第 6 回　汎用レジスタマシンへの拡張

汎用レジスタマシン

前回に学んだアキュムレータマシンは、ループを含むアルゴリズムの実行が可能であったが、以下の問題があった。

- アキュムレータだけしかレジスタがないので、演算の度に結果をメモリに格納する必要がある。
- アキュムレータだけしかレジスタがないため、メモリのアドレスを動的に指し示す機能がない。つまりポインタの実現ができない。

さらに、これはアキュムレータマシンの一般的な問題ではないのだが、アドレスの指定が長くなりすぎると面倒なので、アドレス空間を 8 ビット、すなわち全体で 256 とした点がある。コンピュータの歴史を振り返ると、開発するプログラムサイズは年々増大し、必要とされるアドレス空間は直線的に増加している。現在の高性能のマイクロプロセッサが 64 ビットアーキテクチャを採用している主な原因は 32 ビットアーキテクチャではアドレス空間が不足するためである。このことを考えると、アドレス空間 8 ビットはいくら演習用でも酷いとしか思えない。

そこで、以下の改造を行おう。

- レジスタ数を 8 個に増強する
- アドレスを 16bit に拡張して、アドレス空間を 64K とする

まず、8 個のレジスタは、基本的に全てが同じ機能を持たせるようにする。初期のマシン（現在でも 8 ビットマシンなどは）ではレジスタの機能が専門化され、レジスタによって可能な演算や操作（メモリのアドレスのみ格納する等）が決っている場合が多かった。これを専用レジスタマシンと呼ぶ。しかし、専用レジスタマシンは、レジスタの機能を考えてプログラムをするのが大変で、コンパイラも作りにくいため、半導体の面積の余裕ができるにつれ、一部の例外を除いてすべてのレジスタが同様にすべての機能を実現できるマシン、すなわち汎用レジスタマシンに移行した。我々も汎用レジスタマシンに拡張しよう。

汎用レジスタマシンでは、代表的な演算命令のオペランドが 2 または 3 となり、2 オペランド命令、3 オペランド命令と呼ぶ。

```
ADD r0, r1          r0<-r0+r1
ADD r0, r1, r2      r0<-r1+r2
```

ここで、r0 は結果が格納されるレジスタであり、ディスティネーションレジスタ（オペランド）と呼ぶ。一方、r1(r2) は、元となるデータが格納され、演算によって変化しないレジスタであり、ソースレジスタ（オペランド）と呼ぶ。マシンによっては、オペランドに、アキュムレータマシン同様、メモリのアドレスを書くことができるものもある。

```
ADD r0, 1000           r0<-r0+(1000)
ADD 1000, 2000, 2002   (1000)<-(2000)+(2002)
```

最初の命令は r0 と 1000 番地の中身を足して、答えを r0 に格納し、次の命令は 2000 番地と 2002 番地の中身を足して答えを 1000 番地に格納する。

ここでは、ディスティネーションオペランドを一番左に書く表記法を採用している。この表記法は IBM や Intel のマシンで使われている方法で、現在一般的である（ディスティネーションオペランドを一番右に書くやり方もあり、DEC やモトローラのマシン、日立の SH などに使われている）。

汎用レジスタマシンの分類

汎用レジスタマシンの命令セットは、演算命令のオペランドにメモリを指定できるかどうかによって以下のように分類される。

- まったく指定できない: register-register アーキテクチャ、load-store アーキテクチャまたは RISC(Reduced Instruction Set Computer) と呼ばれる。オペランドにメモリを指定できないため、演算を行う場合、必ずレジスタに値を Load してからレジスタ間で演算し、答えをメモリに Store する形を取る。一定の処理を行うための命令数は多くなるが、個々の命令は単純化され固定長で実装可能である。SPARC,MIPS,ARM,SH-4 などのマイクロプロセッサはこの命令形式を用いている。
- どこでも指定できる: memory-memory アーキテクチャ、オペランドのどの部分でもメモリが指定できる。もちろんレジスタ間でも演算が可能だが、メモリ中のデータ同士を直接計算する場合が多くなる。一定の処理を行うための命令数は少ないが、個々の命令は可変長にする必要があり、複雑になるため CISC(Comptex Instruction Set Computer) と呼ばれる。DEC の VAX-11 で用いられ、80 年代はメジャーであったが、性能向上が難しいことから現在はほとんど使われていない。
- 1 つだけ指定できる: register-memory アーキテクチャ、オペランドに一ヶ所だけメモリを指定できる。多くの場合 2 オペランド命令でソースオペランドに限られる。register-register 型と memory-memory 型の中間的な性質を持つ。一定の処理を行うための命令数はさほど多くならず、命令は可変長にする必要があるが、memory-memory 型ほど悲惨なことにはならない。現在、デスクトップやラップトップで最も良く用いられている Intel/AMD の IA32,IA64 はこの型に属する。

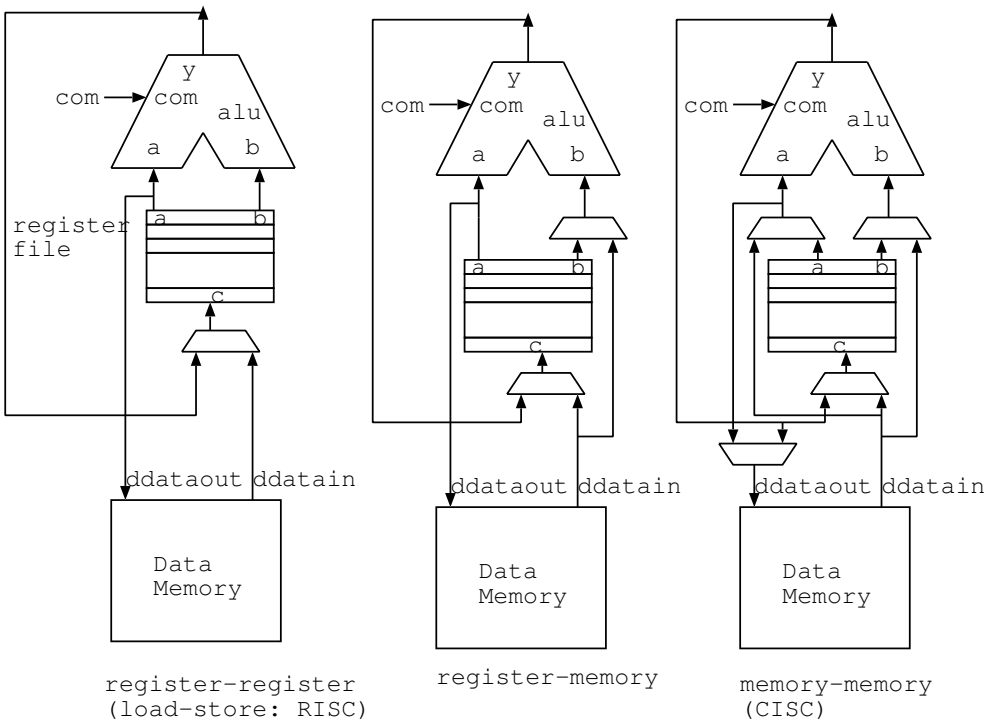

図 1: 汎用レジスタマシンのデータパス

例えば 0x10 番地のデータと 0x11 番地の内容を加算して 0x12 に格納する場合、register-register 型では以下のようになる。

```
LDI r0,#0x10
```

```
LD r1,(r0)
LDI r0,#0x11
LD r2,(r0)
ADD r2,r1
LDI r0,#0x12
ST r2,(r0)
```

ここでは、LDI という命令を使ってレジスタにまずアクセスする番地を格納しておいて、間接的にメモリの読み書きを行う方法を使っている。この方法はレジスタ間接指定と呼ぶ。この方法は後で詳しく説明しよう。まずは見ての通り、register-register 型は、命令数が多いことがわかる。しかし、命令の中に長いアドレスを埋め込む必要がなく、命令長を固定にすることが可能である。

一方、memory-memory 型では一命令ですむ。

```
ADD 0x12,0x11,0x10
```

しかし、オペランドにはレジスタも指定できるので、命令長は長くなったり短くしたりしなければならないことがわかる。

register-memory 型は、ディスティネーションは、レジスタのみ許される場合が多いので以下のようになる。

```
LD r0,0x10
ADD r0,0x11
ST r0,0x12
```

これも、ADD 命令にメモリアドレスを書く場合と書かない場合で、命令長を変える必要があるが、その可変の程度は memory-memory 型よりもひどくはない。一方、一命令では済まず、3 命令必要になっている。register-memory 型は、アキュムレータマシンにレジスタを指定する拡張を行ったという点で、非常に自然な拡張であると言える。

3 つの方式のハードウェア構成を考えてみよう。複数の汎用レジスタは、レジスタファイルというハードウェアモジュールで実装される。レジスタファイルは小規模なマルチポートメモリであり、ここでは図 1 に示すように、読み出しに 2 ポート (a,b)、書き込みに 1 ポート (c) を持っている。これを単純にアキュムレータの代わりに入れ替えた構成が、図の真ん中に示す register-memory 型である。この点で、register-memory 型はアキュムレータマシンのもっとも自然な発展であると言える。次に左に示す register-register 型は、メモリから ALU への直接入力が存在せず、全てのデータが一度レジスタを介して演算される。一方で、memory-memory 型はレジスタとメモリが同様に ALU に接続されている構成である。

さて、register-memory 型は、アキュムレータマシンの自然の発展であり、また、最も良く用いられている Intel/AMD の命令セットは、この形である。しかし、この型が最もうまく行っていると勘違いしてはならない。実は Intel/AMD のマイクロプロセッサは、Pentium II 以降は、register-memory 型の命令を内部的に register-register 型に変換して実行している。register-memory 型の見掛けを維持しているのは、アキュムレータマシンの自然な発展として register-memory 型を採用してしまったが故に蓄積したプログラムとの互換性 (Compativility) を保持するために過ぎない。高速に実行するという点でいうと、register-register 型つまり RISC はとことん優れており、わざわざ変換して実行せざるを得ないのだ。つまり、これらのプロセッサは概観が register-memory 型でも、一皮剥けば RISC であり、組み込み用ではほとんどの CPU が RISC であることを考えると、プロセッサの命令セットという点では RISC の圧勝である。

そこで、ここでも、今までのアキュムレータマシンを register-register 型、すなわち RISC に拡張していくことにする。ここで、この RISC を名前を付け、POCO と呼ぶことにする。

16bit RISC: POCO

基本演算命令

今までのアキュムレータマシンは、オペランドはメモリのアドレス 1 つで済んだ。また命令数も少なかったので、命令コードは 4bit の Opcode に 8bit の operand を備えて 12 ビット構成とした。

```
ADD 0     accum <- accum + 0 番地     0110 00000000
ADDI #5   accum <- accum + 5         1100 00000101
```

しかし、POCO ではレジスタが複数あるので、オペランドは 2 個必要である。すなわち典型的な演算命令は以下の形式を取る。レジスタはそれぞれが今までのアキュムレータの代わりができる 16bit レジスタで r0-r7 まで、計 8 個を選ぶことができる。

```
ADD r0, r1    r0 <- r0+r1
ADDI r0,#5    r0 <- r0+5
```

さて、命令をどのように設計すべきだろうか。まず、オペランドにはレジスタの番号を指定する必要がある。レジスタは 8 個なので、一つあたり 3 ビット必要である。また、Opcode は 4bit では 16 種類しか命令が指定できないので、まずこれを 5bit に拡張し、さらに命令数を増やす工夫をしよう。

さて、ADD r0, r1 などのレジスタ同士を加算する命令と ADDI などのようにイミーディエイトを指定する命令では、命令に必要なビット幅が異なるが、register-register 型、すなわち RISC では、種類毎に命令のビット数を変えたりせず、これを単一命令長で実現して設計上の面倒を省く。POCO では命令長を、データの幅と同じ 16bit とする。このことでハードウェア構成はさらに簡単になる。

まず、イミーディエイト命令は、命令中に数字が入る必要がある。これは 16 ビット入れれば、データ幅と一致するが、これでは命令コードが 16 ビットを越えてしまうし、イミーディエイトはさほど大きな数字を使うことは少ないので効率が良くない。今、Opcode は 5bit、レジスタは 3 ビットで指定できるので、16-5-3+8bit 残る。これをイミーディエイトで指定する数字として、アキュムレータマシン同様符号拡張することにしよう。

```
ADDI rd,#X          01100 ddd XXXXXXXX
```

ここで、rd は結果が格納されるディスティネーションレジスタを示し、r0-r7 を選べる。対応する数字を ddd に入れておく。イミーディエイトデータ X は残りの 8 ビットを利用し、符号拡張される。opcode 部 (01100), レジスタ (ddd)、イミーディエイト (XXXXXXXX) など、命令中のそれぞれ意味を持った部分のことを命令フィールドと呼ぶ。

次に ADD などレジスタを 2 つ指定するタイプの命令について考えよう。同様に opcode は 5 ビットとし、レジスタ二つを指定するフィールドを設けると、残りは、16-5-3*2=5bit 余る。ここで、この余りのフィールドを用いて命令数を増やすことを考える。すまわち、全ての ALU を用いた命令は、opcode を 00000 とし、この余りの 5 ビットのうち下位 3 ビットを ALU のコマンドに入れて演算を指定する。この下位 5 ビットを function フィールドと呼ぶ。3bit 目、4bit 目の 2 ビットが完全に余るが、これは将来のために取っておくことにし、ALU 命令ではここを 00 とする。すなわち、opcode が 00000 で、function が 00XXX であれば、ALU 命令であり、XXX が ALU のコマンドとなる。

今までと同じ ALU を利用すれば以下の命令が定義される。

```
NOP                          00000 --- --- 00000
MV  rd,rs    rd <- rs        00000 ddd sss 00001
AND rd,rs    rd <- rd AND rs 00000 ddd sss 00010
OR  rd,rs    rd <- rd OR rs  00000 ddd sss 00011
SL  rd       rd <- rd<<1     00000 ddd --- 00100
SR  rd       rd <- rd>>1     00000 ddd --- 00101
ADD rd,rs    rd <- rd + rs   00000 ddd sss 00110
SUB rd,rs    rd <- rd - rs   00000 ddd sss 00111
```

ddd 同様、sss にはソースレジスタ rs を示す番号 (r0-r7:000-111) が入る。RISC では先の図 1 に示すように、ALU の B 入力につながっているのはレジスタファイルの B ポートなので、ALU のコマンド 001: THB は、LD ではなくレジスタ間の転送命令である MV(move) となる点に注意されたい。THA は、今まで同様に NOP(No-operation) 命令となり何も行われない。この方法はなんだかセコいようだが、命令数を増やし、ハードウェアを簡単にするのに有効なので、ほとんどの RISC で同様の方法を用いている。

メモリアクセス命令

メモリのアドレスを 16bit に拡張したため、LD 命令と ST 命令で直接アドレス指定することはできない。イミーディエイト命令のように符号拡張しても指定できるアドレスの範囲が広がるわけではないので、この場合あまり意味がない。そこで、POCO では、他の RISC 同様、レジスタの中身で、メモリのアドレスを表す方法を採用する。これをレジスタ間接メモリアクセスと呼ぶ。

```
LD r0, (r1)     r0 <- (r1)
```

上記の命令を実行すると、r1 に中身がアドレスとなり、読み出されたデータが r0 に格納される。r1 が 0 ならば 0 番地、100 ならば 100 番地が読み出される。ST も同様に表される。

```
ST r0, (r1)    r0 -> (r1)
```

この場合、r0 の中身が、r1 で指定するアドレスに書き込まれる。すなわち、r1 が 0 ならば 0 番地、100 ならば 100 番地に r0 の中身が書き込まれることになる。ここで、データの移動の方向が ST 命令のみは左（レジスタ）から右 (メモリ）になる点に注意されたい。LD 命令、ST 命令は、レジスタを 2 つ指定するので ALU 命令と同じ形で表現することにしよう。ここでは function の 5,4 ビット目が 01 で LD, ST を表すこととした。ST を先に持ってきたのはアキュムレータマシンと命令コードを同じにした方が慣れていていいかと思ったためで、深い意味はない。aaa はメモリアドレスを示すレジスタという意味で、やはり r0-r7 までで 000-111 で表す。

```
ST rs, (ra)     rs -> (ra)     00000 sss aaa 01000
LD rd, (ra)     rd <- (ra)     00000 ddd aaa 01001
```

レジスタ間接指定によって、ポインタ、配列、スタックなど様々な方法が実現できる。これは後にプログラムを書いてみるとその威力がわかる。

イミーディエイト命令の追加

POCO では、簡単のため、LD, ST はレジスタ間接指定以外は設けないことにする。このため、メモリのアドレスなどを簡単にレジスタ内に作ることのできる必要がある。これには今の ADDI だけでは不足する。そこで、POCO では以下の命令を設けることにする。

```
LDI rd,#X          01000 ddd XXXXXXXX    rd <- X（符号拡張）
LDIU rd,#X         01001 ddd XXXXXXXX    rd <- X（符号拡張なし）
ADDIU rd,#X        01101 ddd XXXXXXXX    rd <- rd + X（符号拡張なし）
LDHI rd,#X         01010 ddd XXXXXXXX    rd <- X|0
```

LDI(Load Immediate) 命令は、X を符号拡張してレジスタに格納し、LDIU(Load Immediate Unsigned) 命令は符号拡張しないでレジスタに格納する。メモリアドレスなどは符号拡張されない場合が多いので、後者が役に立つ。また、これに付随して加算命令もアドレス計算用に符号拡張しないものを設けてやる。これが ADDIU(Add Immediate Unsigned) 命令である。後に述べるが、Unsigned 命令は符号拡張回路を通さずに 0 を補えばいいので、むしろ簡単に実装できる。opcode はかなりいいかげんに決めてある。ADDIU は ADDI を 1100 としたので、その一つ上の番号を

使った。さて、LDI, LDIU はレジスタの下位 8 ビットに数を入れるのには便利だが、これだと上位 8 ビットは、全て 1 か全て 0 が常に入ることになる。これでは困るので上位 8 ビットに、指定した値 X を入れて、下位 8 ビットは 0 にクリアするのが LDHI(Load High Immediate) である。この命令の働きを図 2 に示す。

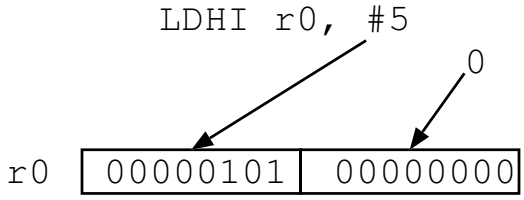

図 2: LDHI の操作

LDHI と ADDIU の組み合わせで、16 ビットの数をレジスタ中に作ることができる。例えば、0x8090 番地をアクセスしたければ、以下のように行えば良い。

```
LDHI r0, #0x80   01010 000 10000000
ADDIU r0, #0x90  01101 000 10010000
LD r1, (r0)      00000 001 000 01001
```

ちなみに、この辺の手口は、多くの RISC で共通に使われている。

分岐命令の拡張

レジスタ数が増えたため、アキュムレータの分岐命令はレジスタのどの中身を調べるか、を指定する必要がある。このため、レジスタ番号のフィールドを付け加えて、以下のように分岐命令を拡張する。

```
BEZ rd, X   if (rd==0) pc <- pc + X    10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X    10001 ddd XXXXXXXX
```

ここで、もう一つ拡張の必要がある。今まではメモリのアドレス空間が狭かったため、8bit で直接指定することができた。すなわち、今までの方法は、8bit で表した X を直接 pc に入れてしまっていた。これを絶対指定:absolute(直接指定:direct) と呼ぶ。絶対指定は分かりやすいが、アドレス空間が広い場合は指定のビット数が長く成りすぎて現実的ではない。

そこで、多くのマシンで使われている方法がプログラムカウンタ (PC) 相対 (relative) 指定である。この方法では飛び先 X を直接 pc に入れるのではなく、現在の pc の値に加えた値を飛び先とする。もちろん X は符号拡張されるので、マイナス方向にも飛ぶことができる。ここで注意したいのは、多くの CPU では、分岐の起点は分岐命令ではなく、その次の命令になっている点である。なぜこのようになるのかは、後ほど説明する。すなわち、X を 0 とすると、分岐しないで次の命令がそのまま実行され、-1 とすると自分自身に分岐する。跳び越す命令数を数える場合，分岐命令の次の命令を起点にするようにしなければならない。

この方法では、分岐命令の飛べるアドレスの範囲は、-方向には 128、+方向には 127 に限定される。これでは困るようだが、一般的にはさほど不便ではない。これは、プログラムの動き方が極めて局所性 (Locality) が強いことから、飛び先は概ね近場に限られているからである。とはいえ、やはり、これだけでは困るので、後でもっと別のタイプの分岐命令を導入する。

前回の 2 番地の中身と 3 番地の中身を掛け算して 0 番地に格納するプログラムは以下のようになる。

```
LDIU r0, #2    01001 000 00000010
LD r1,(r0)      00000 001 000 01001
LDIU r0, #3    01001 000 00000011
LD r2,(r0)      00000 010 000 01001
```

```
LDIU r3, #0    01001 011 00000000
ADD r3,r1       00000 011 001 00110
ADDI r2, #-1    01100 010 11111111
BNZ r2,-3       10001 010 11111101
LDIU r0, #0    01001 000 00000000
ST r3,(r0)      00000 011 000 01000
BEZ r2,-1       10000 010 11111111
```

web から、poco1.v, poco_test.v, alu.v, rfile.v, def.h, dmem.dat, imem.dat をダウンロードし、同じディレクトリに置いて

```
iverilog -o poco_test poco_test.v poco1.v alu.v rfile.v
vvp poco_test
```

と実行し、掛け算の様子を確認しよう。

うんちく 1:古典的な分類（忘れても良い話）

昔、演算のためのスタックというのを持っていて、そこにデータをメモリから持ってきて、演算するスタックマシンというコンピュータがあった。この形は、演算がスタックのトップとその下で常に行われるため、加算などの一般的な演算で、オペランドの数が 0 になる。そこで、昔はコンピュータの命令セットをオペランドの数によって分類する方法が使われた。つまり、

- 0：スタックマシン
- 1:アキュムレータマシン
- 2,3:汎用（専用）レジスタマシン

この分類は、すっきりしていて大変良いのだが、80 年代にスタックマシンが RISC の高性能化についていけずに絶滅（バロース B5000, HP-9000 などの名機があった）し、アキュムレータマシンも汎用レジスタマシンへの道をたどったため、分類としてあまり意味をもたないようになった。

うんちく 2:命令セットアーキテクチャとは

いままで何となく紹介してきたコンピュータの命令セットアーキテクチャというのはソフトウェアつまりプログラムとハードウェアのインタフェースを役割をはたしている。つまり、コンパイラや OS の作者であるプログラマは、コンピュータの構造の詳細を理解する必要はなく、単に命令セットアーキテクチャだけを理解すれば良く、これに従って書いたプログラムは同じ命令セットアーキテクチャを持つ様々なマシンで走る。一方、ハードウェアの設計者は、価格と目的に応じて様々な構成を設計する必要が生じるが、同じ命令セットアーキテクチャに基づいて設計しさえすれば同じプログラムが動作することになる。この概念は 60 年代にメインフレームの名機 IBM360 の登場によって固められ、現在に至っている。このため、世の中には、IA-32,IA-64, SPARC, MIPS, ARM など様々な命令セットアーキテクチャがあり、それぞれの命令セットアーキテクチャに対して様々なマシンが存在する。例えば Intel の Pentium IV と AMD の Xeon は、同じ命令セットアーキテクチャに基づいているが、内部構造は全く異なる。

命令セットアーキテクチャによって、ソフトウェアとハードウェアをうまく切断することに成功したのがプログラム格納型計算機の発展のひとつの大きな原因である。このため、かつては命令セットアーキテクチャの設計こそコンピュータアーキテクトの華であった。しかし、最近は、半導体の面積の増加によって、ある命令セットを別の命令セットに変換して実行することがコスト的にさほど問題なくなった。このため、命令セットに凝ること自体の意味がなくなり、命令セットはソフトウェア互換性を重視してなかなか変わらなくなった。一方で、コンピュータアーキテクチャの主たる発明や興味は命令セットを実現する内部構造をどのように作るかに移っている。

うんちく 3: その他のアドレッシングモード

POCO はレジスタ間接指定しかアドレッシングモードを持っていない。さすがにこれは極端な例で、世の中には以下に示すようなアドレッシングモードが存在する。

- ディスプレースメント付きレジスタ間接指定：レジスタで指定したアドレスに命令コード中の定数（ディスプレースメントまたはオフセット）を加えた値が実効アドレスになる方法。下の例では r2 の値と 10 を足した番地が実効アドレスとなる。

  ```
  LD r1,10(r2)
  ```

- インデックス修飾付きレジスタ間接指定：レジスタで指定したアドレスにもう一つ別のレジスタ（インデックスレジスタ）の値を足した値が実効アドレスになる方法。下の例では r2+r3 が実効アドレスとなる。

  ```
  LD r1,(r2,r3)
  ```

- ポストインクリメント付きレジスタ間接指定：レジスタで指定したアドレスにアクセスした後、指定したレジスタの数値をアクセスしたデータに相当する分のアドレスを足してやる。下の例では、データのロード後に r2 に 2 が足される。

  ```
  LD r1,(r2)+
  ```

- プリデクリメント付きレジスタ間接指定：まずアクセスするデータに相当する分のアドレスを引き算してから、データにアクセスする。ポストインクリメント付きと合わせてスタックの実現に便利。下の例では、r2 から 2 を引いてからデータをロードする。

  ```
  LD r1,-(r2)
  ```

- スケールド：インデックス修飾同様、二つレジスタを指定するが、片方のレジスタが配列の添字の役割をし、これにデータのサイズ分のアドレスを掛けてからもう一つのレジスタと足してやる。配列のアクセスに便利。
- メモリ間接：メモリの中身を実効アドレスとしてさらにもう一度メモリをアクセスする。

多くの RISC は、ディスプレースメント付きレジスタ間接指定を中心にしてメモリをアクセスする。しかし、POCO はさらに簡単にするため、レジスタ間接指定だけにした。これでも通常の場合は十分である。

演習 6-1

0 番地,1 番地,2 番地にそれぞれ P,Q,R が格納されている。(Q-P) OR (Q+R) の演算を行い、3 番地に格納するプログラムを POCO で実行せよ。

演習 6-2

0 番地に格納されている自然数を X とし、X+(X-1)+(X-2)+(X-3)+...1 を演算するプログラムを POCO で実行せよ。

今回の範囲の POCO の基本命令

POCO はだれでも拡張、変更をしやすい構成なのが特徴である。これからどんどん拡張していく。

```
NOP                                          00000 --- --- 00000
MV  rd,rs    rd <- rs                        00000 ddd sss 00001
AND rd,rs    rd <- rd AND rs                 00000 ddd sss 00010
OR  rd,rs    rd <- rd OR rs                  00000 ddd sss 00011
SL  rd       rd <- rd<<1                     00000 ddd --- 00100
SR  rd       rd <- rd>>1                     00000 ddd --- 00101
ADD rd,rs    rd <- rd + rs                   00000 ddd sss 00110
SUB rd,rs    rd <- rd - rs                   00000 ddd sss 00111
ST rs, (ra)     rs -> (ra)                   00000 sss aaa 01000
LD rd, (ra)     rd <- (ra)                   00000 ddd aaa 01001
LDI rd,#X       rd <- X（符号拡張）          01000 ddd XXXXXXXX
LDIU rd,#X      rd <- X（符号拡張なし）      01001 ddd XXXXXXXX
ADDI rd,#X      rd <- rd + X（符号拡張）     01100 ddd XXXXXXXX
ADDIU rd,#X     rd <- rd + X（符号拡張なし） 01101 ddd XXXXXXXX
LDHI rd,#X      rd <- X|0                    01010 ddd XXXXXXXX
BEZ rd, X   if (rd==0) pc <- pc + X          10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X          10001 ddd XXXXXXXX
```