R8Cマイコン アセンブラの基礎

今回は どこから始めたらいいのか 悩みますね。

アセンブラは 他の言語と異なり 対応するCPUのアーキテクチャと 密接に連携しています。特に命令や アドレッシングモードは CPUと 1対1に 対応します。
よって 最初は アセンブラではなく、R8Cマイコンの CPU の話から始める事にします。

最初に レジスタを説明しようと考えましたがそのレジスタを説明するにあたり、最低限出てくる アセンブラの用語を説明しないと レジスタと アセンブラの連携が 分かりにくいと判断しました。 という事で 話が 2転 3転して申し訳ありませんが 最初、概念的な 用語の説明を行う事にしました。

まずは ニーモニックコードを 説明します。ニーモニックコードですが CPUが実行するマシン語と 1対1に 対応する コードです。1と 0で構成されるマシン語では 意味が分かりにくいので もう少し意味が分かりやすい 短い単語で構成されています。 で、一つ疑問が出てきました。

一般的に ニーモニックコードというと MOV命令とか ADD命令とか マシン語の オペコードを指している場合が 多いのですが オペコードだけを指して ニーモニックコードというのか、オペコード、オペランドの両方を指して ニーモニックというのか、気になった次第です。 で、調べましたが 明確には書いてないですね。
通常オペコードを指す場合が多いですが、オペランドも 含めてニーモニックという場合もあるようです。その場合、オペランドは アドレス値、数値は ラベル名や 値を示す名前に なってます。

よってニーモニックは 意味が分かりやすい名前に 置き換わっている状態を指すのかもしれませんね。 で、オペランドですが、日本語風に表現すると 命令語の修飾子で しょうか。？ オペコードに続く パラメータみたいな物です。
例を 少し示します。

```
ｵﾍﾟｺｰﾄﾞ  ｵﾍﾟﾗﾝﾄﾞ
PUSH.w    A0
MOV.w     R1, A0
MOV.b     [A0], R0L
POP.w     A0
RTS
```

上記の PUSH.w や MOV.w や MOV.b や PUSH.w POP.w や RTS は、オペコードです。 それに対し 右に続く 文字列（レジスタ名や アドレス値、データ値）がオペランドになります。 上の例では A0、R1、R0Lなどが オペランドです。で R1,A0 や [A0],R0L は コンマで 区切って2つの オペランドを 並べています。

左が 第一オペランド、右が 第二オペランド と呼びます。 因みに 左の例では RTS命令の 様に オペランドが 無い命令も あります。
通常オペランドが 二つ並ぶ場合は MOV命令などの 転送命令です。
第一オペランドが 転送元で
第二オペランドが 転送先です。
例） mov.w r1, a0
では r1 が 転送元で 、 a0 が 転送先です。

それと、フェッチサイクルとか イグゼキュートサイクルは フェッチサイクルは 1命令を メモリから 取り込むサイクルです。細かくいうと メモリから命令を取り込んだ後、命令解読というか デコード処理も フェッチサイクルに含むようです。
イグゼキュートサイクルは 読み込んだ命令を実行するサイクルです。

通常 フェッチサイクルと イグゼキュートサイクルは 交互に繰り返し行い プログラムを 順次実行して行きます。 遥か昔の機械は 固定語長で フェッチと イグゼキュートを 単純に1サイクルずつ交互に繰り返してプログラムを実行してきました。マイコンが 出てきてから マシン語の 可変語長が 普及して フェッチサイクルは 最初のオペコードを 取り込んだ時点で 何バイト命令であるか判定し その後の2バイト目以降を 読み込むため フェッチサイクルのサイクル数は オペコードにより変則的に変化します。 で、アセンブラのソースですが、1行の書式は [シンボル] [オペコード] [オペランド] [コメント]と なります。 ラベルは シンボルの中に 含まれます。シンボルは 行の先頭カラムから 記述します。 シンボルの先頭文字に 数字を使う事は 出来ません。

逆に 数値を記述する際に 先頭文字に アルファベットは 使えません。シンボルと御認識されます。

よって 16進数の数値で、先頭の文字が A～Fになる場合が ありますが その場合は 先頭に 0 を 付けます。

例） FF00h --> 0FF00h

それと シンボル、オペコード、オペランドの間は1つ以上の スペース または TABも 使えます。

コメントは ; を 入れる事で それより右は コメントになります。

次に アセンブラのソースと オブジェクトコードに変換した物を 1行で見れる リスティングファイルの サンプルを お見せします。

```
SEQ.   LOC.    OBJ.                OXMSDA ....*....SOURCE STATEMENT....7....*....8....*....9.
4425                                          .glb     _poke
4426   00090                               _poke:
4427   00090   C2                    S         push.w     a0
4428   00091   73 14                           mov.w      r1, a0
4429   00093   73 21                           mov.w      r2, r1
4430   00095   72 26                           mov.b      r1l, [a0]
4431   00097   D2                    S         pop.w      a0
4432   00098   F3                              rts
```

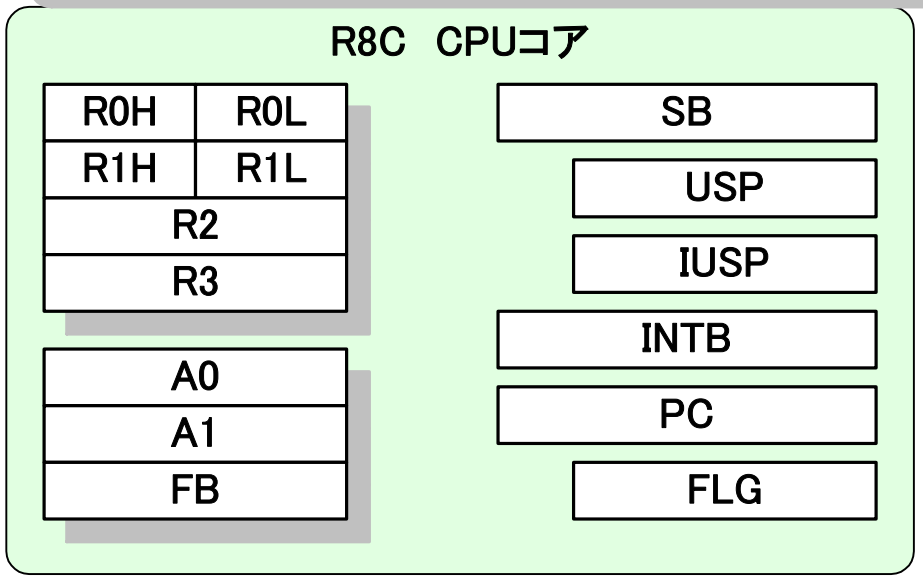
上の テキストは R8C_IOCS_Base.a30の C言語から呼び出す ポーク関数の リス ティング ファイルの一部分を 切り出しました。

一番上の行に 左から SEQ. は 単に行番号です。LOC. は ロケーションで メモリのアドレスです。 OBJ. は 16進表現の マシン語です。

この ポーク関数は ロケーション 90h～98hに配置されています。 この番地は 実行時のアドレスではありません。 リンカで複数のモジュールを連結編集するので 番地は 後方に ズレてきます。push.w a0 の マシン語は C2h で 1byteです。 mov.w r1, a0 の マシン語は 73 14h で 2byteです。 R8Cの オペコードの マシン語は 命令により 1byteの場合と 2byteの場合が あります。

リスティングファイルだと ニーモニックと マシン語を 左右に並べて 見比べ出来るので、いいかなと思い 載せてみました。

R8Cマイコン CPUコアに関して



左は 汎用レジスタバンクと呼ばれるものです。 R0、R1、R2、R3、A0、A1 は 通常よく使います。 A0、A1は ポインタレジスタですが、値の転送、算術、論理演算に使用できます。逆に R0 ～ R3は、ポインタレジスタとしては使えません。R0は 上下 8bitに分けて R0H、R0Lに 使用できます。R1も同様に分けて使用出来ます。それと、R2と R0を連結して 32bit レジスタとして使用できます。 同様に R3と R1を連結して 32bitレジスタとして使用できます。

あと、汎用レジスタバンクの 下にある FB ですが、フレームベースレジスタです。 これは FB相対アドレッシングに使用します。 主な用途はC言語の 関数内で宣言する Auto変数を アクセスするレジスタとなります。 C言語と アセンブラの関数を 組み合わせて使用する場合は C言語側で 使用している可能性が高いので アセンブラ側では使わない方がいいです。

それと、汎用レジスタバンクは 2つあります。汎用レジスタバンクに 影を付けて描いているのは そのためです。

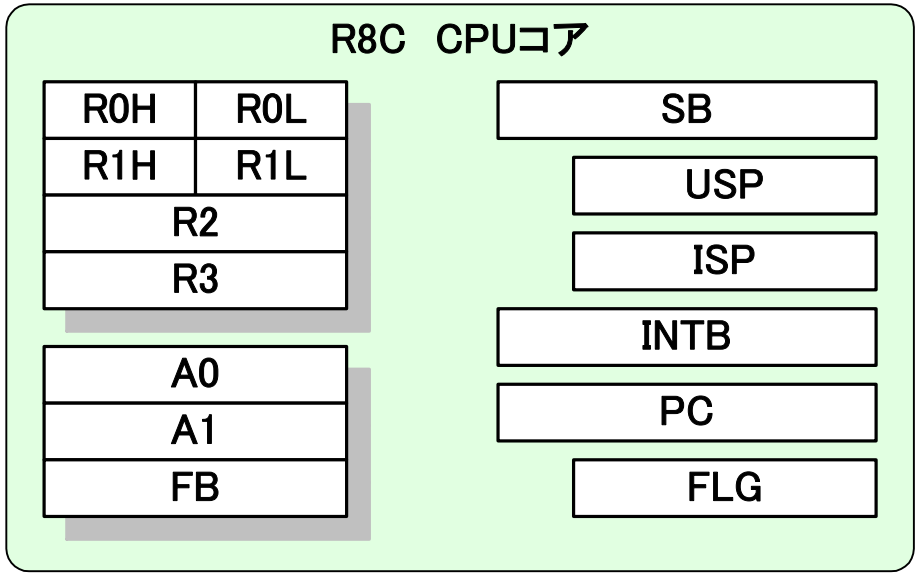
汎用レジスタバンクの切り替えは フラグレジスタの Bフラグを 設定する事により、瞬時に レジスタバンクを 切り替えられます。

汎用レジスタバンクの切り替えは 割り込み処理において絶大な高速応答性を 実現出来ます。
昔 Z80においても同様の機能がありました。

通常の割り込み処理の場合は 全ての汎用レジスタを スタックエリアに退避する事になります。
R8Cマイコンでは R0、R1、R2、R3、A0、A1、FB の 7本の 16bitレジスタを スタックに積み上げる事になります。 且つ R8Cマイコンでは データバスが 8bitのため 1本のレジスタを スタックに積み上げるため データバスを 2回通ってスタックエリアに 積み上げる事になります。 よって レジスタバンクをスタックに積み上げるため計 14回データバスを アクセスする事になります。これじゃ 割り込み応答が 遅くなりますよね。 但し、一つ制限が有りレジスタバンク切り替えは バンクが 2つしか無いので バンク切り替えで、多重割り込みには 対応出来ません。

という事で 複数の割り込み要因がある場合、割り込み処理の優先順位等を 考える必要があります。 最優先の割り込み(非常に高速の応答性を要求される割り込み)にだけ、レジスタバンクの切り替えを行うのが 有効と思います。
それより低い優先順位の割り込み処理には 汎用レジスタの退避、復帰は 通常の push、pop 命令で スタックに積み上げる事になります。
割り込み処理は、ちょっと難しい話になりましたね。 例えば、STEPモーターの相の切り替えを割り込み処理にて行う場合、早い速度で 回転させる場合、1秒間に 10,000回ぐらい 10KHz周期ですね。割り込みが発生する事は あり得ると思います。

レジスタバンクの話から 割り込み処理の話になってしまいましたね。 残りのレジスタの話に移ります。



右側のレジスタですが、システムの起動時初期値を 設定する時ぐらいで 直接アクセスは あまりしないと思います。 フラグレジスタはアセンブラでは ちょこちょこアクセスするかな。

まず、SBですが SB相対アドレッシングに 使用します。USPはユーザスタックポインタです。通常の関数呼び出し時に使用します。 ISPは割り込み用スタックポインタです。

INTBは 割り込みテーブルレジスタで 20bitで構成されており、可変割り込みベクターテーブルの先頭番地を示します。 通常は 固定ベクトルテーブルの 手前に配置されます。可変割り込みベクターテーブルには タイマーや シリアル通信の 内蔵周辺回路の 割り込みを登録します。

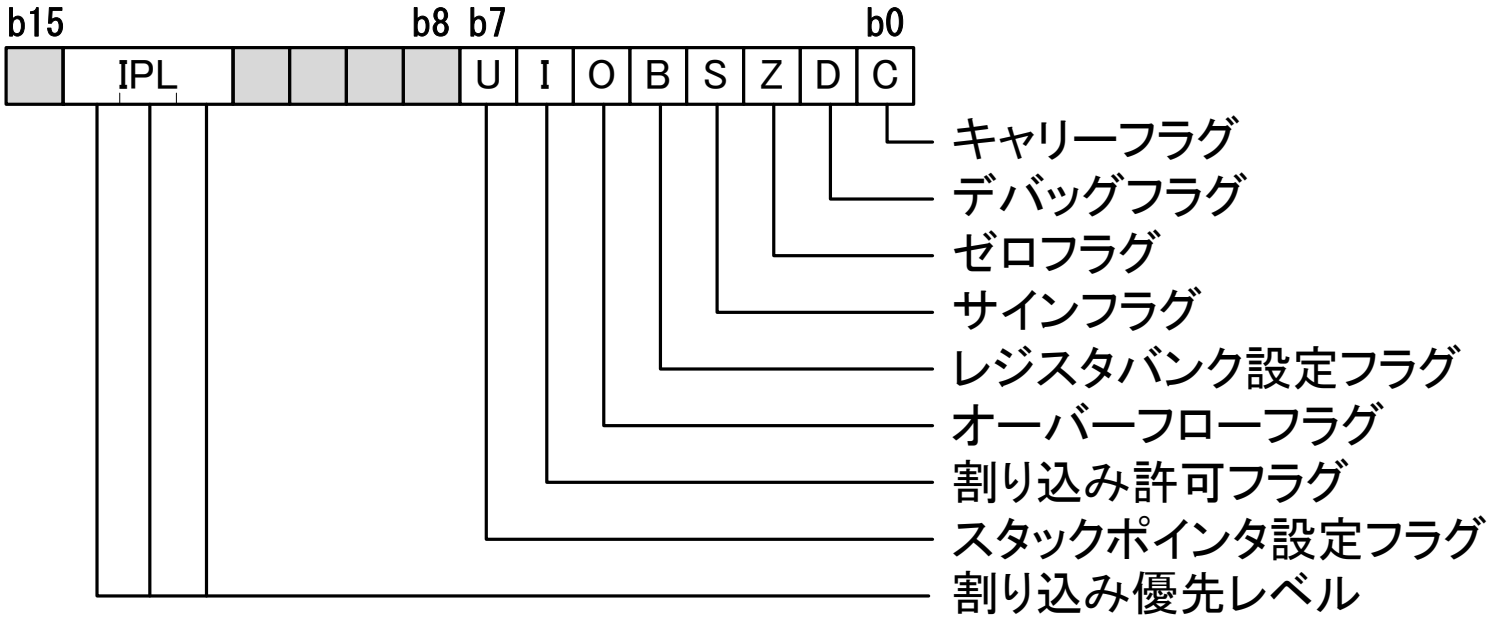
PCは プログラムカウンタです。 次にフェッチする命令の番地を 指しています。 命令のフェッチに連動して インクリメントされます。

JMP命令を実行した場合等は PCの値が 飛び先のアドレスに 設定し直されます。

JSR命令 ジャンプサブルーチン命令の場合は現在の PC値を スタックに積み上げた後にサブルーチン先頭アドレスを PCに設定して飛び先の命令を フェッチし始めます。

フラグレジスタは 次で説明します。

フラグレジスタ



スタックポインタ設定フラグ:
Uフラグが 0 の場合 ISP が指定され 1 の場合 USP が 指定されます。

USP=1の場合でも 割り込みが 発生した場合は、一時的に ISPが 指定され、割り込みから復帰したら USPに 戻ります。
よって 通常の関数呼び出しは USPで 行います。

キャリーフラグ: 算術論理ユニットで発生した キャリー、ボロー、シフトアウトしたビットを保持します。
デバッグフラグ: Dフラグは デバッグ専用です。0 に して下さい。
ゼロフラグ: 演算の結果が 0 のとき 1 になります。 それ以外の時 0 に なります。
サインフラグ: 演算の結果が 負のとき 1 に なります。それ以外の時 0 に なります。
レジスタバンク設定フラグ: Bフラグが 0 の場合、バンク 0 が 指定され、1 の場合 バンク 1 が 指定されます。
オーバーフローフラグ: 演算の結果が オーバーフローしたとき 1 になります。それ以外は 0 です。
割り込み許可フラグ: マスカブル割り込みを許可するフラグです。I フラグが 0 で禁止、1 で許可です。

最後に 割り込み優先レベルですが
IPLは、3 bit で構成され レベル 0 ～ 7 までの 8段階の 割り込み優先レベルを指定します。
要求があった割り込みの優先レベルが IPL より大きい場合、その割り込み要求は 許可されます。

割り込みの優先レベルを設定して割り込み処理を行う事は 多重割り込みを 許可して割り込みを行う事になり高度な使い方になります。
割り込み要因が少なくて 特に高速応答が必要な割り込み処理が 無い場合は 全て割り込み優先レベルは 同一設定でも構わないと思います。
高度な割り込み処理を行う場合、割り込み処理の 実時間を オシロスコープ等で 計測して検討する必要も あります。

以上 CPUコアのレジスタの説明でした。

あとアセンブラで使用する CPUコアに関わる絶対必要な物として重要な CPUの命令を 意味するニーモニックコードが あります。
大雑把に一覧で示します。

機能	ニーモニック	内容
転送	MOV	転送
	MOVA	実行アドレスの転送
	MOVDir	4bitデータ転送
	POP	レジスタ／メモリの復帰
	POPM	複数レジスタの復帰
	PUSH	レジスタ/メモリ/即値の退避
	PUSHA	実行アドレスの退避
	PUSHM	複数レジスタの退避
	LDE	拡張データ領域からの転送
	STE	拡張データ領域への転送
	STNZ	条件付き転送
	STZX	条件付き転送

機能	ニーモニック	内容
転送	XCHG	交換
ビット処理	BAND	ビット論理積
	BCLR	ビットクリア
	BMCnd	条件ビット転送
	BNAND	反転ビット論理値
	BNOR	反転ビット論理和
	BNOT	ビット反転
	BNTST	反転ビットテスト
	BNXOR	反転ビットの論理和
	BOR	ビット論理和
	BSET	ビットセット
	BTST	ビットテスト
	BTSTC	ビットテスト＆クリア
	BTSTS	ビットテスト＆セット
	BXOR	ビット排他的論理和
ローテート	ROLC	キャリー付き左回転
	RORC	キャリー付き右回転
	ROT	回転

機能	ニーモニック	内容
シフト	SHA	キャリー付き左回転
	SHL	キャリー付き右回転
算術演算	ABS	絶対値
	ADC	キャリー付き加算
	ADCF	キャリーフラグの加算
	ADD	キャリー無し加算
	CMP	比較
	DADC	キャリー付き 10進加算
	DADD	キャリー無し 10進加算
	DEC	デクリメント
	DIV	符号付き除算
	DIVU	符号なし除算
	DIVX	符号付き除算
	DSBB	ボロー付き 10進減算
	DSUB	ボロー無し 10進減算
	EXTS	符号拡張
	INC	インクリメント
	MUL	符号付き乗算

機能	ニーモニック	内容
算術演算	MULU	符号なし乗算
	NEG	2の 補数
	RMPA	積和演算
	SBB	ボロー付き減算
	SUB	ボローなし減算
論理演算	AND	論理積
	NOT	全ビット反転
	OR	論理和
	TST	テスト
	XOR	排他的論理和
ジャンプ	ADJNZ	加算＆条件分岐
	SBJNZ	減算＆条件分岐
	JCnd	条件分岐
	JMP	無条件分岐
	JMPI	間接分岐
	JSR	サブルーチン呼び出し
	JSRI	間接サブルーチン呼び出し
	RTS	サブルーチンからの復帰

機能	ニーモニック	内容
ストリング	SMOVB	逆方向のストリング転送
	SMOVF	順方向のストリング転送
	SSTR	ストリングストア
その他	BRK	デバッグ割り込み
	ENTER	スタックフレームの構築
	EXITD	スタックフレームの解放
	FCLR	フラグレジスタの bit クリア
	FSET	フラグレジスタの bit セット
	INT	ソフトウエア割り込み
	INTO	オーバーフロー割り込み
	LDC	専用レジスタへの転送
	LDCTX	コンテキスト復帰
	LDINTB	INTBレジスタへの転送
	LDIPL	割込み許可レベルの設定
	NOP	ノーオペレーション
	POPC	専用レジスタの復帰
	PUSHC	専用レジスタの退避
	REIT	割り込みからの復帰

機能	ニーモニック	内容
その他	STC	専用レジスタからの転送
	STCTX	コンテキストの退避
	UND	未定義命令割り込み
	WAIT	ウェイト

ここで表示したニーモニックの数は 計 87 でした。でも実際は 使用するレジスタの指定はオペコードに 含まれるようで、マシン語の オペコード部分は 個数は 数倍に増えると思われます。オペコードの長さは 1byteで 用が足りる命令は 1byte長で 2つのレジスタ指定を 行う命令は 2byt長になります。それとは別に アドレス値や 固定データ値などが 必要な場合 オペランドとして オペコードの後ろに付いてきます。

次は アドレッシングモードについて 説明します。

アドレッシングモードとは オペランドに関わる種類のような物です。

一番単純なものは NOP命令や RTS命令 のようなオペランドが 無い物で インヘレントと呼びます。

次に 次に第一オペランドに 即値というか 固定的な数値を 置くアドレッシングモードで イミディエイトと呼びます。 データの初期値とか 先頭アドレスとかを 数値で 第一オペランドに 置きます。 R8Cの場合は 即値を 識別するために # を 値の前に 付けます。
例を 示します。

```
mov.b    #30, r1l
mov.w    #0400h, r2
```

絶対アドレスモード:　アブシリュート
オペランドの　絶対値アドレスの メモリ内容を読み出す、あるいは書き込むモードです。
例）

```
;    プログラム セクション
     .section     program, CODE,ALIGN
     mov.w  r1, eadr_hi      ; b19 ～ b16
     mov.w  r2, eadr_low     ; b15 ～  b0

;    データ セクション
     .section     bss_NE, DATA, ALIGN
eadr_hi:  .blkb 2    ; 拡張アドレス上位 4bit
eadr_low: .blkw1     ; 拡張アドレス下位 16bit
```

セクションを　跨っているので ややこしく見えますが　赤のラベルは　データ領域のアドレスが入ってます。 よって 2つの MOV命令の 第二オペランドは アドレスの 絶対値となります。よって絶対アドレスモードとなります。 実用レベルでは、アドレスの数値ではなく、上記の様にラベルを 使用して変数のアドレスを　指定します。
このようにしておくと、変数を 追加した際に　追加した変数より後ろの変数の番地が 後方にズレますが、ラベルにしておけば、変数の番地のズレは　気にしなくて済みます。 変数のアドレス値を　16進数の数値で 指定したりすると、後で修正する時とんでもない事になります。
よって、アドレスは　ラベルで指定して下さい。

因みに データセクションで　.blkbとか .blkw がありますが、データの領域を確保する 疑似命令です。
　.blkb　2　は　2byteの領域を確保します。
　.blkw　1　は　1wordの領域を確保します。
バイト指定か　ワード指定の 違いだけです。
データセクションの場合、ラベルは　変数名と考えて問題ないと思います。

アドレスレジスタ間接指定: レジスタ インダイレクト

このアドレッシングは 以前使いましたが、A0 または A1レジスタを用いて 仮に A0レジスタを使う場合、A0レジスタが 指しているアドレスの内容を 読み出す、あるいは書き込む アドレッシングモード です。 例）

```
MOV.B  [A0], R1L   ; A0で指すメモリ内容を
                   ; R1L に 転送する
MOV.B  R1L, [A0]   ; R1Lの内容を A0で指す
                   ; メモリに 転送する
```

あと、このA0レジスタに ディスプレースメントといって アドレス オフセットを 付ける事も できます。私は 使った事が ありません。

相対アドレス指定: リラティブアドレッシング

これは、条件付き分岐命令で 使用されます。フラグレジスタの C、Z、S、Oの 状態により、条件が 成立すれば 指定された飛び先にブランチします。成立しなければ 次の番地に行きます。飛び先にブランチする時に 現在の PCの値を中心というか 0 として -128 ～ 127の オフセットで 飛びます。PCの相対値で飛ぶので 相対アドレス指定といいます。因みに 無条件 JMP 命令では 近い番地に飛ぶ時は 相対アドレス指定で飛び、離れた 飛び先には 絶対アドレス指定で 飛びます。

あと レジスタ、メモリ間転送では Push命令があります。これは まず スタックポインタを 積み上げるデータ長に合わせ デクリメントまたはダブルデクリメントして、指すアドレスにレジスタ値を 格納します。

Pop命令は Push命令の逆で、これは まず スタックポインタが指すデータを 送り先レジスタに転送します。 その後 データ長に合わせ インクリメントまたは ダブルインクリメントします。

Push、Pop命令を使う時は Push命令で 最後に積み上げたデータを Pop命令で 最初に 降ろします。 例）

```
Push.w  R1
Push.w  R2
Push.w  R3
Push.w  R4

Pop.w   R4
Pop.w   R3
Pop.w   R2
Pop.w   R1
```

このように なります。

アセンブラの説明は このくらいにしておきます。

後は サンプルのアセンブラソースを見て、まねしてソースを 作ってみると 少しずつ コツが見えてくると思います。

今回 この資料を作って思ったのは あっ、こんな命令も あったんだ。という発見もありました。

R8Cマイコンは、100円マイコンの M120A、M110Aには あまり周辺回路が 実装されて無くて ソフトで I2Cや SPIの 機能を作りました。

R8C/38Aや 35Aには I2Cや SPIとして使える周辺回路が あるようなので いつか試してみたいと思います。