■目次（ほげが今回の範囲）

第 7 回の授業では，計算に使うデータだけでなく，CPU に対する機械語命令も 0/1 のならびとしてメモリに格納されていることを知った．また，「プログラムカウンタ」などの仕組みによって CPU が命令を 1 つずつ実行していく手順を学んだ．さらに，演算結果に応じて命令実行の流れを切り替える「条件分岐」を実現する仕組みについても考えた．第 2 部最終回である今回は，機械語をテーマとする．まずはじめに，0/1 をならべて本当に機械語命令を作れそうなことを確かめる．次に，機械語と C 言語などのプログラミング言語との関係について学ぶ．コンピュータ，特に CPU が，全てを 0/1 で表しその計算によって様々な情報処理を行っていることを理解し，C 言語でプログラムを書くことで CPU を自在に操れる喜び (?) を感じることが目標である．

★ 6　CPU と機械語

★ 6.1　機械語命令の例 – がんばれほげおくん ver.4 –

0/1 をならべたビットパターンで機械語命令を構成できることを確かめるために，CPU「ほげおくん」の例で考えてみよう．そのために，前回資料の ver.3 よりも詳細な設定をする．

実行制御装置　機械語命令を解読し，ALU 等に指示を出して命令を実行する．

ALU　指定された 2 つの汎用レジスタの内容の間で演算を行い，結果を指定された汎用レジスタに書き込む．ここでは，12 通りの演算を実行できるとする（符号あり／なし整数の四則演算 8 通り (☆ 1) と論理演算 4 通り (☆ 2)）．また，演算を実行する度にその結果を表す情報 (☆ 3) を専用のレジスタ（フラグレジスタ）に自動的に書き込む．

汎用レジスタ　A,B,C,D の 4 つあり，それぞれ 16 ビットのビットパターンひとつを保持できる．

命令レジスタ　実行中の機械語命令ひとつ分のビットパターンを保持できる．

プログラムカウンタ　次の命令を読み込むメモリ番地を保持する．

また，この CPU を搭載するコンピュータは次のような条件で動作すると考える．

- メモリが接続されている．そのひとつの区画には 16 ビットのビットパターンを格納できる．それらの区画毎に 16 ビットで番地が割り振られている．
- 扱えるデータは符号あり／なしの整数のみとする（符号小数点数等は扱えない）．いずれも 16 ビットで表し，符号あり整数には 2 の補数表現を用いる．

以上からわかるように，このコンピュータでは，汎用レジスタ，ALU，メモリはどれも，16 ビットのビットパターンをひとまとまりとして扱える．

☆ 1) 符号あり整数の加減乗除で 4 通り + 符号なし整数の加減乗除で 4 通り．

☆ 2) 例えば，2 つのビットパターンの間でビット毎の AND,OR,XOR をとる演算と，1 つのビットパターンのビット毎の NOT をとる演算．

☆ 3) 例えば，値が正／0／負だった，オーバーフローした等

Q1. この CPU の汎用レジスタをビットパターンによって区別するとしたら，最小何ビット必要か．また，ALU への入力 2 つと出力先 1 つの計 3 つのレジスタをビットパターンで指定するとしたら，最小何ビット必要か．

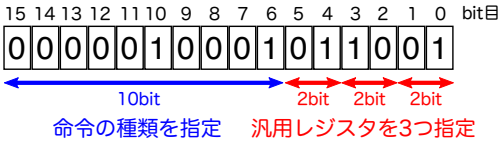
Q2. この CPU の ALU が行う演算の種類をビットパターンで指定するとしたら，最小何ビット必要か．

Q3. 符号ありの整数を 16 ビットの 2 の補数表現で表す場合，表せる最大と最小の数は 10 進数でそれぞれいくつか．

Q4. このコンピュータのメモリは，最大でいくつの区画に分けられるか（つまり，16 ビットのビットパターンを最大で何個格納できることになるか）．

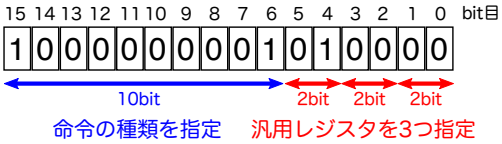
上述のような設定を考えると，例えば 16 ビットのビットパターンを次のように使って機械語命令を作ることができるだろう．

例 1. ALU を使った整数演算の命令などは，16 ビットを右図のように分けて，10 ビットを使って命令の種類を表す．残り 6 ビットは，2 ビットずつに分けて，ALU への入力 2 つと出力先の計 3 つの汎用レジスタを指定する．



命令の種類は，例えば符号あり整数の加算なら 0000010000，同じく減算なら 0000010001, ... のように CPU 設計時に決めておく．汎用レジスタも同様で，例えば A,B,C,D にそれぞれ 00,01,10,11 を割り当てればよい．

例 2. 番地の指定が必要な命令では，番地を表すだけで 16 ビット必要なので，右図のように 16 ビットのビットパターンを 2 つ使って命令を表現する．この例では，最初の 16 ビットの使い方は 1. と同じとしている．



命令の種類を表すビットパターンなどの決め方は 1. と同様にすればよい．ただし，この命令が 16 × 2 ビットでできていることを実行制御装置がわかるように，最初の 10 ビットは 1. と重複しないビットパターンとする必要がある．例えば 1000000000 はメモリからレジスタへの情報の転送，1000000001 は逆にレジスタからメモリへの転送を表す命令とする（これらの命令ではレジスタを 1 つだけ指定するので，例えば 6 ビットのうち上位の 2 ビットだけを使い残りは無視すればよい）．

これらはあくまで説明のための例で，実際の CPU の機械語がこの通りの 0/1 の並びでできているわけではない．また，一般的な CPU で使われるような命令の種類を網羅して説明しているわけでもない．しかし，ここに示したような考え方で機械語命令を作れば，0/1 の並びによって CPU に様々な指示を与えて動作させられそうだ，ということがわかるだろう．

Q5. 「ほげおくん」の機械語命令が上記の例のように作られていたとする．このとき，(1) の図に示されたビットパターンは，「レジスタ [?] とレジスタ [?] を入力として [?] を行い，結果をレジスタ [?] に書き込め」という命令を表すことになる．[?] に入るものを答えなさい．

Q6. 上記と同様に，(2) の図に示されたビットパターンは，「レジスタ ? の内容をメモリの ? 番地に書き込め」という命令を表すことになる．? に入るものを答えなさい．

★ 6.2 機械語とアセンブリ言語

★ 6.2.1 CPU のつくりが違えば機械語も違う

★ 6.1 の例からもわかるように，機械語命令は，ALU にできる演算の種類や内蔵している汎用レジスタの数などに応じて作っておく必要がある．このように，機械語は CPU のつくり（「アーキテクチャ」という）と密接に結びついているので，**CPU のつくりが異なれば機械語も異なる**ものとなる．したがって，2 台のコンピュータで搭載している CPU のつくりが異なっていたとすると，一方で実行できる機械語プログラムを他方に移しても動作しない (☆ 4)．

☆ 4) 同じ機械語命令が使えるかどうかの観点で見た CPU のつくりを「命令セットアーキテクチャ」という．例えばインテル社の Core, Core 2, Core i7 等は (ほぼ) 同じ命令セットアーキテクチャの CPU であり (後者ほど新しい)，Core で動く機械語プログラムは Core i7 でも動く（新しい CPU では新しい命令が追加されていることがあるので，逆のことは可能とは限らない)．

★ 6.2.2 アセンブリ言語

以下の左に示すのは，★ 6.1 で説明した機械語命令の作り方に従って，右の内容欄に示す 4 つの命令（前回資料に登場したものと同じ）を実際に機械語として書いてみたものである (☆ 5)．このように，0/1 がたくさんならぶ機械語プログラムは，人間には非常にわかりにくい．作るのも，バグ取り（デバグ）するのも，読むのも大変である．そこで，少しでも人間が楽にできるようにと，**アセンブリ言語** (☆ 6) というものが考えだされた．以下の真ん中の列がその例である．

☆ 5) 前述のように 16 ビット 2 つで 1 命令のものもあるので，右の欄には空行を入れて対応付けしやすくしてある．

機械語	アセンブリ言語		内容
1000000000000000	LD	rA,X	**メモリの番地 4649 の内容をレジスタ A にコピーする**
0001001000101001			
1000000000010000	LD	rB,Y	**メモリの番地 4650 の内容をレジスタ B にコピーする**
0001001000101010			
0000010000000100	ADDA	rA,rB,rA	**レジスタ A と B の内容を符号付き整数として加算，結果を A に書き込む**
1000000001000000	ST	rA,Z	**レジスタ A の内容をメモリの番地 5963 に書き込む**
0001011101001011			

上の例に示すように，アセンブリ言語では文字を使ってプログラムを書く．1 行 1 行が機械語命令とほぼ一対一に対応しており，文字列を用いて命令の種類 (LD,ADDA など）や汎用レジスタ（rA,rB など）を表せる．さらに，番地も記号に置き換え（X,Y など）て指定することができ，プログラミングの労力をかなり軽減できる．ただし，アセンブリ言語の命令は機械語の命令を一つずつ置き換えただけのものであるから，機械語と同様，アセンブリ言語も CPU のつくり毎に異なるものとなる．

☆ 6) assemble（集める，組み立てる）という英語から．アセンブリ言語: assembly language，アセンブラ: assembler.

アセンブリ言語を使う場合，人間が書いたアセンブリ言語プログラムを，コンピュータ自身に機械語に翻訳させる (☆ 7)．この翻訳を行う機械語プログラムを**アセンブラ**という．

☆ 7) このような処理ができるのも，命令をメモリに格納するプログラム内蔵方式だからこそである．

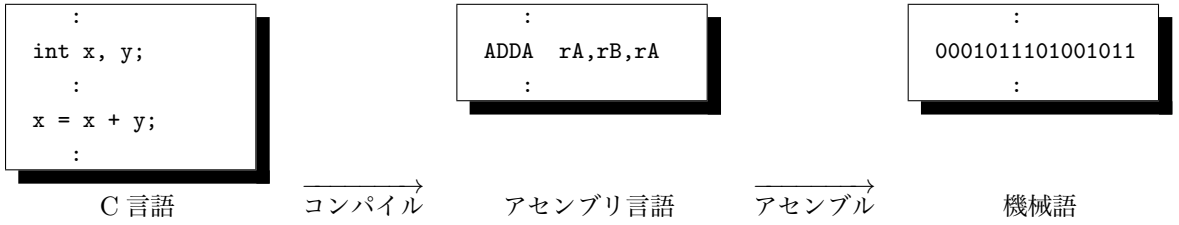
★ 6.3 高水準言語と低水準言語

機械語やアセンブリ言語は CPU のつくりと密接に結びついているので，プログラムを書くためには，対象とする CPU のつくりをよく理解してないといけない．また，つくりの異なる CPU を搭載した複数のコンピュータでプログラムを動かしたければ，それぞれの CPU 用に異なる機械語・アセンブリ言語でプログラムを書かないといけない．これらはとても難しくかつ手間のかかることである．このような困難を克服し，より簡単にプログラミングができるようにと考え出されたのが，**「人間がプログラミングする際にはアセンブリ言語よりもっとわかりやすい形で書き，それをコンピュータに翻訳させる」**という方法である．C 言語は，このような方法のために作り出されたプログラミング言語の一つである (☆ 8)．

C 言語のようなプログラミング言語は，**高水準言語**と呼ばれる．これと対比させて，機械語やアセンブリ言語は**低水準言語**と呼ばれる．高水準言語を用いる場合，人間が書いたプログラム（**ソースプログラム**という）はコンピュータに理解できる形式ではないので，アセンブリ言語／機械語プログラムに翻訳する処理が必要となる．例えば，龍大計算機室の Linux 環境においては，C 言語ソースプログラムを hoge.c という名前のファイルに書いたとすると，

```
$  cc  hoge.c
```

のように cc コマンドを実行して a.out というファイルを作り，この a.out を実行するのだった．この a.out の中身は，hoge.c に書かれた C 言語プログラムを翻訳してできた（ほぼ）機械語プログラムであり，cc が下図のような翻訳作業を行っている．この C 言語の例のような，ソースプログラムをアセンブリ言語／機械語プログラムに翻訳する処理のことを**コンパイル** (☆ 9) といい，その処理を行うプログラムを**コンパイラ**という．高水準言語の中には，ここで説明した「ソースをコンパイルしてできた機械語を実行」という手順とは異なる実行手順をふむものもある (☆ 10)．



高水準言語の場合，言語の設計を CPU のつくりと切り離すことができる．そのため，多くの高水準言語では，様々なコンピュータ用のプログラムを一つのソースプログラムで書くことができる (☆ 11)．プログラミングが容易なだけでなくこのような利点もあるため，現代のプログラミングの場面では高水準言語を使うことが多い．しかし，低水準言語の方が向いている場面（CPU の性能を最大限引き出したい等）もあるので，両者は使い分けられている．

☆ 8) このようなプログラミング言語は，C 言語の他にも FORTRAN, Pascal, Java, Python, Swift 等たくさんある．

☆ 9) compile（(資料を）集める，（書物を）編集する）という英語から．図では C 言語からアセンブリ言語への翻訳作業を「コンパイル」としているが，アセンブル作業も含めて「コンパイル」と言うこともある．コンパイラ: compiler.

☆ 10) あらかじめソースを機械語に翻訳しておいたりせず実行中に逐一翻訳していくなど，様々な方式のものがある．

☆ 11) P,Q というアーキテクチャの異なる 2 つの CPU があった場合，同じソースを P 用 Q 用 2 つのコンパイラで処理し，P で動く機械語プログラムと Q で動く機械語プログラムを別個に作ればよい．