|  |
| --- |
| IT-X アセンブラ  詳細設計仕様書 |

2021/4/12 rev1.0

(株)アイシンク

改訂履歴

|  |  |  |  |
| --- | --- | --- | --- |
| 版数 | 発行日 | 担当 | 改訂内容 |
| 1.0 | 20121/4/12 |  | 新規作成 |
|  |  |  |  |

内容

[概要 1](#_Toc69113898)

[プログラム構成 2](#_Toc69113899)

[処理詳細 3](#_Toc69113900)

[メイン処理 3](#_Toc69113901)

[1 アセンブル処理準備 3](#_Toc69113902)

[2 アセンブル処理 3](#_Toc69113903)

[3 アセンブル後処理 4](#_Toc69113904)

[4 アセンブル結果表示 4](#_Toc69113905)

[アセンブル情報構造体 5](#_Toc69113906)

[5 1行読み込み(& 正規化) 6](#_Toc69113907)

[6 機械語変換 7](#_Toc69113908)

[6.1 命令語判定 8](#_Toc69113909)

[6.2 オペランド判定 9](#_Toc69113910)

[1オペランドの処理 9](#_Toc69113911)

[2オペランドの処理 9](#_Toc69113912)

[6.3 トークン分離 11](#_Toc69113913)

[6.4 コード生成処理 12](#_Toc69113914)

[6.5 ORG処理 13](#_Toc69113915)

[6.6 Data出力処理 13](#_Toc69113916)

[6.7 出力バイナリファイル処理 14](#_Toc69113917)

概要

本文書はIT-Xアセンブラの処理詳細及び関数仕様を説明するものである。

IT-XアセンブラはWindowsコンソールから使用するコマンドラインプログラムである。

コマンドラインから実行されたアセンブラプログラムは以下の手順で処理を実行する。

全体の処理の流れ

1. アセンブル処理準備
   1. コマンライン引数を確認し、ソースファイル、出力バイナリファイルをオープンする
2. アセンブル処理の実行
   1. ソースファイルから命令行を1行読み込む
   2. 命令行を機械語に変換する
   3. 変換した機械語を出力バイナリファイルに書き込む
   4. ソース行の終わりまでアセンブル処理を繰り返す
3. アセンブル処理の後処理
   1. ソースファイル、出力バイナリファイルをクローズする
4. アセンブル処理結果の表示
   1. アセンブル処理結果を確認し、正常終了/エラーの詳細を表示する

* これらの処理は関数として分離し、各処理間でグローバル変数を用いた情報伝達は禁止する　（プログラム実行中に更新されない定数テーブル等は該当しない）
* 各処理間で持続的に参照され記録する情報はアセンブル情報構造体に保持し、メイン関数から呼び出し時に渡される物とする

プログラム構成

main

ShowResult

Deinit

Assemble

Init

Read1Line

LineAssemble

SeparateToken

GetOpecodeInfo

ProcDATA

ProcORG

CodeGenerate

GetOperandInfo

WriteCode

処理詳細

## メイン処理

メイン関数はアセンブル処理準備、アセンブル処理、アセンブル後処理、アセンブル結果表示の各処理を順次呼び出す。

準備に失敗した場合は、アセンブル処理及びアセンブル後処理の呼び出しを行わない。

その他アセンブルに関係する処理は、メイン処理では直接行わない。

## アセンブル処理準備

|  |
| --- |
| int Init(struct AssembleInfo\* pInfo, int argc, char\* argv[]);  pInfo アセンブル情報を格納する構造体のポインタ  Init関数内で初期化されるので、元の内容は参照しない。  argc, argv main関数で受け取ったコマンドライン引数をそのまま渡す。  戻り値 ITX\_OK / ITX\_ERR |

アセンブル処理準備では、main関数から受け取ったコマンドライン引数を確認し、ソースファイル名、出力バイナリファイル名を抽出し、それぞれのファイルをオープンする。

main関数から受け取ったアセンブル情報構造体を、その後のアセンブル処理で使用できるように初期化し、オープンしたファイルハンドル(FILE\*)を格納する。

ソースファイルか出力バイナリファイルのオープンに失敗した場合は、オープン済みのファイルはクローズして、エラー終了する。

* ITX\_OK = 1, ITX\_ERR = 0 と定義(#define)する

## アセンブル処理

|  |
| --- |
| int Assemble(struct AssembleInfo\* pInfo);  pInfo Init関数で初期化済みのアセンブル情報を渡す。  アセンブル結果やエラー情報を含め、アセンブル処理を通じて必要な情報を格納する  戻り値 ITX\_OK / ITX\_ERR |

アセンブル処理は、1行ずつソースファイルを読み込み、各命令行を機械語に変換する。

変換後の機械語は出力バイナリファイルに書き込む。

それぞれの処理において、何らかのエラーが検出された場合は、アセンブル情報構造体にエラー情報を記録し、処理を終了する。

各処理はそれぞれの関数に分けて実行する。

* 1. 1行読み込み処理
  2. 機械語変換
  3. 出力バイナリファイル処理

変換された機械語を出力ファイルに書き込む。

書き込み位置は、変換後の機械語のアドレスと一致するが、書き込み位置のアドレスが適正で無い場合(※)は、エラーとする。

書き込み後は、書き込み位置と、書き込み済みアドレス情報を更新する。

何らかのエラーが判別した場合は、アセンブル情報構造体にエラー情報を記録し、処理を終了する。

※適正でないアドレス：既にコードが書かれている場合、メモリーの有効アドレス範囲外

* 1. ソースファイルの末尾まで処理したかを確認し、終端で無い場合は2.1から処理を繰り返す

## アセンブル後処理

|  |
| --- |
| void Deinit(struct AssembleInfo\* pInfo);  pInfo 有効なアセンブル情報を格納したアセンブル情報のポインタ |

アセンブル後処理では、ソースファイルと出力バイナリファイルのクローズを行う。

Initを一度も通っていないアセンブル情報を渡してはならない。

アセンブル後処理では、アセンブル情報のエラー情報が格納されている場合、上書きを行わない。

（原則ファイルクローズ処理でのエラーしか検出されないので、それ以前に検出されたエラーを優先させる）

## アセンブル結果表示

|  |
| --- |
| void ShowResult(struct AssembleInfo\* pInfo);  pInfo 有効なアセンブル情報を格納したアセンブル情報のポインタ |

アセンブル結果の表示は、アセンブル処理の処理結果に基づき、正常終了か異常終了かを判別し、結果を表示する。

アセンブル処理の結果詳細はアセンブル情報構造体に格納されているので、その情報に基づきエラー情報を表示する。

## アセンブル情報構造体

|  |
| --- |
| struct AssembleInfo {  FILE\* SourceFile, // ソースファイルハンドル(FILE\*)  FILE\* BinaryFile; // 出力バイナリファイルハンドル(FILE\*)  int AssembledLines; // 読み込んだソースファイルの行数  unsigned int CodeAddress; // コード書き込み先アドレス  unsigned int ResultCode; // 処理結果（エラーコード）  GenerateCodeMap GenerateCodeInfo; // 出力済みコードアドレス情報  }; |

アセンブル情報構造体は、アセンブル処理を行うのに必要な情報と、アセンブル処理結果を格納する。

格納する情報は以下の物である。

・ソースファイルハンドル(FILE\*)

　オープンしたソースファイルのを格納する。

オープンできなかった場合はNULLを格納する。

・出力バイナリファイルハンドル(FILE\*)

　オープンした出力バイナリファイルのハンドルを格納する。

　オープンできなかった場合はNULLを格納する。

・読み込んだソースファイルの行数

　初期化後は0であり、1行読み込み処理で更新する。

・コード書き込み先アドレス

　初期化後は0であり、コード生成や.org処理で更新される。

・処理結果（エラーコード）

　初期化後は”エラー無し”になる。アセンブラにおけるエラーが検出されたときに、エラーコードを格納する。

　ただし、アセンブル後処理は、既にエラーコードが格納されているとき限り、上書きを行わない。

・出力済みコードアドレス情報

　コード出力(バイナリファイルへの出力)が行われたアドレスを保存する。

　出力バイナリファイル処理で参照、更新される。

**形式は未決定。(GenrateCodeMap)**

## 1行読み込み(& 正規化)

|  |
| --- |
| int Read1Line(struct AssembleInfo\* pInfo, char\* LineBuffer, int size);  pInfo アセンブル情報  LineBuffer 正規化後の命令行を格納するバッファのポインタ  size LineBufferのサイズ  戻り値 ITX\_OK / ITX\_ERR |

ソースファイルから命令行を1行読み込む。

このときに命令行の正規化を行い、その後の処理で扱いやすい形に変換する。

行末までがあまりに長い場合、itxアセンブラ用ソースコードではない可能性があるので、上限を決めてエラーとする。

（.dataが最も長い行になると考えられるので、.dataでの上限に基づきsizeを決定する）

命令行の正規化処理は、ソースファイルから1文字ずつ読み込んで逐次行う。

正規化後の文字列は、以下の様な形式になる。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **?** | **?** | **?** |  | **?** | **?** |  | **,** |  | **(** |  | **?** | **?** |  | **)** | **\0** |

・行の先頭から文字までの余白文字は取り除く

・文字と文字との間にある余白文字が複数ある場合は1文字のスペースにする

・**カンマ**の前後はスペースにする

・**カッコ**の前後はスペースにする

・最後の文字から行末までの余白文字は取り除く

・アルファベットは全て小文字に変換する

正規化処理では正しい命令行であるかは判別しないので、不適切な命令行であっても、上記のルールに従って処理を行う。

例えば、**“ ABCD, ((EFG HIJ ”** という文字列を処理すると、次のような結果になる。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **c** | **d** |  | **,** |  | **(** |  | **(** |  | **e** | **f** | **g** |  | **h** | **i** | **j** | **\0** |

※空行の場合は、先頭が’\0’になる

正規化処理によってスペースで区切られた単位を**トークン**と呼ぶ。

## 機械語変換

|  |
| --- |
| int LineAssemble(struct AssembleInfo\* pInfo, char\* source);  pInfo アセンブル情報  source 機械語変換する命令行、1行読み込み処理で読み込んだもの  戻り値 ITX\_OK / ITX\_ERR |

正規化された命令行を解析し、機械語に変換する。

命令行からオペコードを取得し、命令コードと後に続くオペランドの形式を判別する。

命令が適切な場合、判別したオペランドの形式に従い、後続のオペランドの解析とオペランドコードの変換を行う。

何らかのエラーが判別した場合は、アセンブル情報構造体にエラー情報を記録し、処理を終了する。

機械語変換は、複数のステップを順次実行して行われる。

各ステップの処理は前段の処理にエラーがあった場合は、先へは進まず、機械語変換処理を終了する。

・命令語判定

・命令語/疑似命令語の判別

・(命令語の場合)

・オペランド判定(オペランドがある場合)

・オペランド判定(複数のオペランドがある場合)

・コード生成

・(疑似命令語)

・各疑似命令の処理

## 命令語判定

アセンブル処理で1行読み取りが行われた直後、最初のトークンがある場合は必ず命令語になっている。まず、この命令語の判定を行う。

先頭のトークンが無い場合、これはその行が空白行なので、何もせずに処理を終了する。（これはエラーではない）

|  |
| --- |
| int GetOpecodeInfo(char\* pOpecode, struct OpecodeInfo\*\* ppOpecodeInfo);  pOpecode 判定するオペコード  ppOpecodeInfo 判定結果(OpecodeInfoのポインタ)を受け取るポインタ  戻り値 ITX\_OK / ITX\_ERR |

ppOpecodeInfo にはテーブルの項目を参照するポインタを返す。

命令語の種別から、その後に続くトークンがどのようなパターンになるかが判別できる。命令語からオペコードのコード、オペランドの数、指定可能なオペランドの種類を取得出来るように、これらをテーブルにして用意する。

命令語が疑似命令であった場合は、それぞれの疑似命令用の処理を実行する。

|  |
| --- |
| struct OpecodeInfo {  char Opecode[4]; // 命令語  unsigned char Code; // 変換する命令コード  int NumOperandt; // 該当する命令語で指定可能なオペランドの個数  unsigned char CombinationType; // 該当する命令語で指定可能なオペランドの組み合わせを判別する情報  }; |

## オペランド判定

命令語種別により、オペランド無し、1オペランド、2オペランドの3つのパターンに分かれる。

例外として、疑似命令.dataの場合は、3つ以上のオペランドになる場合がある。

疑似命令の処理は、命令語判定の後に別の処理を実行するので、このオペランド判定では考慮しない。

## 1オペランドの処理

1オペランドを処理するときは、命令語に続いてオペランドを意味するトークンが続いている。

GetOperandInfoでオペランドの情報を得る。

## 2オペランドの処理

2オペランドを処理するとき、1オペランド目の処理後には次のトークンは必ずカンマになる。

カンマ以外のトークンはエラーである。

カンマ以後のトークン処理は1オペランド目と同様に処理できる。

命令語種別に基づいてオペランド判定を行った後には、一切のトークンが残っていてはならない。

何らかのトークンが残っている場合はエラーである。

必要なオペランドが判別できた後、命令語により有効なオペランドの組み合わせが異なるので、組み合わせが適切であるかの判別を行う。

不適切な組み合わせであった場合はエラーである。

命令語と適切なオペランドが抽出できたら、機械語に変換し出力バイナリファイルにコードを出力する。

|  |
| --- |
| int GetOperandInfo(char\*\* source, struct OperandInfo\* pOperandInfo);  source 判定するオペランドを示すソース(ポインタへのポインタ)  pOperandInfo 判定結果を受け取るOperandInfoのポインタ  戻り値 ITX\_OK / ITX\_ERR |

sourceはオペコード判定にてSeparateTokenでオペコードを分離した後のソースのポインタのポインタである。

オペランド判定は複数回行い後続のトークンを判定する為に、sourceを次のトークンに進める。

オペランドは大きく2種類に分類出来る。

1. カッコを含むオペランド
2. カッコを含まないオペランド

・オペランド判定を呼び出した時の最初のトークンが(であるかどうかで判断ができる

・最初のトークンが(の場合、2つ先のトークンは必ず)にならなくてはならない

カッコを分離したオペランドは2種類に分かれる。

1. 先頭がrで始まるレジスタ
2. 数字で構成されたイミディエイト

先頭がrで始まる場合はレジスタの記述である。rの後ろは必ず0～3のいずれか1文字のみになる。

これ以外の文字がある場合はエラーである。

イミディエイトの場合は0～9, a～fの文字で構成された16進数である。イミディエイトは1文字か2文字で記述される。

これ以外の文字がある場合はエラーである。

オペランドは上記の種類の組み合わせになるので、4種類のパターンになる。

オペランド判定では、4種類のパターンと指定された値（コード）が抽出されることになる。

|  |
| --- |
| struct OperandInfo {  unsigned char Code; // オペランドのコード  unsigned char OperandType; // 直接指定/間接指定 (カッコの有無) と  // レジスタ/イミディエイト の種別を示す  }; |

## トークン分離

正規化された文字列から個別のトークンを取り出す。

|  |
| --- |
| char\* SeparateToken(char\* source);  source 機械語変換処理で処理中の命令行  戻り値 次のトークンの先頭を指すポインタ |

元になる文字列は、正規化処理を実施した後の文字列で、先頭は必ず最初のトークンの先頭になる。

渡された文字列の先頭から順次文字を確認し、スペースを見つけたらスペースをNUL文字(‘\0’)に置き換える。これにより、渡されたポインタは1つのトークンを指すことになる。

残りの文字列から次のトークンを探せるように、NUL文字の次の位置を指すポインタを戻り値として返す。

**1回目、正規化後の文字列からトークンを取り出す：**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **m** | **o** | **v** |  | **r** | **0** |  | **,** |  | **(** |  | **a** | **b** |  | **)** | **\0** |

↑

　　このスペースを探しNUL文字に置き換える

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **m** | **o** | **v** | **\0** | **r** | **0** |  | **,** |  | **(** |  | **a** | **b** |  | **)** | **\0** |

↑

　　　 戻り値は、この位置を指すポインタ

繰り返し処理を実行することで、先頭から順番にトークンを取り出すことが出来る。

**2回目、戻り値から続きのトークンを取り出す：**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **r** | **0** |  | **,** |  | **(** |  | **a** | **b** |  | **)** | **\0** |

↑

　このスペースを探しNUL文字に置き換える

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **r** | **0** | **\0** | **,** |  | **(** |  | **a** | **b** |  | **)** | **\0** |

↑

　　 戻り値は、この位置を指すポインタ

スペースが見つからず、NUL文字(‘\0’)に到達した場合は、最後のトークンなので、NUL文字に置き換える処理は行わない。そして、戻り値はNULLポインタを返し、それ以降のトークンが存在しないことを示す。

## コード生成処理

|  |
| --- |
| int CodeGenerate(struct AssembleInfo\* pInfo, struct OpecodeInfo\* pOpecodeInfo,  struct OperandInfo\* pOperandInfo1, struct OperandInfo\* pOperandInfo2);  pInfo アセンブル情報  pOpecodeInfo オペコード情報  pOperandInfo1 第1オペランド情報(無い場合はNULLを渡す)  pOperandInfo2 第2オペランド情報(無い場合はNULLを渡す)  戻り値 ITX\_OK / ITX\_ERR |

オペコード情報、第1オペランド情報、第2オペランド情報に基づいて、機械語コードを生成する。

機械語コードは、命令コード、オペランドコード1、オペランドコード2の最大3バイトの長さになる。これは命令語によって長さが決まるが、pOpcodeInfoに格納された情報で判別できる。

pOperandInfo1, pOperandInfo2にはそれぞれのオペランド種別が含まれており、命令コードの生成で参照する。

オペランド種別の組み合わせ、オペランドの個数は、コード生成出来ない引数が渡されてはならない。

生成された機械語コードは出力バイナリファイル処理に渡してファイルに書き込む。

## ORG処理

|  |
| --- |
| int ProcORG(struct AssembleInfo\* pInfo, char\* source);  pInfo アセンブル情報  source イミディエイトを指す文字列へのポインタ  戻り値 ITX\_OK / ITX\_ERR |

.org疑似命令はバイナリコードを生成しない特殊な命令である。

この命令語は1つのイミディエイトのオペランドを指定する。

適切なオペランドが検出された場合、アセンブル情報構造体のコード書き込み先アドレスを更新する。

不適切なオペランドはエラーである。

## Data出力処理

|  |
| --- |
| int ProcDATA(struct AssembleInfo\* pInfo, char\* source);  pInfo アセンブル情報  source イミディエイトを指す文字列へのポインタ  戻り値 ITX\_OK / ITX\_ERR |

.data疑似命令は、複数個のイミディエイトをオペランドとして指定する。

行末まで繰り返しトークンを判定し、有効なトークンを検出する都度、出力バイナリファイルにコードを出力する。

トークンがカンマで区切られていない、もしくはイミディエイト以外のトークンを検出した場合は、エラーである。

## 出力バイナリファイル処理

|  |
| --- |
| int WriteCode(struct AssembleInfo\* pInof, unsigned char\* pData, int size);  pInfo アセンブル情報  pData 書き込むデータを示すポインタ  size 書き込むデータのサイズ  戻り値 ITX\_OK / ITX\_ERR |

出力バイナリファイル処理では、機械語変換もしくはData出力処理で生成されたバイナリコードを出力バイナリファイルに書き込む。

まず、バイナリコードの書き込みに先立って、出力先のメモリーに書き込み可能か（他のコードが生成されていないか、アドレスが適切か）どうかをチェックする。

チェックした結果、メモリーに書き込み可能であれば、以下の処理を実行する。

・ファイルポインターをアセンブル情報構造体に記録されているコード書き込み先アドレスの該当位置に変更する

・バイナリデータを書き込む

・アセンブル情報構造体のメモリーアドレスを書き込んだデータ数に応じて更新する

書き込みに失敗したときはエラーである。