体験版

私の卒研に付き合ってくれる優しきテスターの皆様用の簡易版です。

初めに、ファイルに これ、説明書、命令一覧、実行ファイル（2種類）、アンケート先URL が揃っていることを確認してください。
解答冊子は、本文中に出てくる「問題」の答えが書いてあります。
SourseCodes フォルダは、アセンブリ言語のソースコードがまとめてあります。多分使わない。

これの 1章 を行った後、説明書（必要であれば命令一覧も）、これの 3章 の順番で見ると分かりやすいかもです。

そして、体験版を最後（3.3節は、やってもやらなくてもOK）まで行うのに要した時間を知りたいので、お手数ですが **ストップウォッチ** をご準備ください。スマホやPCの機能でかまいません。

コンピュータのお話（本編 1章）

本章では、組み込み機器におけるコンピュータを理解する上で必要な知識を纏める。要は座学パートだ。
結構内容が多いので頑張ってください（他人事）

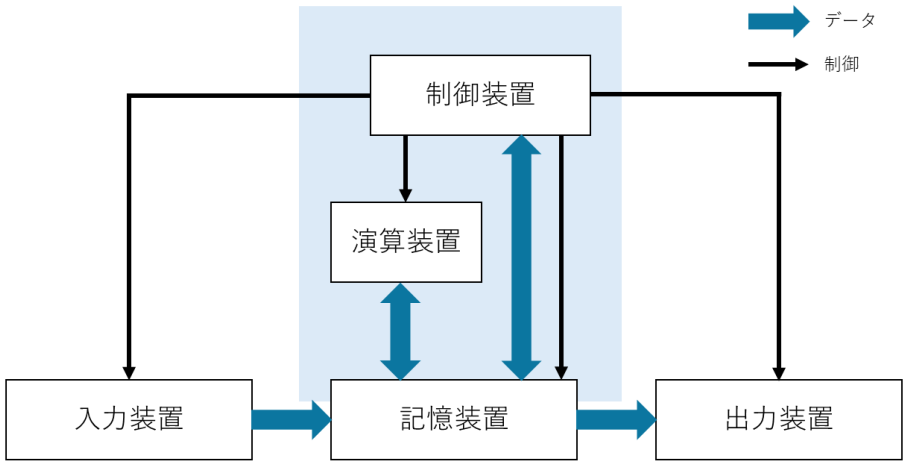
1.1 コンピュータの構成要素

1.1.1 コンピュータの大まかな構成

早速だが、「コンピュータ」と聞いて何が思い浮かぶだろうか。
「パソコン」、「計算機」、「量子コンピュータ」など、色々考えられよう。
実は、Computer は Compute(計算する) -er (接尾辞「~な人」) であるから、元は 計算をする人 や 機械 を指していた。
では、そんな「コンピュータ」は、どのような部分から構成されているだろう。
「パソコン（パーソナルコンピュータ）」（デスクトップPCやノートPCなど何でも良い）を基に考えてみると、まず画面を映す「液晶ディスプレイ」、入力する「キーボード」がある。さらに、データを保存するHDDやSSDなど「ストレージ」もあるだろう。
では、あとは内部に何があるだろうか。
まず、画面描画やキーボード入力、データ保存などを実際に行うために、各部分に命令を送る「制御機構」があるだろう。
そして、Computerの名の通り、計算を行う「演算機構」も備えられている。電卓とかイメージすると良い？
これらを踏まえてコンピュータの構成要素を纏めると、以下のようになる。

- 制御装置 : 各部に制御指令を送る
- 演算装置 : 計算を行う
- 記憶装置 : データを保存する
- 入力装置 : キーボードなど入力する
- 出力装置 : 画面表示など出力する

これら5つの装置を **コンピュータの5大要素** と呼び、概念図にすると次のようになる。

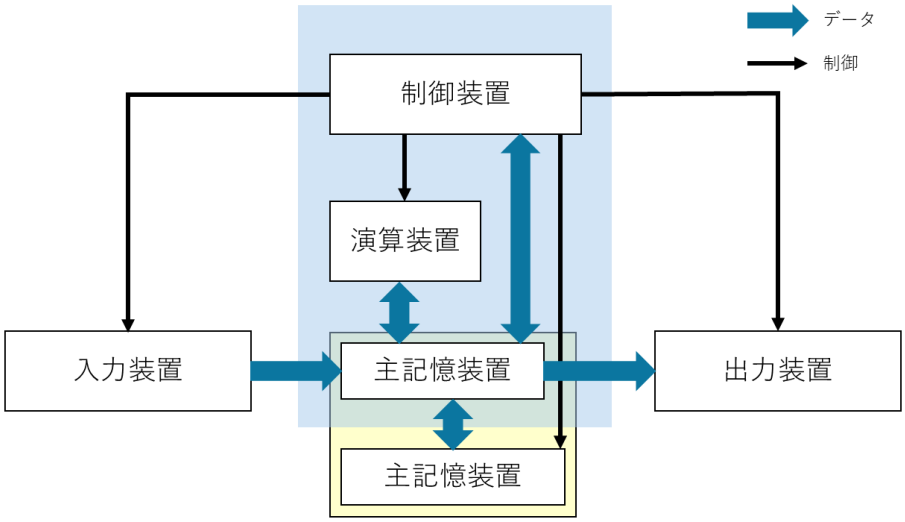


さて、コンピュータの構成が分かったところで、先に挙げた図を確認したい。
まず、入力装置から入力を受け取ると、入力したデータは記憶装置に入る。そして、その入力した文字だったりを出力装置に表示する。
記憶装置は、入力データの他にも、演算結果を保存したり、制御装置が「これを記録しろ」とデータを送りつけてくることもある。
制御装置は、記憶装置からデータを呼び出して、「入ってきた値とこれを足す」や、「画面に出力しろ」といった制御を行ったりする。
よって、演算装置、制御装置と記憶装置は相互にデータをやり取りする。
図において背景色がついている部分は、パソコンなどにおける 内部 に相当する。

と、ここで、勘のいい人は違和感に気付くだろう。図をよく見ると、記憶装置の上半分も色づいているのだ。

実は、コンピュータの内部には 制御装置と演算装置の他に、記憶装置が入っている。
この記憶装置を、HDDやSSDといった「ストレージ」と分けるために、「メモリ」や「主記憶装置」と呼ぶ。
主記憶に対して、外付けのストレージを「補助記憶装置」と呼ぶこともある。

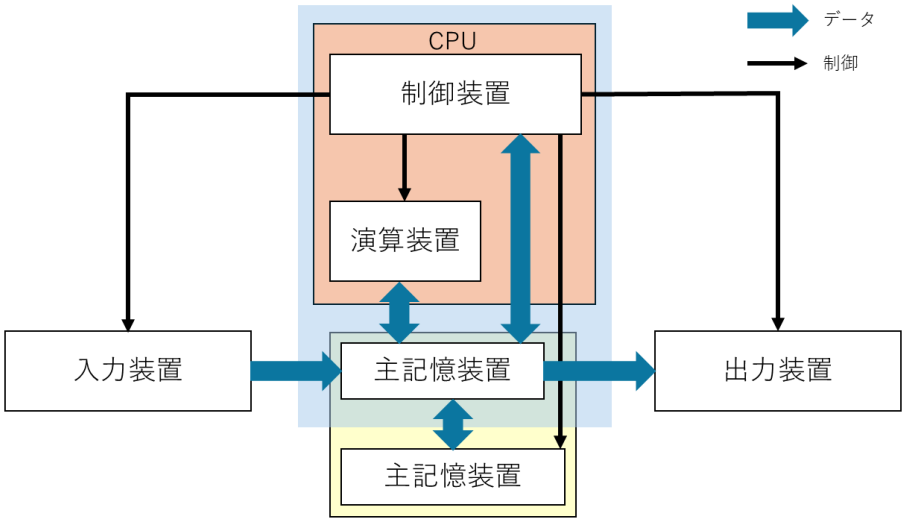
ということで、構成をもう少し正確に書くと、次のようになる。



1.1.2 CPUとは

コンピュータの内部に、具体的に何をする構成要素があるかは分かった。では、実際のところ、物理的には何が入っているだろうか。

世の中には、たくさんの回路が入った「回路の集合体」（集積回路; IC）とした部品が存在する。
このICは、板に張り付けられ ICチップ として、今日では様々な機器に組み込まれている。
コンピュータにもそんな回路の集合体があり、 **CPU** と呼ぶ。中央演算処理装置（**C**entral **P**rocessing **U**nit）の略称であり、名前の通り 演算 と 処理 を行う。構成要素でいうところの、制御装置と演算装置が合体したものだ。
制御装置を積んだ 司令塔 であるCPUは、コンピュータの頭脳などと呼ばれることもある。



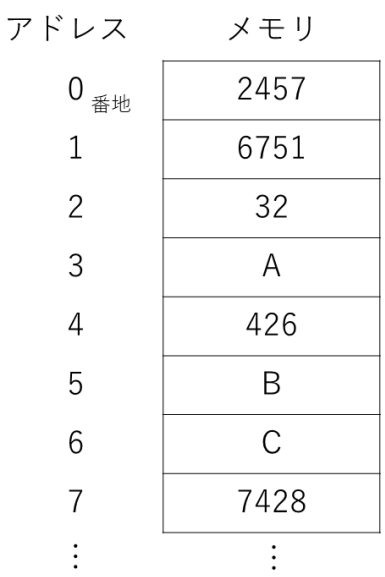
1.1.3 メモリとは

メモリとは、データの保存場所である。データを保存するといえば、HDDやSSD、USBメモリやSDカードなどを思い浮かべるかもしれないけれど、それとはまた別物である。

メモリには、コンピュータを制御するための手順が入っていたり、一時的なデータの保存をするために使われたりする。
手順はコンピュータの中で常に同じであるから、内蔵してしまえばいいと納得できるだろう。勝手に改変されても困るので、読み込みだけできるメモリ、 **ROM（Read Only Memory）** と呼ぶ。
また、一時データは、右クリックや Ctrl+C でコピーを行った内容であったり、保存を行っていないメモ帳の落書きなどが置かれる。
このデータは電源が入っている間しか生きられない一時的なものであるから、パソコンの電源を一度落としてしまえば、「電源落とす前にコピーしたデータ貼り付けたいな」と思っても、既に消えてしまっており、出来ないのである。
こちらのメモリは、蒸発するみたいに消えるから **揮発性メモリ**、あるいは読み書きを好きな場所に出来るから **RAM（Random Access Memory）** と呼ばれる。

そんな一時的なメモリに作業中の進捗を保存しておき、「保存ボタンを押す」などをトリガーに、内容をストレージへまとめて一気に保存する。
なんですぐストレージに書き込まないのかというと、単純に時間がかかるからだ。ストレージは容量が大きい分、空きを見つけたりデータを探すのに時間がかかる（メモリと比べて1000倍以上の時間を要する）。
「何かするたびに処理がカクカクする」となっては困るので、容量が小さい代わりに比較的高速な メモリに、細かい変更を貯めこんでおくのだ。

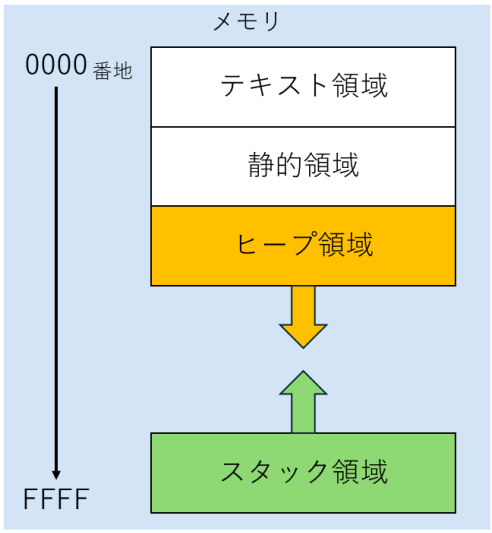
メモリのイメージとしては、先頭から連番のついた、更衣室や駅のロッカーのような、縦並びの箱である。
この箱の一つ一つに、データや命令を詰め込める。
「10番のロッカーに入った物を取り出したい」「30番にこのデータ入れといて」といった形で、番号を指定してロッカーを扱う。
この、ロッカー番号を **アドレス** や **番地** と呼ぶ。つまり住所ですね。
番地は 0番 から始まる。先頭は1番目ではなく0番目であることに注意したい。
下の図だと、「0番地に2457が入ってる」、「1番地に6751が格納されている」といった表現がされる。
ロッカーの大きさは機械によって異なるが、ロッカー自体の大きさと、ロッカー自体の数がメモリの容量に直結する。



また、メモリには4つの領域が存在し、それぞれ役割が異なる。以下に、それぞれの名称と役割を示す。

- **テキスト領域** : 機械語が記されている。これを上から順番に読んで、実行していく。
- **静的領域** : プログラム作成時に宣言されている、初期値がある変数に割り当てる領域。「この箱にはこれを入れるよ」が決まっている。
- **ヒープ領域** : 「大きめの保存領域が欲しいよ」みたいなときに、動的に割り当てられる領域。出現した順に置かれていく。
- **スタック領域** : プログラム作成時から宣言されているものではなく、「ここでしか使わない」変数などに対する領域。メモリの一番後ろから、前の方に上ってくる。

難しい用語が色々出てきたので、簡単に話します。
まず、「これをしてね」「次はこれを動かしてね」といった処理を次々と書いた場所があります。これを上から順番に読んで、その通りに動かすことで、コンピュータは動作します。
次に、「この動作するために絶対必要なデータ」を保存する、動かしてる途中で大きさが変わったりしない (**静的**) データを保存する場所があります。
その下に、途中で大きさが変わったり (**動的**) 、急に大きい保存場所が欲しくなった時に使う場所があります。「名簿表使いたいから、中のデータ入れられるだけのまとまった場所をくれ」「やっぱこのデータも入れるから領域大きくして」みたいな、動的に変動する、まとまった場所を確保する場所です。
最後に、「この処理するときだけ使う、ちょっとしたデータ」とか、「あっ、ちょっとこのタスク先にやってくれる？」みたいに命令の順番が変わったときの、「元々やってた仕事の進捗具合」、戻り先（本の しおり みたいな）とかが保存される場所があります。これはちょっと特殊で、一番後ろから前に進んでいきます。下から上に積み上げていくので、「スタックを 積む 」と表現したりします。



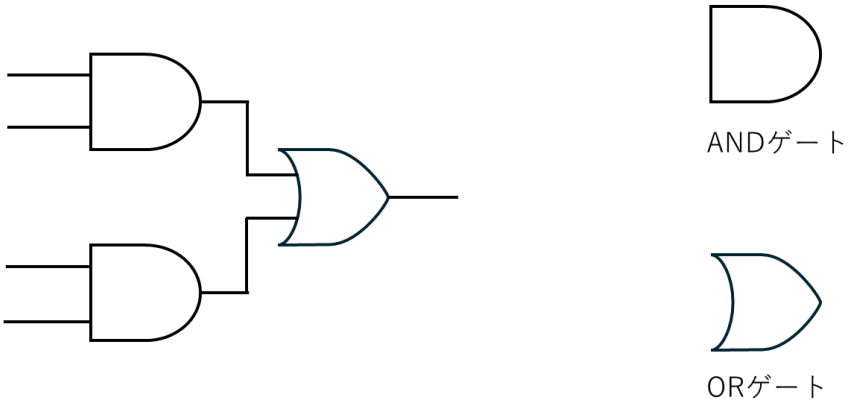
問題

(1) メモリにおいて、データを格納する一つ一つの場所を指す番号を何というか。
(2) 前のページにあるメモリの図について、A が格納されている (1) は、いくつか。

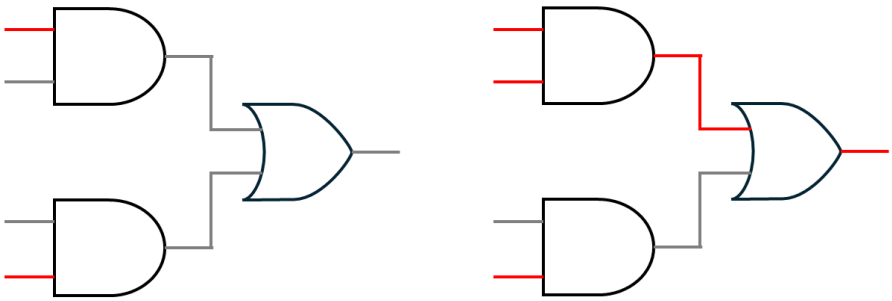
1.2 機械語とアセンブリ言語

1.2.1 コンピュータの内部表現と機械語、アセンブリ言語

コンピュータは電気で動く電子機器である。つまり、「電気が流れている」「電気が流れていない」の2状態しか扱うことが出来ない。
制御装置は、コンピュータの各部分に「電気を流す」「電気を流さない」を制御することで、「今は足し算をしてくれ」「いまはデータ読み込んでくれ」と、行うことを切り替えることで望んだ操作が出来るようにする。
「ここととここに電気を流せば、この回路が動くから足し算が出来る」みたいな感じで、複雑にオンオフを切り替えるのだ。



上の図では、4本の線が左側から来ており、中央にはよくわからない図形、右側に1本の線が伸びている。
線は電気を流す銅線であり、左側から電気を流して、中央で処理して、最終的に電気を右に流すか、流さないかを決定する。
中央左側にあるかまぼこみたいなやつは ANDゲート と呼び 、「全部の線からの入力がON (電気が流れている) とき、電気を流してあげるよ！」といった処理をする。
また、中央右側にあるかまぼこだけど先がとがってるやつは ORゲート と呼び、「入力がどれか一つでもONなら、電気を流してあげるよ！」といった処理をする。
この状況において、左側のどこに電気を流すか、2パターン考えてみよう。



左の図では、一番上と一番下に電気を流した。この場合、どちらも ANDゲート でOFFが出力されるから、最終的に電気は流れない。
右の図では、さらに上から2番目に電気を流した。この場合、上の ANDゲート でONが出力されるから、次の ORゲート もONとなり、最終的に電気が流れる。

このように、コンピュータは電気の流し方で、「どの回路に電気が流れて動作するか」を制御する。
これを行い、どこに電気を流すかを決定する回路が 制御装置 であり、「どこに電気を流すか」の指示書が ROM に記されている。
ROM では、各電線に電気を流すか、流さないかを記録しておくので、流す→1, 流さない→0 と表現すると都合がいい。電線の順番を決めて、01の羅列を記すだけで手順書になる。
そのため、コンピュータの内部では、データは全て 1 と 0 の2種類を組み合わせて表現される。

1 と 0 の組み合わせにより記された、制御を行うための手順を **機械語** と呼ぶ。機械が理解できる言語なので機械語だ。しかし、これでは人間に理解できない。手順を作ってデータとして組み込むのは人間なのに、肝心の人間が理解できなければ意味がない。
そこで、機械語と一対一で対応した、ちょっと人間向きの言語を作ることにした。これを **アセンブリ言語** と呼ぶ。jump や add など、英単語（の略）をキーワードに使うことで、ちょっとだけ読みやすくしている。
本教材では、このアセンブリ言語を学ぶことで、コンピュータへの理解を高めたい。

また、用語として、命令を書き記した機械語の「手順書」のことを **プログラム** と呼ぶ。
プログラムを作るために頑張ることを、最近流行りの **プログラミング** と呼んで、その時に使う言語を **プログラミング言語** と呼ぶ。アセンブリ言語もプログラミング言語の一種ですね。
プログラミング言語にも色々と種類があって、アセンブリ言語の他にも、C言語、Pythonなど挙げればキリがない。この中で、「人間にとって分かりやすい」言語を 高級言語 、「機械の動きに忠実な、機械語に近い」言語を 低級言語 と言ったり。
また、プログラミング言語 で書いた、人間向きの命令群を **ソースコード** や、単に **コード** と呼ぶ。

1.2.2 ビットとデータ表現

前節で出てきた、「1 と 0 の組み合わせ」によるデータの表現を、**2進数** や **ビット列** と呼ぶ。
0 と 1 で構成された文字列の、一つ一つを **ビット（bit）** と呼び、そんなビットが列になっているから、ビット列である。
そして、8個のビットでまとめた単位を **バイト（Byte）** と呼ぶ。10100000 や 00010001 などを「8ビット」や「1バイト」、11110000 10110011 を「16ビット」や「2バイト」と数える。
ストレージの容量なんかで、「ギガバイト GB」やら「2TB」などといった表現がされるが、その「バイト B」である。
キロメートルkm の「キロ」が 1000 を表すように、1KB は 1000Bということである。そんな KB が 1000個あって、1000KB = 1MB （メガバイト）になる。同様に、1000MB = 1GB、1000GB = 1TBである。つまり1TBは1兆バイトです。デカすぎ。
ちなみに、1000 区切りではなく 2^{10} (1024) を基準にすることもある。1024B = 1KB といったように。
この場合、上の単位系と混同しないように、KBではなく KiB（キ**ビ**バイト）、MBではなく MiB（メ**ビ**バイト）など、間に i を入れた書き方をすることもある。

さて、実際のところ、そんなビット列でデータをどのように表現しているのだろう。
これは非常に単純なもので、「データの一つ一つに対応する番号を割り振る」のだ。
アルファベットを書きたいなら、a から z まで26文字あるから、00000 から 00001, 00010, ..., 11010 までを対応させる。
対象のコンピュータ、空間の中で統一された対応表があれば、一対一で対応しているから一意に復元できる。小学生の時に経験があるかも、友達間で使う秘密の暗号文字みたいな感じです。

そういうことで、世界的に統一された「文字とビット列の対応表」が存在する。
文字コード と呼ばれ、「この文字はこのビット列だよ」が取り決められている。

文字コードにも unicode や Shift-JIS, ISO-2022-JP などたくさんの種類があり、
文字コードが違うと復元結果が変わるので、文字化け が発生したりするのだが……。
例えば、
unicode だと「あ」は `11100011 10000001 10000010` というビット列になる。
Shift-JIS だと「あ」は `10000010 10100000` というビット列になり、全然違う。
Shift-JIS だと思って `11100011 10000001 10000010`（unicodeの「あ」）を読もうとすると「縺」になってしまう。
そのため、「unicode でビット列に変換したものを Shift-JIS と思って元に戻す」など、文字コードを間違えて変換を行うと、文字化けする。

次のページに、今回使用する文字コード「asciiコード」を紹介する。

行＼列	2	3	4	5	6	7
0	間隔	0	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
10	*	:	J	Z	j	z
11	+	;	K	[	k	{
12	,	<	L	\	l	|
13	-	=	M	]	m	}
14	.	>	N	^	n	~
15	/	?	O	_	o	削除

表の見方は、列→行 の順番で数字を当てはめる。例えば、
20 が 「間隔」（半角スペース）に対応する。
41 が 大文字の A に対応する。

といったようになる。
なぜ 0 ~ 15 なのかは、次節の 16進数 が関係する。

1.2.3 2進数と10進数、16進数

私たちが普段使用している数字は、10進数 と呼ばれる。0 ~ 9 の数字を使い、10を基準に桁が上がるから、**10** で桁が **進** む **数** である。
それに対し、先で話しているビット列は、0 と 1 しかないので、2を基準に桁を上げる **2進数** である。0, 1, と来て、2 は使えないので繰り上がって 10。その次は 11。次は 12 にはできないので繰り上がって 20、さらに繰り上がって 100。というように2進数は数えられる。0 は 0、1 は 1、2 は 10、3 は 11、4 は 100, ...

2進数と10進数は相互に変換が可能であり、2進数から10進数は以下のように求められる。

各位の数字が a_i である二進数について（1011 なら $a_1 = 1, a_2 = 0, a_3 = 1, a_4 = 1$ ）

$$
\begin{aligned}
a_1 a_2 \cdots a_n &= \sum_{i=1}^{n} a_i \times 2^{n-i} \\
&= a_1 \times 2^{n-1} + a_2 \times 2^{n-2} + \cdots + a_{n-1} \times 2^1 + a_n \times 2^0
\end{aligned}
$$

数式が出てきて拒否反応が表れた方もいると思うので、簡単に10進数で話をします。

まず、1024という数字は、一の位や十の位など、それぞれの位の数字と、基準となる10を使って、次のように表現できます。

$$
\begin{aligned}
1024 &= 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 \\
&= 1 \times 1000 + 0 \times 100 + 2 \times 10 + 4 \times 1
\end{aligned}
$$

お金のイメージです。1000円が1枚と、100円が0枚と、10円が2枚と、1円が4枚 みたいに、
桁ごとに「基準（を何回か掛け合わせたもの；累乗）が何個あるか」を見ます。

同様に、917235は、以下のようになります。

$$
\begin{aligned}
917235 &= 9 \times 10^5 + 1 \times 10^4 + 7 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 \\
&= 9 \times 100000 + 1 \times 10000 + 7 \times 1000 + 3 \times 100 + 2 \times 10 + 5 \times 1
\end{aligned}
$$

このように、数字は「それぞれの位の数と、基準の累乗」の組み合わせで表現できます。

これを使って、2進数についても同じように考えてあげると、基準は 10 ではなく 2 になるから、

$$
\begin{aligned}
1011 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
&= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\
&= 11
\end{aligned}
$$

$$
\begin{aligned}
100110 &= 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
&= 32 + 4 + 2 \\
&= 38
\end{aligned}
$$

というように、2進数から10進数が得られます。
なので、「各位の数値について、2 の (桁目-1)乗 したものを足す」ことで、2進数から10進数に変換できるというわけです。

また、逆向きには、「2の乗数の組み合わせに分解する」ことで、表現が得られる。

$$
\begin{aligned}
495 &= 256 + 1284 + 32 + 8 + 4 + 2 + 1 \\
&= 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
&= 11101111
\end{aligned}
$$

しかし、これでは、同じ数を表すにも、2進数と10進数で全然違う値になってしまう。関係性が見えづらくて、計算しなければ変換しづらいのだ。
そこで、2進数を基準に、「2進数の4桁を対応させた新しい数字」を作る。これは、2進数5桁目で桁上がりするから、2^4 = 16が基準な **16進数** と呼ぶ。
とはいっても、私たちの知っている一桁の数字は 0 ~ 9 までの10個しかない。
なので、16進数では、 0 ~ 9, A, B, C, D, E, F で値を表現する。
10進数で言う10をA, 11をBといったように対応している。
なお、16進数であることを分かりやすく書くために、先頭に `0x` や `#` を書くことがある。 `0x21` とか `#1BF6` とか。
この16進数を使って、2進数を変換すると、

1010 0110 → 0xA6
0011 0101 → #35

といったように、4桁ずつそのまま値が対応する。
16進数から2進数は特に「35だから 3 → 0011, 5 → 0101 で 00110101だな」といったように、すぐに変換が出来る。計算が要らない。
そのため、機械語などコンピュータの中身を表現する際は、10進数より「内部は2進数、人間が読むときは16進数」といった棲み分けをすることが多い。

ここの内容はちょっと難しいので、例題と問題を用意しておきます。
10進数と 2進数・16進数の相互変換に慣れてみましょう。

123 を 8桁の2進数 に直しなさい。また、これを16進数に直しなさい。
ただし、$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16,\ 2^5 = 32,\ 2^6 = 64,\ 2^7 = 128$　である。

解答
まず、2の累乗の組み合わせに分解する。
大きい数から考えると、まず 64 が使えそう。
123 から 64 を引くと、59 になるから、残り 59 を同様に 2の累乗 で表したい。
59 には 32 が使えそうだ。残りは 59 - 32 で 27 になる。
27 には 16 が使えそうだ。残りは 27 - 16 で 11 になる。
11 には 8 が使えそうだ。残りは 11 - 8 で 3 になる。
3 には 2 が使えそうだ。残りは 3 - 2 で 1 になる。
1 には 1 を使える。残りが 0 になったので、
123 は 64 + 32 + 16 + 8 + 2 + 1 で表せることが分かる。
これを累乗に直して、$123 = 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0$　となる。
空いている 2^2 と、8桁にするので 2^7 のところには 0 が入る。
$123 = 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
よって、二進数に直すと `01111011` となる。

また、これを16進数に直すには、4桁ずつ対応を考えるので、
`0111` は `7`、 `1011` は `B` より `0x7B` となる。

問題
(1) 173 を 8桁の2進数と、16進数に直しなさい
(2) `01001011` を 10進数に直しなさい

1.2.4 2進数と負の数（二の補数表現）

さて、コンピュータ上で正の数を表現する方法はわかった。では、負の数はどのように表現しよう。
コンピュータには 0 と 1 しかないから、「マイナス」を意味する記号は存在しない。
そこで、「一番左のビット（最上位ビット）」を **符号ビット** とし、「ここが 0 なら正の数、1 なら負の数」といった解釈を行う方法を考えてみよう。
たとえば、1 は 0001 なので -1 は 1001、2 は 0010 なので -2 は 1010、といった感じである。
しかし、これでは足し算を行うときに不都合が生じる。
例えば、$1+(-1)$ は 0 なので 0000 になってほしいが、上の方法では 0001 + 1001 で 1010 になる。
これでは不便なので、足し算で都合が良くなるように 逆順で マイナスを割り振っていく。
0000 を基準に、1 は 0001，-1 は 1111、2 は 0010，-2 は 1110、といったように割り当てる。
こうすると、符号違いの同じ数を足したときに、結果が（0 + 0以外）必ず 10000 になるのだ。
この方式で負の数を実現する表現方法を、**二の補数表現** という。
補数とは、「元の数と足した結果が、基準となる数と等しくなるような数」である。
60について「100の補数」なら、「60と足した結果、基準100になるような数」なので 40 である。
これで考えると、正直 二の補数 というより「2のべき乗の補数」といった方が正しい気もするが……。

補数を用いない普通の2進数を、マイナスの符号を考えないという意味で **符号なし** 2進数と呼ぶ。
この方法では、$0 \sim 2^{(桁数)}-1$ までの整数を表現できる。
それに対し、負の数を扱える2進数を、マイナスの符号を考えるという意味で **符号付き** 2進数と呼ぶ。
この方法では、最上位ビットが符号になるから、実質的に数字を表現するのは (全体の桁数 - 1) 桁である。
よって、表現できる整数の範囲は $-2^{(桁数-1)} \sim 2^{(桁数-1)}-1$ となる。

2の補数表現で負の数を表すには、以下の手順を踏む。

1. 普通の正の数でのビット列を考える。 例えば 4 を4桁で `0100` と表す。
2. これに、0 と 1 を反転したものを考える。0 のところを 1 に、1 のところを 0 にする。 `1011`
3. 1 を足した値を考える。繰り上がりに注意。 `1100`

これで、2の補数表現による -4 が完成する。普通の二進数を考え、ビットを反転、+1 という手順だ。

2進数	16進数	符号付き	符号なし	2進数	16進数	符号付き	符号なし
0000	0	0	0	1000	8	8	-8
0001	1	1	1	1001	9	9	-7
0010	2	2	2	1010	A	10	-6
0011	3	3	3	1011	B	11	-5
0100	4	4	4	1100	C	12	-4
0101	5	5	5	1101	D	13	-3
0110	6	6	6	1110	E	14	-2
0111	7	7	7	1111	F	15	-1

問題
(1) 二進数8桁で -20 を表現せよ。ヒント：普通の 20 は `00010100` である。
(2) `10000110` は、符号なし二進数でいくつか。また、符号付き二進数でいくつか。

1.3 命令セットアーキテクチャ

機械語やアセンブリ言語といっても、実は様々な種類が存在する。
というのも、コンピュータを作る会社は一つではなく、さまざまな会社が自分の都合のいい命令や仕様を作っている。
コンピュータによって、電線の組み合わせ方も、内部に入っている回路の順番や種類も異なる。そのため、同じ機械語を適用しても、電気の流れ方が全然変わってしまう。同じアセンブリ命令を適用しても、「この命令を動かすための回路が実装されてないよぉ」なんてことも起こる。
そのせいで、世界的に統一した規格が作られなかったのだ。

世の中には、様々な会社による規格が存在する。それらを分類するため、CPUの論理的な構成を **アーキテクチャ** と呼び、分類を作った。
「こうすればこう動く」、つまり、どのような機械語で特定の動作をするか、による分類を **命令セットアーキテクチャ** と呼び、例えば以下のような命令セットとCPUの実装例がある。

- x86 (Intel社のCoreシリーズ, AMD社のRizenシリーズ)
- ARM (Apple社のA/Mシリーズ)
- RISC-V (EsperantoTechnologies社のAIプロセッサ)

本教材では、情報処理技術者試験の問題の中で扱われる仮想的なコンピュータである **COMETⅡ** および、COMETⅡに対応した命令セット、アセンブリ言語である **CASLⅡ** を基に、仮想的なCPUを使ってアセンブリ言語を学習する。

1.4 仮想CPUの構成

この仮想CPUは、命令とデータを一つの同じメモリの中に内蔵する **プログラム内蔵方式** であり、命令を上から順番に実行する **逐次制御方式** を採用している。
また、このようなプログラム内蔵方式かつ逐次実行方式を採用したコンピュータを、この方法を提唱した数学者の名前から、**ノイマン型コンピュータ** と呼ぶ。

と、堅く言ってもわかりづらいので、要するに「実行する前にやりたいことと使うデータを全部書き込んで、それを順番に実行するよ」ということだ。
当たり前じゃんといったところだが、実は最近話題の量子コンピュータは、非ノイマン型のコンピュータである。

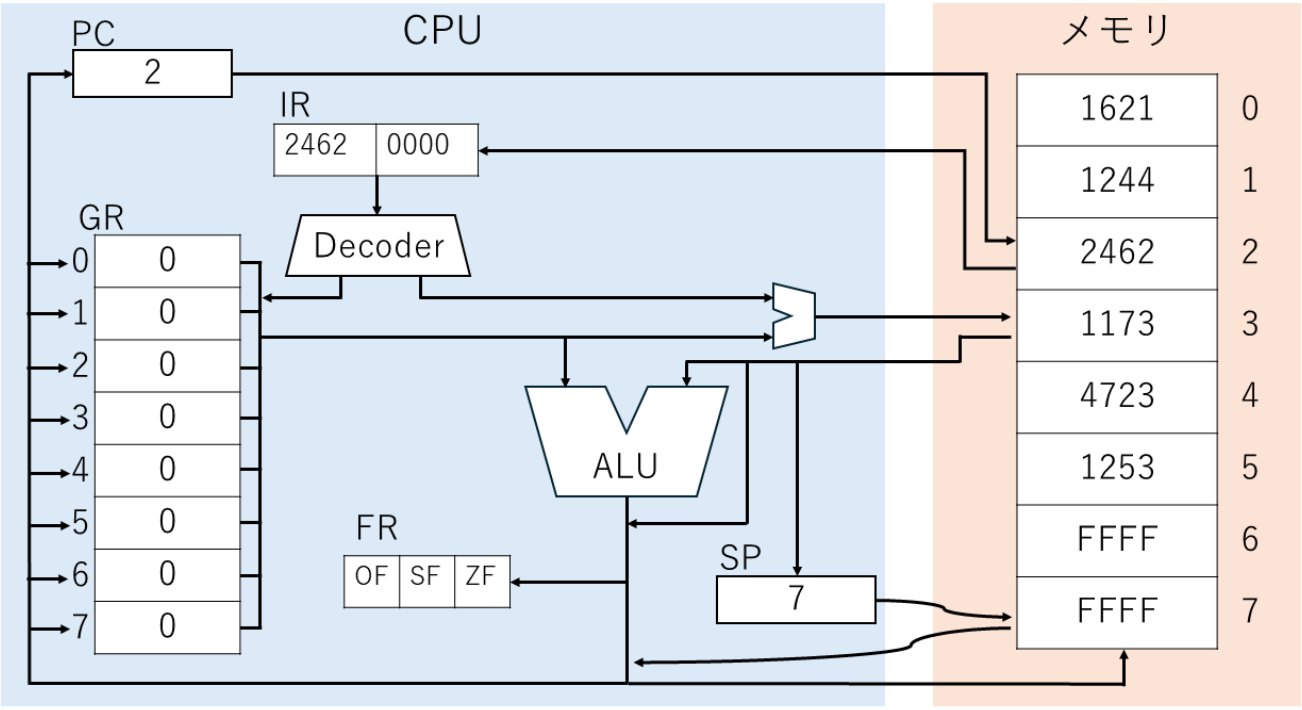
この仮想CPUは、以下の要素で構成されている。

- PC プログラムカウンタ (次に実行するべき命令を持ったメモリのアドレスを保持する)
- IR 命令レジスタ (命令を保持する)
- GR 汎用レジスタ (計算結果とかデータとかを保持しておく)
- FR フラグレジスタ (数値に応じて変わる。「今読み込んだ値は負の数！」「計算結果が0！」とか)
- SP スタックポインタ (スタック領域の先頭であるアドレスを保持する)
- メモリ (各16ビット、0 から 65535 番地まで 65536個の領域がある)
- ALU (Arithmetic Logic Unit; 算術論理演算器 の略。演算装置のこと)

フラグレジスタには、以下のフラグが 1ビットずつ存在する。

- OV (オーバーフローフラグ) 値が大きすぎたり小さすぎて表現できなくなった場合に、1となる
- SF (サインフラグ) 値が 負の数 になった場合に、1となる
- ZF (ゼロフラグ) 値が 0 になった場合に、1となる

なお、COMETⅡでは通常、プログラムカウンタのことを「プログラムレジスタ PR」と呼ぶが、本CPUでは PC とする。



1.5 コンピュータの基本動作

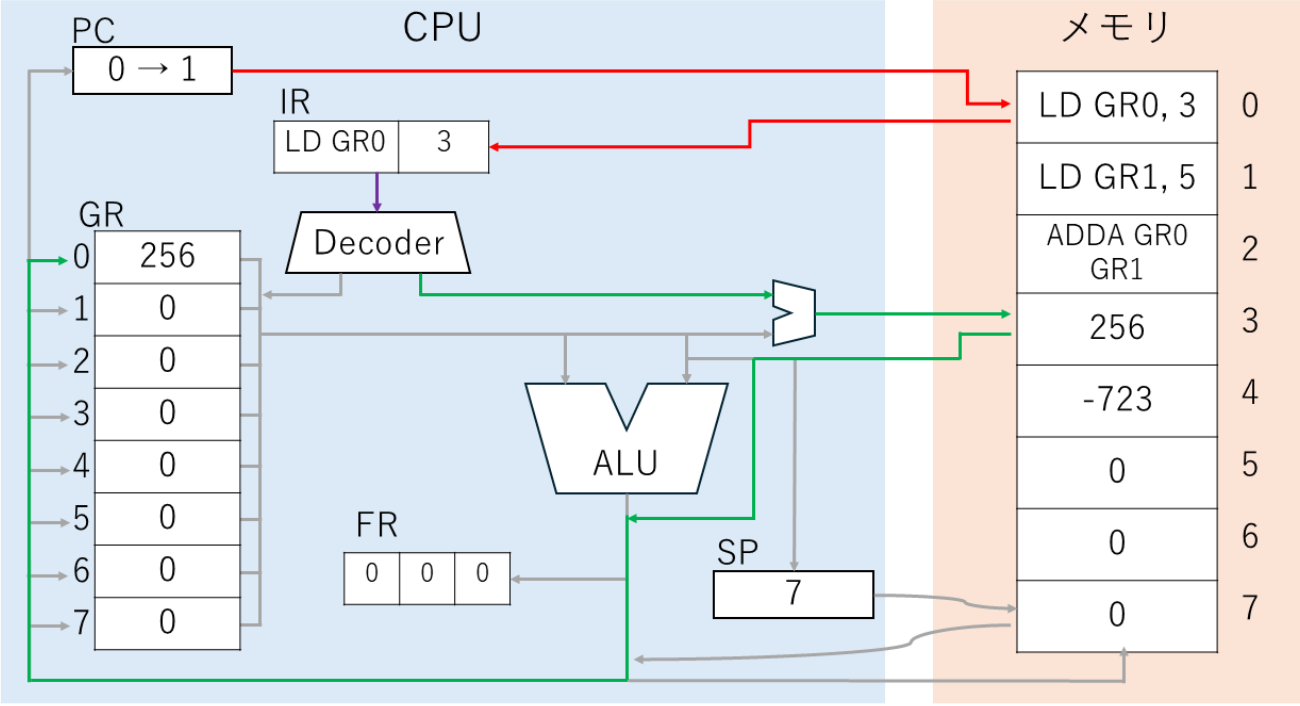
ノイマン型コンピュータは、大きく分けて、次の3つを繰り返して動作している。

1. fetch（フェッチ）：PCの値を参照し、それをアドレスとしてメモリの中身を IR に保存する。そして、PCの値を増やす。
2. decode（デコード）：IRに入った命令を解読する。
3. execute（実行）　：解読した結果に基づき、どのような処理をするか制御装置で逐次制御・実行する。

これを繰り返すことで、「メモリに書かれた命令を上から順番に読み実行する」逐次制御方式を実現する。

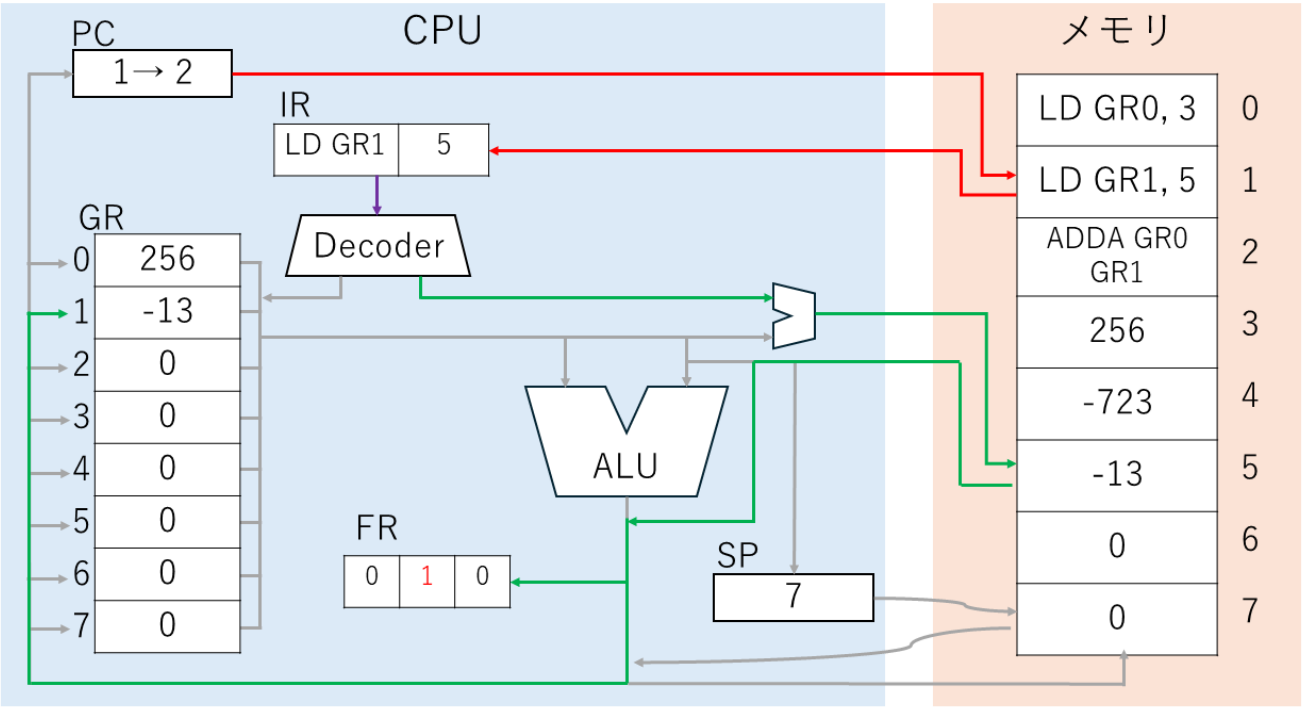
例えば、メモリからデータを読み込んで、汎用レジスタ0に保存する場合、次のように制御とデータが流れる。

fetch を 赤色、decode を 紫色、execute を 緑色 でそれぞれ色分けしてみた。
赤 → 紫 → 緑 の順番で処理が行われる。

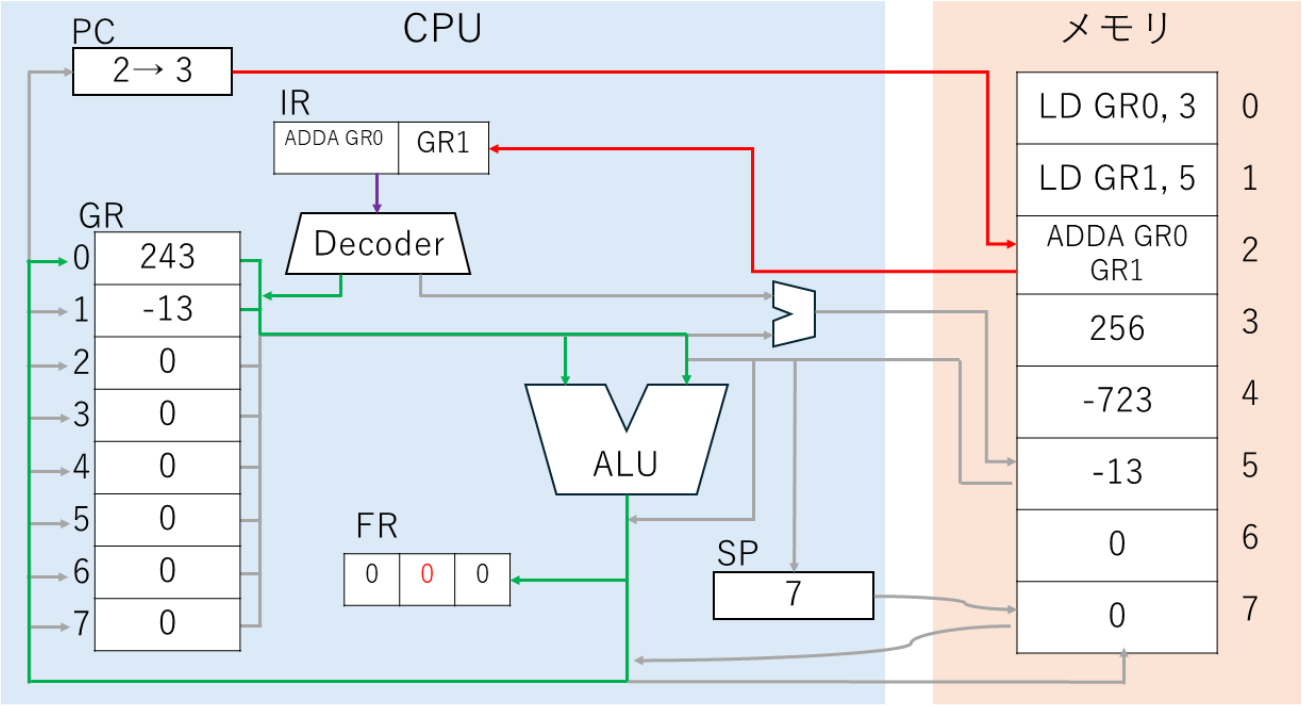


まず、（fetch；赤色）PCの値を参照して、0番地の命令をIRに読み込む。そして、PCの値を増やす。
次に、（deocode；紫色）デコーダーに命令を渡し、解読を行う。
その後（execute；緑色）解読結果から、どの回路を動かすか決定し、処理を行う。今回はメモリの3番地に入ったデータをGR0に格納する。
そして、これが終われば、またPCの値を参照して命令を読み込む。……といったように処理が続く。

次の命令も続けて見てみよう。



PCの値を参照して、1番地の命令をIRに読み込む。そして、PCの値を増やす。
次に、デコーダーに命令を渡し、解読を行う。
解読結果から、どの回路を動かすか決定し、処理を行う。今回はメモリの5番地に入ったデータをGR1に格納する。
ここで、GR1に読みだした値が 負の数 だったため、SFが1になる。
そして、これが終われば、またPCの値を参照して命令を読み込む。……といったように処理が続く。



PCの値を参照して、2番地の命令をIRに読み込む。そして、PCの値を増やす。
次に、デコーダーに命令を渡し、解読を行う。
解読結果から、どの回路を動かすか決定し、処理を行う。今回はGR0の値とGR1の値を足して、GR0に格納する。足す操作はALUにて行う。
結果、256 + (-13) により、GR0の値は 243 に変わった。
そして、これが終われば、またPCの値を参照して命令を読み込む。……といったように処理が続く。

アセンブリ言語で世界に挨拶 (本編 3章)

長かった座学パートも終わり。次は実際にアセンブリ言語を触ってみよう。
説明書を読んで、仮想CPUを起動してください。

3.1 「Hello, World!」

プログラミング言語を学習すると、大体初めにこれを行う。
まあ、文字を出力できなければ「こいつ何か出来るんか？まず動いてる？」案件になってしまうので、最初に入出力を学ぶんですかね。知らんけど。

ということで、世界に挨拶しよう。「Hello, World!」という文字列を出力する。
文字列を出力するには、`OUT` 命令を使う。使い方は `OUT 出力文字列領域, 文字長領域` だ。
詳しくは 2章 を見返してほしい。
出力文字列領域は 「文字列」を意味する character string から `string` と名付けてみる。
文字長領域は「長さ」 length から `len` とでもしておこうか。

```
MAIN    START
        OUT     string, len
        RET
        END
```

そしたら、`string` と `len` に出力したい文字列とその長さを指定しよう。
領域の確保には、`DC` という命令を使う。「メモリにこのデータを入れておいて欲しいな」と要求をする。
これを高級言語では「変数宣言」と呼んだりする。
文字列は `'` （シングルクオーテーション） で囲う。シフトキーを押しながら 7 を押そう。
文字列は `'Hello, World!'` としよう。
`len` には文字列の長さを指定する。文字列 Hello, World! は 13文字 だから、 13 を指定する。

```
MAIN    START
        OUT     string, len
        RET
string  DC      'Hello, World!'
len     DC      13
        END
```

これで「世界に挨拶」出来そうだ。実行して Output 欄を見てみよう。

このコードが理解出来たら、本節は以上になる。理解できるまで見返そう。ポイントは、

- 文字列の出力には `OUT` 命令を使う
- `OUT` 命令は「出力文字列領域, 文字長領域」を指定する。
- それぞれラベルで宣言して、`DC` 命令でメモリに格納する。
- 文字列は `'` で囲う必要があり、文字長には適切な文字数を指定する必要がある。

といったところだろう。

3.2 くどいくらいの挨拶を

文字を一回だけでなく、たくさん表示したい。他人が見たらドン引きするくらいの変人ムーブをしよう！
一つ思いつくのは、「表示したい数だけ `OUT` 命令を書く」だろう。

```
MAIN    START
        OUT     string, len  ; この行を何個も書く
        OUT     string, len
        OUT     string, len
        OUT     string, len
        OUT     string, len
        RET
string  DC      'Hello, World!'
len     DC      13
        END
```

確かにこれでもたくさん表示できる。しかし、「10000回表示してくれ」となったらどうだろう。
コピー&ペーストでも手打ちでも、10000回ぶん律儀に `OUT string, len` する？
流石に大変が過ぎるから、別の手を考えよう。

同じことを何回もする。つまり「繰り返す」のだ。
ところで、命令の中には、「ここから実行してね！」と、実行する場所を強制的に変える物がある。
これらは「分岐命令」と呼ばれ、例えば「引き算の結果が 0 より小さかったら警告を表示する」など、場合によって処理を「分岐」させることが多い。
しかし一方で、「実行する場所を変える」というのは、別の使い方も考えられる。
これを上手く使って、「前に実行したことをもう一度実行する」が出来ないだろうか。
命令は上から順番に 逐次実行 される。「OUTして、前に戻ってもう一回OUTする」と考えてみよう。

```
MAIN    START               ; <──────────────────────┐
        OUT     string, len ;                        │ 戻る
        JUMP    MAIN        ; MAIN、つまり先頭に戻る  ──┘
        RET
string  DC      'Hello, World!'
len     DC      13
        END
```

これを実行すると、無条件で先頭へ戻り続けるので、無限に出力され続けてしまう。終わらない。
[▶] ボタンで実行した方は [||] ボタンを押して止めてください。
[F] をした方は残念。一生終わらないので、右上の × を押してアプリを消してください。開きなおしです。

ということで、指定された回数だけ戻るように改修しよう。

考え方は、「繰り返した回数（今、何週目？）が、n回 より小さければ、もっと繰り返す」である。
繰り返しに応じて、繰り返した回数 `カウンタ` が増えるように、カウンタを作る。
カウンタと 指定する繰り返し回数 を比較して、カウンタの方が小さければ、ジャンプして戻る。

次のページで、流れを追って実装しよう。

まず、繰り返し回数を宣言しよう。とりあえず 5回 繰り返すことを想定する。
`count` と名付けて、`DC` 命令で宣言して 5 を入れてみよう。

今回は GR0 をカウンタとして運用してみる。繰り返すたびに、GR0 の値が 1 増えるように設計しよう。
足し算を行うには、`ADDA` 命令がある。 `ADDA レジスタ1, リテラル` の形で 1 を指定して使ってみる。
これ以外にも方法はあるので、好きに書き換えてみてほしい。
値 1 を格納したアドレスをラベルで宣言して、そのアドレスを指定しても良い。
例えば、GR1 に 1 を入れて、`ADDA GR0, GR1` のようにレジスタ同士の加算でも良い。
`ADDA` 命令の代わりに `ADDL` 命令を使ってもいい。

そしたら、カウンタGR0 と、繰り返し回数 `count` を比較する。
比較を行うには、`CPA` 命令がある。`CPA レジスタ, アドレス` の形で使うと、
レジスタ - アドレス を計算して、FRの値を書き換えてくれる。
今回は、「実際に繰り返した回数（今、何週目？）- 繰り返すべき回数 `count` 」なので、
繰り返すときには、引き算の結果がどういう値になるだろうか。

条件に応じてジャンプしよう。`JPL` かな、`JMI` だろうか、`JZE` かも、`JOV` だったり……？
無条件でジャンプする `JUMP` 命令を、上で考えた条件に合わせて書き換える。

以上を組み合わせて、指定回数だけ繰り返す **forループ** を作ろう。
2章の分岐命令の項目にも、forループのサンプルプログラムが存在する。そちらを参考にしても良い。

```
MAIN    START
        LAD     GR0, 0        ; GR0 をカウンタとして使う。初期値0 を代入

FOR     OUT     string, len   ; 繰り返したい部分の先頭を FOR とラベル付け  <---┐
        ADDA    GR0, =1       ; カウンタを 1 増やす                            ¦
        CPL     GR0, count    ; カウンタ と 繰り返し回数 を比較                ¦
        JMI     FOR           ; カウンタ < 繰り返し回数 なら FOR に戻る    ----┘

        RET
string  DC      'Hello, World!'
len     DC      13
count   DC      5             ; 繰り返し回数
        END
```

まとめ

- 繰り返しには 分岐命令 を使って、自分より前（上）に飛ぶようにする。`JUMP` とか `JMI` とか。
- 条件を適切に考えることで、「○○回だけ繰り返す」のような処理が行える。
- リテラルを使って宣言を省くと、コードを短く書くことが出来る。

と、これでお試しは終了です。

本編で言う1章と3章をほぼ丸々やったことになるので、結構なボリュームがありましたね。

ストップウォッチを止めて、かかった時間を覚えておいてください。アンケートに使います。

記憶が新しいうちに、アンケートに答えてもらえると嬉しいです。

アンケート先URLは、配布のフォルダの中にテキストファイル形式で入っていると思います。

次のページからはおまけです。ここまでの内容で「おもしれ〜」や「分かった！」となった方向けです。
ちょっとした応用と、関連する課題を出しておきます。

3.3 出力する回数を指定しよう（おまけ）

今のプログラムでは、命令の出力回数を変えるために、 `count` の数をいちいち書き直して Assemble し直さなければならない。非常に手間である。
そこで、「プログラムは変えずに、ユーザーが好きなように回数を変えられる」ように変更したい。
具体的には、繰り返す回数を入力にて指定する。

まず、入力を受け取るには `IN` 命令を用いる。 `IN 入力文字列領域, 文字長領域` の形だ。OUT命令と似てますね。
入力を受け取って、それを `count` に格納すれば、繰り返す回数を好きなように弄れそうだ。
なので、入力文字列領域 には `count` を指定して、文字長領域には `=1` でも入れておく。とりあえず 1 ~ 9 回で指定できればいいと思うので、 `=1` で十分だ。
リテラルを使う代わりに、ラベルを使って `inlen DC 1` など宣言しても良い。

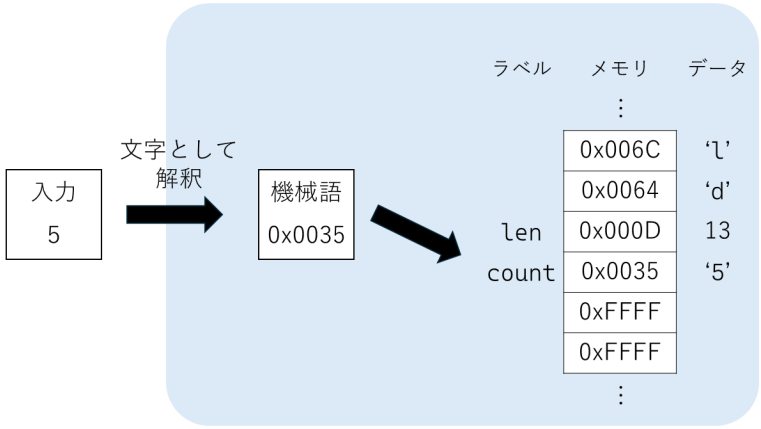
```
MAIN    START
        IN      count, =1    ; 繰り返し回数を標準入力

        LAD     GR0, 0       ; GR0 をカウンタとして使う。初期値0 を代入

FOR     OUT     string, len  ; 繰り返したい部分の先頭を FOR とラベル付け
        ADDA    GR0, =1      ; カウンタを 1 増やす
        CPL     GR0, count   ; カウンタ と 繰り返し回数 を比較
        JMI     FOR          ; カウンタ < 繰り返し回数 なら FOR に戻る

        RET
string  DC      'Hello, World!'
len     DC      13
count   DS      1            ; 繰り返し回数
        END
```

これを実行して、Input欄に 5 を入力する。5回繰り返されるはずだ。

……。
終わらない。ぜんっぜん終わらない。
ずっと待つと、終了時には GR0 が 53 になっている。53回も繰り返したらしい。どうして？

原因は「数字」と「数値」の違いにある。「文字としての数字」と「実際の数」が違うのだ。
長い座学の中に、「文字コード」「asciiコード」というものがあった。思い出してみてほしい。
文字は、コンピュータ上では 2進数 として扱われる。それは asciiコード として変換される。
例えば `A` は `0x41`（10進数で 65）に、`a` は `0x61`（10進数で 97）に変換される。
数字に対しても変換が行われて、`0` は `0x30`（48）、`1` は `0x31`（49）、…、`9` は `0x39`（57）となる。

`IN` 命令で受け取った文字列は、「文字の列」であるから、文字として処理される。
`5` と入力したなら、コンピュータは `0x35`、10進数では 53 という数値として処理してしまう。
これを `count` に格納するから、上のコードは「53回繰り返す」ことになる。



入力した数が、しっかりと「その数値」として解釈してもらえるように、プログラムを改修しよう。

ヒントはasciiコードの並びにある。`0` は `0x30`、`1` は `0x31`、…、`9` は `0x39` となる。
よく見てみると、数字は `0x3` がついている。数値 8 に `0x30` を足すと、数字 `0x38` になる。
つまり、`0x30` を引いてあげれば、数字→数値の変換が出来そうだ。
一度、これで変換が出来るか、数値入力のテストコードを書いてみよう。

```
TEST    START
        IN      count, =1
        LD      GR0, count  ; count の中身（つまり入力した数字）を GR0 にロード
        SUBA    GR0, =#0030 ; GR0の値 から、0x30 を引く。10進数で 48 を指定しても良い
        ST      GR0, count  ; 計算結果を count に再格納
        LD      GR1, count  ; 確認するために、count の中身を GR1 に呼ぶ
        RET
count   DS      1
        END
```

実行したら、0から9までの適当な数字を入力しよう。
GR1 の値が、入力した数値と同じになっていたら成功だ。

これを、先程まで作っていたプログラムに移植する。

```
MAIN    START
        IN      count, =1    ; 繰り返し回数を標準入力

        LD      GR0, count   ; 移植部分
        SUBA    GR0, =#0030
        ST      GR0, count

        LAD     GR0, 0       ; GR0 をカウンタとして使う。初期値0 を代入

FOR     OUT     string, len  ; 繰り返したい部分の先頭を FOR とラベル付け
        ADDA    GR0, =1      ; カウンタを 1 増やす
        CPL     GR0, count   ; カウンタ と 繰り返し回数 を比較
        JMI     FOR          ; カウンタ < 繰り返し回数 なら FOR に戻る

        RET
string  DC      'Hello, World!'
len     DC      13
count   DS      1            ; 繰り返し回数
        END
```

5 を入力したら、しっかり 5回 だけで実行が終わることを確認しよう。

余談だが、「テスト」は大事な考え方になる。
初めから、一気にたくさんの要素を詰め込もうとすると、いざミスがあったときに「どこが間違っているか分からない」状態になってしまうことがある。
そこで、実際に機能を詰め込む前に、「その機能だけ」あっているか確かめるといい。
この検証プログラムを「テストコード」と呼び、小さい機能単体をテストするような場合を「単体テスト」という。
単体テストを行って、いけそうなら実際のプログラムに組み込む。
さらに、組み込んだことで、ほかの部分と噛み合わずバグが起きないか、などもテストする。少し大規模なテストだ。
このように、テストを繰り返しながら 安全に プログラムを作成すると、一見 遠回りに見えても完成が早かったりする。致命的なミスを予防できるからね。

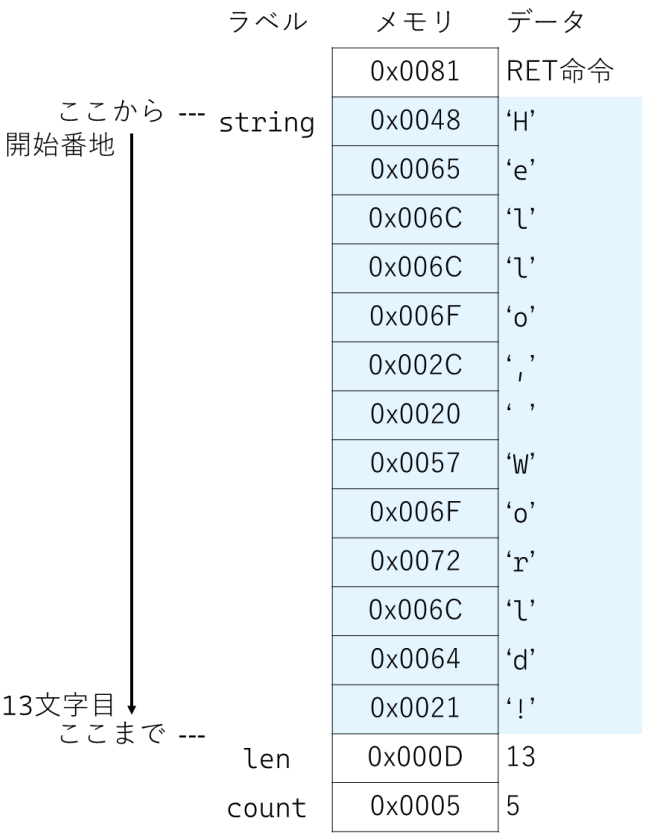
3.4 おまけ課題

今のプログラムは、何行ぶん出力したのかが分かりにくい。
そこで、`1: Hello, World!` のように、先頭に何個目の Hello, World! なのか分かるようにしたい。

ヒント
出力する文字列は、「出力文字列領域」と「文字長」で指定する。また、メモリは 命令を上から**連続させて**書き込む。
上手く工夫して、「何行目」「: Hello, World!」を連続させて、文字長を適切に指定しよう。

また、出力する文字列は「文字」である。「数値」のままだと上手くいかないぞ！
では、数値から「数字」にするには、どうしたらいいだろう？

OUT　string, len の処理



ヒント2
次のコードの、一部分を書き換えよう。

```
MAIN    START
        IN      count, =1    ; 繰り返し回数を標準入力

        LD      GR0, count   ; 移植部分
        SUBA    GR0, =#0030
        ST      GR0, count

        LAD     GR0, 0       ; GR0 をカウンタとして使う。初期値0 を代入

FOR     ADDA    GR0, =1      ; カウンタを 1 増やす

; ここに、数値→数字の変換を記述
; row に、変換した数字を格納
; 文字列を出力

        CPL     GR0, count   ; カウンタ と 繰り返し回数 を比較
        JMI     FOR          ; カウンタ < 繰り返し回数 なら FOR に戻る

        RET
row     DS      1
string  DC      ': Hello, World!'
len     DC      16           ; 行目が 1文字、': 'が 2文字、全体で 1 + 2 + 13 = 16文字
count   DS      1            ; 繰り返し回数
        END
```