目次

1　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。

2　実習４

掛け算を以下の図 1 のように実装した。まず、掛け算 ($z \leftarrow x \times y$) の命令コードを 70h とし、x,y,z を続く３つの連続する数値とする。例えば

```
|000: 70 3 4 0
```

としたとき、3×4 が実行され、0 の領域に計算結果 (=Ch) が代入されるようにした。まず、x の値を pc の値の番地のメモリの値とし、y も同様に pc を +1 した pc の値の番地のメモリの値を格納する。計算結果の z を０に初期化し、一次保存する値 tmp も初期化する。掛け算は各ビットを見て１ならばその桁数個左にシフトする事で計算ができる。例えば、左に１、２，３回シフトしたとき、その値は２、４，８倍となる。このことを利用して $y = k_0 2^0 + k_1 2^1 + k_2 2^2 + k_3 2^3$ のとき $x \times y = x(k_0 2^0 + k_1 2^1 + k_2 2^2 + k_3 2^3)$ となる。つまり x に１、２、４，８倍した値を足し合わせることで $x \times y$ の計算を行うことができる。ここから以下のアルゴリズムをつくることができる。
まず y の各ビットが１かどうか判定し、１ならば x をその桁分左にシフトさせた値を実行結果の値に足す。そして、次の桁のビッチも同様に判定して処理を行っていく。最大桁４まで見たときの実行結果の値を z とする。
y の各ビットが１かどうか判定することを論理積で表すと

```
if y&0b0001 == 0b0001
if y&0b0010 == 0b0010
if y&0b0100 == 0b0100
if y&0b1000 == 0b1000
```

で見ることができる。また x をその桁分左にシフトさせることをシフト演算を用いると

```
tmp = x << 1 //right shift 1
tmp = x << 2 //right shift 2
tmp = x << 3 //right shift 3
```

となる。

ここから発展させ、シフト演算を SLA、SLL を用いて表すことができると考えた。（図 2）先ほどの実装では、シフト演算を以下のように行っていた。

```
case 0x70:
        x = cpub->mem[cpub->pc++];
        y = cpub->mem[cpub->pc++];
        z = 0;
        tmp = 0;
        if((y&0b1) == 0b1){
                z += x;
        }
        if((y&0b10) == 0b10){
                tmp = x << 1;
                z += tmp;
        }
        if((y&0b100) == 0b100){
                tmp = x << 2;
                z += tmp;
        }
        if((y&0b1000) == 0b1000){
                tmp = x << 3;
                z += tmp;
        }
        cpub->mem[cpub->pc++] = z;
        break;
```

図 1: 実習 4 における掛け算の実装

```
tmp1 = x << 1
tmp2 = x << 2
```

これを自作関数 SLA を用いると

```
tmp = x
SLA(cpub,&tmp) //tmp1 = x << 1
SLA(cpub,&tmp) //tmp2 = x << 2
```

で書き換え可能となる。さらに、SLA を自作関数 SLL に置き換えることで算術シフトだけでなく、論術シフトも可能となる。(図 3)。そして if 文の部分で各桁比べているがこれを変数 num とおいて siht したものを用いることで、if 文ではなく while のループを用いて実装することができる。(図 4)
つぎにこのコードをアセンブリ言語で表してみると図 5 のようになる。これらの説明をしていく。x の番地を 100H,y の番地を 101H,z の番地を 102 番地,tmp の番地を 103H とする。

```
LD ACC,(00H)
LD IX,(01H)
ST ACC,(03H)
```

ある桁のビットが１のとき、tmp に x を左シフトしたものをコピーしておき、z に加える作業を考える。左シフトしたものは SLL を用いることができるので、一桁目が１のとき、tmp の 103H 番地にコピーする。その値を z の 102H 番地に格納する。

```
LD ACC,(03H)
ADD ACC,(02H)
ST ACC,(02H)
```

２、３、４桁目が１のときは、tmp の 103H にある値を SLL で左シフトし tmp の 103H 値を SLL して tmp に入れ、z の 102H に tmp の値を加えてやる。

```
LD ACC,(03H)
SLL ACC
ADD ACC,(02H)
ST ACC,(02H)
```

また、y の各桁のビットを１か０か確認するとき、フラグを用いて確認する。たとえば、最下位ビットが１の時、cf が１で 0 のとき cf が 0 となる SRL を用いることができる。また、SRL を４回実行することで、４桁のビットを確認することができる。SRL を用いると各ビットの１が最大１つずつ減少する。この y が 0 となるとき y に関して各ビットがすべて０となることを意味するので、y が０となったときをループの終了条件とする。

```
LOOP:    SRL IX
・・・
BNZ LOOP
HLT
```

3　実習５

掛け算 ($z \leftarrow x \times y$) の命令コードを 70h を実行したときの結果を確認してみる。(図 6) 「70 3 4 0」を実行したとき、これは 3×4 が実行され、0 のところに Ch(=12) が格納されることが予想できる。pc=0 で i 実行したとき、pc=4 にインクリメントされ、0 のところ（0x03 番地）に 0c が格納されていることが分かる。また、続けて、「70 5 6 0」を実行したところ pc=8 にインクリメントされ、0 のところ (0x07 番地) に 1e が格納されていることが分かる。1eh $= 16 + 14 = 30$ より確かに 5×6=30 が実行され、連続した番地に格納されていることが分かる。

```
case 0x70:
        x = cpub->mem[cpub->pc++];
        y = cpub->mem[cpub->pc++];
        z = 0;
        tmp = x;
        if((y&0b1) == 0b1){
                z += x;
        }
        SLA(cpub,&tmp);
        if((y&0b10) == 0b10){
                z += tmp;
        }
        SLA(cpub,&tmp);
        if((y&0b100) == 0b100){
                z += tmp;
        }
        SLA(cpub,&tmp);
        if((y&0b1000) == 0b1000){
                z += tmp;
        }
        cpub->mem[cpub->pc++] = z;
        break;
```

図 2: 実習 4 における SLA を用いた掛け算の実装

```
prog.txt>
70 3 4 0
70 5 6 0
Console>
CPU0,PC=0x0> r prog.txt
CPU0,PC=0x0> m
| 000: 70 03 04 00 70 05 06 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

CPU0,PC=0x0> i
Program Halted.
CPU0,PC=0x4> m
| 000: 70 03 04 0c 70 05 06 00 | 008: 00 00 00 00 00 00 00 00
CPU0,PC=0x4> i
Program Halted.
CPU0,PC=0x8> m
| 000: 70 03 04 0c 70 05 06 1e | 008: 00 00 00 00 00 00 00 00
```

図 6: 実習 5 における prog.txt と Console 結果

```
case 0x71:
                x = cpub->mem[cpub->pc++];
                y = cpub->mem[cpub->pc++];
                z = 0;
                //tmp = 0;
                tmp = x;
                if((y&0b1) == 0b1){
                        z += x;
                }
                SLL(cpub,&tmp);
                if((y&0b10) == 0b10){
                        //tmp = x << 1;
                        z += tmp;
                }
                SLL(cpub,&tmp);
                if((y&0b100) == 0b100){
                        //tmp = x << 2;
                        z += tmp;
                }
                SLL(cpub,&tmp);
                if((y&0b1000) == 0b1000){
                        //tmp = x << 3;
                        z += tmp;
                }
                cpub->mem[cpub->pc++] = z;
                break;
```

図 3: 実習 4 における SLL を用いた掛け算の実装

```
case 70:
  Uword num =1;
 while(num <=8){
   if((y&num) == num){
     z += x*num;
   }
   SLA(cpub,&num);
 }
 cpub->mem[cpub->pc++] = z;
 break;
```

図 4: 実習 4 におけるループを用いた掛け算の実装

```
1  START:LD ACC,(00H) 65 00
2      LD IX,(01H) 6D 01
3      ST ACC,(03H) 75 03
4  LOOP: SRL IX 48
5      BNC goto 35 E
6      LD ACC,(03H) 65 03
7      ADD ACC,(02H) B5 02
8      ST ACC,(02H) 75 02
9  goto: LD ACC,(03H) 65 03
10     SLL ACC 43
11     ST ACC,(03H) 75 03
12     AND IX,0 EC 0
13     BNZ LOOP 31
14     HLT 0F
```

図 5: 実習 4 におけるアセンブリ言語の掛け算の実装

そして、図 5 のコードを実行した結果についてみてみる。仮に５×３の計算をしてみる。まず、prog.txt を読み込み、100H 番地に５を 101H 番地に３を格納する。PC ０から５番まで実行すると、acc に５、ix に３、103H に５が格納されていることが分かる。

```
CPU0,PC=0x6> d
  acc=0x05(5,5)    ix=0x03(3,3)   cf=0 vf=0 nf=0 zf=0
  ibuf=0:0x00(0,0)    obuf=0:0x00(0,0)
CPU0,PC=0x6> m
| 100:  05 03 00 05 00 00 00 00    | 108:  00 00 00 00 00 00 00 00
```

次に、IX の値を SRL する。IX は 0011(3) なので最下位ビットは１。よってｃｆは１にセットされる。なので、BNC では cf は０でないので、goto にいかず、pc が９となる。そのあと、103H の値（05）が ACC に格納される. そして、ACC(05) と 02H の値（０）の和が ACC に格納され、ACC は 05 となる。この ACC の値を 02H 番地に格納される。実行してみると、確かに、

```
CPU0,PC=0x6> i
Program Halted.
CPU0,PC=0x7> d
  acc=0x05(5,5)    ix=0x03(3,3)   cf=1 vf=0 nf=0 zf=0
  ibuf=0:0x00(0,0)    obuf=0:0x00(0,0)
CPU0,PC=0x7> i
Program Halted.
CPU0,PC=0x9> i
CPU0,PC=0xb> i
Program Halted.
```

```
CPU0,PC=0xd> i
Program Halted.
CPU0,PC=0xd> d
  acc=0x05(5,5)    ix=0x03(3,3)   cf=1 vf=0 nf=0 zf=0
  ibuf=0:0x00(0,0)    obuf=0:0x00(0,0)
```

そのあと、03H の値（０５）を ACC にコピーし、ACC（0101）を SLL で左シフトする。その値 (1010) を 103H 番地に格納する。そして、IX が０かどうか確認する。IX は 0011(3) なので、０でないため、６に移動することが予想される。実行結果を見てみると確かに、６に移動されていることが分かる。

```
CPU0,PC=0x12> i Program Halted.
CPU0,PC=0x14> i Program Halted.
CPU0,PC=0x16> i Program Halted.
CPU0,PC=0x6>
```

以下、LOOP を繰り返していく。ｙの 3 は 0011 だから、２個の桁で１のビットがある。つまり、ｘの５を２回左シフトして足し合わせることになる。ループは２回で終わることが予想される。そして、実行が終了したとき、z の 102 番地に 5 × 3 の値（0f）が格納されることが期待される。図 8 は全体の結果を表したものである。ループは pc が６から 16 までのときである。ｐｃが１６から６に移動するのは１回ある。つまり、ループ内を２回実行したこと分かる。そしてｐｃが１８のとき、実行を終了する前のメモリは

```
CPU0,PC=0x18> m
| 100:  05 03 0f 14 00 00 00 00    | 108:  00 00 00 00 00 00 00 00
```

となっている。ｚの値である 102 番地には 0f(15) が格納されており、確かに 3 × 15 が実行されたことが確認できる。

```
prog.txt>
65 00
6D 01
75 03
48
35 0F
65 03
B5 02
75 02
65 03
43
75 03
EC 0
31
0F
```

図 7: 実習 5 におけるアセンブリ言語の prog.txt

```
Console>
CPU0,PC=0x0> i Program Halted.
CPU0,PC=0x2> i Program Halted.
CPU0,PC=0x4> i Program Halted.
CPU0,PC=0x6> i Program Halted.
CPU0,PC=0x7> i Program Halted.
CPU0,PC=0x9> i Program Halted.
CPU0,PC=0xb> i Program Halted.
CPU0,PC=0xd> i Program Halted.
CPU0,PC=0xf> i Program Halted.
CPU0,PC=0x11> i Program Halted.
CPU0,PC=0x12> i Program Halted.
CPU0,PC=0x14> i Program Halted.
CPU0,PC=0x16> i Program Halted.
CPU0,PC=0x6> d
    acc=0x0a(10,10) ix=0x01(1,1) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
CPU0,PC=0x6> i Program Halted.
CPU0,PC=0x7> i Program Halted.
CPU0,PC=0x9> i Program Halted.
...
CPU0,PC=0x11> i Program Halted.
CPU0,PC=0x12> i Program Halted.
CPU0,PC=0x14> i Program Halted.
CPU0,PC=0x16> i Program Halted.
CPU0,PC=0x18> m
| 100: 05 03 0f 14 00 00 00 00 | 108: 00 00 00 00 00 00 00 00
CPU0,PC=0x18> i
```

図 8: 実習 5 におけるアセンブリ言語の Console 結果

さらに発展として、他の演算と同様なアドレッシングモードを用いた掛け算を実装することは可能であると考えた。(図 9)
まず、まだ命令コードで使われていない 0x50 番台を掛け算の演算を行う命令コードとした。そして、ADD 関数と同様に MLT 関数を作った。これは実習 4 で作った掛け算の実行とアルゴリズムは同じである。抽象度を上げるため、関数化とし、値はポインタを通じて変更した。

```
// In switch(cpub->ir)
case 0x50 ... 0x5F: //MLT
            MLT(cpub,opa,opb);
            break;

// In Step Function
void MLT(Cpub *cpub,Uword *a, Uword *b){
    Uword c = 0;
    Uword tmp = *a;
    if(((*b)&0b1) == 0b1){
            c += *a;
    }
    SLL(cpub,&tmp);
    if(((*b)&0b10) == 0b10){
            c += tmp;
    }
    SLL(cpub,&tmp);
    if(((*b)&0b100) == 0b100){
            c += tmp;
    }
    SLL(cpub,&tmp);
    if(((*b)&0b1000) == 0b1000){
            c += tmp;
    }
    *a = c;
}
```

図 9: 実習 5 における MLT の実装

```
prog.txt>
50 //acc * acc
51 //acc * ix
52 2 //acc * 2
54 7 //acc * mem[007]
55 4 //acc * mem[107]
56 5 //acc * mem[007 + ix]
57 6 //acc * mem[106 + ix]
```

図 10: 実習 5 における prog.txt

確認として図 10 のように prog.txt を書いた。まず、命令コード 50 が実行できたか確認する。prog.txt を読み込み、acc を 2 にセットする。50 は acc*acc の値を acc に格納するを意味するので 2*2 が実行され acc に 4 がはいることが予想される。実行結果は図 11 のようになり、acc が 4 が格納されていることが分かる。つまり 50 は実行された。

```
CPU0,PC=0x0> r prog.txt
CPU0,PC=0x0> s acc 2
    acc=0x02(2,2) 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> d
    acc=0x04(4,4) ix=0x00(0,0) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 11: 実習 5 における命令コード 51 の結果

次に、命令コード 51 が実行されたか確認する。ix を 3 にセットする。51 は acc*ix の値を acc に格納するを意味するので、4*3 が実行され acc に 0ch(12) がはいることが予想される。実行結果は図 12 のようになり、acc が 0ch(12) が格納されていることが分かる。つまり 51 は実行された。

```
CPU0,PC=0x1> s ix 3
    acc=0x04(4,4) ix=0x03(3,3) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
CPU0,PC=0x1> i
Program Halted.
CPU0,PC=0x2> d
    acc=0x0c(12,12) ix=0x03(3,3) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 12: 実習 5 における命令コード 51 の結果

次に、命令コード [52 2] が実行されたか確認する。acc は 0ch(12) が格納されており即値 2 がある。52 は acc*即値の値が acc に格納されるを意味するので、0ch*2 = 12*2 = 18h(24) が acc に格納されることが予想される。実行結果は図 13 のようになり、acc が 18h(24) が格納されていることが分かる。つまり 52 2 は実行された。

```
CPU0,PC=0x2> i
Program Halted.
CPU0,PC=0x4> d
    acc=0x18(24,24) ix=0x03(3,3) cf=0 vf=0 nf=1 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 13: 実習 5 における命令コード 52 の結果

次に、命令コード [54 7] が実行されたか確認する。acc は 18h(24) が格納されており prog 領域の 7 番地

には 04 が格納されている。54 は acc*prog 領域の絶対値アドレスの値が acc に格納されるを意味するので、18h*4 = 24*4 = 60h(96) が acc に格納されることが予想される。実行結果は図 14 のようになり、acc が 60h(96) が格納されていることが分かる。つまり 54 7 は実行された。

```
CPU0,PC=0x4> m
    | 000: 50 51 52 02 54 07 55 04 | 008: 56 05 57 06 00 00 00 00
CPU0,PC=0x4> i
Program Halted.
CPU0,PC=0x6> d
    acc=0x60(96,96) ix=0x03(3,3) cf=0 vf=0 nf=1 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 14: 実習 5 における命令コード 54 の結果

次に、命令コード [55 4] が実行されたか確認する。acc に 3 を格納し data 領域の 4 番地に 05 を格納する。55 は acc*data 領域の絶対値アドレスの値が acc に格納されるを意味するので、3*5 = 0fh(15) が acc に格納されることが予想される。実行結果は図 15 のようになり、acc が 0fh(15) が格納されていることが分かる。つまり 55 4 は実行された。

```
CPU0,PC=0x6> s acc 3
    acc=0x03(3,3) ix=0x03(3,3) cf=0 vf=0 nf=1 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
CPU0,PC=0x6> w 104 5
    | 100: 00 00 00 00 05 00 00 00 | 108: 00 00 00 00 00 00 00 00
CPU0,PC=0x6> i
Program Halted.
CPU0,PC=0x8> d
    acc=0x0f(15,15) ix=0x03(3,3) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 15: 実習 5 における命令コード 55 の結果

次に、命令コード [56 5] が実行されたか確認する。acc に 0fh が格納されており prog 領域の 7 番地に 4 が格納されている。56 5 は 5 と ix の和の値と等しい prog 領域のメモリ番地のデータ*acc の値を acc に格納されるを意味する。ix は 2 だから prog 領域の 7 番地のデータと acc の積を acc に格納する。7 番地には 4 があるので 0fh*4 = 15*4 = 3ch(60) が acc に格納されることが予想される。実行結果は図 16 のようになり、acc が 3ch(60) が格納されていることが分かる。つまり 56 5 は実行された。

```
CPU0,PC=0x8> s ix 2
    acc=0x0f(15,15) ix=0x02(2,2) cf=0 vf=0 nf=0 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
CPU0,PC=0x8> m
    | 000: 50 51 52 02 54 07 55 04 | 008: 56 05 57 06 00 00 00 00
    | 100: 00 00 00 00 05 00 00 00 | 108: 00 00 00 00 00 00 00 00
CPU0,PC=0x8> i
Program Halted.
CPU0,PC=0xa> d
    acc=0x3c(60,60) ix=0x02(2,2) cf=0 vf=0 nf=1 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 16: 実習 5 における命令コード 56 の結果

次に命令コード [57 6] が実行されたかどうか確認する。acc に 3ch(60) が格納されており、data 領域 8 に 2 を格納したととする。[57 6] は 6 と ix の綿の値と等しい data 領域のメモリ番地のデータ*acc の値を格納する。ix には 2 があるので 108 番地に注目する。108 番地には 2 があるので、3ch*2 = 60 * 2 = 78h(120) が acc に格納されることが予想される。実行結果は図 17 のようになり、acc が 78h(120) が格納されていることが分かる。つまり 57 2 は実行された。

```
CPU0,PC=0xa> w 108 2
    | 100: 00 00 00 00 05 00 00 00 | 108: 02 00 00 00 00 00 00 00
CPU0,PC=0xa> i
Program Halted.
CPU0,PC=0xc> d
    acc=0x78(120,120) ix=0x02(2,2) cf=1 vf=0 nf=1 zf=0
    ibuf=0:0x00(0,0) obuf=0:0x00(0,0)
```

図 17: 実習 5 における命令コード 57 の結果

4　演習 4

JAL と JR 命令を実行するとき、PC 付近に新たに sel の追加と pc に 2 インクリメントする生成器をつくる必要がある。JAL は PC+2 ⇒ ACC と B' ⇒ PC を実行する命令である。元のブロック図は PC と MAR が一直線につながっているので PC に＋ 2 インクリメントする生成器を創る必要がある。また、JR は ACC ⇒ PC を実行する命令である。元のブロック図では ACC から PC につながる経路がないので、PC につながる経路をつくる必要がある。(図 18) また命令フェーズの表は表 1 のようになる。

次に各フェーズの動きをみてみる。JAL はまず、P 0 では命令フェッチで PC から実行する命令のアドレ

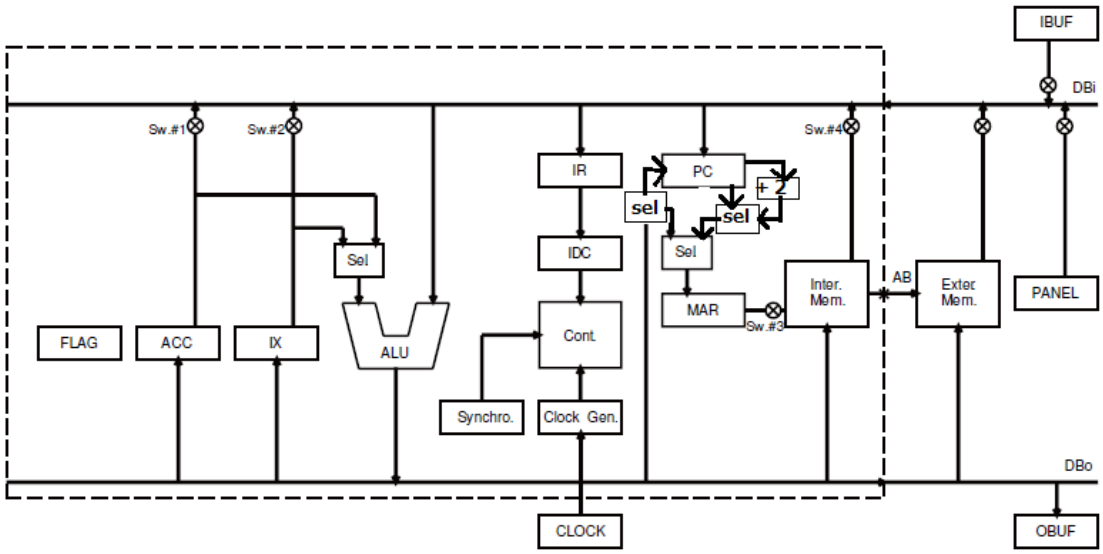

図 18: 演習 4 におけるブロック図

表 1: 演習 4 命令実行フェーズ

	P0	P1	P2	P3	P4
JAL	PC → MAR, PC++	Mem → IR	PC → +2 → MAR	MEM → ACC	Mem → PC
JR	PC → MAR, PC++	Mem → IR	ACC → PC		

スを MAR にコピーする。そして PC をインクリメントする。P 1 では MAR にコピーしたアドレスのメモリのデータを読み出し IR に格納する。P2 では PC の値を＋２インクリメントする生成器を通過し、MAR に格納する。PC に 2 インクリメントされた番地のメモリのデータを ACC に格納する。P4 では次実行する命令の番地の値を Mem を通って PC に代入する。（図 19）
JR は JAL と P0 と P 1 は同じである。P 0 では命令フェッチで PC から実行する命令のアドレスを MAR にコピーする。そして PC をインクリメントする。P 1 では MAR にコピーしたアドレスのメモリのデータを読み出し IR に格納する。P 2 では acc の値を新たに作った PC の経路を通って PC に格納される。acc から ALU を通る。その後、PC へ通る。(図 20)

5　JAL と JR について

JAL は Jump And Link である。現在の命令の次の命令をレジスタ acc にコピーして、指定された場所へ pc を移動する命令である。JAL の命令語は 2 語目 B’があるので、次の命令は PC+ 1 ではなく PC ＋２となる。JR は Register Jump である。指定されたレジスタの値が次実行する命令のアドレスとなる。この JAL と JR は組み合わせて使われる。まず、JAL で次命令するアドレス

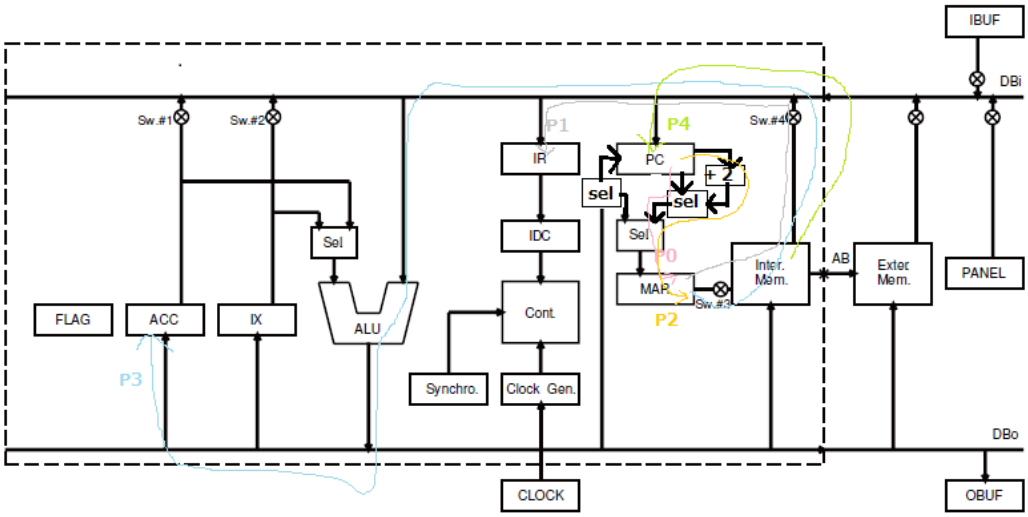

図 19: 演習 4 における JAL のブロック図

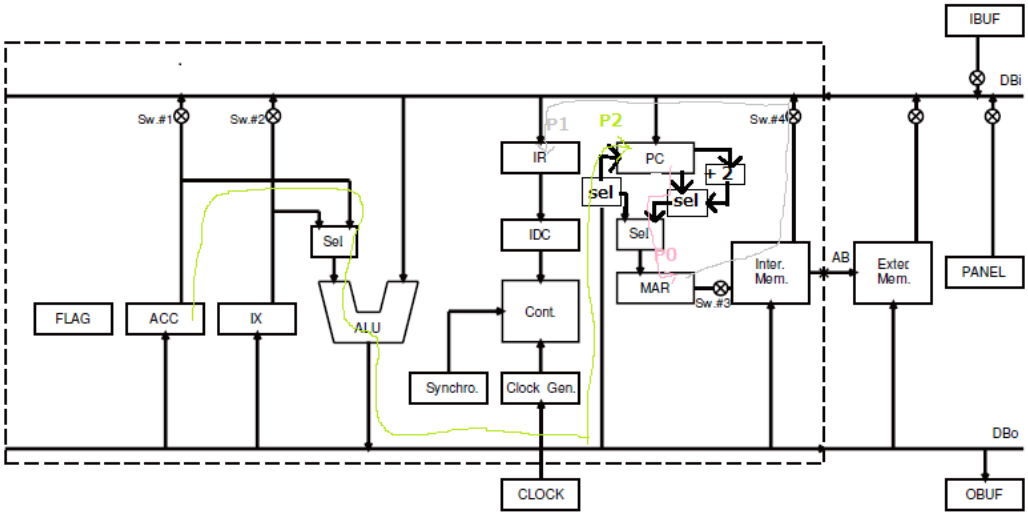

図 20: 演習 4 における JR のブロック図

6 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] https://brain.cc.kogakuin.ac.jp/ kanamaru/lecture/MP/final/part06/node3.html