目次

0　実験の心構え

初回授業において先生が言っていたことを簡単にまとめた。レポートに書く必要もないのかもしれないが、自分への意識付けの意味も込めて書いた。

> コロナ禍において今一度「何のために大学にくるのか」について考えてみる。勉強をするためだけであれば、本を呼んだ方が効率的に学べるかもしれない。また、４年で学べる量や質というものはほんの少しである。たとえ今最先端なことを学んだとしても１０年２０年もすればそれは当たり前のこと、または廃れている分野になっているかもしれない。ではどうすればいいのか。ずばり、社会に出ても学び続ける必要があるのだ。就職面接時、学生は今まで自分がしてきたこと、実績を得意気に話す傾向にある。一方面接官というと会社に入ったあとこの学生は本当に自分で学び続けるだろうかということを見ているというのだ。そのことも知らず、よく学生は就活は難しいという。そんなことはない、就活は簡単なのだ。大学期間中、とりあえず課題をこなす人もいるだろう。しかし単位の為だけにこの４年間を費やすのは非常にもったいない。きちんと疑問をもって取り組む姿勢が大切である。これは何も大学生活だけでなく、社会に出たときにも活きてくる。ある会社では研修中の新入社員に電話番を担当させるという。上司が「何か起きないか見といて」と新入社員に言って、出かけた。その間に会社では火事が起きた。新入社員は火事を見ていただけだったという。上司が新入社員に「どうしてみていただけか」と訳を聞くと「見といてと言われたから」と答えたそうだ。本当にあったかはさておき、これをレポートの話に置き換えてみよう。レポートにおいて課される問題がある。問題を解けと言われたから解いたのであればそれは新入社員と同じ思考である。そうでなく、どうしてこのことが問われているのか。本当の意味は何なのかを考えることがレポートの本質ではないのだろうか。

1　論理と用語の確認

1.1　固定長命令と可変長命令について [1]

命令語のオペランド数を固定するか可変にするかによって命令形式を分類することができる。例えば、命令コード ST のときオペランドは A、B の２個、HLT のときオペランド数０と固定されている。一方、命令コードが ADD のときオペランド数は固定されていない。もしアドレス方式がレジスタ指定（第二オペランドが ACC や IX）のとき命令コード１語目の B フィールドに 000,001 を指定する。このとき２語目の命令コードがいらなくなる。一方で、アドレス方式が絶対アドレスのとき２語目の命令コードにアドレス番号を指定する。このとき命令語は２語となる。このように命令語が１語、２語と変わることが可能なものが可変長命令となる。

1.2　キャリーとオーバーフローについて [2]

r 進数の１桁の加算の結果が r 以上となる場合キャリー（桁上げ）という。演算結果が表現可能な範囲を超える場合オーバーフロー（桁あふれ）という。。簡単にいうと、筆算で桁が移動することをキャリーといい、筆算結果が正しい答えにならない場合をオーバーフローという。符号なし整数の和の場合、最上位ビットのキャリーが生じたときオーバーフローが生じたことになる一方、符号つき整数の和の場合、最上位ビットのキャ

リーが生じたといってオーバーフローが生じるとは限らない。例えば、2 ビットの符号なし整数の和 110+101 を考える。最上位ビットのキャリーが生じ、011 となりオーバーフローが生じたことになる。次に、符号つき整数の和 110+101 を考える。最上位ビットのキャリーが生じ、011 となるが、演算結果は正しく, オーバーフローは生じていないことになる。

1.3　アドレッシングモードについて

アドレッシングモードとは、オペランドフィールドで指定されたデータの格納場所の方式である。データを格納できる場所はレジスタとメモリがある。レジスタの場合、レジスタ番号で格納場所を識別する。メモリの場合、アドレスで格納場所を識別する。さらにこのアドレスに対する考え方として絶対と相対の 2 つがある。前者はメモリにアドレスが付いてあるのだからそれをそのまま使用しましょうというものするもの。つまり、直接アドレスを指定するもの。一方後者はあるアドレスを基準として、そこからどれだけ離れているか指定しましょうというもの。つまりベースアドレスとオフセットを用いて相対的にアドレスを指定するもの。

1.4　命令実行サイクルについて [3]

cpu である命令語を実行するとき、ステージ (過程) を経る。1 つの命令に対する一連のステージのことを命令実行サイクルと呼ぶ。大きく、6 つのステージに分けることができる。命令フェッチ、命令デコード、オペランドフェッチ、演算実行、演算書き込み、PC の更新。

1 つ目は命令フェッチで実行する命令の読み出しを行う。ちなみに、フェッチとは読み出しを意味する。PC（Program Counter）には次実行すべき命令文が格納された MM（メインメモリ）のアドレスが格納されている。この PC を MAR（Memory Address Register）にコピーする。この MAR に格納された値のアドレスを基に、MM から命令文を読みだす。この命令文は IR（Instruction Register）に格納され、命令文を読み出す。

2 つ目は命令デコード。IR 内の命令語を解読する。例えば、OP コードからこの命令を処理するのにあと何サイクルかかるか求め、フェーズ信号生成器をセット、起動する。また、このフェーズ信号生成器の出力や OP コードから各部を制御する制御信号を生成する。このように機械なので実際に解読はできてないが、命令を解読したかのようふるまう。

3 つ目はオペランドフェッチ。命令に必要なデータをアドレッシングモードに従って読みだす。例えば、ソースオペランドがレジスタの場合、レジスタから値を読みだす。メモリの場合、デコードしたソースオペランドをメモリアドレスに変換し、MAR に設定する。そして MM に読み出しを指令し、読みだしたデータは ALU の入力ラッチ（一時的保持機構）におく。

4 つ目は演算実行で ALU を用いて演算を実行する。計算そのものを担当する演算装置は ALU（Arithmetic and Logic Unit）とも呼ばれる。ALU では 1 つまたは 2 つのソースオペランド（演算前の値のオペランド）を入力として、1 つのデスティネーションオペランドへ出力する。（実行結果は ALU の出力ラッチに置かれる。）演算の種類として、符号反転や平方根の単項算術演算、足し算、掛け算の 2 項算術演算、大小比較の関係演算がある。

5 つ目は演算書き込みでデスティネーションオペランド（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。メインメモリへの格納が必要な場合にはメモリオペランドから得たメモリアドレスを MAR に設定してメインメモリに実行結果の書き込みを指令する。

6 つ目は PC の更新で次に実行する命令のアドレスを PC へ設定する。引き続くアドレスの場合は暗黙的に、

表 1　演習 1

固定長命令	可変長命令
NOP	LD
HLT	ADD
OUT	ADC
IN	SUB
RCF	SBC
SCF	SMP
ST	OR
Ssm	EOR
Rsm	
Bbc	
JAL	
JR	

分岐命令によって分岐する場合は明示的にアドレスを指定する。

2　実験の目的

コンピュータで扱う数値の表現方法、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。およびアドレッシングモードなどを理解する。

3　演習と実習

3.1　演習１　命令の分類

命令を固定長命令と可変長命令に分類すると、表 1 のようになる。表 2 の B’(2 語目) に注目して、×と◎が固定長命令。○が可変長命令となる。

3.2　演習２　ADC 命令の VF と CF の検出

とある整数 A,B があったとする。式 $A + B = C$ が成り立つような演算結果 C があったとする。この A,B,C それぞれの最高位ビットを MSB_A,MSB_B,MSB_C とする。

3.2.1　A と B と C が符号つき整数のとき

最高位ビットの状態を場合分けをして考えると、表 3 のように全 8 通り考えることができる。ここから条件を導くと
オーバーフローが生じる条件は、

表 2　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	-	-	-	×	$(\mathrm{ACC}) \to \mathrm{OBUF}$
IN	0	0	0	1	1	-	-	-	×	$(\mathrm{IBUF}) \to \mathrm{ACC}$
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	$sm^{\dagger}$		×	$(A) \to \mathrm{shift}^{*} \to A$
R$sm^{\dagger}$	0	1	0	0	A	1	$sm^{\dagger}$		×	$(A) \to \mathrm{rotate}^{*} \to A$
B$bc^{\ddagger}$	0	0	1	1	$bc^{\ddagger}$				◎	条件成立時は B’ $\to$ PC
JAL	0	0	0	0	1	0	1	0	◎	PC + 2 $\to$ ACC, B’ $\to$ PC
JR	0	0	0	0	1	0	1	1	×	ACC $\to$ PC

* shift/rotate 命令の詳細は表 3 を参照のこと。

- MSB_A が 0 かつ MSB_B が 0 で演算結果 C が 1 のとき
- MSB_A が 1 かつ MSB_B が 1 で演算結果 C が 0 のとき

キャリーが生じる条件は、

- MSB_A が 1 かつ MSB_B が 0 で演算結果 C が 0 のとき
- MSB_A が 0 かつ MSB_B が 1 で演算結果 C が 0 のとき
- MSB_A が 1 かつ MSB_B が 1 で演算結果 C が 0 のとき

である。

3.2.2　A と B と C が符号なし整数のとき

符号なし整数のとき、符号つき整数のキャリーが生じたときオーバーフローも発生したと考えることができる。よってオーバーフローとが生じる条件は、

- MSB_A が 0 かつ MSB_B が 0 で演算結果 C が 1 のとき
- MSB_A が 1 かつ MSB_B が 1 で演算結果 C が 1 のとき
- MSB_A が 1 かつ MSB_B が 0 で演算結果 C が 0 のとき
- MSB_A が 1 かつ MSB_B が 0 で演算結果 C が 0 のとき

表 3　演習 2

MSB_A	MSB_B	MSB_C	→	Flag
0	0	0		
0	0	1		VF
0	1	0		CF
0	1	1		
1	0	0		CF
1	0	1		
1	1	0		VF,CF
1	1	1		

表 4　演習 3 の LD のスイッチの開閉

	p0	p1	p2	p3	p4
	PC → MAR	Mem → IR	PC → MAR	Mem → ALU → MAR	Mem → ALU → A
sw3	×	open	×	open	open
sw4	×	open	×	open	open

キャリーが生じる条件はオーバーフローと同じく

- MSB_A が 0 かつ MSB_B が 0 で演算結果 C が 1 のとき
- MSB_A が 1 かつ MSB_B が 1 で演算結果 C が 1 のとき
- MSB_A が 1 かつ MSB_B が 0 で演算結果 C が 0 のとき
- MSB_A が 1 かつ MSB_B が 0 で演算結果 C が 0 のとき

3.3　演習 3　スイッチの開閉の制御

3.3.1　LD の場合

図 1 の instruction ”LD の [d](d)” に注目する。すると、図 2 において命令実行の通る手順は図 3 のようになる。SW 3 を通るのは P1,P3,P4 のとき。同じく、SW4 を通るのは P1,P3,P4 のとき。よってこのときそれぞれスイッチを開ける。まとめると、表 4 のようになる。

3.3.2　ST の場合

図 1 の instruction ”ST の [d](d)” に注目する。すると、図 2 において命令実行の通る手順は図 4 のようになる。SW 3 を通るのは P1,P3 のとき。同じく、SW4 を通るのは P1,P3 のとき。よってこのときそれぞれスイッチを開ける。まとめると、表 5 のようになる。

3.3.3　ADD の場合

図 1 の instruction ”ADD の [d](d)” に注目する。すると、図 2 において命令実行の通る手順は図 5 のようになる。SW 3 を通るのは P1,P3,P4 のとき。同じく、SW4 を通るのは P1,P3,P4 のとき。よってこのとき

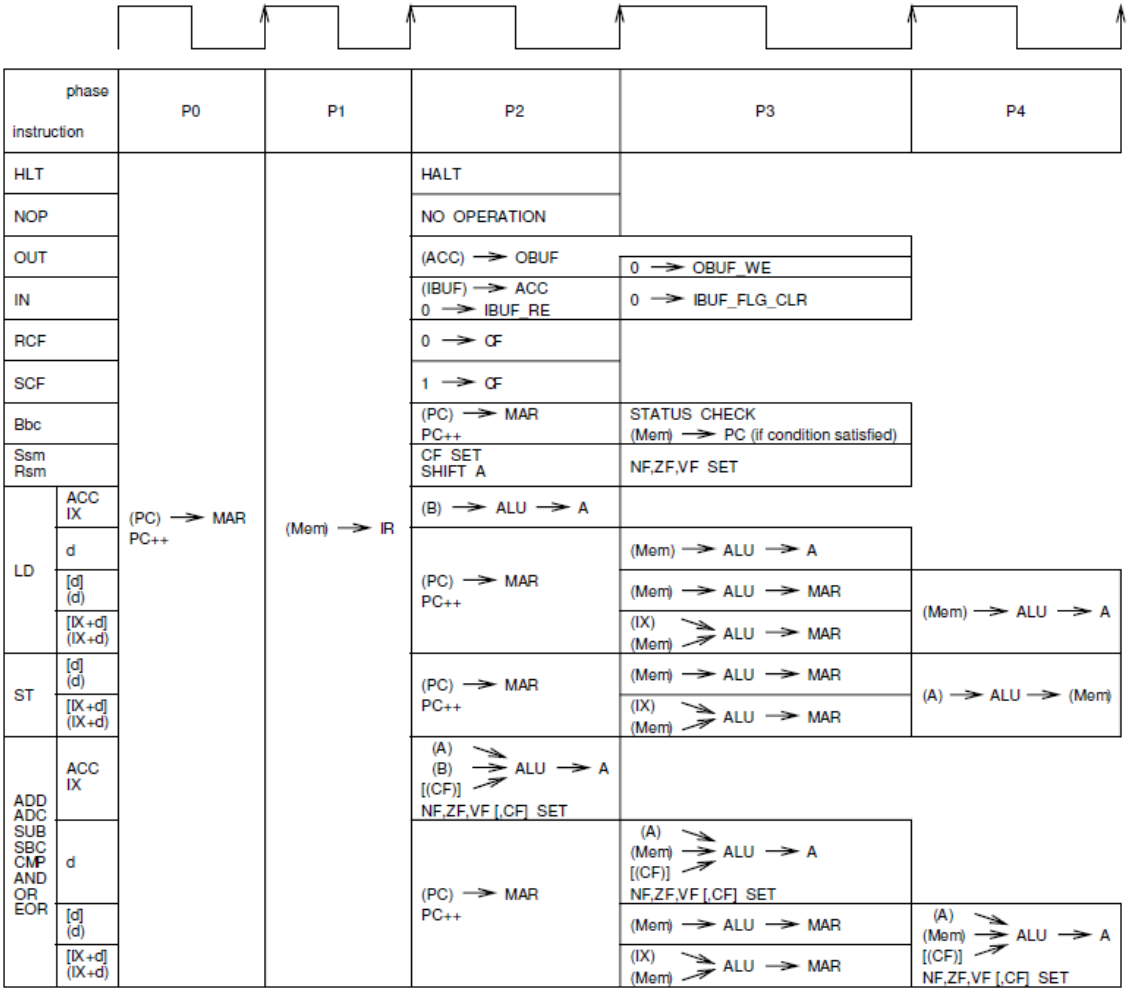

図 1　命令実行フェーズ

表 5　演習 3 の ST のスイッチの開閉

	p0	p1	p2	p3	p4
	PC → MAR	Mem → IR	PC → MAR	Mem → ALU → MAR	A → ALU → Mem
sw3	×	open	×	open	×
sw4	×	open	×	open	×

それぞれスイッチを開ける。まとめると、表 6 のようになる。

3.3.4　BA の場合

図 1 の instruction ”BA の [d](d)” に注目する。すると、図 2 において命令実行の通る手順は図 6 のようになる。SW 3 を通るのは P1,P3 のとき。同じく、SW4 を通るのは P1,P3 のとき。よってこのときそれぞれスイッチを開ける。まとめると、表 7 のようになる。

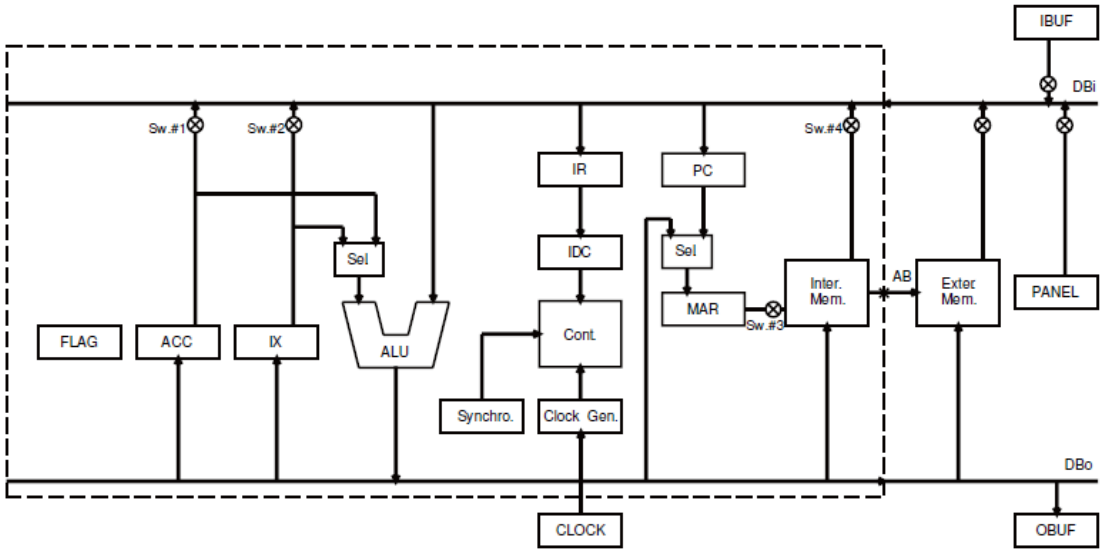

図 2　ブロック構成

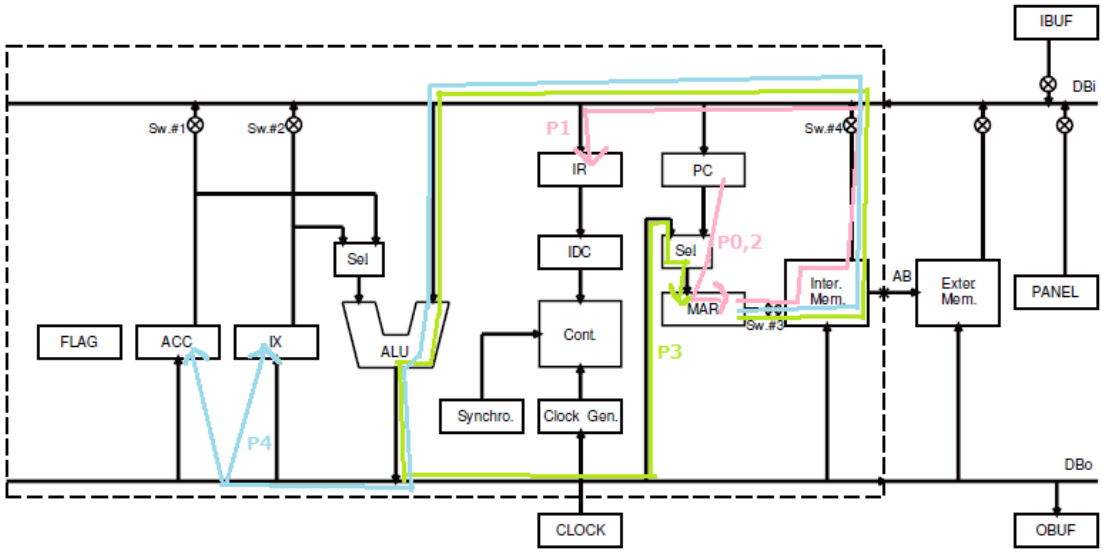

図 3　演習 3 に置ける LD のブロック

表 6　演習 3 の ADD のスイッチの開閉

	p0	p1	p2	p3	p4
	PC → MAR	Mem → IR	PC → MAR	Mem → ALU → MAR	A Mem → ALU → A CF
sw3	×	open	×	open	open
sw4	×	open	×	open	open

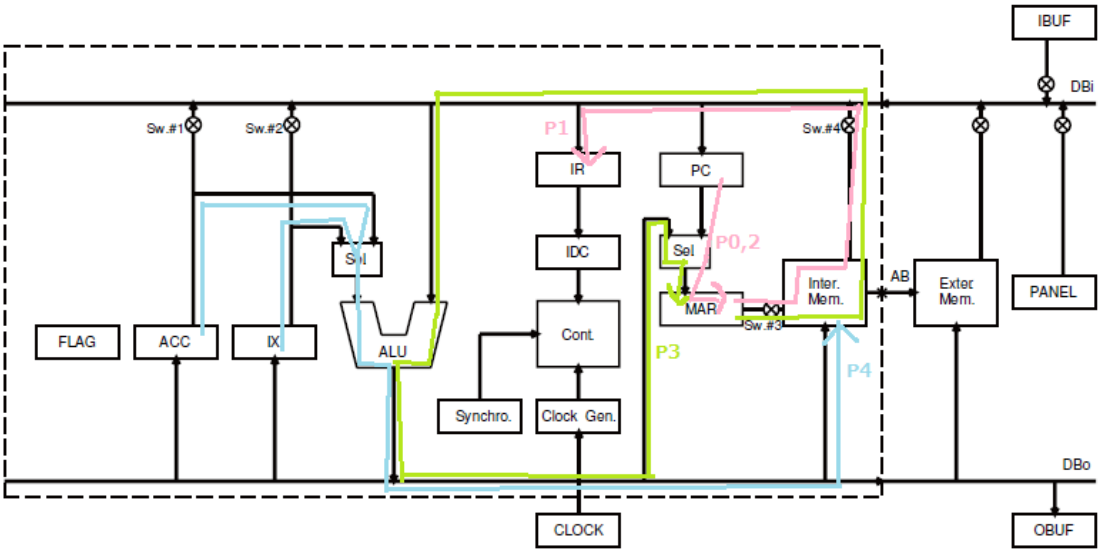

図 4　演習 3 に置ける ST のブロック

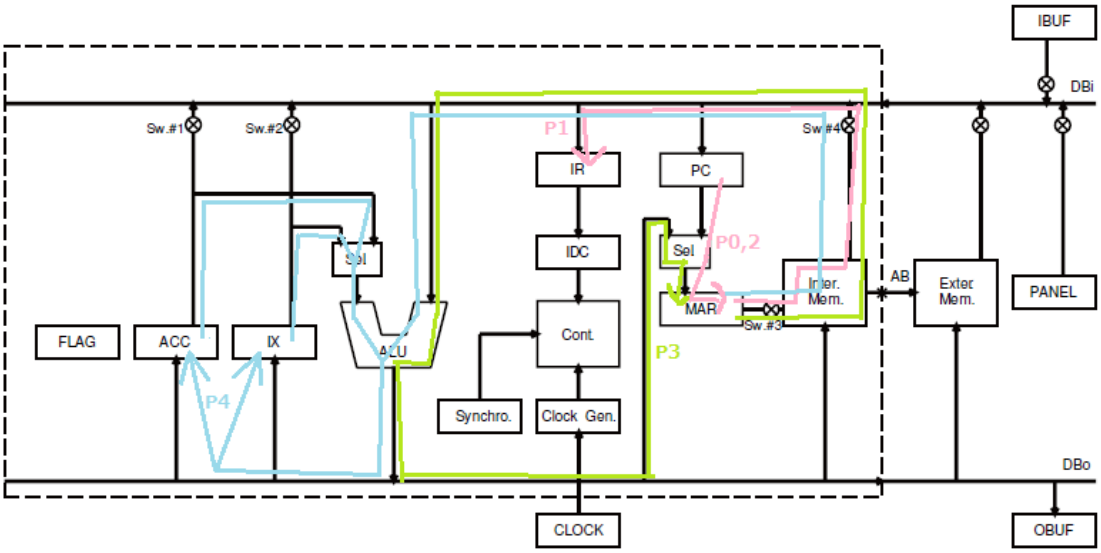

図 5　演習 3 に置ける ADD のブロック

表 7　演習 3 の BA のスイッチの開閉

	p0	p1	p2	p3
	PC → MAR	Mem → IR	PC → MAR	Mem → PC
sw3	×	open	×	open
sw4	×	open	×	open

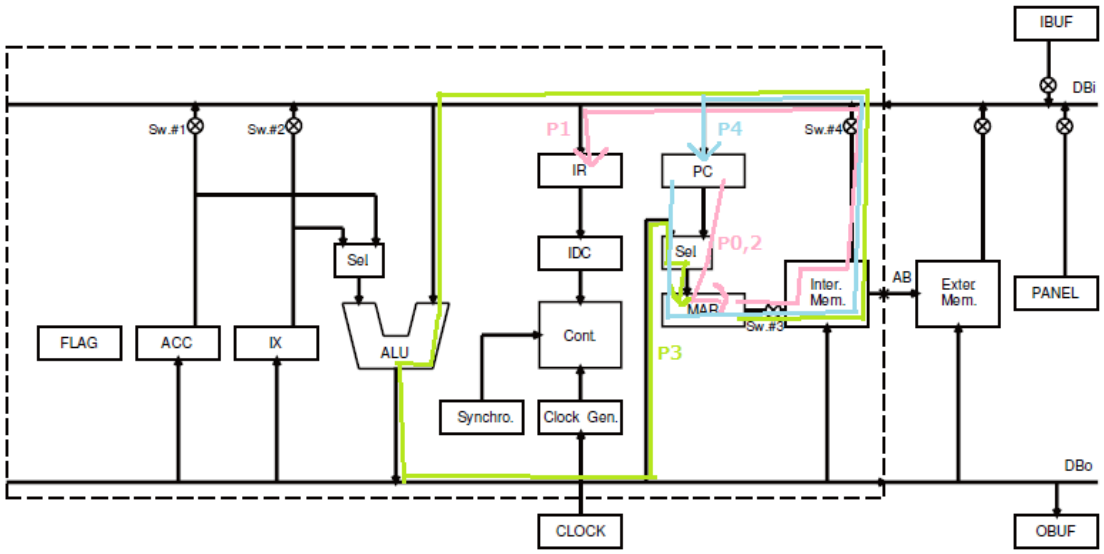

図 6　演習 3 に置ける BA のブロック

3.4　実習 1　ST の実装

ST 命令を実装するため、cpubord.c と cpubord.h を図 7, 図 8 のように書き換えた。cpubord.h では構造体 cpubord の定義を設定している。図 1 よりフェーズ 0 で MAR が使われており、フェーズ 1 で IR が使われている。そこで、cpubord のメンバーに mar,ir を追加した。図 1 にのっとって step() を作成した。まず、P1 では PC が MAR にコピーされるので、

```
cpub->mar = cpub->pc++;
```

プログラミングカウンターはコピー後更新（インクリメント）されるので後置インクリメントしておく。次に、MAR の値と等しいメモリの番地の値に変換され IR に格納される。

```
cpub->ir = cpub->mem[cpub->mar];
```

ここまでが命令フェッチでありまたこのとき、IR には実行コードが格納されている。ここから命令コードを分類（解読）していく。アドレッシングモードがデータ領域の ST に対応させたいので、ir が 75 のときの処理を考える。

```
switch (cpub->ir){
case 0x75:
```

P2 において、PC の値が MAR にコピーされ PC が更新される。そして、P 3 では MAR の値の番地のメモリの値が ALU を経由して MAR にコピーされる。このとき、MAR には命令コードの 2 語目が格納

されている。P 4では A が格納されている値が指定されたプログラム領域にコピーされる。プログラム領域の始まりは 0x100 番から 10 進数にすると 256 番目からであるから acc のコピーする先は MAR に IMEMORY_SIZE(256) 加えた番地である。

```
cpub->mar = cpub->pc++;
cpub->mar=cpub->mem[cpub->mar];
cpub->mem[cpub->mar +IMEMORY_SIZE] = cpub->acc;
```

実行結果は、図 9 のようになった。 <prog.txt >はプログラムファイルの中身を表し、 <Console >はコンソールの入力と出力をコピーしたものである。prog.txt で「75 03」を入力した。75 は 2 進数で 01110101 となる。これを 0111 0 101 と三つに分けることができる。0111 は ST の命令コードをさし、0 は第一オペランドが ACC であることをさし、101 は第二オペランドが絶対アドレスのデータ領域にであることをさす。命令語 2 語目の 03 はその第二オペランドのアドレスがデータ領域の 03H 番地にあることを示す。acc に 0x10 をセットして、命令を実行してやると、確かにデータ領域（100 番以降）の 03H 番地に 10 が格納され ST 命令が実装されたことが分かる。

```
1  int step(Cpub *cpub){
2  //P0
3  cpub->mar = cpub->pc++;
4  //P1
5  cpub->ir = cpub->mem[cpub->mar];
6    switch (cpub->ir){
7    case 0x75:
8    //P2
9    cpub->mar = cpub->pc++;
10   //P3
11   cpub->mar=cpub->mem[cpub->mar];
12   //P4
13   cpub->mem[cpub->mar +IMEMORY_SIZE] = cpub->acc;
14   break;
15   }
16 }
```

図 7　実習 1 における cpubord.c の step()

```
typedef struct cpuboard {
    Uword pc;
  //add
    Uword mar;
  //add
    Uword ir;
    Uword acc;
    Uword ix;
    Bit cf, vf, nf, zf;
```

図 8　実習 1 における cpubord.h の構造体 cpubord

```
<prog.txt>
    75 03

<Console>
    CPU0,PC=0x0> r prog.txt
    CPU0,PC=0x0> s acc 10
    acc=0x10(16,16) ix=0x00(0,0) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
    CPU0,PC=0x0> i
    Program Halted.
    CPU0,PC=0x2> m
      | 000: 75 03 00 00 00 00 00 00 | 008: 00 00 00 00 00 00 00 00
      | 010: 00 00 00 00 00 00 00 00 | 018: 00 00 00 00 00 00 00 00
      | 020: 00 00 00 00 00 00 00 00 | 028: 00 00 00 00 00 00 00 00

      | 0e0: 00 00 00 00 00 00 00 00 | 0e8: 00 00 00 00 00 00 00 00
      | 0f0: 00 00 00 00 00 00 00 00 | 0f8: 00 00 00 00 00 00 00 00
      | 100: 00 00 00 10 00 00 00 00 | 108: 00 00 00 00 00 00 00 00
      | 110: 00 00 00 00 00 00 00 00 | 118: 00 00 00 00 00 00 00 00
      | 120: 00 00 00 00 00 00 00 00 | 128: 00 00 00 00 00 00 00 00
```

図 9　test

4　考察

4.1　演習 1 における考察

問題を解くだけでは表 3 の B’ の列を見るだけでできる。しかしどうしてその命令が可変または固定の長さになるか考えてみる。B’ フィールドが〇（不要 or 必要）の場合は演算式の左辺に（B）がある特徴があった。

表 8 演習 3 の ADD のスイッチの開閉

	p0	p1	p2	p3	p4
	PC → MAR	Mem → IR	PC → MAR	Mem → ALU → MAR	A Mem → ALU → A CF
sw1	×	×	×	×	open(if B == ACC)
sw2	×	×	×	×	open(if B == IX)
sw3	×	open	×	open	open(if B == Mem)
sw4	×	open	×	open	open(B == Mem)

例えば LD は (B) → A、ADD は (A)+(B) → A といったように左辺に B がある。逆を言えば左辺に (B) がないものが固定長命令となっていた。当たり前かもしれないが、B のアドレスが、レジスタ指定であれば、2 語目はいらなくなるし絶対アドレスで有れば 2 語目が必要となってくる。一方で A は ACC か IX の 2 択であるため左辺に A があっても命令の種類には影響しないと考えた。またどうして固定長命令と可変長命令があるか考えた。固定長命令の場合命令語数が決まってるので語長による分岐がなく高速に命令処理ができる。一方ですべて固定の長さの命令にすると本来はいらない命令語も使う必要があったりしメモリを余分に使ってしまう欠点がある。逆に可変長命令は使う分の命令長にすることができメモリを節約できるが、処理に時間がかかってしまう。そこで可変長命令と固定長命令が 2 つできたのだと考えた。

4.2 演習３における考察

P0 から P2 まで同じ経路で処理されていることに気付いた。P0 から P2 までは命令フェッチにあたるため、どんな命令であっても命令フェッチの手順は同じであるためだと考えられる。また、スイッチは３と４がよく使われていて、スイッチ１、２は使われていなかった。そこでどんなときに使われるか考える。ADD の命令処理について考える。B は絶対値アドレスではなくレジスタ指定も含めた場合、B のアドレス方式によってスイッチ１、２が図 10 のように使われることがあると考えれる。(表 8)。このことからスイッチはをどのアドレス

4.3 実習１における考察

図 7 では、CPU の動きに基づき、実装した。しかし、ソフトウエアの観点からコードを簡略できるのではないかと考え、新たに実装し直してみた。（図 11）まず、mar は余分なので省き、ir に配列 mem の pc 番目を代入した。そして、switch 文で 2 語目の命令語にあるアドレスとデータ領域のベースアドレス（256）を足したアドレス番地に acc の値を代入する。このコードで再度実行すると実習 1 と同様の結果 (図 9) が得られコードを簡略化することも確認できた。また、第一命令コードが 75 のときデータ領域に acc の値を格納する。一方、74 のときプログラム領域に acc の値を格納する。なので、acc の値を代入するメモリ番号を 75 のときから 256 引いたものにすれば、74 も実装できると考えた。結果は図 12 のようになった。acc の値は確かにプログラム領域の 03 番地に格納された命令コード 74 も実装することができた。

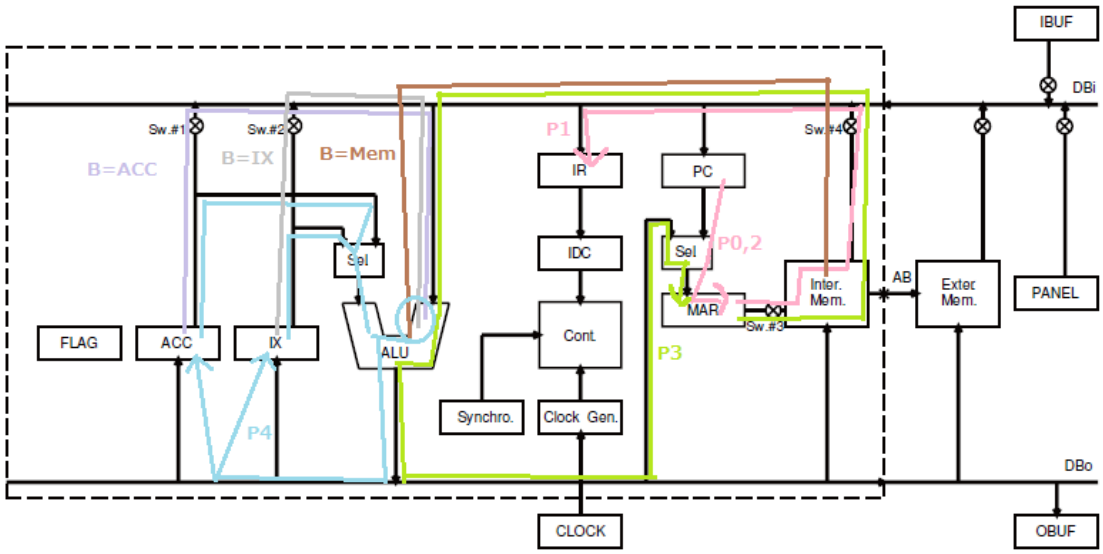

図 10　演習 3 に置ける ADD のブロック

```
1  int step(Cpub *cpub){
2    cpub->ir = cpub->mem[cpub->pc++];
3    switch (cpub->ir){
4    case 0x74:
5    cpub->mem[cpub->mem[cpub->pc]] = cpub->acc;
6    break;
7    case 0x75:
8    cpub->mem[cpub->mem[cpub->pc]+IMEMORY_SIZE] = cpub->acc;
9    break;
```

図 11　考察における cpubord.c の step() の再実装

```
<prog.txt>
    74 03

<Console>
    CPU0,PC=0x0> r prog.txt
    CPU0,PC=0x0> s acc 10
        acc=0x10(16,16) ix=0x00(0,0) cf=0 vf=0 nf=0 zf=0
        ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
    CPU0,PC=0x0> i
    Program Halted.
    CPU0,PC=0x1> m
        | 000: 74 03 00 10 00 00 00 00 | 008: 00 00 00 00 00 00 00 00
        | 010: 00 00 00 00 00 00 00 00 | 018: 00 00 00 00 00 00 00 00
```

図 12　実習 1 における実行結果

5　追加項目

5.1　演習 2 について、CF/VF をどのようにセット/リセットするか

MSB で判断できることが分かった。では実際にどのように CF/VF をどのようにセット/リセットするか考えてみる。符号なしの場合、キャリーフラグを cf、オーバーフローフラグを vf とする。キャリーが 0（リセット）されるときは実行結果が実行前の値より大きいときである。キャリーが 1（セット）されるときは実行結果が実行前の値より小さいときである。また、符号なしの場合、キャリーが生じたときはオーバーフローも生じているので、vf に cf を代入しておく。このことを利用すると、図 13 のようにかける。

```
1    result = a + b;
2    cf = a < result ? 0 : 1;
3    of = cf;
```

図 13　符号なしの CF/VF の実装

次に符号なしについて考えてみる。符号なしの場合はオーバーフローとなるときは 2 つの場合がある。一つは正と正を足して結果が負となったとき。二つ目は負と負を足して結果が正となったとき。これらをコードで書くと図 14 のようにかける。

```
result = a + b;
if ((result >= 0) && (a < 0) && (b < 0)) {
        vf = 1;
    } else if ((result <= 0) && (a > 0) && (b > 0)) {
        vf = 1;
    } else {
        vf = 0;
    }
```

図 14　符号付きの VF の実装

CF は符号付きの場合符号なしの場合と同様に実行結果より大きいか小さいかで判断することができる。また、符号ありの場合キャリーを利用して計算するので CF がセット、リセットされたかはさほど重要にはならない。

```
result = a + b;
cf = a < result ? 0 : 1;
```

図 15　符号付きの CF の実装

では次に実際符号ありかなしかをハードウエアではどのように判断できるのか考える。フラグの種類に NF（Negative Flag）がある。これは演算結果が（符号付き整数と仮定したとき）0 より小さいかどうかを判定するものでもあるが、実際 MSB が 1 かどうかを見ているものにすぎない。符号ありにするのか符号なしにしないかは最終的には設計者が決める。とは言ってもここを符号付きでここを符号なしということをいちいち覚えることはできない。そこでフラグが用いられる。キャリーフラグを用いて計算したとき符号付き整数とみなすことができる。例えば a と b を足し合わせるとき命令は ADD と ADC の 2 種類ある。ADD は実行語にキャリーフラグの影響はなく、ADC はキャリーが発生したとき次の桁に CF を加える。符号無しの場合、キャリーフラグで桁上げをすることで多倍長の計算が可能となる。一方で符号付き整数のときは負と正の足し算のようにキャリーが発生するがオーバーフローが発生しない場合があるのでキャリーが桁上げをすることはできない。用いられているフラグの種類に注目して、符号ありか無しかを判定できる。CF を演算に用いる、ADC、SBC のとき符号なし整数であると判断、または設計することができる。

5.2　演習 3 の ST 命令 [4]

図 1 の instruction ”ST の [d](d)” に注目する。すると、図 2 において命令実行の通る手順は図 16 のようになる。SW 3 を通るのは P1,P3,P4 のとき。同じく、SW4 を通るのは P1,P3 のとき。よってこのときそれぞれスイッチを開ける。まとめると、表 9 のようになる。ここで注意したいことは、図 17 のようにならないということだ。CPU とメモリの間でデータをやり取りは必ず MAR を介して行われる。詳しく言うと、MAR と MDR を介して行われる。例えば、メモリからデータを読みだす場合、読みだすデータのあるアドレスを MAR に設定し、MAR の番地のデータを MDR に設定する。またメモリからデータを書き込む場合、書き込み先のアドレスを MAR に設定し、MAR 番地に MAR のデータを書き込む。

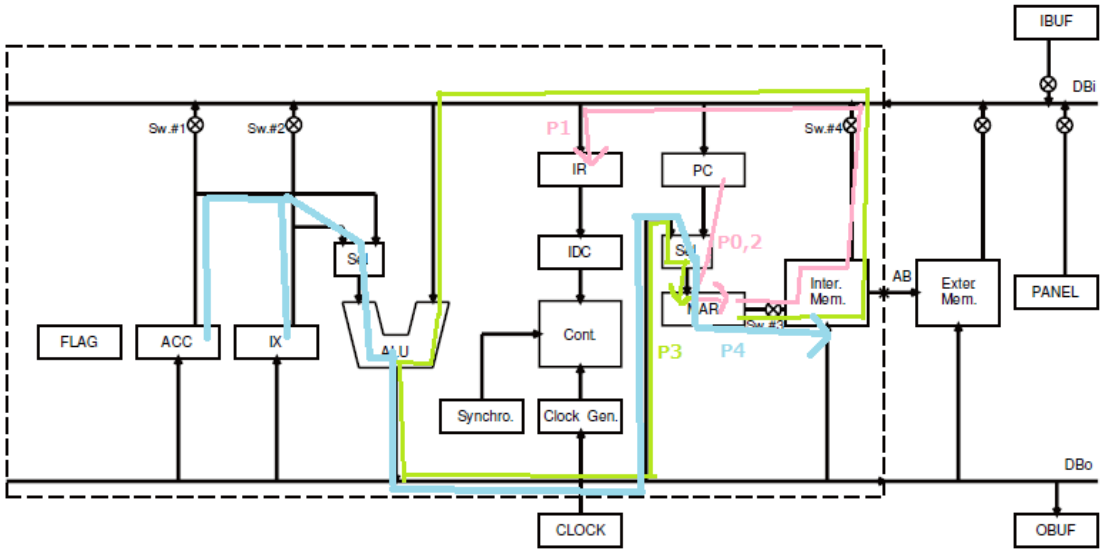

図 16　演習 3 に置ける ST のブロック

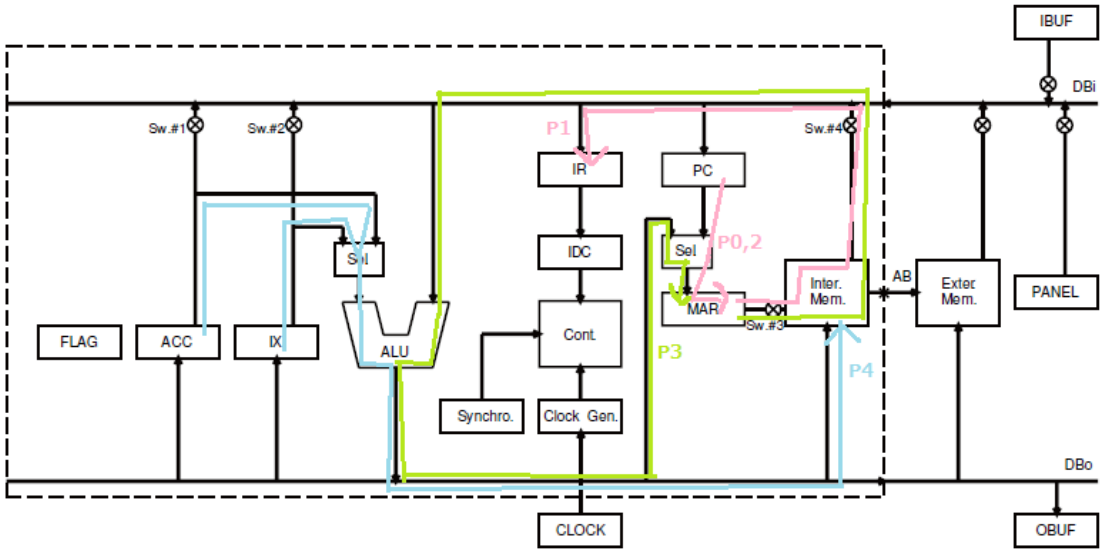

図 17　演習 3 に置ける間違った ST のブロック

表 9　演習 3 の ST のスイッチの開閉

	p0	p1	p2	p3	p4
	PC → MAR	Mem → IR	PC → MAR	Mem → ALU → MAR	A → ALU → Mem
sw3	×	open	×	open	open
sw4	×	open	×	open	×

6 追加項目 2

6.1 演習 2 について、CF/VF をどのようにセット/リセットするか

CF は最大桁で桁あがりが起こったとき 1 にセットする。桁上がりがないとき 0 にリセットする。例えば 1111+1 のとき最大桁では桁上がりが発生するので CF は 1 にセットされる。

```
 1111
+0001
-----
 0000
```

VF は符号付きと符号なし整数で場合分けする。符号なしの場合、キャリーフラグが立ったとき VF も立つ。符号付きのとき、A と B の符号が等しく実行結果の符号と異なるとき VF が 1 にセットされる。それ以外のとき 0 にリセットされる。たとえば、1111+1 のとき符号なしとすると、CF が 1 にセットされるので VF もセットされる。符号ありとみなすと最大桁は 1 と 0 で異なる。つまり A と B で符号が異なるので VF は 0 にリセットされる。

```
 1111
+0001
-----
 0000
```

このように ADD のとき CF と VF はセット、リセットされる。

7 参考文献

参考文献

[1] コンピュータアーキテクチャの基礎, 柴山潔 著, 2.2 基本命令セットアーキテクチャ, p43
[2] コンピュータアーキテクチャの基礎, 柴山潔 著, 6.1 固定小数点数の算術演算装置, p164
[3] コンピュータアーキテクチャの基礎, 柴山潔 著, 2.2 基本命令セットアーキテクチャ, p55
[4] http://www.ics.teikyo-u.ac.jp/wcasl2/tutorial/lesson01/comet10.html Hiroyoshi Watanabe 閲覧日 2022 年 5 月 5 日