(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第5910197号
(P5910197)
(45)発行日　平成28年4月27日(2016.4.27)　(24)登録日　平成28年4月8日(2016.4.8)

(51)Int.Cl.　F I
G06F 9/45 (2006.01)　G06F 9/44 322C

請求項の数 6 (全 15 頁)

(21)出願番号　特願2012-57118(P2012-57118)
(22)出願日　平成24年3月14日(2012.3.14)
(65)公開番号　特開2013-191038(P2013-191038A)
(43)公開日　平成25年9月26日(2013.9.26)
審査請求日　平成27年2月17日(2015.2.17)

(73)特許権者　000006747
株式会社リコー
東京都大田区中馬込１丁目３番６号
(74)代理人　100060690
弁理士　瀧野　秀雄
(72)発明者　岸川　準
東京都大田区中馬込１丁目３番６号　株式会社リコー内

審査官　杉浦　孝光

最終頁に続く

(54)【発明の名称】組み込み用プログラム開発装置、コンパイル方法およびコンパイラプログラム

(57)【特許請求の範囲】
【請求項１】
ソースプログラムを読み込む読み込み手段およびアセンブリ言語コードを生成するアセンブリ言語コード生成手段を備えたコンパイラと、前記アセンブリ言語コードを読み込みオブジェクトファイルを生成するアセンブラと、を有する組み込み用プログラム開発装置において、
前記ソースプログラムが、前記コンパイラにてアセンブリ言語に変換されるコンパイル言語および前記アセンブリ言語で構成されており、
前記コンパイラが、
前記読み込み手段が読み込んだ前記ソースプログラムから前記アセンブリ言語部分を検出するアセンブリ言語検出手段と、
前記アセンブリ言語検出手段が検出した前記アセンブリ言語部分から前記コンパイル言語で宣言されている変数と同じ変数を検出する変数検出手段と、
前記変数検出手段が検出した前記変数について、前記コンパイル言語上と前記アセンブリ言語上でのサイズまたは符号のうち少なくともいずれか一方を比較する比較手段と、
前記コンパイル言語の変数宣言に対する前記アセンブリ言語上の不正な命令と正しい命令の対応テーブルを備え、前記比較手段が比較した結果、前記サイズまたは符号のうち少なくともいずれか一方が異なる場合は、前記対応テーブルに基づいて前記アセンブリ言語を変更する変更手段と、
を備えていることを特徴とする組み込み用プログラム開発装置。

【請求項２】
前記変数が、ポインタ変数であることを特徴とする請求項１に記載の組み込み用プログラム開発装置。
【請求項３】
ソースプログラムからアセンブリ言語コードを生成するコンパイル方法において、
前記ソースプログラムが、アセンブリ言語に変換されるコンパイル言語および前記アセンブリ言語で構成されており、
前記ソースプログラムから前記アセンブリ言語部分を検出するアセンブリ言語検出ステップと、
前記アセンブリ言語検出ステップで検出した前記アセンブリ言語部分から前記コンパイル言語で宣言されている変数と同じ変数を検出する変数検出ステップと、
前記変数検出ステップで検出した前記変数について、前記コンパイル言語上と前記アセンブリ言語上でのサイズまたは符号のうち少なくともいずれか一方を比較する比較ステップと、
前記比較ステップで比較した結果、前記サイズまたは符号のうち少なくともいずれか一方が異なる場合は、前記コンパイル言語の変数宣言に対する前記アセンブリ言語上の不正な命令と正しい命令の対応テーブルに基づいて前記アセンブリ言語を変更する変更ステップと、
を順次実行することを特徴とするコンパイル方法。
【請求項４】
前記変数が、ポインタ変数であることを特徴とする請求項３に記載のコンパイル方法。
【請求項５】
ソースプログラムからアセンブリ言語コードを生成するコンパイラとしてコンピュータを機能させるコンパイラプログラムにおいて、
前記ソースプログラムが、前記コンパイラにてアセンブリ言語に変換されるコンパイル言語および前記アセンブリ言語で構成されており、
前記コンパイラが、
前記ソースプログラムから前記アセンブリ言語部分を検出するアセンブリ言語検出手段と、
前記アセンブリ言語検出手段が検出した前記アセンブリ言語部分から前記コンパイル言語で宣言されている変数と同じ変数を検出する変数検出手段と、
前記変数検出手段が検出した前記変数について、前記コンパイル言語上と前記アセンブリ言語上でのサイズまたは符号のうち少なくともいずれか一方を比較する比較手段と、
前記コンパイル言語の変数宣言に対する前記アセンブリ言語上の不正な命令と正しい命令の対応テーブルを備え、前記比較手段が比較した結果、前記サイズまたは符号のうち少なくともいずれか一方が異なる場合は、前記対応テーブルに基づいて前記アセンブリ言語を変更する変更手段と、
して前記コンピュータを機能させることを特徴とするコンパイラプログラム。
【請求項６】
前記変数が、ポインタ変数であることを特徴とする請求項５に記載のコンパイラプログラム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、ソースプログラムからアセンブリ言語コード生成するコンパイラを備えた組み込み用プログラム開発装置と、コンパイル方法およびコンパイラプログラムに関する。
【背景技術】
【０００２】
従来の組み込み用プログラム開発は、図１３に示すように、例えばホストコンピュータ上でソースファイル（ソースプログラム）をコンパイラによりコンパイルし、アセンブリ

言語コードファイル生成後、アセンブラによりオブジェクトファイルが生成され、そのオブジェクトファイルをリンカによりライブラリや他のオブジェクトファイルなどとの結合処理が行われることで実行形式プログラムが生成されていた。

【０００３】

ここで、コンパイラには、例えばＣ言語などのコンパイル言語上のプログラミングでは冗長なコードになり易い部分を、直接アセンブリ言語記述できるインラインアセンブル機能を持つものがあり、一般的に用いられている。このインラインアセンブルが可能なコンパイラにおいて、インラインアセンブリ言語中のレジスタと、コンパイル言語部分にて展開され使用されるレジスタとの競合のチェックや、よりコード量の少ない、または実行速度が速くなるような最適なレジスタの使用に関するコンパイラの技術は、例えば特許文献１～３に提案されている。

【０００４】

特許文献１には、レジスタで引き渡される仮引数に関しては、仮引数自体をコンパイラ内部で生成した変数に置き換え、さらに、変数へのレジスタ割り付け時に、変数の生存区間がインラインアセンブラルーチンを含む場合、インラインアセンブラルーチンで更新されるレジスタとは異なるレジスタを変数に割り付けることが記載されている。

【０００５】

特許文献２には、インラインアセンブラ内で参照される外部変数、定義される外部変数を指定可能な記述仕様と、高級言語記述部分とインラインアセンブラ部分とのデータ授受をレジスタを介して行なう記述仕様を設けて、この記述仕様の情報を使って、インラインアセンブラに跨がった外部変数に関する最適化によるコード効率向上や、コンパイラ自体の解析コストを軽減することが記載されている。

【０００６】

特許文献３には、プログラム開発装置が、ターゲットプロセッサ上で実行される目的プログラムのための原始プログラム中に、ターゲットプロセッサのレジスタを直接指定している１以上の代入文を含むか否かを解析する構文解析部と、原始プログラムがコンパイルされ割り付けられるレジスタの生存区間情報を検出し、検出した生存区間情報に基づき割り付けられるレジスタが１以上の代入文を跨いで生存しているか否かを検出する検出部とを有して、アセンブリ言語の部分を跨いで生存するレジスタとアセンブリ言語の部分において指定されるレジスタとが実際にバッティングする場合のみ検出することが記載されている。

【発明の概要】

【発明が解決しようとする課題】

【０００７】

上述したインラインアセンブルが可能なコンパイラにおいて、インラインアセンブリ言語（コンパイル言語内のアセンブリ言語）中に使用されるコンパイル言語で宣言された変数の使い方に矛盾があっても、コンパイラはおろか、その後段に実行されるアセンブラでも検出されることはなく、プログラムの不具合の要因となる問題があった。

【０００８】

上述した問題を図１４を参照して詳細に説明する。例えば、インラインアセンブルが可能なＣ言語用コンパイラにおいて、図１４（ａ）のように宣言されたグローバル変数があったとする。この変数をインラインアセンブル記述内で使用する場合に図１４（ｂ）のような記述を行なったとする。ここで、図１４（ｂ）の＃ａｓｍと＃ｅｎｄａｓｍは、両者で挟まれる部分がアセンブリ言語記述であることを示すものであり、Ｃ言語側で宣言された変数ａｂｃは、アセンブリ言語側では先頭に“＿”を付加して、同じ領域を表すものとする。また、ｌｄ．ｗ　＿ａｂｃ，ｇ０は、アドレス＿ａｂｃから１ワード（＝１６ビット）分のデータを対象のプロセッサの１６ビット汎用レジスタであるｇ０レジスタにロードする命令、ｃｍｐ　＃１０，ｇ０は即値１０とレジスタｇ０の内容を比較し、その結果をプロセッサのステータスフラグに設定する命令、ｂｇｔ　ｌａｂｅｌ１０ は、そのステータスフラグの状態に基いて１０より大きかったらｌａｂｅｌ１０に分岐する命令であ

る。
【０００９】
図１４（ａ）のＣ言語側の記述も、図１４（ｂ）のアセンブリ言語側の記述も、文法的に問題はなく、図１３に示すコンパイラや、その後段のアセンブラでもエラーとなることはなく、正常にリンカを通して実行形式プログラムが生成される。しかし、Ｃ言語側で宣言された変数ａｂｃは、ｓｉｇｎｅｄ　ｃｈａｒで宣言されているので８ビットデータであり、確保されるメモリ領域も１バイト分である。アセンブリ言語側では、このアドレスより１ワード分のロードを行っているため、ｇ０レジスタの上位８ビット側には、どのようなデータが入るのかが不定となる。そして、このｇ０レジスタと即値１０とを比較し、その結果により分岐を行っているため、上位８ビットの内容によっては、間違った比較となる可能性がある。これは、プログラムの不具合の元となり、この部分が意図した動作にはならない可能性が出てくる。
【００１０】
あるいは、インラインアセンブル記述部分が、図１４（ｃ）のような記述を行ったとする。ここで、ｌｄ．ｂ　＿ａｂｃ，ｇ０は、アドレス＿ａｂｃから１バイト分のデータをｇ０レジスタの下位８ビットにロードし、上位８ビットは０詰めを行うものである。この時も同様に、コンパイラもアセンブラもエラーとは認識せず、正常に実行形式プログラムが生成される。しかし、変数ａｂｃはｓｉｇｎｅｄであるので、符合付きの８ビットデータとして宣言されているため、データのサイズ指定は正しいものの、ｇ０へのロード時の上位８ビットは、符号ビットで埋められるのが正しい動作となる。これもプログラムの不具合の元となる可能性が出てくる。
【００１１】
正しくは、例えば図１４（ｄ）のような記述となる。ここで、ｌｄ．ｂｓ　＿ａｂｃ，ｇ０ は、アドレス＿ａｂｃから１バイト分のデータをｇ０レジスタの下位８ビットにロードし、上位８ビットは符号拡張を行うものである。
【００１２】
即ち、図１４に例示したように、コンパイラやアセンブラではエラーと認識しないにもかかわらず、インラインアセンブリ言語中の変数の使い方によってはプログラムの不具合となる場合があった。
【００１３】
本発明はかかる問題を解決することを目的としている。
【００１４】
すなわち、本発明は、Ｃ言語などのコンパイル言語で宣言した変数をインラインアセンブリ言語中で使用した際に矛盾が生じないようにすることができる組み込み用プログラム開発装置、コンパイル方法およびコンパイラプログラムを提供することを目的としている。
【課題を解決するための手段】
【００１５】
上記に記載された課題を解決するために請求項１に記載された発明は、ソースプログラムを読み込む読み込み手段およびアセンブリ言語コードを生成するアセンブリ言語コード生成手段を備えたコンパイラと、前記アセンブリ言語コードを読み込みオブジェクトファイルを生成するアセンブラと、を有する組み込み用プログラム開発装置において、前記ソースプログラムが、前記コンパイラにてアセンブリ言語に変換されるコンパイル言語および前記アセンブリ言語で構成されており、前記コンパイラが、前記読み込み手段が読み込んだ前記ソースプログラムから前記アセンブリ言語部分を検出するアセンブリ言語検出手段と、前記アセンブリ言語検出手段が検出した前記アセンブリ言語部分から前記コンパイル言語で宣言されている変数と同じ変数を検出する変数検出手段と、前記変数検出手段が検出した前記変数について、前記コンパイル言語上と前記アセンブリ言語上でのサイズまたは符号のうち少なくともいずれか一方を比較する比較手段と、前記コンパイル言語の変数宣言に対する前記アセンブリ言語上の不正な命令と正しい命令の対応テーブルを備え、

前記比較手段が比較した結果、前記サイズまたは符号のうち少なくともいずれか一方が異なる場合は、前記対応テーブルに基づいて前記アセンブリ言語を変更する変更手段と、を備えていることを特徴とする組み込み用プログラム開発装置である。

【発明の効果】

【００１６】

請求項１に記載の発明によれば、コンパイル言語で宣言されている変数と同じ変数をアセンブリ言語中で使用しているか変数検出手段で検出し、検出された変数のサイズまたは符号のうち少なくともいずれか一方を比較手段で比較し、比較結果が異なる場合は、アセンブリ言語を変更手段で変更するので、インラインアセンブリ言語とコンパイル言語とで矛盾を生じないようにすることができる。

【図面の簡単な説明】

【００１７】

【図１】本発明の第１の実施形態にかかる組み込み用プログラム開発装置の構成図である。

【図２】図１に示された組み込み用プログラム開発装置の機能的な構成を示した構成図である。

【図３】図２に示されたコンパイラの構成を示した構成図である。

【図４】図３に示された構文解析部の構成を示した構成図である。

【図５】図３に示された構文解析部のインラインアセンブリ言語部分の解析動作を示したフローチャートである。

【図６】本実施形態のコンパイラの対象となるプロセッサの命令コードのうち、変数を扱うことのできる命令の動作内容一覧の例を示した表である。

【図７】Ｃ言語命令対応テーブルの例を示した表である。

【図８】正しくないインラインアセンブリ言語部分を持つＣ言語ソースの例である。

【図９】図８に示されたソースを正しい命令に変換した例である。

【図１０】ポインタ変数の場合の正しくないインラインアセンブリ言語部分を持つＣ言語ソースの例である。

【図１１】図１０に示されたソースを正しい命令に変換した例である。

【図１２】ポインタ変数の場合のＣ言語命令対応テーブルの例を示した表である。

【図１３】従来の組み込み用プログラム開発装置の機能的な構成を示した構成図である。

【図１４】コンパイル言語で宣言された変数がインラインアセンブリ言語内で不正に扱われていることを示すコード例である。

【発明を実施するための形態】

【００１８】

以下、本発明の一実施形態を、図１乃至図１２を参照して説明する。図１は、本発明の第１の実施形態にかかる組み込み用プログラム開発装置の構成図である。図２は、図１に示された組み込み用プログラム開発装置の機能的な構成を示した構成図である。図３は、図２に示されたコンパイラの構成を示した構成図である。図４は、図３に示された構文解析部の構成を示した構成図である。図５は、図３に示された構文解析部のインラインアセンブリ言語部分の解析動作を示したフローチャートである。図６は、本実施形態のコンパイラの対象となるプロセッサの命令コードのうち、変数を扱うことのできる命令の動作内容一覧の例を示した表である。図７は、Ｃ言語命令対応テーブルの例を示した表である。図８は、正しくないインラインアセンブリ言語部分を持つＣ言語ソースの例である。図９は、図８に示されたソースを正しい命令に変換した例である。図１０は、ポインタ変数の場合の正しくないインラインアセンブリ言語部分を持つＣ言語ソースの例である。図１１は、図１０に示されたソースを正しい命令に変換した例である。図１２は、ポインタ変数の場合のＣ言語命令対応テーブルの例を示した表である。

【００１９】

図１に示した組み込み用プログラム開発装置１は、コンピュータ装置であり、本体装置７と、各種処理の実行結果を表示する表示部２と、キーボード３と、マウス４等の入力部

と、を含んで構成されている。なお、入力部として示したキーボード３と、マウス４は一例であり、タッチパネルなど他の入力装置であってもよい。本体装置７は、中央処理装置（以下、ＣＰＵという）５と、記憶装置６とを含んで構成されている。記憶装置６には、コンパイラ言語としてＣ言語及びアセンブリ言語により記述されたソースプログラムが記憶されているとともに、コンパイラやアセンブラおよびリンカなどの実行形式プログラムを生成する各ツールも記憶されている。

【００２０】

ユーザは、各種指示をＣＰＵ５に与えるためのキーボード３及びマウス４等の入力部を操作することにより、上述したソースプログラムをコンパイラへの入力として指定することができる。このような組み込み用プログラム開発装置１において上述した各ツールをＣＰＵ５上で実行することにより、ソースプログラムから実行形式プログラムを生成することができる。即ち、各ツールはＣＰＵ５で実行されるコンピュータプログラムとして構成されている。なお、ソースプログラムは図示しない外部記憶装置から入力したりネットワーク経由で外部から入力されるようにしてもよいし、実行形式プログラムも外部記憶装置に出力したりネットワーク経由で外部へ出力されるようにしてもよい。

【００２１】

図２に組み込み用プログラム開発装置１の機能的な構成を示す。組み込み用プログラム開発装置１は、コンパイラ１１と、アセンブラ１２と、リンカ１３と、を備えている。コンパイラ１１は、ソースプログラム２１を読み込んでアセンブリコードファイル（アセンブリ言語コード）２２を生成する。コンパイラは図３に示したように構文解析部１１１と、最適化・資源割付部１１２と、コード生成部１１３と、を備えている。

【００２２】

構文解析部１１１は、入力されるソースプログラムに対し、字句解析、構文解析及び意味解析を行うとともに、Ｃ言語で宣言されている変数がインラインアセンブリ言語（Ｃ言語内のアセンブリ言語部分）中に使用されている場合における矛盾のチェックも行っている。

【００２３】

構文解析部１１１は、図４に示したように解析部１１１ａと、アセンブリ言語検出部１１１ｂと、変数検出部１１１ｃと、比較部１１１ｄと、変更部１１１ｅと、を備えている。読み込み手段としての解析部１１１ａは、入力されるソースプログラム２１を読み込んで、字句解析、構文解析及び意味解析を行う。

【００２４】

アセンブリ言語検出手段としてのアセンブリ言語検出部１１１ｂは、解析部１１１ａの処理が終了したソースプログラム２１について、ソースプログラム２１内のインラインアセンブリ言語部分を検出している。変数検出手段としての変数検出部１１１ｃは、ソースプログラム２１内のインラインアセンブリ言語部分で、コンパイラ言語で宣言されている変数を検出している。比較手段としての比較部１１１ｄは、変数検出部１１１ｃで検出された変数がサイズの扱いと符号の扱いについてコンパイル言語とアセンブリ言語とで相違があるかを検出している。変更手段としての変更部１１１ｅは、比較部１１１ｄでの比較結果で相違があると判断された場合にアセンブリ言語部分の当該箇所の変更を行う。

【００２５】

最適化・資源割付部１１２は、構文解析部１１１の出力に対し、プログラムのコードサイズ削減及び実行速度向上を目的とした最適化処理およびプログラム実行時に割り付けられるレジスタ等のＣＰＵ資源を決定する。

【００２６】

アセンブリ言語コード生成手段としてのコード生成部１１３は、最適化・資源割付部１１２の出力に対し、ターゲットプロセッサが実行可能なアセンブリ命令列に変換し、アセンブリコードファイル２２を生成する。

【００２７】

アセンブラ１２は、アセンブリコードファイル２２を読み込んでオブジェクトファイル

２３を生成する。リンカ１３は、オブジェクトファイル２３や図示しないライブラリなどを読み込んで実行形式プログラム２４を生成する。

【００２８】

なお、ソースプログラム２１、実行形式プログラム２４は記憶装置６に記憶されることは上述したとおりであるが、アセンブリコードファイル２２とオブジェクトファイル２３も必要に応じて記憶装置６に記憶される。

【００２９】

次に、上述した構成の組み込み用プログラム開発装置１において、コンパイラ１１でのインラインアセンブリ言語部分の解析動作を図５のフローチャートを参照して説明する。

【００３０】

まず、アセンブリ言語検出ステップとしてのステップＳ１において、ニーモニックを取り出してステップＳ２に進む。つまり、インラインアセンブリ言語を検出してその中のニーモニックを１つ読み込む。

【００３１】

次に、変数検出ステップとしてのステップＳ２において、Ｃ言語側で宣言された変数を使っている命令か否かを判断し、Ｃ言語側で宣言された変数を使っている命令である場合（Ｙの場合）はステップＳ３に進み、そうでない場合（Ｎの場合）はステップＳ８に進む。即ち、Ｃ言語で宣言されている変数と同じ変数を検出している。本ステップでは、ステップＳ１で取り出したニーモニックに、Ｃ言語側で宣言されている変数が使用されているか否かを走査している（検出している）。通常、インラインアセンブリ言語にて変数を扱うのは、ロード／ストア系命令、アドレス取得命令等であり、インラインアセンブリ命令コードの一部に限られる。したがって、予め対象となる命令を指定しておくことで検出が容易かつ高速に行える。図６に、本実施形態のコンパイラ１１の対象となるプロセッサの命令コードのうち、変数を扱うことのできる命令の動作内容一覧の例を示す。なお、当該プロセッサのバイト順はリトルエンディアンとする。

【００３２】

次に、比較ステップとしてのステップＳ３において、Ｃ言語命令対応テーブルを使って命令の比較を行いステップＳ４に進む。本ステップでは、ステップＳ２で該当する変数が見つかったので、その使用されているニーモニックを解析している。具体的には、その変数に対して、どのようなサイズでアクセスするのか、また符号付きか符号無しかを確認する。そして、その確認結果と、Ｃ言語部分で宣言されている変数のサイズと符号付き／符号無しかの情報を比較する。即ち、Ｃ言語上とインラインアセンブリ言語上でのサイズおよび符号を比較している。

【００３３】

図７に、Ｃ言語による変数宣言に対するインラインアセンブリ言語の命令コードの正しい命令と不正な命令の対応テーブルの例（Ｃ言語命令対応テーブル）を示す。図７の例では、ｃｈａｒ型は８ビット、ｓｈｏｒｔ型は８ビット、ｉｎｔ型は１６ビット、ｌｏｎｇ型は１６ビットであるものとする。また、ｃｈａｒ型はｓｉｇｎｅｄ　ｃｈａｒ型を示すものとする。図７の対応する命令コードや不正な命令コードに記載されているように命令コードを確認すれば、当該変数のサイズと符号付き／符号無しかの情報は確認できる。なお、図７中「不正とは限らない場合有り」と書かれている部分は、プログラマが２バイトの領域から故意に１バイトのアクセスを行う場合があるためである。このＣ言語命令対応テーブルは図４の変更部１１１ｅに含まれるが、比較部１１１ｄからも参照可能となっている。

【００３４】

次に、ステップＳ４において、ステップＳ３で比較した結果が正しい命令か否かを判断し、正しい命令である場合（Ｙの場合）はステップＳ８に進み、正しい命令で無い場合（Ｎの場合）はステップＳ５に進む。即ち、図７に示したＣ言語命令対応テーブルにより当該ニーモニックが正しい命令か否かを判断している。

【００３５】

次に、ステップＳ５において、不正であることを示す、または、その可能性があることを示すワーニングメッセージを出力してステップＳ６に進む。例えば、表示部２にワーニングメッセージが出力される。或いは実行結果のログファイル等にワーニングメッセージを出力するようにしてもよい。

【００３６】

次に、ステップＳ６において、正しい命令に置換するか否かを判断し、置換する場合（Ｙの場合）はステップＳ７に進み、置換しない場合（Ｎの場合）はステップＳ８に進む。本ステップでは、不正であると判断された命令のニーモニックを変更するか否かを選択している。なお、この選択は、コンパイルオプションにてプログラマが行う。

【００３７】

次に、変更ステップとしてのステップＳ７において、Ｃ言語命令対応テーブルを基に正しい命令に変更してステップＳ９に進む。即ち、Ｃ言語命令対応テーブルに基づいてニーモニックを変更する。即ち、比較した結果、サイズまたは符号のうち少なくともいずれか一方が異なる場合は、インラインアセンブリ言語部分を変更している。

【００３８】

一方、ステップＳ８においては、ニーモニック変更なしでステップＳ９に進む。即ち、ステップＳ３で比較した結果が正しい命令であった、または、Ｃ言語側で宣言された変数を使っている命令でなかった、または、ワーニングメッセージのみの出力としたので、ニーモニック変更なしてそのまま出力する。

【００３９】

次に、ステップＳ９において、インラインアセンブル言語部分の終わりか否かを判断し、終わりの場合（Ｙの場合）は最適化・資源割付などの次のコンパイラ処理へ進み、終わりでない場合（Ｎの場合）はステップＳ１に戻る。つまり、次のニーモニックを読み込んで同様の処理を繰り返す。

【００４０】

ここで、上述したコンパイラ１１でのインラインアセンブリ言語部分の解析の具体的な例を説明する。図８は、正しくないインラインアセンブリ言語部分を持つＣ言語ソースの例である。

【００４１】

この例の中で、（ａ）、（ｂ）、（ｃ）、（ｄ）がＣ言語ソース側で宣言した変数をインラインアセンブリ言語記述内で使用しているものである。それぞれ、インラインアセンブリ言語上の問題はなく、アセンブルエラーとなることはない。しかし、Ｃ言語部分で宣言している変数の扱いとは相違がある。

【００４２】

（ａ）は、ｓｉｇｎｅｄ　ｃｈａｒ型の変数に対して、ｌｄ．ｂ、即ち１バイトのデータを２バイトのレジスタｇ０の下位側バイトにロードするが、その上位側バイトは、０詰めされてしまう。これはｕｎｓｉｇｎｅｄ扱いとなり、Ｃ言語の宣言とは異なってしまう。ここは、図７のＣ言語命令対応テーブルにより、

ｌｄ．ｂｓ　＿ｓｃ，ｇ０

として、上位側バイトを符号拡張してロードすることが正しいため、命令変更の対象となり、命令変更（ニーモニック変更）が行われる。

【００４３】

同様に（ｂ）は、ｕｎｓｉｇｎｅｄ　ｃｈａｒ型の変数に対して、ｓｔ、即ち２バイトのレジスタｇ１の書き込みを行っている。これでは、この変数の領域外の部分にデータの書き込みが行われ、その部分もデータを破壊してしまうことになる。ここは、図７のＣ言語命令対応テーブルにより、

ｓｔ．ｂ　ｇ１，＿ｕｃ

として、下位側バイトのみを書き込むことが正しいため、命令変換の対象になり、命令変更（ニーモニック変更）が行われる。

【００４４】

同様に（ｃ）は、ｓｉｇｎｅｄ　ｉｎｔ型の変数に対して、ｌｄ．ｂ、即ち２バイトのデータの１バイト目を２バイトのレジスタｇ２の下位側バイトにロードするが、その上位側バイトは、０詰めされる。これはロードされるサイズが異なってしまっている。ここは、図７のＣ言語命令対応テーブルにより、

ｌｄ．ｗ　＿ｓｉ１,ｇ２

または、

ｌｄ　＿ｓｉ１，ｇ２

として、２バイトのデータをロードすることが正しいと判断され、命令変更の対象になり得る。ただしこの場合、プログラマが意図して１バイト目のみをロードしている可能性もあるため、ワーニングメッセージを出力するだけに留めるだけにしてもよい。なお、変更候補の命令が複数ある場合は、予めデフォルトで置き換える命令を決めておいてもよい。

【００４５】

同様に（ｄ）は、ｕｎｓｉｇｎｅｄ　ｉｎｔ型の変数に対して、ｌｄ．ｂｓ、即ち２バイトのデータの１バイト目を２バイトのレジスタｇ３の下位側バイトにロードし、その上位側バイトは符号拡張される。これもロードされるサイズが異なっており、さらに符号拡張が正しいのかさえ疑われる。ここは、図７のＣ言語命令対応テーブルにより、

ｌｄ．ｗ　＿ｕｉ１,ｇ３

または、

ｌｄ　＿ｕｉ１，ｇ３

として、２バイトのデータをロードすることが正しいと判断され、命令変更の対象になり得る。ただし、（ｃ）の場合と同様に、プログラマが意図して１バイト目のみをロードしている可能性があるため、ここも、ワーニングメッセージを出力するだけに留めるだけにしてもよい。

【００４６】

図８に示したソースを正しい命令に変換した例を図９に示す。ここでは、（ｃ）、（ｄ）も変換を行うものとする。（ａ）、（ｂ）、（ｃ）、（ｄ）はそれぞれ、（ｅ）、（ｆ）、（ｇ）、（ｈ）に変換される。

【００４７】

続いてポインタ変数の場合の例を示す。ポインタ変数の場合、通常そのアドレスがレジスタに書き込まれ、そのレジスタからの間接参照が行われる。図１０にポインタ変数を用いたインラインアセンブリ言語の例を示し、その変換結果の例を図１１に示す。

【００４８】

図１０の（ｉ）、（ｊ）、（ｋ）、（ｌ）は、全て１バイトアクセスを行っており、ポインタとして扱うためにはプロセッサのメモリ領域をアクセスできるように２バイトでのアクセスが必要で有り、これら全てが正しくない。図１０の（ｉ）、（ｊ）、（ｋ）、（ｌ）部分が、それぞれ図１１の（ｍ）、（ｎ）、（ｏ）、（ｐ）に変換される。この変換には、図１２に示すポインタ変数に関するＣ言語命令対応テーブルが用いられる。ただし、ポインタ変数でない場合と同様にプログラマが意図して１バイト目のみをロードしている可能性があるため、ワーニングメッセージを出力するだけに留めるようにしてもよい。即ち、図１０の（ｉ）、（ｊ）、（ｋ）、（ｌ）部分を変換しないことをプログラマが選択できるようにしてもよい。

【００４９】

本実施形態によれば、Ｃ言語で宣言されている変数と同じ変数をインラインアセンブリ言語中で使用しているかを検出し、検出された変数のサイズおよび符号をＣ言語における宣言とインラインアセンブリ言語中とで比較し、比較結果が異なる、つまり不正な命令であると判断された場合は、Ｃ言語命令対応テーブルに基いてインラインアセンブリ言語のニーモニックを変更するので、Ｃ言語記述内のインラインアセンブリ言語内の、変数への不正なアクセスを検出し、必要に応じて、正しい記述へ変換することができ、インラインアセンブリ言語中で矛盾が生じないようにすることができる。

【００５０】

また、正しい命令か不正な命令かは、Ｃ言語命令対応テーブルに基いて判断し、当該Ｃ言語命令対応テーブルに基いてニーモニックを変更するので、予めＣ言語命令対応テーブルを用意することで正しい命令か不正な命令かの判断やニーモニックの変更が容易となる。

【００５１】

また、変数は、ポインタ変数であってもよく、その場合も通常の変数と同様に、Ｃ言語で宣言されている変数と同じ変数をインラインアセンブリ言語中で使用しているかを検出し、検出された変数のサイズおよび符号をＣ言語における宣言とインラインアセンブリ言語中とで比較し、比較結果が異なる、つまり不正な命令であると判断された場合は、Ｃ言語命令対応テーブルに基いてインラインアセンブリ言語のニーモニックを変更するので、Ｃ言語記述内のインラインアセンブリ言語内の、変数への不正なアクセスを検出し、必要に応じて、正しい記述へ変換することができ、インラインアセンブリ言語中で矛盾が生じないようにすることができる。

【００５２】

なお、上述した実施形態では、変数のサイズと符号の両方を比較していたが、いずれか一方のみを比較するようにしてもよい。

【００５３】

また、上述した実施形態では、構文解析部１１１が、アセンブリ言語検出部１１１ｂ、変数検出部１１１ｃ、比較部１１１ｄ、変更部１１１ｅを備えていたが、最適化・資源割付部１１２に備えてもよい。その場合は、アセンブリ言語検出部１１１ｂ、変数検出部１１１ｃ、比較部１１１ｄ、変更部１１１ｅの各処理後に最適化や資源割付処理を行ってもよいし、最適化や資源割付処理後にアセンブリ言語検出部１１１ｂ、変数検出部１１１ｃ、比較部１１１ｄ、変更部１１１ｅの各処理を行ってもよい。或いは、コード生成部１１３のアセンブリコード生成前にアセンブリ言語検出部１１１ｂ、変数検出部１１１ｃ、比較部１１１ｄ、変更部１１１ｅの各処理を行うようにしてもよい。要するにコンパイラ１１が、アセンブリ言語検出部１１１ｂ、変数検出部１１１ｃ、比較部１１１ｄ、変更部１１１ｅを備えていればよい。

【００５４】

なお、本発明は上記実施形態に限定されるものではない。即ち、本発明の骨子を逸脱しない範囲で種々変形して実施することができる。

【符号の説明】

【００５５】

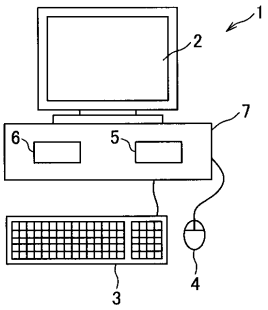
１	組み込み用プログラム開発装置
５	ＣＰＵ
６	記憶装置
１１	コンパイラ
１２	アセンブラ
２１	ソースプログラム
１１１ａ	構文解析部（読み込み手段）
１１１ｂ	アセンブリ言語検出部（アセンブリ言語検出手段）
１１１ｃ	変数検出部（変数検出手段）
１１１ｄ	比較部（比較手段）
１１１ｅ	変更部（変更手段）
１１３	コード生成部（アセンブリ言語コード生成手段）
Ｓ１	ニーモニックを取り出す（アセンブリ言語検出ステップ）
Ｓ２	Ｃ言語側で宣言された変数を使っている命令か（変数検出ステップ）
Ｓ３	Ｃ言語命令対応テーブルを使って命令の比較（比較ステップ）
Ｓ７	Ｃ言語命令対応テーブルを元に正しい命令に変更（変更ステップ）

【先行技術文献】

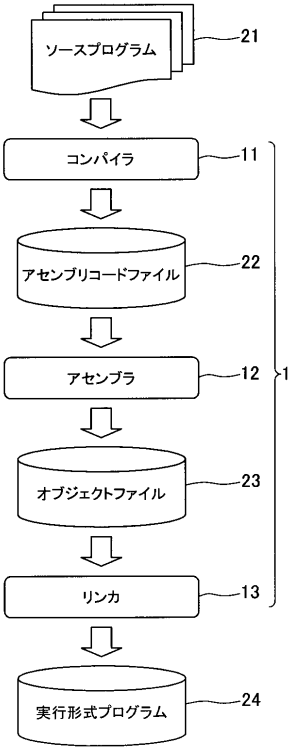
【特許文献】

【００５６】
【特許文献１】特許４０４１２４８号公報
【特許文献２】特許３６３００８６号公報
【特許文献３】特開２００９－２５８７９６号公報

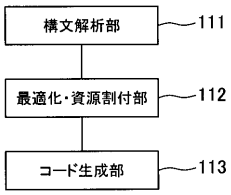
【図１】



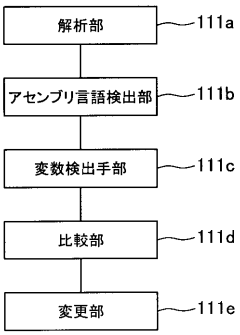
【図２】



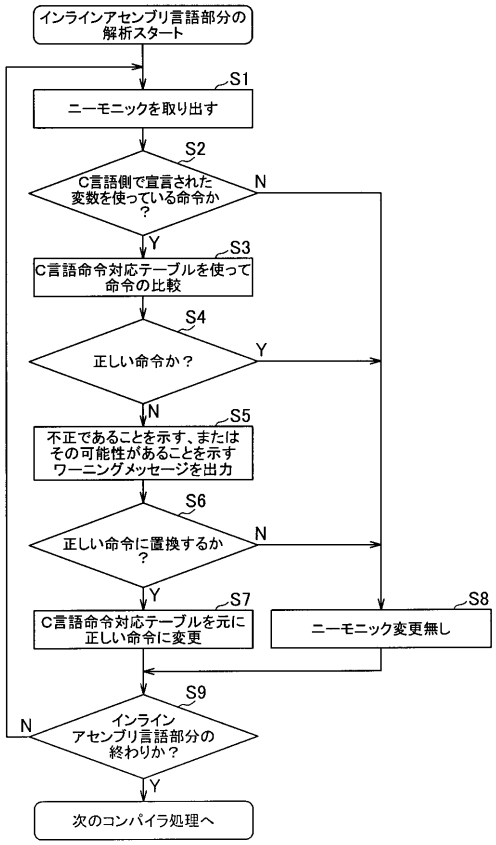
【図３】



【図４】



【図５】



【図６】

表記	動作内容
ld　<dir>,Gd	アドレス<dir>で示すデータRAMから2バイトのデータをGdで示すレジスタに読み出す。ld.wと同値。
ld.b　<dir>,Gd	アドレス<dir>で示すデータRAMから1バイトのデータをGdで示すレジスタに16ビットまでO拡張して読み出す。ld.bzと同値。
ld.bs　<dir>,Gd	アドレス<dir>で示すデータRAMから1バイトのデータをGdで示すレジスタに16ビットまで符号拡張して読み出す。
ld.bz　<dir>,Gd	アドレス<dir>で示すデータRAMから1バイトのデータをGdで示すレジスタに16ビットまでO拡張して読み出す。
ld.w　<dir>,Gd	アドレス<dir>で示すデータRAMから2バイトのデータをGdで示すレジスタに読み出す。
ldi　#dir,Gd	#dirで示すアドレス値をGdで示すレジスタに書き込む。
st　Gs,<dir>	Gsで示すレジスタの内容をアドレスdirとその次のアドレスのデータRAMに2バイト分書き込む。
st.b　Gs,<dir>	Gsで示すレジスタの下位1バイトの内容をアドレスdirで示すデータRAMに書き込む。
st.w　Gs,<dir>	Gsで示すレジスタの内容をアドレスdirとその次のアドレスのデータRAMに2バイト分書き込む。

【図７】

C言語による宣言	対応する命令コード	不正な命令コード
char signed char	ld.bs	ld.b ld.bz ld.w ld
	st.b	st.w st
unsigned char	ld.b ld.bz	ld.bs ld.w ld
	st.b	st.w st
short short int signed short signed short int	ld.bs	ld.b ld.bz ld.w ld
	st.b	st.w st
unsigned short signed short int	ld.b ld.bz	ld.bs ld.w ld
	st.b	st.w st
int signed int signed	ld.w ld	ld.bz ld.bs ld.b ※ただし、不正とは限らない場合有り。
	st.w st	st.b ※ただし、不正とは限らない場合有り。
unsigned int unsigned	ld.w ld	ld.bz ld.bs ld.b ※ただし、不正とは限らない場合有り。
	st.w st	st.b ※ただし、不正とは限らない場合有り。
long long int signed long signed long int	ld.w ld	ld.bz ld.bs ld.b ※ただし、不正とは限らない場合有り。
	st.w st	st.b ※ただし、不正とは限らない場合有り。
unsigned long unsigned long int	ld.w ld	ld.bz ld.bs ld.b ※ただし、不正とは限らない場合有り。
	st.w st	st.b ※ただし、不正とは限らない場合有り。

【図８】

```
signed char     sc1 ;
unsigned char   uc1 ;
signed int      si1 ;
unsigned int    ui1 ;

func()
{
      :
#asm
     ld.b   _sc1,g0          ...(a)
      :
     st     g1,_uc1          ...(b)
      :
     ld.b   _si1,g2          ...(c)
      :
     ld.bs  _ui1,g3          ...(d)
      :
#endasm
      :
}
```

【図９】

```
signed char     sc1 ;
unsigned char   uc1 ;
signed int      si1 ;
unsigned int    ui1 ;

func()
{
      :
#asm
     ld.bs  _sc1,g0          ...(e)
      :
     st.b   g1,_uc1          ...(f)
      :
     ld.w   _si1,g2          ...(g)
      :
     ld.w   _ui1,g3          ...(h)
      :
#endasm
      :
}
```

【図１０】

```
signed char     *psc1 ;
unsigned char   *puc1 ;
signed int      *psi1 ;
unsigned int    *pui1 ;

func()
{
      :
#asm
     ld.b    _psc1,g2        ...(i)
      :
     st.b    g0,_puc1        ...(j)
      :
     ld.bz   _psi1,g2        ...(k)
      :
     ld.bs   _pui1,g2        ...(l)
      :
#endasm
      :
}
```

【図１１】

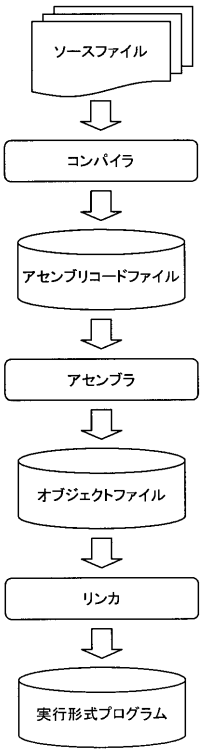
```
signed char     *psc1 ;
unsigned char   *puc1 ;
signed int      *psi1 ;
unsigned int    *pui1 ;

func()
{
      :
#asm
     ld.w    _psc1,g2        ...(m)
      :
     st.w    g0,_puc1        ...(n)
      :
     ld.w    _psi1,g2        ...(o)
      :
     ld.w    _pui1,g2        ...(p)
      :
#endasm
      :
}
```

【図１２】

signed *C言語による宣言	対応する命令コード	不正な命令コード
char * signed char * unsigned char * short * short int * signed short * signed short int * unsigned short * signed short int * int *	ld.w ld	ld.bz ld.bs ld.b ※ただし、不正とは限らない場合有り。
signed int * signed * unsigned int * unsigned * lond * lond int * signed long * signed long int * unsigned long * unsigned long int *	st.w st	st.b ※ただし、不正とは限らない場合有り。

【図１３】



【図１４】

（ａ）

```
signed char  abc ;
```

（ｂ）

```
#asm
        ld.w  _abc, g0
        cmp   #10, g0
        bgt   label10
          :
#endasm
```

（ｃ）

```
#asm
        ld.b  _abc, g0
        cmp   #10, g0
        bgt   label10
          :
#endasm
```

（ｄ）

```
#asm
        ld.bs  _abc, g0
        cmp   #10, g0
        bgt   label10
          :
#endasm
```

フロントページの続き

(56)参考文献 特開昭５８－１６９６３７（ＪＰ，Ａ）
特開２００４－１１８４９４（ＪＰ，Ａ）
特開２００８－０２０９７２（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.，ＤＢ名)
Ｇ０６Ｆ　９／４４　－　９／４５５
Ｇ０６Ｆ　１１／２８　－１１／３４