設ける必要がある。教育用 CPU

[2] 教育用 CPU ボードでは、JAL/LR 命令は HLT 命令と同一の機能をもつ命令として扱われるため、これらの命令を実行するとプログラム実行を停止する。2 週目以降の実習で作成する CPU シミュレータでは、JAL/LR 命令を正式にサポートする

表 4: 命令語コードとその機能

♠ 命令コード一覧

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	-	-	-	×	何もしない
HLT	0	0	0	0	1	1	-	-	×	停止
OUT	0	0	0	1	0	-	-	-	×	(ACC) → OBUF
IN	0	0	0	1	1	-	-	-	×	(IBUF) → ACC
RCF	0	0	1	0	0	-	-	-	×	$0 \to CF$
SCF	0	0	1	0	1	-	-	-	×	$1 \to CF$
LD	0	1	1	0	A		B		○	$(B) \to A$
ST	0	1	1	1	A		B		◎	$(A) \to B$
ADD	1	0	1	1	A		B		○	$(A)+(B) \to A$
ADC	1	0	0	1	A		B		○	$(A)+(B)+CF \to A$
SUB	1	0	1	0	A		B		○	$(A)-(B) \to A$
SBC	1	0	0	0	A		B		○	$(A)-(B)-CF \to A$
CMP	1	1	1	1	A		B		○	$(A)-(B)$
AND	1	1	1	0	A		B		○	$(A) \wedge (B) \to A$
OR	1	1	0	1	A		B		○	$(A) \vee (B) \to A$
EOR	1	1	0	0	A		B		○	$(A) \oplus (B) \to A$
S$sm^{\dagger}$	0	1	0	0	A	0	$sm^{\dagger}$		×	$(A) \to$ shift$^{*} \to A$
R$sm^{\dagger}$	0	1	0	0	A	1	$sm^{\dagger}$		×	$(A) \to$ rotate$^{*} \to A$
B$bc^{\ddagger}$	0	0	1	1		$bc^{\ddagger}$			◎	条件成立時は B' → PC
JAL**	0	0	0	0	1	0	1	0	◎	PC + 2 → ACC, B' → PC
JR**	0	0	0	0	1	0	1	1	×	ACC → PC

* shift/rotate 命令の詳細は表 3 を参照のこと。

** JAL 命令と JR 命令は教育用 CPU ボードには実装されていない。

♠A（1 ビット）

0	ACC
1	IX

♠B'（2 語目）

× : 不用
○ : 不用 or 必要
◎ : 必須

♠$^{\dagger}sm$: Shift Mode

RA	0	0
LA	0	1
RL	1	0
LL	1	1

♠B（3 ビット）　（B' に格納されるのは以下の d の値）

000	ACC	アキュムレータを選択
001	IX	インデックスレジスタを選択
01–	d	即値アドレス
100	[d]	絶対アドレス（プログラム領域）
101	(d)	絶対アドレス（データ領域）
110	[IX+d]	IX 修飾アドレス（プログラム領域）
111	(IX+d)	IX 修飾アドレス（データ領域）

♠$^{\ddagger}bc$: Branch Condition

A	0	0	0	0	常に成立
VF	1	0	0	0	桁あふれ　$VF=1$
NZ	0	0	0	1	$\neq 0$　$ZF=0$
Z	1	0	0	1	$=0$　$ZF=1$
ZP	0	0	1	0	≥ 0　$NF=0$
N	1	0	1	0	<0　$NF=1$
P	0	0	1	1	>0　$(NF \vee ZF)=0$
ZN	1	0	1	1	≤ 0　$(NF \vee ZF)=1$
NI	0	1	0	0	$IBUF\_FLG\_IN=0$
NO	1	1	0	0	$OBUF\_FLG\_IN=1$
NC	0	1	0	1	$CF=0$
C	1	1	0	1	$CF=1$
GE	0	1	1	0	≥ 0　$(VF \oplus NF)=0$
LT	1	1	1	0	<0　$(VF \oplus NF)=1$
GT	0	1	1	1	>0　$((VF \oplus NF) \vee ZF)=0$
LE	1	1	1	1	≤ 0　$((VF \oplus NF) \vee ZF)=1$

に CF と VF とが個別に設けられているのはこのような理由によるもので、符号付き整数か符号なし整数かで CF と VF をプログラマが使い分ける必要がある。

3　プログラムの作成と実行

図 2 に示すプログラムは、ACC の値を IX の値の回数だけ加えることにより、(ACC) × (IX) → ACC の計算を行う非常に簡単なプログラムである。最左列から順番

表 5: 命令語コード早見表（16 進数）

(a) データ移動命令／算術演算命令／論理演算命令

オペランド B ⇒		ACC	IX	d	[d]	(d)	[IX+d]	(IX+d)
LD	ACC/IX,	60/68	61/69	62/6A	64/6C	65/6D	66/6E	67/6F
ST	ACC/IX,	−−	−−	−−	74/7C	75/7D	76/7E	77/7F
ADD	ACC/IX,	B0/B8	B1/B9	B2/BA	B4/BC	B5/BD	B6/BE	B7/BF
ADC	ACC/IX,	90/98	91/99	92/9A	94/9C	95/9D	96/9E	97/9F
SUB	ACC/IX,	A0/A8	A1/A9	A2/AA	A4/AC	A5/AD	A6/AE	A7/AF
SBC	ACC/IX,	80/88	81/89	82/8A	84/8C	85/8D	86/8E	87/8F
CMP	ACC/IX,	F0/F8	F1/F9	F2/FA	F4/FC	F5/FD	F6/FE	F7/FF
AND	ACC/IX,	E0/E8	E1/E9	E2/EA	E4/EC	E5/ED	E6/EE	E7/EF
OR	ACC/IX,	D0/D8	D1/D9	D2/DA	D4/DC	D5/DD	D6/DE	D7/DF
EOR	ACC/IX,	C0/C8	C1/C9	C2/CA	C4/CC	C5/CD	C6/CE	C7/CF

(b) 制御命令

NOP	00
HLT	0F
OUT	10
IN	1F
RCF	20
SCF	2F

(c) シフト演算命令

SRA	ACC/IX	40/48
SLA	ACC/IX	41/49
SRL	ACC/IX	42/4A
SLL	ACC/IX	43/4B
RRA	ACC/IX	44/4C
RLA	ACC/IX	45/4D
RRL	ACC/IX	46/4E
RLL	ACC/IX	47/4F

(d) 分岐命令

BA	30	BVF	38
BNZ	31	BZ	39
BZP	32	BN	3A
BP	33	BZN	3B
BNI	34	BNO	3C
BNC	35	BC	3D
BGE	36	BLT	3E
BGT	37	BLE	3F

Address	Obj. Code	Source Code		
00	75 03	START:	ST	ACC,(03H)
02	C0		EOR	ACC,ACC
03	B5 03	LOOP:	ADD	ACC,(03H)
05	AA 01		SUB	IX,1
07	31 03		BNZ	LOOP
09	0F		HLT	
			END	

図 2: サンプルプログラム

に、オブジェクトコードのアドレス（16 進数）、オブジェクトコード（16 進数）、および、アセンブリ言語によるソースコードである。

このプログラムの最初の命令は ST 命令であり、ACC の内容をメモリ（データ領域）のアドレス *03*H に書き込む。この命令のコードは表 4 を参照することによって求められる。表 4 より、ST 命令の 1 語目の上位 4 ビットは 0111 であり、第 1 オペランドが ACC であるから A の部分は 0、第 2 オペランドがデータ領域の絶対アドレスであるから B の部分は 101 で、かつ 2 語目が必要（この 2 語目でアドレスの *03*H を指定する）であることがわかる。以上より、この命令の 1 語目の命令コードは 16 進数で 75 となる。表 5 を用いれば、命令コードをより直接的に得ることができる。「ST ACC/IX」の行と「(d)」の列の交わるところには「75/7D」と記述されており、第 1 オペランドが ACC の場合は左側の 75、IX の場合は 7D とわかる。

図 2 のサンプルプログラムはプログラム領域のアドレス *00*H から始まる。最初の ST 命令に 2 語を要したので、次の命令はアドレス *02*H から格納されることになる。以後、同様にしてアセンブリ言語のソースコードからオブジェクトコードに変換すればよい。

プログラムの 5 行目の BNZ 命令は分岐命令である。分岐の条件は、表 1 より「Not Zero」とわかる。つまり、「演算結果が 0 でない場合に LOOP とラベルの付けられた命令に分岐せよ」という意味である。図 2 のプログラムでは、LOOP とラベルのついた命令のアドレスは *03*H なので、この分岐命令の 2 語目にそのアドレスを指定している。さて、「演算結果が 0 でない」かどうかは ZF フラグを参照することによって判断される。BNZ 命令の 1 つ前の命令は SUB 命令であり、表 6 より、SUB 命令が ZF フラグを設定することがわかる。よって、この BNZ 命令による教育用 CPU の動作は「SUB 命令で減算を行った結果が 0 でなければアドレス *03*H に分岐する」となる。

以上は、主に教育用 CPU を使用するプログラマの視点からの説明である。一方、教育用 CPU を設計する側の立場では、オブジェクトコードから上記の意味を解読して動作するコンピュータを、論理回路を用いて設計することになる。

4　教育用 CPU のハードウェア構成

一般に、CPU で 1 つの命令を実行するには複数の段階を経る必要がある。命令処理の機能として見た場合は、お

表 6: フラグ機能

略記号	実行への影響 † CF	VF	NF	ZF	実行後の状態 ‡ CF	VF	NF	ZF
NOP/HLT	—	—	—	—	—	—	—	—
OUT/IN	—	—	—	—	—	—	—	—
RCF	—	—	—	—	0	—	—	—
SCF	—	—	—	—	1	—	—	—
LD/ST	—	—	—	—	—	—	—	—
ADD	—	—	—	—	—	V	N	Z
ADC	c	—	—	—	C	V	N	Z
SUB	—	—	—	—	—	V	N	Z
SBC	c	—	—	—	C	V	N	Z
CMP	—	—	—	—	—	V	N	Z
AND	—	—	—	—	—	0	N	Z
OR	—	—	—	—	—	0	N	Z
EOR	—	—	—	—	—	0	N	Z
SRA	—	—	—	—	$b0$	0	N	Z
SLA	—	—	—	—	$b7$	V	N	Z
SRL	—	—	—	—	$b0$	0	N	Z
SLL	—	—	—	—	$b7$	0	N	Z
RRA	$b7$	—	—	—	$b0$	0	N	Z
RLA	$b0$	—	—	—	$b7$	V	N	Z
RRL	—	—	—	—	$b0$	0	N	Z
RLL	—	—	—	—	$b7$	0	N	Z
BA	—	—	—	—	—	—	—	—
BVF	—	VF	—	—	—	—	—	—
BNZ	—	—	—	$\overline{ZF}$	—	—	—	—
BZ	—	—	—	ZF	—	—	—	—
BZP	—	—	$\overline{NF}$	—	—	—	—	—
BN	—	—	NF	—	—	—	—	—
BP	—	—	$\overline{NF \vee ZF}$		—	—	—	—
BZN	—	—	$NF \vee ZF$		—	—	—	—
BNI	—	—	—	—	—	—	—	—
BNO	—	—	—	—	—	—	—	—
BNC	$\overline{CF}$	—	—	—	—	—	—	—
BC	CF	—	—	—	—	—	—	—
BGE	—	$\overline{VF \oplus NF}$		—	—	—	—	—
BLT	—	$VF \oplus NF$		—	—	—	—	—
BGT	—	$\overline{(VF \oplus NF) \vee ZF}$			—	—	—	—
BLE	—	$(VF \oplus NF) \vee ZF$			—	—	—	—

フラグ略称
- CF: Carry Flag
- VF: oVerflow Flag
- NF: Negative Flag
- ZF: Zero Flag

† 実行への影響
- c: 最下位への carry/borrow 入力となる
- $b0$: オペランド A の第 0 ビットとなる
- $b7$: オペランド A の第 7 ビットとなる
- 式: 分岐の成立する条件（論理）を示す
- —: 影響なし

‡ 実行後の状態
- 0: 0 にリセット
- 1: 1 にセット
- C: carry/borrow の発生によりセット/リセット
- V: オーバフローの発生によりセット/リセット
- N: 演算結果の第 7 ビットの値に設定
- Z: 演算結果が 0 ならセット, 0 以外ならリセット
- $b0$: オペランド A の第 0 ビットの値に設定
- $b7$: オペランド A の第 7 ビットの値に設定
- —: 変化なし

おまかに以下の 6 段階に分けられ、これを**命令実行サイクル**という。

1. **命令フェッチ**　PC（プログラムカウンタ）には、これから実行するべき命令の主記憶装置上でのアドレスを格納している。そこで、PC の値をもとに主記憶装置から命令語を読み出し、CPU 内の IR（命令レジスタ）にコピーする。
2. **命令解読**　IR 内の命令語を解読し、命令内容に応じて CPU 内で処理を行うための各種制御信号を生成する。
3. **オペランドフェッチ**　命令語のオペランドで指定されたアドレッシングモードに基づいて、演算に必要なデータを読み出す。
4. **演算実行**　ALU（演算装置）で演算を行う。
5. **結果書き込み**　命令語のオペランドの指定に基づいて、演算結果をレジスタや主記憶装置に格納する。
6. **PC 更新**　次に実行するべき命令のアドレスを算出して PC を更新する。

また、時間的な観点でも 1 つの命令を実行するのに複数の段階を経る必要があり、本実習で使用する教育用 CPU では、これを「フェーズ」と呼ぶことにする。1 つのフェーズの時間的長さは 1 クロック周期に相当する。各フェーズにおける命令語やデータの流れを図 3 に示す。教育用 CPU では、オペランドフェッチ、演算実行、結果書き込み、の 3 つの機能は、それぞれ個々にフェーズとして明確に分離されていない。また、命令フェッチ後に命令解読

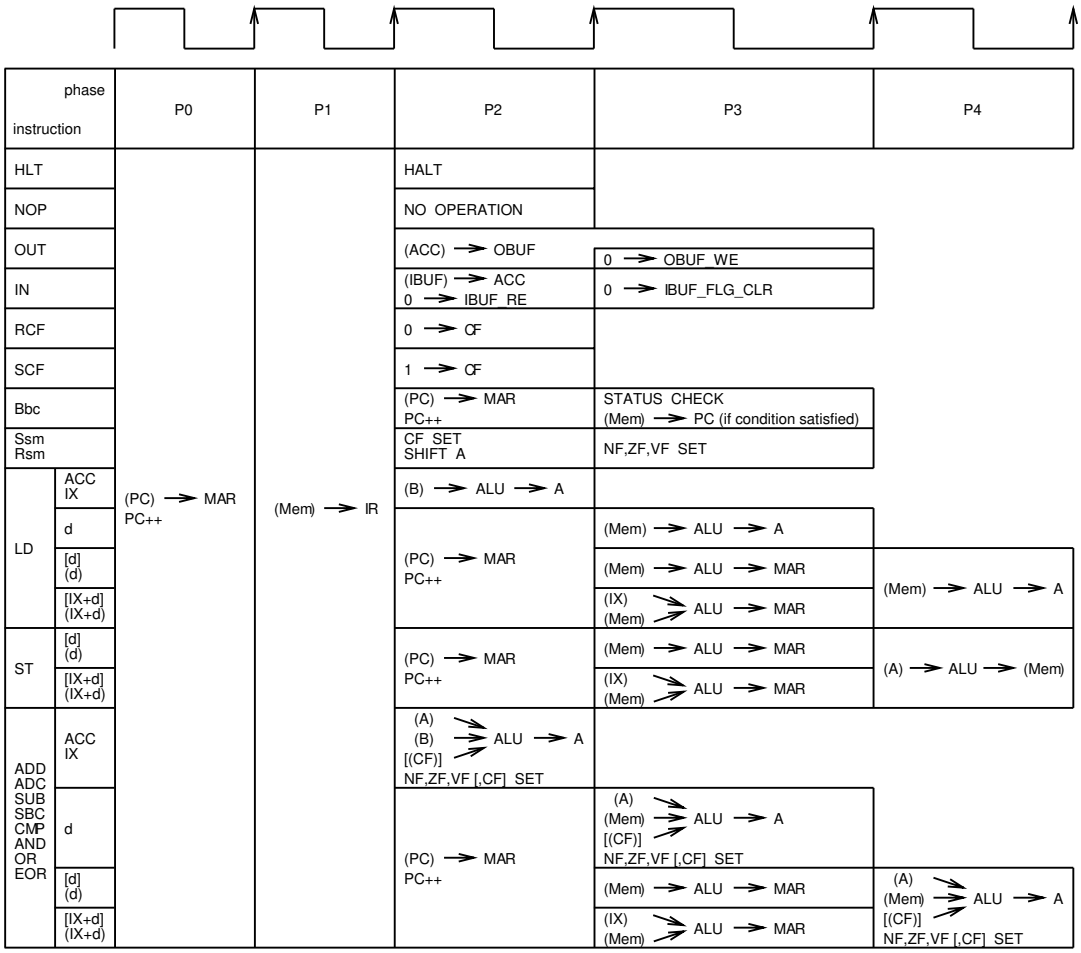


図 3: 命令実行フェーズ

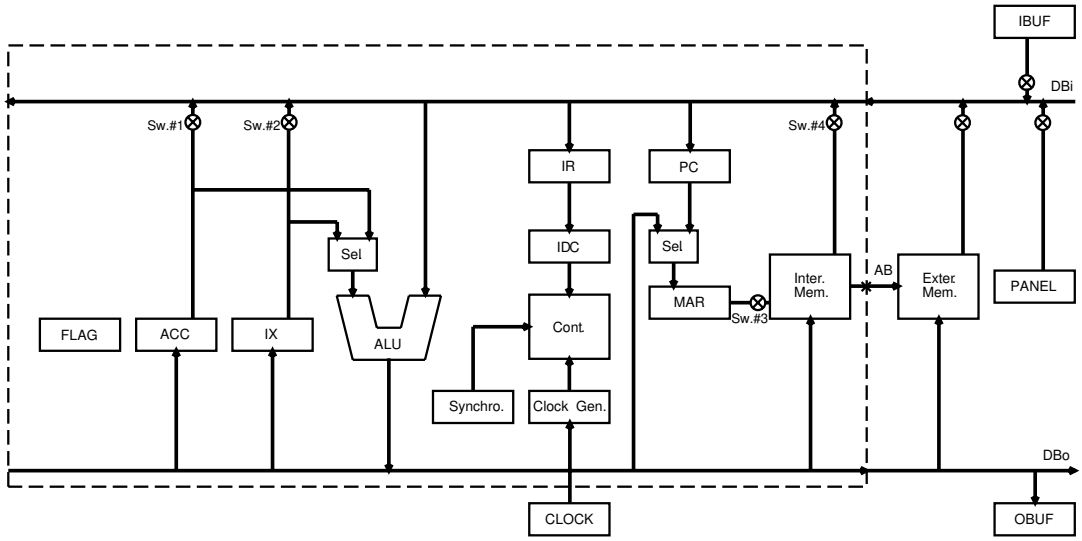


図 4: ブロック構成

のフェーズが独立して設けられているわけではなく、オペランドフェッチや演算実行と並行して、動作に必要な制御信号が生成されている。

図 3 に示すように、命令の種類だけでなく、オペランドの指定によっても、その命令を実行するのに必要なフェーズ数が異なる。ここでは、ADD 命令を例に、各フェーズでの処理をみてみよう。

まず、ADD 命令で表 4 のオペランド B に ACC または

IX を指定した場合（図 3 で下から 4 行目を参照のこと）には、フェーズ P0 とフェーズ P1 で命令フェッチを行い、フェーズ P2 で命令解読、オペランドフェッチ、演算実行、結果書き込みの処理を行う。PC 更新はフェーズ P0 で命令フェッチと同時に行う。

次に、オペランド B に即値 (d) を指定した場合（図 3 で下から 3 行目を参照のこと）には、フェーズ P0〜P2（および P3 の一部）で命令フェッチを行う。

最後に、オペランド B で絶対アドレスやインデックス修飾を指定した場合（図 3 で下から 2 行目および 1 行目を参照のこと）には、フェーズ P3 でアドレッシングモードに対応する処理（オペランドフェッチ）を行う。なお、絶対アドレス指定時にフェーズ P3 で ALU を使用しているが、これはメモリから読み出した値（命令語の 2 語目）を MAR に転送する際の通路として使用しているだけであり、実質的な演算は行っていない。

図 4 は教育用 CPU とその周辺回路のブロック構成図であり、このうち細い破線で囲まれた部分が、教育用 CPU として、教育用 CPU ボード上では 1 個の LSI で実装されている。図 3 と図 4 とをよく見比べて、各命令の動作を理解すること。なお、MAR（メモリアドレスレジスタ）などの図中の用語や命令実行の詳細については、文献 [1] などを参考にせよ。

5　教育用 CPU ボードを用いた実習

【演習 1】 表 4 の命令セット表に示された各命令を、固定長命令と可変長命令に分類せよ。

【実習 1】 図 2 のサンプルプログラムを教育用 CPU ボードに入力して実行せよ、表示などをチェックし、正しく乗算が行われることを確認せよ。

なお、実際の教育用 CPU ボードの操作にあたっては、マニュアル「教育用 CPU ボードの仕様&教育用 CPU ボードの操作例」を参照すること。

【実習 2】 加算に使用するデータの値を変化させて ADC 命令を実行し、どのような条件でキャリーフラグ（CF）やオーバフローフラグ（VF）がセットされるかを調べることによって、2 進数の加算におけるオーバフローとキャリーの意味を確認せよ。最小限の手間で目的を果たすために、教育用 CPU ボードをどのように用いるか、どのようなプログラムを作成するか、どのようなデータを用いるか、など、実習方法の効率と有効性を検討・工夫すること。

【演習 2】 図 3 の「**命令実行フェーズ**」と図 4 の「**ブロック構成**」とを比較対照しながら、教育用 CPU の各命令ごとに CPU 内部の動作を追跡してみよ。そのうち、特に “LD”、“ST”、“ADD”、“BZ” の 4 つの命令について、命令実行フェーズの進行に伴って、ブロック構成図中の $\otimes$ で示されたスイッチの開閉（on/off）をどのように制御すればよいかを考えよ（メモリ参照のアドレス方式は「絶対アドレス」とせよ）。ALU の左/右入力と、命令語の A/B オペランドとの関係にも留意せよ。

以下の**実習 3 以降**の実習（ただし、実習 5 を除く）では、各課題のプログラムを作成してマシン語にアセンブルした後、教育用 CPU ボード（またはシミュレータ）のメモリに書き込み、実行の様子を観察せよ。レポートでの報告の際は、特に以下の注意事項を守ること。

- レポートには、**ソースコードとオブジェクトコードだけを書くのではなく、各命令ごとにそれがプログラム中で持つ ‘意味’ についても記述する**こと。また、仮想コードやフローチャート、PAD などの適当な手段を用いて、採用したアルゴリズムの説明も行うこと。
- **入力値（初期値）として与えるデータについては、プログラムの正当性をチェックするのに十分であるように、適当な値をよく考えて選ぶ**こと。
- 実行結果については、**入力値（初期値）と出力値（最終値）だけでなく、PC, ACC, IX などの値の変化についても記録しておく**こと。例えば、SI スイッチで 1 命令ずつ実行させ、PC, ACC, IX などの値の変化を記録する。ACC については、OUT 命令を用いてボード上の OBUF の LED に値を表示させることもできる。
- できれば実行ステップ数についての評価も行うこと。

なお、複数のプログラムを同時に教育用 CPU のメモリ上に載せるためには、お互いに使用するメモリ領域が重ならないように、あらかじめプログラムの開始（先頭）アドレスをずらしておけばよい。

【実習 3】 符号なし整数の乗算 ‘$z \leftarrow x \times y$’ を計算するプログラムを作成せよ。x, y はメモリ上ではそれぞれ 1 語として格納するが、下位 4 ビットで表現される（上位 4 ビットはすべて 0）ものとし、x, y, z をメモリ上の連続した 3 語に割り付けること。また、シフトと加減算を組み合わせて、x, y の値に依存しない定数回（実際には、y のビット数である 4 回）のループで積が求まるプログラムにすること。

なお、レポートでの実習報告については、上記の注意事項を守ること。

♣ **注** ♣ y の各ビットを順に調べ、‘1’ の時にだけ x の値を部分積に加えればよい。この加算の際に、x と部分積との桁合わせを行う必要があり、そのためにシフト操作が必要になる。

【演習 3】 JAL 命令と JR 命令を教育用 CPU ボードに実装する場合には、図 4 の**ブロック構成**に対してどのような変更が必要か検討せよ。また、JAL 命令と JR 命令の命令実行フェーズを設計し、図 3 にならって各実行フェーズにおける動作を説明せよ。

【実習 4】 2~128 個の 1 バイトデータから成る配列を用意し、バブルソート法により要素を昇順に整列するプログラムを作成せよ。要素数 n はメモリ上の 1 語で与えること。配列データの格納開始番地は固定でよい。

なお、レポートでの実習報告については、上記の注意事項を守ること。

♣ **注** ♣ バブルソート法は、例えば次のような擬似 C プログラムになる。

```
for(i = n-1; i > 0; i--)
    for(j = 0; j < i; j++)
        if(a[j] > a[j+1])
            swap(a[j], a[j+1]);
```

ここで、'swap' とは「2 数の値の交換」を意味する。

---

6 教育用 CPU の機能シミュレータ

ここでは、教育用 CPU ボードの機能をシミュレートするプログラムを、パソコン上のアプリケーションプログラムとして開発する。実際にこのようなシミュレータを開発する必要が生じるケースとして、以下のような状況が挙げられる。

- CPU などのハードウェアを新たに開発する際に、本来は、そのハードウェアが完成しなければそのハードウェア上で実行するソフトウェアの開発に着手できない。しかし、ハードウェア/ソフトウェアを含めたシステム全体の開発期間を短縮するためには、ハードウェアの設計とソフトウェアの開発を並行して進めることが望ましい。そこで、まだ現実のものとして存在していないハードウェアの代わりにシミュレータを作成して用いることで、ソフトウェア開発を早期に着手することが可能になる。
- ある命令セット（ここでは、「命令セット A」とする）をもつ CPU で、これとは異なる命令セット（ここでは「命令セット B」とする）をもつ CPU 用のバイナリプログラムを実行したい場合がある。このようなときに、命令セット B のマシン命令を解釈・実行するシミュレータを用いる。見方を変えれば、このシミュレータは、ソフトウェアによって命令セット B の CPU を仮想的に実現していると捉えることができる。なお、このような「仮想マシン」の概念を基に、広く用いられているプログラミング言語として Java がある。

CPU のシミュレーションには、大別して、対象とする CPU の機能のみをシミュレートする機能シミュレーションと、CPU のハードウェア構造や動作を忠実にシミュレートして性能予測を行う時間シミュレーションとがある。上記の例は主に機能シミュレーションに関するものであり、一方、新たに CPU などのハードウェアを開発する場合には時間シミュレーションを行って設計の詳細を検討する。

本実習では、教育用 CPU ボードの機能シミュレータを C 言語を用いて開発する。

6.1 シミュレータの機能

本実習のシミュレータで実現する機能は、大きく以下の 2 つである。

1) 教育用 CPU の命令シミュレーション機能
2) 上記 1) を使用するためのコンソール機能

まず、1) の命令シミュレーション機能について検討する。

教育用 CPU が内部に備えている記憶装置は、すべて C 言語の変数として定義する。教育用 CPU の 1 語は 1 バイトであるので、PC、ACC、IX などはすべて char 型の変数として定義する。また、メモリ（主記憶）は char 型の配列として定義し、アドレスをその配列に対する添字として扱えばよい。なお、signed char 型とするか unsigned char 型とするかについては、それぞれの記憶装置の用途から都合の良い方を選ぶ。

本実習では、JAL/JR 命令を含む教育用 CPU のすべての命令をシミュレータに実装する。また、命令シミュレーションの制御機能として、教育用 CPU ボードの SI スイッチ（1 命令実行）と SS スイッチ（全プログラム実行）に相当する機能を実装する。

例えば、SI スイッチに対応する関数として、

```
int   step(void);
```

を設けたとしよう。教育用 CPU の 1 命令をシミュレートする場合は、シミュレータ内でこの関数 `step()` を呼び出す。関数 `step()` 内部の処理は、図 3 に従って行えばよい[3]。命令を主記憶から読み出して解読し、演算を行う過程をプログラムとして記述する。命令解読は、表 4 に従って、C 言語の論理演算やシフト演算などの機能（付録参照）を用いて処理すればよい。また、命令実行においては、1 バイトのデータを short 型や int 型に変換して処理することにより、プログラムとしての記述を簡素化できる場合もある。

関数 `step()` の戻り値として、HALT 命令を実行したときには 0 を、それ以外の命令を実行したときには 1 を返すものと定めれば、SS スイッチに対応する処理は、

```
while( step() );
```

と記述することができる。

次に、2) のコンソール機能について検討する。コンソール機能として、以下の機能をシミュレータに対するコマンドとして実装する。

a) **プログラムの入力機能**： 教育用 CPU 用のプログラムをシミュレータ内に読み込む機能。プログラムをハンドアセンブルしたものをファイルに記述し、シミュレータはそのファイルからプログラムを入力するものとする。

b) **データ設定機能**： レジスタやメモリに値を設定（入力）する機能。

c) **データ表示機能**： レジスタやメモリに記憶されている内容を表示する機能。

d) **実行制御機能**： 教育用 CPU ボードの SI スイッチ（1 命令実行）と SS スイッチ（全プログラム実行）に相当する命令シミュレーションの制御機能。

e) **その他の機能**： シミュレータの終了など、上記以外に必要となる機能。

[3]ただし、本実習では、教育用 CPU ボードの SP スイッチ（1 フェーズ実行）に相当する機能を実装しないので、フェーズの切れ目やフェーズの数については無視してよい。

6.2 シミュレータの作成

本実習のサポートページ（Moodle システム）に未完成のシミュレータのソースコードが用意されている。このソースコードには、(i) CPU 内のレジスタや主記憶の定義、(ii) 基本的なコンソール機能を実現するための各種コマンドの処理、などが既に記述されている。

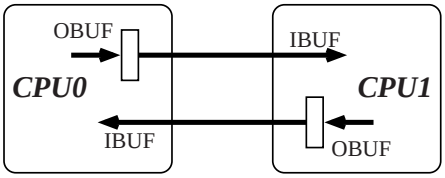


図 5: シミュレーション対象システムの構成

また、2 台の教育用 CPU ボードを図 5 のように接続した構成をシミュレートすることを想定している。CPU0 の OBUF を CPU1 は IBUF とみなし、同様に、CPU1 の OBUF を CPU0 は IBUF とみなす。CPU0 で OUT 命令を実行すると、CPU0 の OBUF にデータが書き込まれるのと同時に *OBUF_FLG_IN* がセットされる。この *OBUF_FLG_IN* は、CPU1 からは *IBUF_FLG_IN* として見えるので、CPU1 では、BNI 命令を用いて IBUF に有効なデータが存在するか否かを検査することができる。CPU1 で IN 命令を実行すると、CPU1 の IBUF（CPU0 の OBUF）内のデータを CPU1 の ACC に転送するとともに、*IBUF_FLG_IN* をリセットする。このようにして、2 台の教育用 CPU ボード間でデータを通信することができる。

【実習 5】 シミュレータを C 言語を用いて作成せよ。必要に応じて追加・変更を行うことも含めて、本実習のサポートページに用意されているソースコードを利用してよいものとする。ただし、入力ファイルの仕様については、変更する場合は上位互換性を保つこと。

実習レポートでは、**グループ（班）内でのメンバーの役割分担を明記**した上で、プログラムのソースコードとそれに対する担当部分の解説を行うこと。また、追加･変更の有無にかかわらず、サポートページのソースコードを利用した場合にはその解説も行うこと。

【実習 6】 シミュレータを用いて**実習 3** および**実習 4** の再実習を行い、シミュレータの動作を確認すること。シミュレータに入力するプログラムは、必ずしも前回の実習時に作成したものをそのまま用いる必要は

なく、改良や修正などの手を加えてよい。従って、実習レポートでは**実習 3** および**実習 4** のプログラムについても再度報告し、その動作の様子についてまとめること。

【成果発表】 実習 3、実習 4 のプログラムをテストプログラムに用いて、実習 5 で作成したシミュレータの動作をデモンストレーションせよ。シミュレータが正しく動作することを示すためには「何を」「どの順序で」「どの程度詳細に」示せばよいか、また、グループ（班）内でどのようにメンバーの役割を分担するか、を十分に検討した上で発表すること。さらに、シミュレータ作成において特に工夫した点があれば、それもアピールすること。

7 プログラミング言語とマシン語プログラム

プログラミング言語を習うとき、一般に、「変数」は「値を入れておくための入れ物である」と説明される。この「入れ物」は、実際のコンピュータハードウェアにおいては、メモリ（主記憶装置）上に用意する。1 個のプログラムにおいて複数の変数が定義された場合も、それらの変数の値を記憶する場所（すなわち「入れ物」）は、原則としてすべてメモリ上に用意する。プログラミング言語では、その変数にアクセスする際には変数名を使用するが、一方、マシン命令レベル（アセンブリ言語）では、その変数の値が格納されるメモリ領域の先頭アドレス [4] を使用する。つまり、マシン命令レベルでは、変数名ではなく、メモリアドレスが、変数を識別するための名前となる。

C 言語などでポインタを扱う場合には、変数に対して、その値だけでなく、変数のアドレスも意識しなければならない。ポインタ変数は、整数値や実数値を記憶するための変数ではなく、変数の記憶場所すなわちアドレスを記憶内容とする変数である。

C 言語の構造体は、原則として、各メンバをメモリ上の連続した領域に順に割り付ける。例えば、

```
struct sample {
    int    ivalue;
    short  hvalue;
    char   bvalue;
} values;
```

に対して、変数 values をアドレス *6c*H から割り付ける場合（図 6 参照）、&values の値は *6c*H であり、また、

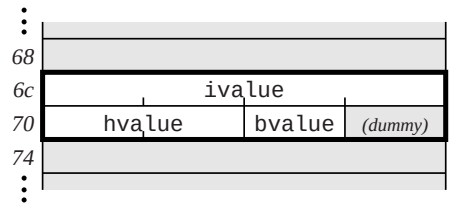


図 6: 構造体のメモリへの割り付け例

int 型、short 型、char 型のデータサイズをそれぞれ 4 バイト、2 バイト、1 バイトとすると、メンバ ivalue の値はアドレス *6c*H～*6f*H に、メンバ hvalue の値はアドレス *70*H～*71*H に、メンバ bvalue の値はアドレス *72*H に、それぞれ格納される。変数 values が他のアドレスに割り付けられたとしても、構造体内のメンバのレイアウトは変わらない。従って、&values をベースアドレス *base* とすると、values のメンバ ivalue、hvalue、bvalue のアドレスは、それぞれ、$base+0$、$base+4$、$base+6$ と計算できる。

【実習 7】 以下は、一方向の連結リスト（ただし要素数は 1 以上とする）の中で、データ（構造体メンバ data）の値が最大のノードを検索するプログラムを C 言語風に記述したものである。このプログラムを教育用 CPU のマシン語プログラムとして作成し、その動作を確認せよ。

```
struct node {
    char          data;
    struct node   *next;
};
struct node  *head, *max, *p;
max = head;
for( p = head->next ; p ; p = p->next )
    if( max->data < p->data )
        max = p;
```

♣ **注** ♣ IX をインデックスレジスタとしてではなく、ベースレジスタとして使用するとよい。

【演習 4】 一般に、サブルーチンを構成するためには、**a)**「呼び出し（call）」; **b)**「戻り（return）」; **c)**「引数の授受」; の各操作を行う命令列が必要であるが、ここでは c) についてはひとまず置いておき、a), b) に論点を絞って検討することにする。

a), b) のためにそれぞれ JAL 命令、JR 命令が定義されているが、教育用 CPU ボードではこれらの命令は実装されていない。実際の教育用 CPU ボードでサブルーチンを実現する方法があり得るかどうか、その可能性について検討せよ（可能な場合にはその方法を、また不可能な場合にはその理由を説明

[4] 例えば、変数 count のサイズが 4 バイトであり、コンパイラ（またはプログラマ）がこの変数の値をアドレス *c4*H～*c7*H に格納するものと決定した場合、変数 count を演算に使用するマシン命令のオペランドにはアドレス *c4*H を指定する。

せよ）。

なお、**プログラミングスタイル**（プログラム作成上の作法）としては、サブルーチンのコードサイズや格納場所などが変わっても、呼出し側（メインルーチン）では “サブルーチンの開始アドレスだけの変更” で済むようにするのが望ましい。

また、**再帰呼出し**については考慮しなくてよい。したがって、必ずしも「戻りアドレス」をスタック（Last–in–First–out）で管理するようなスタイルにする必要はない。

8　データ通信プログラム

2 台の教育用 CPU ボードを図 5 のように接続する場合、通信バッファIBUF と OBUF は、通信相手の CPU ではそれぞれ OBUF、IBUF とみなす。受信側の CPU が通信バッファを読み出す前に送信側の CPU がそのバッファに書き込むと、送信データが失われてしまう。これを防ぐためには、OUT 命令と BNO 命令を組み合わせて使用し、OBUF が空であることを確認してから OBUF に書き込まなければならない。また、送信側の CPU が通信バッファに書き込む前に受信側の CPU がそのバッファを読み出すと、無効なデータが全通信データ中に混入することになる。そこで、IN 命令を実行する前に BNI 命令を用いて IBUF にデータが存在することを確認しなければならない。

【実習 8】 2 台の教育用 CPU を用いて、1 から N までの数の総和を計算する。CPU0 では、1 から N までの整数を、順次、生成して CPU1 に送信する。データの終了を示すために、最後は 0 を送信する。一方、CPU1 では、CPU0 から受信した値の総和を計算する。CPU0/1 用のそれぞれのプログラムを作成し、その動作を確認せよ。

参考文献

[1] 柴山 潔：“改訂新版 コンピュータアーキテクチャの基礎,” 第 2 章「基本アーキテクチャ」, 近代科学社, 2003.

付録: C 言語におけるビット演算

ビットごとの論理演算子

演算子	意味
`&`	ビットごとの論理積 (AND)
`|`	ビットごとの論理和 (OR)
`^`	ビットごとの排他的論理和 (XOR)
`<<`	左シフト
`>>`	右シフト
`~`	1 の補数（単項演算子）

（注）　シフト演算については、操作対象のデータ（変数）の型が signed の場合は算術シフト、unsigned の場合は論理シフト、をそれぞれ行う。

使い方（例）

♣ 論理積

- `n` の下位 4 ビットのみを取り出す（残す）

```
unsigned char n;
n = n & 0x0f;
```

- `n` の最上位ビットを 0 にする

```
unsigned char n;
n = n & ~0x80;
```

または

```
unsigned char n;
n = n & 0x7f;
```

♣ 論理和

- `n` の最上位ビットを 1 にする

```
unsigned char n;
n = n | 0x80;
```

♣ 排他的論理和

- `n` の最上位ビットを反転（1→0、0→1）する

```
unsigned char n;
n = n ^ 0x80;
```

♣ シフト

- `n` を 3 ビット右シフトする

```
n = n >> 3;
```

- `n` を 2 ビット左シフトする

```
n = n << 2;
```