

(19) 日本国特許庁(JP)

(12) 公開特許公報(A)

(11) 特許出願公開番号

特開2006-79226

(P2006-79226A)

(43) 公開日 平成18年3月23日(2006.3.23)

(51) Int. Cl. F I テーマコード (参考)
G06F 17/50 (2006.01) G06F 17/50 654M 5B046
 G06F 17/50 654A

審査請求 有 請求項の数 17 O L (全 36 頁)

(21) 出願番号 特願2004-260466 (P2004-260466)
 (22) 出願日 平成16年9月8日(2004.9.8)
 (11) 特許番号 特許第3735723号 (P3735723)
 (45) 特許公報発行日 平成18年1月18日(2006.1.18)

(71) 出願人 504160781
 国立大学法人金沢大学
 石川県金沢市角間町ヌ7番地
 (74) 代理人 100094525
 弁理士 土井 健二
 (74) 代理人 100094514
 弁理士 林 恒徳
 (72) 発明者 北川 章夫
 石川県金沢市角間町ヌ7番地 国立大学法人 金沢大学内
 (72) 発明者 尾形 秀範
 石川県金沢市角間町ヌ7番地 国立大学法人 金沢大学内

最終頁に続く

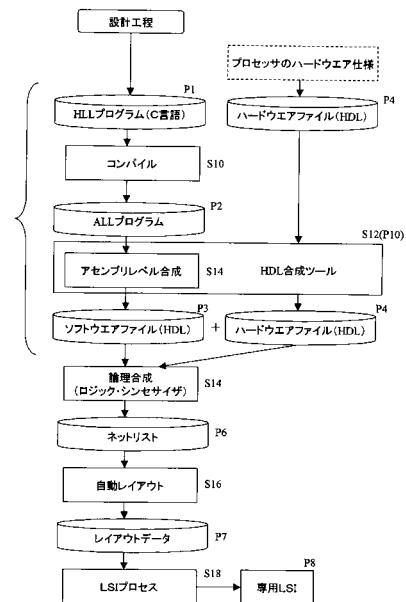
(54) 【発明の名称】 ハードウェア記述言語合成ツール及びそれを利用した集積回路の設計方法

(57) 【要約】

【課題】 C言語などの高位レベル言語で記述されたプログラムから自動的にハードウェア記述言語で記述されたハードウェア設計ファイルを形成することができるハードウェア記述言語合成ツール及びそれを利用した集積回路の設計方法を提供する。

【解決手段】 マイクロプロセッサのプログラムメモリとプログラムコードなどを除くハードウェアコンポーネントを、ハードウェア記述言語で記述したハードウェアファイルと、高級レベル言語プログラムをコンパイルして得られたアセンブリレベル言語プログラムを解析して、その命令コードをハードウェアコンポーネントに対応させてハードウェア記述言語で記述したソフトウェアファイルに変換するアセンブリレベル合成プログラムとを有するハードウェア記述言語合成ツールである。そして、このツールは、ハードウェアファイルとソフトウェアファイルとを、ハードウェア設計ファイルとして出力する。

【選択図】 図1



【特許請求の範囲】**【請求項 1】**

少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリ及びプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルと、

アセンブリレベル合成プログラムとを有し、

前記アセンブリレベル合成プログラムは、

前記マイクロプロセッサが実行可能な高級レベル言語で記述された高級レベル言語プログラムを当該マイクロプロセッサのコンパイラにより変換したアセンブリレベル言語プログラムを、解析する手順と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する手順と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルとを出力する手順とをコンピュータに実行させることを特徴とするハードウェア記述言語合成ツール。

10

【請求項 2】

請求項 1 において、

更に、複数の前記命令コードにそれぞれ対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成し、

当該プロセッサソフトウェアファイルは、前記アセンブリレベル言語プログラムの命令コードに対応する前記プロセッサソフトウェアモジュールを有することを特徴とするハードウェア記述言語合成ツール。

20

【請求項 3】

請求項 2 において、

前記プロセッサソフトウェアモジュールは、前記プログラムカウンタのカウント値が前記アセンブリレベル言語プログラムのプログラムアドレスと一致する時に、当該プログラムアドレスの命令コードに対応する所定の動作を行う制御回路の記述、を有することを特徴とするハードウェア記述言語合成ツール。

30

【請求項 4】

請求項 1 において、

更に、複数の前記命令コードそれぞれに対応して、及び複数の一連の命令コードからなる複合命令コードに対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コード及び複合命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成し、

当該プロセッサソフトウェアファイルは、前記アセンブリレベル言語プログラムの命令コード及び複合命令コードに対応する前記プロセッサソフトウェアモジュールを有することを特徴とするハードウェア記述言語合成ツール。

40

【請求項 5】

請求項 4 において、

前記変換手順は、前記複合命令コードを個別の命令コードに優先して、当該複合命令コードに対応するプロセッサソフトウェアモジュールに変換することを特徴とするハードウェア記述言語合成ツール。

50

【請求項 6】

請求項 1 において、

前記アセンブリレベル合成プログラムは、更に、

前記アセンブリレベル言語プログラムの命令コードを解析して、前記プロセッサハードウェアファイルから、使用されないハードウェアコンポーネントに対応する HDL 記述を削除したプロセッサハードウェアファイルを作成する手順を有し、

前記出力手順は、当該削除されたプロセッサハードウェアファイルを出力することを特徴とするハードウェア記述言語合成ツール。

【請求項 7】

請求項 1 において、

前記プロセッサソフトウェアファイルは、前記プログラムカウンタのカウントが前記アセンブリレベル言語プログラムのプログラムアドレスと一致する時に、当該プログラムアドレスの命令コードに対応する所定の動作を行う制御回路の記述、を有することを特徴とするハードウェア記述言語合成ツール。

10

【請求項 8】

請求項 2 または 4 において、

前記プロセッサハードウェアファイルと前記対応テーブルとを、複数のマイクロプロセッサに対応して、複数組有することを特徴とするハードウェア記述言語合成ツール。

【請求項 9】

請求項 2 または 4 において、

前記プロセッサハードウェアファイルと前記対応テーブルとを、仮想的マイクロプロセッサに対応して有すると共に、

20

更に、前記仮想マイクロプロセッサに対応して前記高級レベル言語プログラムを前記アセンブリレベル言語プログラムに変換する仮想コンパイラプログラムを有することを特徴とするハードウェア記述言語合成ツール。

【請求項 10】

少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリとプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルがあらかじめ生成され、集積回路をハードウェア記述言語で記述したハードウェア記述言語ファイルを生成する集積回路の設計方法において、

30

前記マイクロプロセッサが実行可能な高級レベル言語で記述された高級レベル言語プログラムを、当該マイクロプロセッサのコンパイラによりアセンブリレベル言語プログラムに変換する工程と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する工程と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルを、前記ハードウェア記述言語ファイルとして出力する工程とを有する集積回路の設計方法。

40

【請求項 11】

請求項 10 において、

更に、前記ハードウェア記述言語ファイルを、論理合成ツールにより、前記集積回路のネットリストに変換する工程を有することを特徴とする集積回路の設計方法。

【請求項 12】

請求項 10 において、

更に、複数の前記命令コードにそれぞれ対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルがあらかじめ生成され、

50

前記変換工程は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成する工程を含み、

当該プロセッサソフトウェアファイルは、前記アセンブリレベル言語プログラムの命令コードに対応する前記プロセッサソフトウェアモジュールを複数有することを特徴とする集積回路の設計方法。

【請求項 1 3】

マイクロプロセッサが実行可能な高級レベル言語で記述された高級レベル言語プログラムを、当該マイクロプロセッサのコンパイラにより変換したアセンブリレベル言語プログラムを、解析する手順と、

10

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する手順と、

前記プロセッサソフトウェアファイルを、前記マイクロプロセッサの少なくともプログラムメモリとプログラムデコーダを除くハードウェアコンポーネントをハードウェア記述言語で記述したプロセッサハードウェアファイルと共に、出力する手順とをコンピュータに実行させることを特徴とするハードウェア記述言語合成ツール。

【請求項 1 4】

請求項 1 3 において、

更に、複数の前記命令コードにそれぞれ対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

20

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成することを特徴とするハードウェア記述言語合成ツール。

【請求項 1 5】

請求項 1 3 において、

更に、複数の前記命令コードそれぞれに対応して、及び複数の一連の命令コードからなる複合命令コードに対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

30

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コード及び複合命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成することを特徴とするハードウェア記述言語合成ツール。

【請求項 1 6】

請求項 1 5 において、

前記変換手順は、前記複合命令コードを個別の命令コードに優先して、当該複合命令コードに対応するプロセッサソフトウェアモジュールに変換することを特徴とするハードウェア記述言語合成ツール。

【請求項 1 7】

40

請求項 1 3 において、

前記アセンブリレベル合成プログラムは、更に、

前記アセンブリレベル言語プログラムの命令コードを解析して、前記プロセッサハードウェアファイルから、使用されないハードウェアコンポーネントに対応する HDL 記述を削除したプロセッサハードウェアファイルを作成する手順を有し、

前記出力手順は、当該削除されたプロセッサハードウェアファイルを出力することを特徴とするハードウェア記述言語合成ツール。

【発明の詳細な説明】

【技術分野】

【0001】

50

本発明は、ハードウェア記述言語合成ツール及びそれを利用した集積回路の設計方法に関し、特に、高級レベル言語で記述された高級言語プログラムから集積回路設計用のハードウェア記述言語ファイルを自動的に生成することを可能にするハードウェア記述言語合成ツール及びそれを利用した集積回路の設計方法に関する。

【背景技術】

【0002】

半導体集積回路の設計において、近年、C言語をもとにしたハードウェア記述言語（Hardware Description Language：HDL）によりハードウェア回路を記述し、それを論理合成して集積回路のネットリストを作成し、集積回路を作成することが行われている。それに伴って、Verilog-HDLやVHDLなどの標準化されたHDLが普及してきており、

10

【0003】

ハードウェアを記述するHDLは、レジスタ・トランスファ・レベル（RTL：Register Transfer Level）でハードウェア回路を具体的に記述する独特の記述法が用いられており、ハードウェア回路の知識をもつハードウェア開発者によってハードウェア回路がHDL記述により開発されているのが現状である。一方、一般的なC言語などの高位レベル言語は、HDLにより記述されるハードウェア回路に比較するとより抽象的な記述になりがちであり、かかる抽象度の違いから、ソフトウェアツールによってC言語で記述されたプログラムからHDLで記述されたハードウェアに合成することは、一般的に困難である

20

【0004】

一般的なC言語で記述されたアルゴリズム記述から、HDLによるレジスタ・トランスファ・レベル（RTLレベル）の記述に変換することは一部で提案されている。例えば、特許文献1に示される通りである。この提案によれば、C言語で記述されたアルゴリズム記述を、機能C記述に変換し、更に、この機能C記述に時間の概念であるクロック記述を挿入して、RTLレベルのC記述に変換する。

【特許文献1】特開平2003-16122号公報

【発明の開示】

【発明が解決しようとする課題】

30

【0005】

しかしながら、上記の特許文献1によれば、各変換の段階で必ずしも1対1に対応する変換が行われるわけではなく、変換の前後で記述の抽象度が異なるため、変換前と変換後とで論理的に等価であるか否かを数学的に検証したり、各段階で変換後の記述に対して機能シミュレーションを行ったりすることが必要になり、実用化には適切ではない。

【0006】

そこで、本発明の目的は、C言語などの高位レベル言語で記述されたプログラムから自動的にハードウェア記述言語で記述されたハードウェア設計ファイルを形成することができるハードウェア記述言語合成ツール（プログラム）及びそれを利用した集積回路の設計方法を提供することにある。

40

【課題を解決するための手段】

【0007】

上記目的を達成するために、本発明の第1の側面によれば、（1）マイクロプロセッサのプログラムメモリとプログラムデコーダなどを除くハードウェアコンポーネントを、ハードウェア記述言語で記述したハードウェアファイルと、（2）高級レベル言語プログラムをコンパイルして得られたアセンブリレベル言語プログラムを解析して、その命令コードをハードウェアコンポーネントに対応させてハードウェア記述言語で記述したソフトウェアファイルに変換するアセンブリレベル合成プログラムとを有するハードウェア記述言語合成ツールである。そして、このツールは、ハードウェアファイルとソフトウェアファイルとを、ハードウェア設計ファイルとして出力する。

50

【0008】

上記第1の側面において、好ましい実施例によれば、複数の前記命令コードにそれぞれ対応して、ハードウェア記述言語で記述された複数のソフトウェアモジュールを有する対応テーブルを有し、当該対応テーブルを参照して、アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換してソフトウェアファイルを生成する。

【0009】

更に、上記第1の側面において、好ましい実施例によれば、複数の一連の命令コードからなる複合命令コードに対応して、ハードウェア記述言語で記述されたソフトウェアモジュールを有する対応テーブルを有し、アセンブリレベル言語プログラム内の複合命令コードに対しては、それに対応するソフトウェアモジュールに優先的に変換してソフトウェアファイルを生成する。この複合命令に対応するソフトウェアモジュールは、複合命令を構成する個々の命令コードをそれぞれ対応するソフトウェアモジュールに変換した場合よりも、より最適化されより少ない記述量になっている。

10

【0010】

更に、上記の第1の側面において、好ましい実施例によれば、アセンブリレベル言語プログラムの命令コードを解析して、ハードウェアファイルから、使用されないハードウェアコンポーネントに対応するHDL記述を削除したハードウェアファイルを作成し、当該削除されたハードウェアファイルを出力する。アセンブリレベル言語プログラムの命令コードは比較的高級言語に近いので、その解析は、HDL記述された後のファイルの解析よりも容易に行うことができ、より効率的に使用されないハードウェアコンポーネントを削除することができる。

20

【0011】

上記の目的を達成するために、本発明の第2の側面によれば、少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリ及びプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルと、

アセンブリレベル合成プログラムとを有し、

30

前記アセンブリレベル合成プログラムは、

前記マイクロプロセッサが実行可能な高級レベル言語で記述された高級レベル言語プログラムを当該マイクロプロセッサのコンパイラにより変換したアセンブリレベル言語プログラムを、解析する手順と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する手順と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルとを出力する手順とをコンピュータに実行させることを特徴とするハードウェア記述言語合成ツールである。

40

【0012】

更に、上記目的を達成するために、本発明の第3の側面によれば、少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリとプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルがあらかじめ生成され、集積回路をハードウェア記述言語で記述したハードウェア記述言語ファイルを生成する集積回路の設計方法において、

前記マイクロプロセッサが実行可能な高級レベル言語で記述された高級レベル言語プログラムを、当該マイクロプロセッサのコンパイラによりアセンブリレベル言語プログラム

50

に変換する工程と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する工程と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルを、前記ハードウェア記述言語ファイルとして出力する工程とを有することを特徴とする。

【発明の効果】

【0013】

上記発明の側面によれば、マイクロプロセッサが実行可能な高級レベル言語プログラムを、コンパイラによりアセンブリレベル言語プログラムに変換し、そのアセンブリレベル言語プログラムをマイクロプロセッサのハードウェアコンポーネントに対応させてハードウェア記述言語で記述したプロセッサソフトウェアファイルに変換する。アセンブリレベル言語プログラムの命令コードは、例えばあるレジスタにある定数を加算してあるレジスタに格納するというように、マイクロプロセッサのハードウェアコンポーネントに対応しているため、かかるアセンブリレベル言語プログラムであれば、1対1でハードウェア記述言語で記述したハードウェア設計ファイルに置き換えることができ、従来例のような検証やシミュレーションを行う必要がなく、実用的である。

10

【発明を実施するための最良の形態】

【0014】

以下、図面にしたがって本発明の実施の形態について説明する。但し、本発明の技術的範囲はこれらの実施の形態に限定されず、特許請求の範囲に記載された事項とその均等物まで及ぶものである。

20

【0015】

図1は、本実施の形態における集積回路の設計工程を示すフローチャート図である。本実施の形態では、所定のマイクロプロセッサで実行可能なC言語などの標準的な高級レベル言語（以下HLL：High Level Language）のプログラムP1を、そのマイクロプロセッサのコンパイラによるコンパイル工程（S10）により、アセンブリレベル言語プログラム（以下ALL：Assembly Level Language）P2に変換する。

【0016】

ALLプログラムP2の命令コードは、例えばあるレジスタにメモリ内の定数を加算して別のレジスタに格納するなど、マイクロプロセッサのハードウェアコンポーネントであるレジスタや演算回路に対する制御命令である。したがって、あらかじめマイクロプロセッサのハードウェアコンポーネントについてハードウェア記述言語（HDL）で記述しておけば、そこで宣言されたレジスタやワイヤ（信号）や演算子に対応する制御回路として、ALLプログラムP2の命令コードをHDLで記述することが可能である。しかも、ALLプログラムの命令コードとHDLで記述したハードウェアとは、その抽象度に違いがなく1対1に対応する。

30

【0017】

そこで、本実施の形態におけるHDL合成ツールP10は、そのアセンブリレベル合成工程S14により、コンパイルされたALLプログラムP2の命令コードを、HDLで記述したソフトウェアファイルP3に変換する。この変換についての具体例は、後で詳述する。

40

【0018】

さらに、本実施の形態の設計工程では、所定のマイクロプロセッサの、プログラムメモリとプログラムデコーダを除く、レジスタ群や演算ユニット、それにプログラムカウンタを含むハードウェアコンポーネントを、HDLで記述したハードウェアファイルP4をあらかじめ準備しておき、HDL合成ツールP10が、前述のソフトウェアファイルP3と共に、ハードウェアファイルP4も出力する。つまり、ハードウェアファイルP4は、マイクロプロセッサのソフトウェアの部分を除いたハードウェアコンポーネントをHDLで記述した設計ファイルであり、一方、ソフトウェアファイルP3は、そのハードウェアコ

50

ンポーネントを制御する制御回路等をHDLで記述した設計ファイルに対応する。

【0019】

そこで、本実施の形態の設計方法では、このHDLで記述されたソフトウェアファイルP3とハードウェアファイルP4とをロジック・シンセサイザで論理合成し(S14)、集積回路のネットリストP6を生成する。現在、一部のHDLは標準化されており、その標準化されたHDLからネットリストを生成するロジック・シンセサイザも、前述したとおりすでに実現化されている。したがって、本実施の形態においては、現在普及しているロジック・シンセサイザを利用することで、HDL記述の両ファイルP3、P4をネットリストP6に変換することができる。

【0020】

その後の工程は、通常のLSIの設計工程と同じである。すなわち、レイアウトツールによる自動レイアウト工程S16により、ネットリストP6からレイアウトデータP7が生成され、レイアウトデータP7を利用してLSIプロセスS18により専用の集積回路装置P8が生成される。

【0021】

従来では、ハードウェア技術者がハードウェア回路をHDLで記述して設計し、それを論理合成してネットリストに変換していたが、本実施の形態では、マイクロプロセッサが実行可能なHLLプログラムから、HDL合成ツールP10により自動的にHDLで記述したハードウェア設計ファイルP3を生成することができるので、ハードウェア技術者でないソフトウェア技術者でも、専用LSIを設計することができる。しかも、HLLプログラムからコンパイラによりコンパイルされたALLプログラムの命令コードをHDL記述のハードウェア回路に変換しているため、変換前後で抽象度が同じであり、1対1に対応させて変換でき、従来例のような変換前後の論理の検証を行う必要がない。

【0022】

図2は、本実施の形態におけるHDL合成ツールの構成図である。HDL合成ツールP10は、プロセッサハードウェアファイルP4と、ALLプログラムの命令コードとHDLで記述したハードウェアファイルとの対応テーブルTB12と、対応テーブルTB12を参照してALLプログラムP2をプロセッサソフトウェアファイルP3に変換するALLプログラム変換プログラムP14とで構成される。プロセッサハードウェアファイルP4は、所定のマイクロプロセッサのプログラムメモリ、プログラムデコーダなどのソフトウェアに必要なハードウェアコンポーネントを除いて、例えば、入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタなどのハードウェアコンポーネントについてHDLで記述したファイルである。また、対応テーブルTB12は、上記マイクロプロセッサのコンパイラによりコンパイルされる全ての命令コードそれぞれに対応して、各命令コードをHDLで記述したハードウェア回路のファイル(プロセッサソフトウェアモジュール)を有する。このプロセッサソフトウェアモジュールは、後述するように、ALLの命令コードをプロセッサハードウェアファイルP4のハードウェアコンポーネントに対応して、それらを制御する制御回路をHDLで記述したものである。図2には、ALL命令コード「ADDWF」に対応するHDL記述されたプロセッサソフトウェアモジュールが例示されている。

【0023】

図3は、マイクロプロセッサの一般的構成例を示す図である。このマイクロプロセッサMPは、プログラムカウンタPCと、スタックレジスタSRと、プログラムメモリ10と、命令レジスタ12と、プログラムデコーダ14とを有する。プログラムカウンタPCのプログラムアドレスに対応するプログラム命令コードが、プログラムメモリ10からフェッチされ、命令レジスタ12に格納され、その命令レジスタ内のプログラム命令コードが、プログラムデコーダ14でデコードされ、デコード結果による制御信号等C14が生成される。また、マイクロプロセッサMPは、入出力端子として、クロック入力端子clk_{in}と、メインクリア端子mclrと、入出力データ群Ra、Rb(それぞれ8ビット)とを有し、更に、データメモリ16と、特殊レジスタ18と、定数データメモリ20と、タイマーなど

10

20

30

40

50

の周辺回路 22 とを有する。そして、マイクロプロセッサ MP は、各種の演算器を有する演算ユニット 24 と、演算結果が格納される Wレジスタ 26 とを有する。演算器には、例えば、AND、OR、EOR などの論理回路や、加算器などの演算器が含まれる。

【0024】

入出力端子 Ra、Rb に入力されるデータは、一旦特殊レジスタ 18 に格納され、内部バス BUS を介して演算ユニット 24 に供給され、命令コードに対応した演算が実行され、その演算結果が、Wレジスタ 26 に格納され、または特殊レジスタ 18 の所定のレジスタに格納される。また、所定の処理結果が特殊レジスタ 18 に格納され、入出力端子 Ra、Rb から出力される。プログラムメモリ 10 に格納されている命令コードは、C 言語などの HLL プログラムではなく、それをコンパイルした ALL プログラムである。もちろん、ALL プログラムコードをアセンブラーにより翻訳された機械語コード（ニーモニック）の形態で、プログラムメモリ 10 に格納されている。そして、ALL プログラムの命令コードは、例えば、特殊レジスタの所定のレジスタの値に、定数データメモリの定数を加算して、データメモリ 16 内の所定の領域に格納するなどの形態になっており、プロセッサのレジスタ群や演算器などのハードウェアコンポーネントへの制御に対応するものである。

10

【0025】

更に、マイクロプロセッサ MP におけるタイミングの概念は、主に、プログラムカウンタ PC と、制御クロック clkin とにある。つまり、プログラムカウンタ PC のアドレスに対応する命令コードが、制御クロック clkin に同期して、実行されるからである。そして、命令コードの実行が完了するたびに、プログラムカウンタ PC がインクリメントされ、次の命令コードが同様に実行される。

20

【0026】

図 4 は、本実施の形態において、前述のプロセッサハードウェアファイル P4 内に HDL 記述されるハードウェアコンポーネントの一例を示す図である。図 3 のマイクロプロセッサ MP を構成するハードウェアコンポーネントのうち、プログラムメモリ 10 と、命令レジスタ 12 と、命令デコーダ 14 とを除く、全てのハードウェアコンポーネントが、HDL 記述されたハードウェア設計ファイルとしてのプロセッサハードウェアファイル P4 内に記述される。図 4 の例では、入出力端子 clkin、mclr、Ra、Rb と、データメモリ、特殊レジスタ、定数データメモリ、周辺回路、演算ユニット、Wレジスタ、プログラムカウンタ PC 及びスタックレジスタ SR などが、HDL 記述される。HDL は標準化された言語によるものであり、マイクロプロセッサのハードウェア回路の仕様がわかれば、その回路のコンポーネント（回路要素）を HDL 記述することが可能になる。この具体例については、後述する。

30

【0027】

図 2 にもどり、プロセッサハードウェアファイル P4 と、対応テーブル TB12 とは、所定のマイクロプロセッサ毎にあらかじめ作成され、HDL 合成ツール P10 に添付されている。そして、C 言語などの HLL プログラムが作成されると、対応するマイクロプロセッサのコンパイラにより ALL プログラムに変換され、その ALL プログラム P2 の各命令コードが、HDL 合成ツール P10 の ALL プログラム変換プログラム（ALL 合成ツール）P14 により、逐一、ハードウェア回路を HDL 記述したプロセッサソフトウェアモジュールに置き換えられる。この置き換えは、ALL 命令コードと HDL 記述モジュールとの対応を有する対応テーブル TB12 を参照して行われる。そして、HDL 合成ツール P10 は、プロセッサソフトウェアファイル P3 と、あらかじめ添付されていたプロセッサハードウェアファイル P4 とをハードウェア設計ファイル（P3 + P4）として出力する。

40

【0028】

このように、プロセッサハードウェアファイル P4 と対応テーブル TB12 とは、所定のマイクロプロセッサに対応してあらかじめ作成しておけば、その後の HDL 合成では、繰り返して使用することができる。また、複数のマイクロプロセッサに対応する HLL プ

50

プログラムに対応可能にするためには、その複数のマイクロプロセッサに対応して、プロセッサハードウェアファイルP4と対応テーブルTB12とをあらかじめ生成してツールに添付しておく必要がある。

【0029】

図5は、本実施の形態におけるHDL合成手順を示すフローチャート図である。最初に、HLLプログラムからコンパイラによりコンパイルされているALLプログラムを読み込む(S20)。この読み込まれたALLプログラムからどのマイクロプロセッサに対応するものかを判別し、その判別されたマイクロプロセッサのプロセッサハードウェアファイルP4が出力される(S22)。工程S22では、あらかじめ作成されて添付されているプロセッサハードウェアファイルP4がそのまま出力される。

10

【0030】

次に、HDL合成では、 $i = 0$ として(S24)、アドレス*i*番地のALLプログラム命令コードが解析される。そして、対応テーブルTB12を参照して、その命令コードに対応するプロセッサソフトウェアモジュール(HDL記述ファイル)が選択され、出力される(S28)。そして、全てのALLプログラムの命令コードに対して(S30、S32)、上記の工程S26とS28とが繰り返される。その結果、ALLプログラムの全ての命令コードが、対応するHDL記述のプロセッサソフトウェアモジュールに変換される。その結果、C言語などの標準HLLプログラムが、ハードウェア回路に対応するHDL記述ファイルP3、P4として出力される。上記の手順S20、S24、S26、S28、S30、S32が、アセンブリレベル合成手順S14に対応する。

20

【0031】

図6は、最終的に生成されるLSIの構造図である。本実施の形態のHDL合成ツールにより、マイクロプロセッサのプログラムメモリやプログラムデコーダ等を除くハードウェアコンポーネントが、プロセッサハードウェアファイルP4として出力され、プログラムメモリに格納されているALLプログラムがプロセッサソフトウェアファイルP3として出力される。そして、プロセッサソフトウェアファイルP3は、ハードウェアコンポーネントに対する制御回路30をHDL記述したものである。したがって、これらのハードウェア設計ファイルであるHDLファイルP3、P4から、論理合成、レイアウトにより生成されるLSIは、図6に示されるようになる。つまり、マイクロプロセッサのハードウェアコンポーネントである、入出力端子群clk_{in}、mclr、Ra、Rb、プログラムカウンタPC、スタックレジスタSR、データメモリ16、特殊レジスタ18、定数データメモリ20、周辺回路22、演算ユニット24、Wレジスタ26などに加えて、ソフトウェアファイルP3に対応する制御回路ユニット30で構成される。そして、制御回路ユニット30の制御信号群C30~C36が、前述のハードウェアコンポーネントに供給される。

30

【0032】

以上が、本実施の形態の概略的な説明である。以下、具体例を示して、プロセッサハードウェアファイルP4、対応テーブルTB12、プロセッサソフトウェアファイルP3について詳述する。

【0033】

[プロセッサハードウェアファイルP4]

40

図7~図13は、プロセッサハードウェアファイルP4の一例を示す図である。なお、これらの図には、HDL記述が含まれ、それを説明するコメントがブロック内に示されている。また、VerilogHDLには、レジスタ宣言文reg、ワイヤ(信号)宣言文wire、アサイン文assign、オールウエイズ文alwaysなどが含まれる。

【0034】

ハードウェアファイルP4は、前述したとおり、図4に示されたプロセッサのハードウェアコンポーネントをHDLで記述したものである。図7には、ハードウェアファイルP4に含まれる入出力ポート定義部100と、レジスタ及びワイヤ(信号)の宣言部とデータメモリ設定部106の一部が示される。この例のプロセッサでは、16の入出力端子ra0-ra7、rb0-rb7と、クロック端子clk_{in}、マスタクリア端子mclrとがあり、入出力ポート定義

50

部100には、時間の単位の宣言101と、モジュール名sampleの宣言102と、16個の入出力端子の宣言102と、入力端子の宣言104とが含まれる。

【0035】

プロセッサは、回路の各設定を行う特殊レジスタやデータを格納するデータレジスタなどのレジスタと、レジスタや演算器の出力信号などのワイヤがあり、それらが、レジスタ及びワイヤ宣言部とデータメモリ設定部106で宣言される。即ち、「reg」がレジスタの宣言を、「wire」がワイヤ(信号)の宣言文である。そして、図7に示されるように、13ビットのプログラムカウンタpc_regとクロック信号clkの宣言108と、8ビットのワイヤportain1_wire, portain2_wire, portbin1_wire, Portbin2_wireと、入力ポート部における8ビットのレジスタportout_reg, portbout_reg, option_reg, Trisa_regの宣言110と、割込関連のワイヤ(信号)とレジスタの宣言112とが含まれる。

10

【0036】

図8には、前述のレジスタ及びワイヤ宣言部とデータメモリ設定部106の一部が示され、ここでは、32バイトの8ビットレジスタファイルとしてのram_regの宣言114と、演算ユニット関連のレジスタとワイヤの宣言116とが示されている。演算ユニット関連116では、Wレジスタw_regと、入力レジスタin_regと、加算出力信号addout_node、論理和出力信号andout_nodeなど、スタックレジスタstack_reg及びスタックポインタレジスタstack_pnt_regが宣言されている。

【0037】

図9には、前述のレジスタ及びワイヤ宣言部とデータメモリ設定部106の一部が示され、ここでは、特にデータメモリ設定部118が示される。特殊レジスタ群INDF~INTCONとデータメモリのアドレス空間内の配置が示されている。このデータメモリ設定部118は、前述のレジスタファイルram_regにおけるレジスタの配置を示している。

20

【0038】

図10は、入出力切り替え制御部の記述120であり、それぞれ8ビットの入出力ポートAの記述122と入出力ポートBの記述124とが一部省略して示されている。入出力ポートAの記述122には、3つのassign文が含まれている。Assign文とは、ある条件の下でワイヤ(信号)に他のワイヤや回路の出力を割り付けるものである。即ち、1行目のassign文は、信号portain1_wire[7]を、レジスタtrisa_reg[7]が1ビット「1」の時(trisa_reg[7]==1'b1)はポートra7を割り当てて、それ以外の時はハイインピーダンス(1'bz)を割り当てることを意味する。2行目のassign文も同様である。そして、3行目のassign文では、ポートra7を、レジスタtrisa_reg[7]が1ビット「0」の時(trisa_reg[7]==1'b0)は信号portout_reg[7]を割り当てて、それ以外の時はハイインピーダンス(1'bz)を割り当てることを意味する。このような3つのassign文を含む入出力ポートAの記述122によれば、図14に示すハードウェア回路図が定義される。

30

【0039】

図14に示されたハードウェア回路図は、ゲート回路300、302、304が含まれ、入出力端子Ra7に対する内部信号との関係についての入出力回路を構成する。つまり、ゲート回路300、302、304は、レジスタtrisa_reg[7]の状態に応じて、図10に示した信号を出力とする。つまり、レジスタtrisa_reg[7]が「1」の時は、ゲート回路300は入力ポートRa7を出力し、ゲート回路302はそれを出力し、ゲート回路304は出力をハイインピーダンスにし、レジスタtrisa_reg[7]が「0」の時は、ゲート回路300は出力をハイインピーダンスにし、ゲート回路302は信号portout_reg[7]を出力し、ゲート回路304は信号portout_reg[7]を入出力ポートRa7に出力する。このように、図10の入出力ポートAのHDL記述122によれば、図14のような入出力回路が一義的に定義されるのである。

40

【0040】

図11は、ピン変化割込回路の記述126を示す。この記述は、入出力ポートrb7-rb4、rb0の変化に対応する割込回路を記述するものである。最初に割込信号int0~int7を信号portbin1_wire[0]-[7]にそれぞれ割り当てるアサイン文の記述128が示され、次に、

50

2つのオールウエイズ文alwaysの記述130が示される。この記述130を説明するために、行番号1301～1317を記載している。1301行のalways文は、割込信号int0の立ち上がりエッジ(posedge int0)または内部クリアレジスタintclr_reg0の出力の立ち上がりエッジ(posedge intclr_reg0)が発生した場合に、以下の動作を行うことを記述する。また、1303～1307行では、内部クリアレジスタintclr_reg0が「1」ならレジスタinrise_reg[0]を1ビット「0」(1'b0)にし、それ以外であれば1ビット「1」にすることを記述する。つまり、この記述により、信号int0またはレジスタintclr_reg0の立ち上がりエッジが発生した時に、レジスタintclr_reg0が「0」であれば、その割込を許可してレジスタinrise_reg[0]を「1」に設定し、レジスタintclr_reg0が「1」であれば、その割込を禁止してレジスタinrise_reg[0]を「0」にする回路が定義される。1309～1316行も同様の記述である。

【0041】

そして、1317行では、内部割込信号rb0_intは、割込受付レジスタoption_reg[6]が「1」であれば、レジスタinrise_reg[0]に、「0」であれば、レジスタintdown_reg[0]になることを記述している。3つのassign文の記述140により、特に1403行により最終的な内部割込信号int_nodeを生成する論理回路が記述される。従って、これらの記述128, 130, 140によれば、割込回路が定義される。

【0042】

図12には、各演算器や制御信号の設定の記述142が示される。Assign文144は、プログラムカウンタのインクリメント端子incpc_nodeが、プログラムカウンタレジスタpc_regに13ビットの「000...001」を加算したものであることを記述する。また、演算ユニット内の演算器出力信号の記述146が示される。この記述146の1行目を例にして説明すると、加算器出力信号addout_nodeは、8ビットのWレジスタw_regに9ビット目の「0」(1'b0)を加えた9ビットの値と、8ビットの入力レジスタin_regに9ビット目の「0」(1'b0)を加えた9ビットの値との加算結果であることが記述されている。つまり、加算によるキャリーを考慮して、入力レジスタは9ビット構成になっているのである。各行の//の後にそれぞれの演算論理が示されている。他のアサイン文も同様である。この記述146により、演算ユニット内の複数の演算器が定義される。さらに、記述148は、スリープ信号powerdown_nodeや、スリープ解除信号startclk_nodeや、クロック選択信号selclk_nodeをそれぞれ記述する。

【0043】

演算器の記述146において、comout_nodeは入力レジスタin_regの反転論理、decout_nodeは入力レジスタin_regを-1、incout_nodeは入力レジスタを+1、iorout_nodeは入力レジスタとWレジスタとの論理和、rlfout_nodeは入力レジスタの左シフト、rrfout_nodeは入力レジスタの右シフト、subout_nodeは入力レジスタとWレジスタの除算結果、swapout_nodeは入力レジスタの上位ビットと下位ビットを入れ替えた出力、xorout_nodeは入力レジスタとWレジスタとの排他的論理和、bcfout_nodeは入力レジスタをマスクレジスタbit_regでクリアした出力、bsfout_nodeは入力レジスタをマスクレジスタでセットした出力、addlowout_nodeは入力レジスタとWレジスタの加算下位4ビット出力、sublowout_nodeは入力レジスタとWレジスタの減算下位4ビット出力をそれぞれ示している。

【0044】

図13は、シンクロナイザー、リセット時の動作設定、タイマ、プリスケアラ、割込制御回路の記述150の一部であり、図13には、シンクロナイザーの記述152と、リセット時の動作設定の記述154とが示される。シンクロナイザーの記述152では、クロックclkの立ち上がりエッジ(posedge)が発生すると、マスククリア端子mclrがマスククリアレジスタmclr_sync_regに取り込まれることが記述され、これによりマスククリア端子mclrがクロックclk同期でレジスタに取り込まれる回路が定義される。また、リセット動作設定の記述154は、そのマスククリアレジスタmclr_sync_regが「0」(1'b0)になると、プログラムカウンタレジスタpc_regやステータスレジスタstatus_regなどが所定の値にリセットされることを記述している。

【 0 0 4 5 】

以上が、ハードウェアファイルP4の記述例である。そして、ハードウェアファイルP4の最後に、「end else begin」の記述 1 5 6 が挿入され、これ以降に、ソフトウェアファイルP3の記述が加えられる。

【 0 0 4 6 】

[対応テーブルTB12]

図 1 5 ~ 図 2 3 は、対応テーブルTB12の具体例を示す図である。対応テーブルTB12は、前述のとおり、C言語のプログラムをコンパイルした A L L 記述プログラムの全ての命令

コードと、その命令コードを実行するための制御回路について H D L 記述したファイルとの対応を有する。したがって、図 1 5 ~ 図 2 4 は、命令コードと H D L 記述されたソフトウェアモジュールとの対応を有する。本実施の形態のマイクロプロセッサは、A L L 記述の命令として全部で 3 5 命令を有する。その命令には、1 8 のバイト対応のレジスタファイル命令群 (図 1 5 ~ 図 1 9) と、4 つのビット対応のレジスタファイル命令群 (図 2 0 ~ 図 2 1) と、1 3 のリテラル (定数) 及びコントロール命令群 (図 2 1 ~ 図 2 4) とからなる。

【 0 0 4 7 】

図 1 5 には、バイト対応のレジスタファイル命令 2 0 0 の一つである命令 A D D W F に対応するソフトウェアモジュールの H D L 記述 2 0 2 が示されている。この命令 A D D W F は、ニーモニック (機械言語) では「000111dffffff」であり、Wレジスタとレジスタファイルffffff番地のデータ (データソース) を加算し、dで指定された格納先 (デスティネーション) に保存する命令である。この A D D W F 命令の H D L 記述 2 0 2 は、プログラムカウンタレジスタpc_regが「xxxx」番地の時に (2 0 2 1 行) 、第 1 のクロックサイクル (ステートレジスタstate_regが「0」 (1'b0) のとき) で (2 0 2 2 行) 、データソースであるレジスタファイルffffff番地のデータを入力レジスタin_regに格納し (2 0 2 3 行) 、第 2 のクロックサイクルに移り (ステートレジスタstate_regを「1」 (1'b1) にする) (2 0 2 2 行) 、第 2 のクロックサイクルにおいて、加算器出力の下位 8 ビットaddout_node[7:0]をdで指定された格納先に格納し (2 0 2 6 行) 、プログラムカウンタレジスタpc_regをインクリメントし (2 0 2 7 行) 、加算出力のキャリービットaddout_node[8]などをステータスレジスタstatus_reg[0]などに格納し (2 0 2 8 行) 、加算器出力の下位 8 ビットaddout_node[7:0]がすべて「0」 (1'b0) のときはステータスレジスタstatus_reg[2]を「1」 (1'b1) にしそれ以外では「0」にし (2 0 3 0 ~ 2 0 3 4 行) 、最後に第 1 クロックサイクルに移る (ステートレジスタstate_regを「0」にする) (2 0 3 5 行) 。

【 0 0 4 8 】

なお、データソース部とは、バイト対応のレジスタファイル命令では、データソースとなりうるのは特殊レジスタとデータメモリであり、そのアドレスは、図 9 に示した通りである。また、データディスティネーションとなりうるのは、特殊レジスタとデータメモリ及びWレジスタであり、d = 0 の時はWレジスタ、d = 1 の時は特殊レジスタとデータメモリ (データソースに上書き) になる。

【 0 0 4 9 】

この H D L 記述 2 0 2 において、タイミング制御として、プログラムカウンタpc_regが所定の番地「xxxx」である場合に動作すること (2 0 2 1 行) 、2 つのクロックサイクルに同期してそれぞれ動作すること (ステートレジスタstate_regが「0」のサイクルと、「1」のサイクル) が記述されている。つまり、H L L であるC言語のプログラムをコンパイルした A L L プログラムは、それぞれプログラムアドレスが割り当てられているので、そのプログラムアドレスを処理のタイミングに利用しているのである。

【 0 0 5 0 】

図 1 6 は、対応テーブルの命令 A N D W F に対応するソフトウェアモジュールの H D L 記述 2 0 4 が示されている。この命令 A N D W F は、ニーモニック (機械言語) では「00

0101dfffffff」であり、Wレジスタとレジスタファイルffffffff番地のデータ（データソース）の論理積をとり、dで指定された格納先（デスティネーション）に保存する命令である。この記述204では、図15の記述例と同様に、プログラムアドレスが「xxxx」の時に開始され（2041行）、1クロック目（state_reg=0）でデータソースのデータを入力レジスタin_regに格納し（2043行）、2クロック目（state_reg=1）で論理積出力andout_nodeをデータデスティネーションに格納し（2046行）、プログラムカウンタをインクリメントし（2047行）、演算結果が0であればゼロフラグであるステータスレジスタstate_reg[2]を「1」（1'b1）にし（2048、2049行）、演算結果が0以外であれば「0」（1'b0）にする（2051行）。そして、1クロック目に戻す（2053行）。

10

【0051】

図16の記述204Aは、実際にアセンブリレベル合成されて出力されるソフトウェアモジュール例である。つまり、この例では、プログラムカウンタが「8」になったときに（1行目）、所定の番地のメモリレジスタram_regのデータを入力レジスタin_regに格納し（3行目）、演算結果andout_nodeをWレジスタw_regに格納する例である。このように、後述するALLプログラムのアドレスに対応して対応テーブルのモジュールの「xxxx」が特定され（HDL204の2041行目）、その命令コードに対応してデータソース及びデータデスティネーションが特定され（HDL204の2043行、2046行）、対応テーブル内のHDL記述モジュール204が出力例204Aとして出力される。

【0052】

図17は、対応テーブルにおけるバイト対応のレジスタファイル命令のうち、命令CLRF、CLRW、COMF、DECF、DECF SZに対応するHDL記述のモジュール206、208、210、212を示している。それぞれの命令の説明は、図中に記載されるとおりであり、HDL記述モジュールは、プログラムカウンタのカウント値「xxx x」を条件とするif文と、所定の処理（省略）と、プログラムカウンタをインクリメントする記述（pc_reg<=incpc_node）とが含まれる。

20

【0053】

図18、図19も図17と同様であり、バイト対応のレジスタファイル命令の命令INCF、INCF SZ、IORWF、MOVF、MOVWF、NOP、RLF、RRF、SUBWF、SWAPF、XORWFに対応するHDL記述のモジュール214～232をそれぞれ示している。各HDL記述モジュールは、図17と同様に、所定の制御記述に加えて、プログラムカウンタ値の条件とプログラムカウンタのインクリメントの記述とを有する。

30

【0054】

図20は、ビット対応のレジスタファイル命令204のうち、命令BCFに対応するHDL記述のモジュール236が示されている。命令BCFは、レジスタファイルffffffff番地のデータを別途テーブル235で指定されたbbbに対応する8ビットマスクデータで指定されたビットの値をクリア「0」にする命令である。つまり、データソースのデータを入力レジスタに格納し（2363行）、bbbで指定されたマスクデータをマスクレジスタbit_regに格納し（2364行）、それらの論理積出力bcfout_nodeをデータデスティネーションのレジスタに格納する（2367行）。

40

【0055】

図20には、命令BSF、命令BTFS Cに対応するHDL記述のモジュールファイル238、240が示されている。但し、具体的な部分は省略している。

【0056】

図21には、命令BTFS Sに対応するHDL記述のモジュールファイル242が示されている。また、図21には、定数演算命令（リテラル）とコントロール命令243のうち、命令ADDLW、CLRWD T、ANDLW、CALLLに対応するHDL記述のモジュールファイル244、246、248、250が示される。各命令については、図中の説明に示されている。命令CALLLは、プログラムアドレス「kkkkkkkkkkk」番地のサブ

50

ルーチン呼び出す命令である。したがって、HDL記述のモジュール250には、アドレス「kkkkkkkkkkkk」にPCラッチレジスタの2ビットpclath_reg[4:3]を上位ビットとして加えた13ビットのアドレスが、プログラムカウンタレジスタpc_regに格納され(2502行)、スタックポイントレジスタstack_pnt_regが示すスタックレジスタstack_regに、プログラムカウンタをインクリメントしたアドレス(リターンアドレス)が格納され(2503行)、スタックポイントレジスタが「111」であればそれを「000」に戻して(2504、2505行)、スタックポイントレジスタが「111」以外であれば、単にそのポイント値を+1する(2507行)。

【0057】

図22は、命令GOTO、IORLW、MOVLW、RETURN、RETFIEに対応するHDL記述のモジュール252、254、256、258、260を示し、図23は、命令RETLW、SLEEP、SUBLW、XORLWに対応するHDL記述のモジュール262、264、266、268を示す。例えば、命令GOTOは、プログラムアドレス「kkkkkkkkkkkk」にジャンプする命令であり、HDL記述252には、プログラムカウンタにそのアドレスにPCラッチレジスタの2ビットを上位に加えたアドレスを格納することが記述されている。また、命令RETURNは、サブルーチンから復帰する命令であり、そのHDL記述258には、スタックポイントレジスタの-1のポイントにあるスタックレジスタに格納されたアドレスをプログラムカウンタに格納し(2582行)、スタックポイントレジスタを-1減算する(2583~2586行)ことが記述されている。

【0058】

以上のように、対応テーブルTB12には、全てのALLの命令に対応して、HDL記述されたモジュールファイルが設けられている。そして、後述するとおり、HDL合成ツールが、コンパイルされたALLプログラムに示されたプログラムアドレスと、命令コードと、レジスタファイルアドレスを解析して、命令コードに対応するHDL記述モジュールにプログラムアドレスとレジスタファイルアドレスの情報を反映させて、ソフトウェアファイルP3を出力する。

【0059】

[具体例]

次に、具体的なC言語記述のプログラムに基づき、それがコンパイルされ、コンパイルされたALL記述されたプログラムからHDL合成ツールにより、専用集積回路をHDL記述するハードウェア設計ファイルが生成されることを説明する。

【0060】

図24は、HLLとしてC言語で記述された乗剰余算のプログラムP1を示す図である。このC言語記述プログラムは、 $X \times Y \div P$ の余り(%)を求める演算を実行するものである。まずヘッダファイルが定義され(記述300)、変数としてin1~in3とregがそれぞれ整数として定義され(記述302)、そして、ポートa0(pin_a0)とポートa1(pin_a1)とを共にLレベルに初期化する(記述304)。なお、ポートa0がHレベルで変数X,Y,Pが順次入力され、ポートa1がHレベルで演算結果が出力されることが前提である。

【0061】

また、図24のプログラムは、無限ループ(記述305)として、ポートa0=HにしてXを変数in1として入力し、ポートa0=Lにし(記述306)、ポートa0=HにしてYを変数in2として入力し、ポートa0=Lにし(記述308)、ポートa0=HにしてPを変数in3として入力し、ポートa0=Lにし(記述310)、変数in1に変数in2を乗算し、変数in3で除算し、その余りを変数regに与え(記述312)、そして、ポートa1=Hにして変数regを出力し、ポートa1=Lにする(記述314)。

【0062】

このようなC言語記述のプログラムP1が生成されると、所定のマイクロプロセッサのコンパイラによりコンパイルされ、ALL記述プログラムが生成される。

【0063】

10

20

30

40

50

図 25 ~ 図 29 は、図 24 の C 言語をコンパイルした ALL プログラムの例を示す図である。図 25 に記載されるように、ALL プログラム P2 は、プログラムアドレスの欄 400 と、命令コード欄 402 と、レジスタファイルアドレスの欄 404 とを有する命令コードからなる。つまり、対応ファイル TB12 に含まれる命令コードからなる。図 25 の例では、プログラムアドレス「0000」には、命令コード MOV LW、定数データ 8 ビット「00000000」が記述されている。つまり、定数「00」（= 8 ビットで 00000000）の定数（リテラル）データを W レジスタに移動する命令コードである。以下、同様に複数の命令コードが記述されている。

【0064】

図 30 は、図 25 ~ 29 に示された ALL プログラム P2 をニーモニックコード（機械語コード）に変換した例である。 10

【0065】

図 31 は、図 30 に示した ALL プログラム P2 のニーモニックコードを HDL 合成ツールに与えた結果合成された HDL 記述ファイルの一例を示す図である。つまり、図 1、2 に示したように、HDL 合成ツールは、まずプロセッサのハードウェアコンポーネントを HDL 記述したプロセッサハードウェアファイル P4 を出力し、その最終行を「end else begin」で終了させる。さらに、アセンブリレベル合成ツールである ALL プログラム変換プログラム P14 は、図 30 の ALL プログラムの命令コードを 1 行毎に解析し、対応テーブル TB12 を参照して、各命令コードを HDL 記述されたモジュールに変換し、プロセッサソフトウェアファイル P3 として出力する。図 13 のソフトウェアファイル P 20
3 には、図 25 の ALL プログラムの命令「MOV LW 00」と、命令「MOV WF 0A」とに対応する HDL 記述のモジュールファイルが含まれている。これらのモジュールファイルには、それぞれのプログラムアドレス「0000」「0001」を有する if 文と、データディスティネーションとが含まれている。つまり、図 1 に示したように、共に HDL で記述されたハードウェアファイル P4 とソフトウェアファイル P3 とからなるハードウェア設計ファイルが、図 31 のように生成される。

【0066】

[論理合成]

図 1 に示されるように、この HDL 記述されたファイルは、論理合成ツール（ロジックシンセサイザ）により、ネットリスト P6 に変換される。論理合成は、一般的に次のよう 30
に行われる。論理合成ツールは、HDL 記述されたファイルと、面積や速度などの制約条件とを入力し、加算器や乗算器などのリソースを有するリソースライブラリと、チップ上のセルやマクロを有するテクノロジライブラリを参照して、論理回路を合成し、ネットリストの形態で出力する。この論理合成処理は、例えば、(1) リソース割り当てとリソースシェアリング、(2) レジスタの推定、(3) 状態コードの割り当て、(4) 組み合わせ回路の合成、(5) 論理最適化、(6) テクノロジマッピングの処理からなる。(1) リソース割り当てとリソースシェアリング処理では、HDL 記述中の「+」「*」などの演算子に対して、リソースライブラリに登録されている加算器や乗算器といったハードウェアリソースを割り当て、さらに、HDL 記述を解析し、複数の演算子が同時に使用され 40
ない場合を条件として、それらの演算子に対して共通のハードウェアリソースを割り当てる。(2) レジスタの推定処理では、HDL 記述中のあらかじめ決められたレジスタを表現するパターンに一致した部分をレジスタとして認識する。(3) 状態コード割り当て処理では、状態に対してそれぞれ状態コードを割り当てる。(4) 組み合わせ回路の合成処理では、HDL 記述中の代入文、if 文、case 文、for 文、while 文などの文を論理式に変換する。つまり、ある信号の状態を条件に所定のレジスタのデータが別の信号に割り当てられる場合は、それらを論理回路に対応するように論理式に変換する。そして、(5) 論理最適化処理では、例えば、使用されていない HDL 記述やハードウェアコンポーネントを削除して、無駄なハードウェア回路の発生を回避し、所定の論理式を合成して最適化された論理式に変換するなど、様々な論理式の最適化が行われる。最後に、 50
(6) テクノロジマッピング処理では、最適化された論理式にテクノロジライブラリに登

録されたセルやマクロを割り当てる。このとき、面積や速度などの制約条件を考慮してセルを割り当てる。さらに、レジスタに対してはテクノロジライブラリに登録されているフリップフロップのセルを割り当てる。

【0067】

[変形例1]

図32は、本実施の形態におけるHDL合成ルールの変形例を示す図である。図2に示したHDL合成ツールとの相違点は、複数のマイクロプロセッサ(プロセッサA、プロセッサB)に対応して、プロセッサハードウェアファイルP4(A)、P4(B)、対応テーブルTB12(A)、TB12(B)を有することである。このようにすることにより、それぞれのマイクロプロセッサに対応するALLプログラムP2(A)、P2(B)に対しても、HDL合成することができる。複数のマイクロプロセッサに対応させるためには、それぞれに対応するプロセッサハードウェアファイルと対応テーブルを作成する必要があるが、一度作成すれば、その後繰り返し使用することができる。

10

【0068】

また、別の変形例として、仮想的なマイクロプロセッサを開発し、その仮想的なマイクロプロセッサに対応するコンパイラを開発することで、既存の複数のマイクロプロセッサに対応させて複数のハードウェアファイルや対応テーブルを作成する必要をなくすることができる。つまり、仮想的なマイクロプロセッサを開発することで、それに対応するハードウェアファイルと対応テーブルだけをあらかじめ作成しておけば、HLL記述のプログラムが与えられた時、その仮想的なマイクロプロセッサ用のコンパイラでALLプログラムを作成し、HDL合成することができる。上記のような仮想的なマイクロプロセッサは世の中に存在しないので、本実施の形態のHDL合成ツールが、そのコンパイラも有しておき、HLL記述のプログラムをコンパイルする機能を備えておくことが必要になる。

20

【0069】

[変形例2]

上記実施の形態では、HDL合成手順P10において、あらかじめ作成されているプロセッサハードウェアファイルP4をそのまま出力している。しかしながら、例えば、小規模なHLLプログラムの場合は、全てのハードウェアコンポーネントを使用することがなく、また、全てのALL命令コードを利用したALLプログラムに変換されることもない。あるいは、大規模なHLLプログラムであっても、特定の機能のみしか備えていない場合も、全てのハードウェアコンポーネントを使用することがなく、全てのALL命令コードを利用したALLプログラムに変換されることもない。そのような場合は、使用しないハードウェアコンポーネントを含んだプロセッサハードウェアファイルP4を出力すると、そのファイルが不必要に大規模になり、集積回路装置が不必要に大規模化され、動作速度も遅くなる可能性がある。

30

【0070】

HDLファイルからネットリストに変換する論理合成工程において、このように使用しないハードウェアコンポーネントに対応するHDL記述を削除して最適化することも提案されているが、一旦複雑なHDL記述にされた状態で、不必要な記述を削除することは容易ではない。

40

【0071】

そこで、変形例2では、ALLプログラムに含まれるALL命令コードをチェックし、所定のハードウェアコンポーネントの使用に対応するALL命令コードや、ALL命令コードとレジスタファイルアドレスの組合せが存在しないかどうかをチェックする。そして、もしそのようなALL命令コードや、ALL命令コードとレジスタファイルアドレスの組合せが存在しない場合は、対応するハードウェアコンポーネントに対応するHDL記述を、プロセッサハードウェアファイルP4から削除し、その削除により最適化されたプロセッサハードウェアファイルを出力する。

【0072】

更に、上記実施の形態では、HDL合成手順P10において、各ALL命令コードを対

50

応テーブルTB12を参照してそれぞれ対応するHDLモジュールに変換している。しかしながら、複数の一連のALL命令コードの組合せにおいて、各命令コードで無駄な処理が行われることがある。例えば、ある演算結果をWレジスタに格納するALL命令コードと、そのWレジスタのデータをデータメモリのある番地に格納するALL命令コードとが連続する場合は、Wレジスタへの書き込みと読み出しの処理は無駄な処理である。したがって、それらの命令コードが連続する場合は、Wレジスタへの書き込みと読み出しを省略することで、より最適化することができる。

【0073】

そこで、変形例2では、比較的多く頻出する連続するALL命令コードの組合せからなる複合命令コードに対して、最適化されたHDLモジュールを、対応テーブルTB12に追加する。そして、ALLプログラムをHDL記述のプロセッサソフトウェアファイルに変換する時は、複合命令コードに対しては、最適化されている複合命令に対応するHDLモジュールに変換し、複合命令コードでない場合は、各ALL命令コードに対応するHDLモジュールに変換する。また、複合命令コードに対しては、優先的に対応するHDLモジュールに変換することが望ましい。

10

【0074】

上記の処理を行うことで、変形例2によれば、変換後のHDL記述ファイルであるプロセッサハードウェアファイルP3とプロセッサソフトウェアファイルP4のファイル規模を小さくし、集積回路規模を抑制し、処理スピードを高めることができる。以下、例を示して説明する。

20

【0075】

図33は、変形例2におけるHDL合成手順P10のフローチャート図である。図5と同じ工程には同じ番号を与えている。ALLプログラムの読み込み工程S20の次に、読み込んだALLプログラムを解析して、使用しないハードウェアコンポーネントを検出する。例えば、入出力ポートは、図9に示した特殊レジスタに接続されているので、ALL命令コードのレジスタファイルアドレス欄に特殊レジスタに対応するアドレスがあるか否かをチェックすることで、使用されない特殊レジスタと入出力ポートとを検出することができる。レジスタ・ワイヤー宣言については、ALL命令コードから使用予定の演算器を検出することができ、その結果演算器の出力となるワイヤーが判明するので、不使用のワイヤーを検出することができる。同様に、各演算器の設定に関しても使用しない演算器を検出することができる。データメモリの設定については、ALL命令コードのレジスタファイルアドレス欄をチェックすることで、使用されないデータメモリ領域を特定することができる。

30

【0076】

また、割り込みやタイマー回路については、ALL命令コードのBSF(レジスタファイルの所定ビットを「0」にする命令)のレジスタファイルアドレスが「0b.3」「0b.4」が割り込み発生に対応し、「0b.5」がタイマー回路に対応する。つまり、レジスタファイルアドレス「0b」は、図9のメモリアドレスマップに示したように、割り込みレジスタに対応する。従って、このようなALL命令コード「BSF 0b.3」「BSF 0b.4」「BSF 0b.5」が存在しない場合は、割り込み回路、タイマー回路が必要ないことが判明する。そして、割り込み回路が使用されなければ、リセット時の動作定義も不要になる。

40

【0077】

上記のように、ALL命令コードをチェックすることで、使用しないハードウェアコンポーネントを検出することができるので、その検出した不使用のハードウェアコンポーネントのHDL記述を削除して、プロセッサハードウェアファイルP4を出力する(S22B)。これにより、プロセッサハードウェアファイルP4の記述量を減らすことができる。

【0078】

更に、図33のフローチャートによれば、プロセッサソフトウェアファイルP4を生成

50

するためのALL命令コードからHDLモジュールへの変換において、アドレス*i*番地のALLプログラムが複合命令か否かがチェックされる(S27)。そして、複合命令に該当する場合は、それに対応するHDLモジュールのファイルが出力され(S29)、複合命令に該当しない場合は、図5と同様に各ALL命令コードに対応するHDLモジュールのファイルが出力される(S28)。それ以外の処理は、図5と同じである。

【0079】

図34は、複合命令の一例を示す図である。この例は、ALL命令コードmovlw(あるデータをレジスタw_regに格納する命令)とmovwf(レジスタw_regのデータをデータメモリのある番地に格納する命令)とからなる複合命令500である。この2つの命令が連続する場合、あるデータをレジスタw_regに格納し、その格納したデータをデータメモリのある番地に格納する処理になるので、データを一旦レジスタw_regに格納する処理が無駄になっている。

10

【0080】

図34中、各命令movlwとmovwfをそれぞれ対応テーブルTB12のHDLモジュールに変換した例が、HDL記述502に示されている。前述のとおり、この命令が連続すると、プログラムアドレス「0005」の時に、あるデータ「30」をレジスタw_regに格納し、プログラムカウンタレジスタpc_regをインクリメントし、プログラムアドレス「0006」の時に、レジスタw_reg内のデータをデータメモリ20番地に格納し、プログラムカウンタをインクリメントすることになる。動作完了には2クロックを要している。

【0081】

一方、図34中、複合命令movlw、movwfに対して最適化したHDLモジュール504が示されている。これによれば、プログラムカウンタ「0005」の時に、あるデータ「30」がデータメモリ20番地に格納し、プログラムカウンタを2つインクリメントしている。つまり、レジスタw_regへの格納が省略され、プログラムカウンタ「0006」の判断が削除されている。その結果、HDLモジュールの記述量が減り、1クロックで動作完了している。

20

【0082】

図35、図36は、別の複合命令を示す図である。この複合命令は、ALL命令コードbtfss(データメモリのある番地のあるビットが1なら次の命令をスキップする命令)、goto(あるプログラムアドレスにジャンプする命令)の組合せである。図35中のALL命令コード列506では、命令コードbtfss 30h,3(データメモリ30番地の3ビット目が1なら次の命令をスキップ)と、命令コードgoto 0010(プログラムアドレス0010にジャンプ)と、命令コードgoto 0025(プログラムアドレス0025にジャンプ)とが連続している。従って、命令コードbtfssの結果に応じて、次の命令をジャンプまたは次の命令の実行を行うことなく、プログラムアドレス0010または0025にジャンプすれば、より効率的な処理になる。

30

【0083】

図35中のHDL記述ファイル508は、各命令コードをそれぞれ対応するHDLモジュールに変換した例である。命令コードbtfssでは、プログラムカウンタ「0005」の時に、データメモリ30番地のデータをレジスタin_regに格納し、マスクビット「8'b11110111」をマスクレジスタmask_regに格納し、それらレジスタのOR値の出力ノードbsfout_node「8'b11111111」の場合は、データメモリ30番地のデータの3ビット目が「1」であったことになるので、プログラムカウンタを+2し、それ以外ならプログラムカウンタを+1する。そして、命令コードgotoでは、それぞれのジャンプ先アドレスをプログラムカウンタに格納する。このHDL記述ファイル508によれば、動作完了に3クロックを要し、記述量も多い。

40

【0084】

それに対して、図36には、複合命令btfss,goto,gotoに対して最適化されたHDLモジュール510が示されている。これによれば、プログラムカウンタ「0005」の時に、1クロック目の動作はHDL記述ファイル508と同じであるが、2クロック目では、

50

演算結果ノードbsfout_nodeに依じて、プログラムカウンタを「0025」にするか「0010」にするかの処理が行われて完了している。つまり、2クロックで動作が完了し、その記述量もHDL記述ファイル508より大幅に減少している。

【0085】

上記のように複数のALL命令コードからなる複合命令に対して、最適化されたHDLモジュールを対応テーブルTB12に追加登録し、コンパイルされたALLプログラム中に当該追加登録された複合命令が存在する場合は、それに対応するHDLモジュールに優先的に変換する。登録された複合命令に該当しないALL命令に対しては、従前通りに対応するHDLモジュールに変換すれば良い。このように複合命令を優先して最適化されたHDLモジュールに変換することで、全体のHDL記述ファイルを縮小することができ、動作に必要なクロック数を減らすことができる。

10

【0086】

マイクロプロセッサの場合は、一般論として、ALL命令の数を増やすことは、命令メモリや命令デコーダの回路規模が増大する。例えば、使用頻度の高い複合命令をマクロ命令などで置き換える場合である。そのため、マイクロプロセッサでは、ALL命令の数をむやみに増やすことは好ましくない。しかし、本実施の形態では、最終的な集積回路装置は命令メモリや命令デコーダを有しないので、上記の問題はない。つまり、HDL合成手順の処理の負担が増大しない限りでは、複合命令の登録をできるだけ多くして、より最適化されたHDLモジュールへの変換を可能にすることが望ましい。

【0087】

本変形例2によれば、HDL記述されたプロセッサハードウェアファイルP3からは、使用されないハードウェアコンポーネントに対応するHDL記述が削除され、HDL記述されたプロセッサソフトウェアファイルP4では、複合命令に対しては最適化されたHDLモジュールに変換されているので、その記述数が減少し、使用クロック数も減少している。そして、これらの減少に必要な処理は、HDL合成手順P10の段階で、ALLプログラムの命令コードを解析することにより行われる。したがって、一旦HDL記述に変換されたファイルP3、P4を解析して最適化する従来の論理合成処理よりも、解析処理が簡単で且つ効率的である。最終的に生成される集積回路装置についていえば、図6の回路において、プロセッサハードウェアファイルP3の最適化により制御回路ユニット30以外の回路構成が単純化され、プロセッサソフトウェアファイルP4の最適化により制御回路ユニット30の回路構成が単純化されることになる。

20

30

【0088】

以上のとおり、本実施の形態によれば、C言語などのHLL記述されたプログラムから、自動的にHDL記述のハードウェア設計ファイルを作成することができ、専用のLSIの設計効率を高めることができる。また、HLL記述のプログラムとALL記述のプログラムとの間の検証は不要であり、ALL記述プログラムとHDL記述のハードウェアとの間の抽象度も同じであり、両者の間の検証も不要であり、実用性の高いツールである。

【図面の簡単な説明】

【0089】

【図1】本実施の形態における集積回路の設計工程を示すフローチャート図である。

40

【図2】本実施の形態におけるHDL合成ツールの構成図である。

【図3】マイクロプロセッサの一般的構成例を示す図である。

【図4】本実施の形態において、前述のプロセッサハードウェアファイルP4内にHDL記述されるハードウェアコンポーネントの一例を示す図である。

【図5】本実施の形態におけるHDL合成手順を示すフローチャート図である。

【図6】最終的に生成されるLSIの構造図である。

【図7】プロセッサハードウェアファイルP4の一例を示す図である。

【図8】プロセッサハードウェアファイルP4の一例を示す図である。

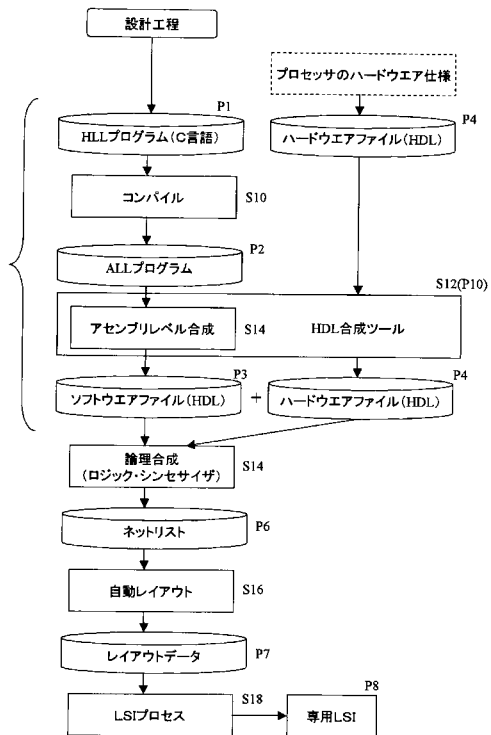
【図9】プロセッサハードウェアファイルP4の一例を示す図である。

【図10】プロセッサハードウェアファイルP4の一例を示す図である。

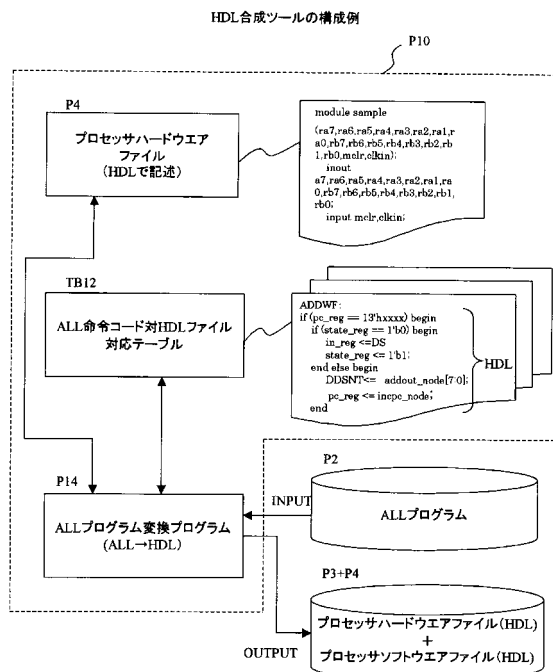
50

- 【図 1 1】プロセッサハードウェアファイル P4 の一例を示す図である。
- 【図 1 2】プロセッサハードウェアファイル P4 の一例を示す図である。
- 【図 1 3】プロセッサハードウェアファイル P4 の一例を示す図である。
- 【図 1 4】プロセッサハードウェアファイル P4 の一例を示す図である。
- 【図 1 5】対応テーブル TB12 の具体例を示す図である。
- 【図 1 6】対応テーブル TB12 の具体例を示す図である。
- 【図 1 7】対応テーブル TB12 の具体例を示す図である。
- 【図 1 8】対応テーブル TB12 の具体例を示す図である。
- 【図 1 9】対応テーブル TB12 の具体例を示す図である。
- 【図 2 0】対応テーブル TB12 の具体例を示す図である。 10
- 【図 2 1】対応テーブル TB12 の具体例を示す図である。
- 【図 2 2】対応テーブル TB12 の具体例を示す図である。
- 【図 2 3】対応テーブル TB12 の具体例を示す図である。
- 【図 2 4】HLL として C 言語で記述された乗剰余算のプログラム P 1 を示す図である。
- 【図 2 5】図 2 4 の C 言語をコンパイルした ALL プログラムの例を示す図である。
- 【図 2 6】図 2 4 の C 言語をコンパイルした ALL プログラムの例を示す図である。
- 【図 2 7】図 2 4 の C 言語をコンパイルした ALL プログラムの例を示す図である。
- 【図 2 8】図 2 4 の C 言語をコンパイルした ALL プログラムの例を示す図である。
- 【図 2 9】図 2 4 の C 言語をコンパイルした ALL プログラムの例を示す図である。
- 【図 3 0】図 2 5 ~ 2 9 に示された ALL プログラム P 2 をニーモニックコード (機械語 20
コード) に変換した例を示す図である。
- 【図 3 1】図 3 0 に示した ALL プログラム P 2 のニーモニックコードを HDL 合成ツ
ールに与えた結果合成された HDL 記述ファイルの一例を示す図である。
- 【図 3 2】本実施の形態における HDL 合成ルールの変形例を示す図である。
- 【図 3 3】変形例 2 における HDL 合成手順 P 1 0 のフローチャート図である。
- 【図 3 4】複合命令の一例を示す図である。
- 【図 3 5】複合命令の別の例を示す図である。
- 【図 3 6】複合命令の一例を示す図である。
- 【符号の説明】
- 【 0 0 9 0】 30
- P 1 : HLL 記述プログラム (C 言語) 、 P 2 : ALL 記述プログラム
- P 3 : プロセッサソフトウェアファイル (HDL)
- P 4 : プロセッサハードウェアファイル (HDL)
- P 1 0 : HDL 合成ツール

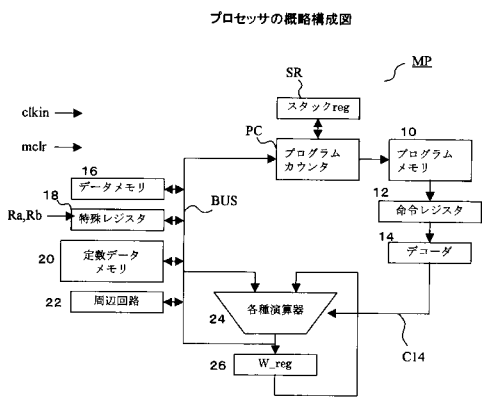
【 図 1 】



【 図 2 】

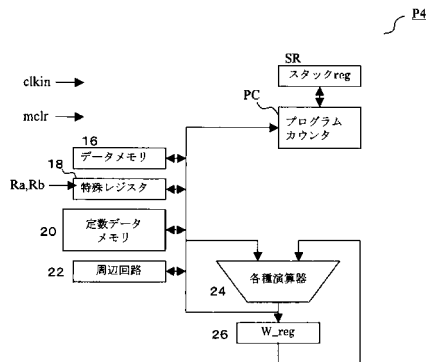


【 図 3 】

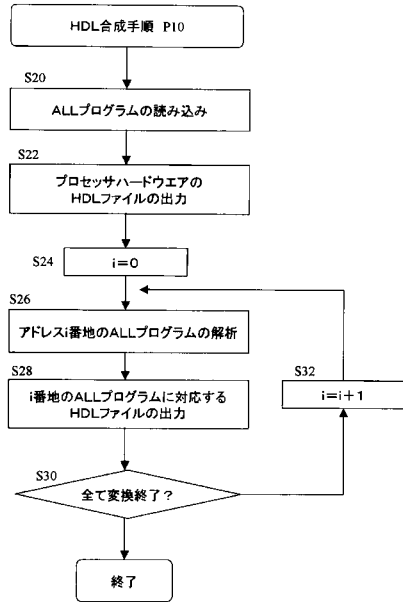


【 図 4 】

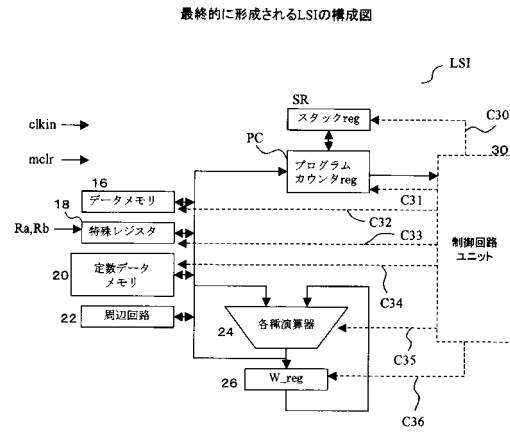
HDLで記述されるプロセッサハードウェアの構成図



【 図 5 】



【 図 6 】



【 図 7 】

ハードウェアファイル P 4 (1)

```

    * 入出力ポート定義部 ~ 100
    `timescale 1ns/1ns } 101

    module sample
      (ra7,ra6,ra5,ra4,ra3,ra2,ra1,ra0,rb7,rb6,rb5,rb4,rb3,rb2,rb1,rb0,mclr,clkln); } 102
      inout ra7,ra6,ra5,ra4,ra3,ra2,ra1,ra0,rb7,rb6,rb5,rb4,rb3,rb2,rb1,rb0; } 104
      input mclr,clkln;

    * レジスタ、ワイヤ宣言部 データメモリ設定部 ~ 106

    reg [12:0] pc_reg; } 108
    wire clk; } プログラムカウンタとクロック信号

    -----
    wire [7:0] portain1_wire; }
    wire [7:0] portain2_wire; }
    reg [7:0] portaout_reg; }
    wire [7:0] portbin1_wire; }
    wire [7:0] portbin2_wire; }
    reg [7:0] portbout_reg; }
    reg [7:0] option_reg; }
    reg [7:0] trisa_reg; }

    wire int0,int4,int5,int6,int7; }
    wire inte,rbint; }
    wire rb0_int,rb4_int,rb5_int,rb6_int,rb7_int; }
    reg [4:0] intrise_reg; }
    reg [4:0] intdown_reg; }
    reg intclr_reg0,intclr_reg1,intclr_reg2,intclr_reg3,intclr_reg4; }
    wire int_node; }
    reg writeintcon_reg; }

    110 入力ポート部の信号とレジスタ
    112 到達関連の信号とレジスタ
  
```

【 図 8 】

ハードウェアファイル P 4 (2)

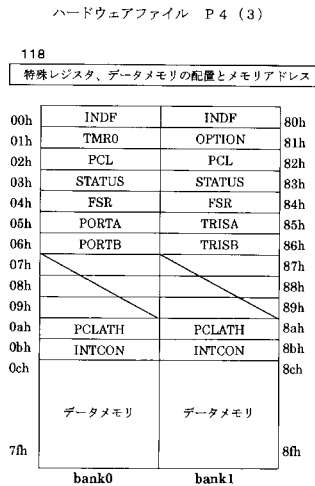
```

    reg [7:0] ram_reg [0:31]; } 114
    } 32ビットの8ビットレジスタファイル

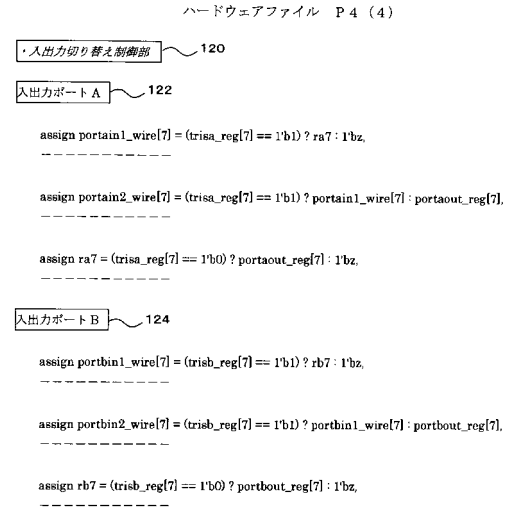
    reg [7:0] w_reg; }
    wire [12:0] incpc_node; }
    reg state_reg; }
    reg [7:0] in_reg; }
    wire [8:0] addout_node; }
    wire [4:0] addlowout_node; }
    wire [7:0] andout_node; }
    wire [7:0] comout_node; }
    wire [7:0] decout_node; }
    wire [7:0] incout_node; }
    wire [7:0] iorout_node; }
    wire [7:0] rlfout_node; }
    wire [7:0] rrfout_node; }
    wire [8:0] subout_node; }
    wire [4:0] sublowout_node; }
    wire [7:0] swapout_node; }
    wire [7:0] xorout_node; }
    wire [7:0] bcfout_node; }
    wire [7:0] bsfout_node; }
    reg [2:0] stack_pnt_reg; }
    reg [12:0] stack_reg [0:7]; }

    116 演算ユニット関連のレジスタとワイヤ
  
```

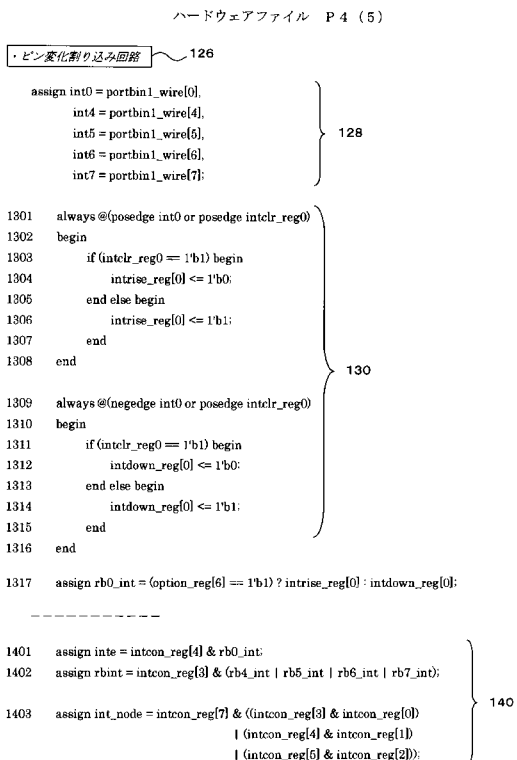
【 図 9 】



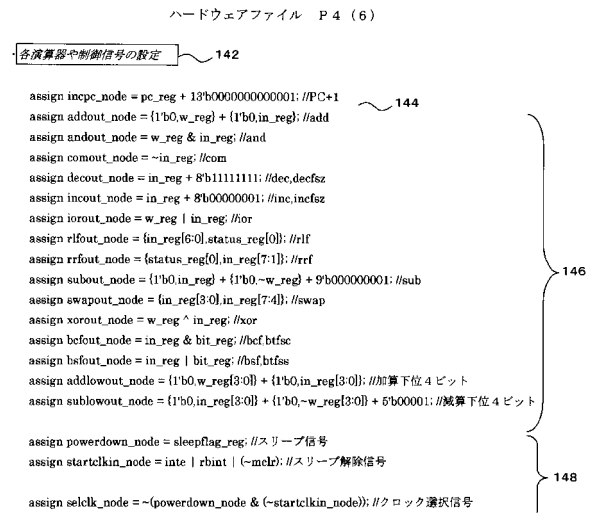
【 図 1 0 】



【 図 1 1 】



【 図 1 2 】



【 図 1 3 】

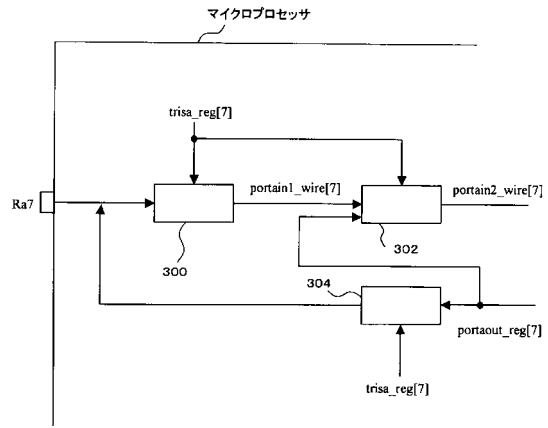
ハードウェアファイル P 4 (7)

```

150 シンクロナイザー、リセット時の動作設定、タイマ、プリスター、割り込み制御回路
always @(posedge clk) begin
    mclr_sync_reg <= mclr; //シンクロナイザー } 152
    if (mclr_sync_reg == 1'b0) begin //リセット
        pc_reg <= 13'h0000000000000000;
        status_reg[7:4] <= 4'b0001;
        status_reg[3] <= ~sleepflag_reg;
        pclath_reg <= 5'b000000;
        intcon_reg <= 8'b00000000;
        option_reg <= 8'b11111111;
        trisa_reg <= 8'b11111111;
        trisb_reg <= 8'b11111111;
        state_reg <= 1'b0;
        stack_pnt_reg <= 3'b000;
        sleepflag_reg <= 1'b0;
        intstart_reg <= 1'b0;
        intclr_reg0 <= 1'b0;
        intclr_reg1 <= 1'b0;
        intclr_reg2 <= 1'b0;
        intclr_reg3 <= 1'b0;
        intclr_reg4 <= 1'b0;
        pscale_reg <= 8'b00000000;
        ps_full_reg <= 1'b1;
        writetmr0_reg <= 1'b0;
        writeintcon_reg <= 1'b0;
    end else begin
154
156
    end else begin ~ 156

```

【 図 1 4 】



【 図 1 5 】

対応テーブル TB 1 2 (1)

全 3 5 命令に対応する HDL ファイル例

1. バイト対応のレジスタファイル命令 (18 種類) 200

```

①命令: ADDWF 一モニック: 000111dfffff クロック数: 2
説明: W_reg とレジスタファイル ffffff 番地のデータを加算し、d で指定された格納先に保存する。
出力 HDL:
2021 if (pc_reg == 13'hxxxx) begin
2022     if (state_reg == 1'b0) begin
2023         in_reg <= データソース;
2024         state_reg <= 1'b1;
2025     end else begin
2026         データディスティネーション <= addout_node[7:0];
2027         pc_reg <= incpc_node;
2028         status_reg[0] <= addout_node[8];
2029         status_reg[1] <= addlowout_node[4];
2030         if (addout_node[7:0] == 8'b00000000) begin
2031             status_reg[2] <= 1'b1;
2032         end else begin
2033             status_reg[2] <= 1'b0;
2034         end
2035         state_reg <= 1'b0;
    end
end
202

```

【 図 1 6 】

対応テーブル TB 1 2 (2)

②命令: ANDWF 一モニック: 000101dfffff クロック: 2
説明: W_reg とレジスタファイル ffffff 番地のデータの論理積をとり、d で指定された格納先に保存する。

```

出力 HDL:
2041 if (pc_reg == 13'hxxxx) begin
2042     if (state_reg == 1'b0) begin
2043         in_reg <= データソース;
2044         state_reg <= 1'b1;
2045     end else begin
2046         データディスティネーション <= andout_node;
2047         pc_reg <= incpc_node;
2048         if (andout_node == 8'b00000000) begin
2049             status_reg[2] <= 1'b1;
2050         end else begin
2051             status_reg[2] <= 1'b0;
2052         end
2053         state_reg <= 1'b0;
    end
end
204

```

```

HDL 出力例
1 if (pc_reg == 13'h0000) begin
2     if (state_reg == 1'b0) begin
3         in_reg <= ram_reg[stare_reg[5],7b0001111];
4         state_reg <= 1'b1;
5     end else begin
6         w_reg <= andout_node;
7         pc_reg <= incpc_node;
8         if (andout_node == 8'b00000000) begin
9             status_reg[2] <= 1'b1;
10        end else begin
11            status_reg[2] <= 1'b0;
12        end
13        state_reg <= 1'b0;
14    end
15 end
204A

```

【 図 1 7 】

対応テーブル TB12 (3)

③命令: CLRIF ニーモニック: 0000011fffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータをゼロクリアする。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  pc_reg <= inipc_node;
end
```

 206

④命令: CLRW ニーモニック: 000010fffff クロック: 1
 説明: W_reg をゼロクリアする。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  pc_reg <= inipc_node;
end
```

 208

⑤命令: COMF ニーモニック: 001001dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータをビット反転し、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 209

⑥命令: DECF ニーモニック: 000011dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータをデクリメント (-1) し、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 210

⑦命令: DECFSZ ニーモニック: 001011dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータをデクリメント (-1) し、d で指定された格納先に保存する。デクリメントした結果が 0 であれば、次命令をスキップする。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 212

【 図 1 8 】

対応テーブル TB12 (4)

⑧命令: INCF ニーモニック: 001010dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータをインクリメント (+1) し、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 214

⑨命令: INCFSZ ニーモニック: 001111dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータをインクリメント (+1) し、d で指定された格納先に保存する。インクリメントした結果が 0 であれば、次命令をスキップする。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 216

⑩命令: IORWF ニーモニック: 000100dfffff クロック: 2
 説明: W_reg とレジスタファイル ffffff 番地のデータの論理和をとり、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 218

⑪命令: MOVE ニーモニック: 001000dfffff クロック: 1
 説明: レジスタファイル ffffff 番地のデータを、d で指定された格納先に移動する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 220

⑫命令: MOVWF ニーモニック: 0000001fffff クロック: 2
 説明: W_reg のデータを、レジスタファイル ffffff 番地に移動する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 220

【 図 1 9 】

対応テーブル TB12 (5)

⑬命令: NOP ニーモニック: 000000000000 クロック: 1
 説明: 何もしない。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  pc_reg <= inipc_node;
end
```

 222

⑭命令: RLF ニーモニック: 001101dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータを左シフトし、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 224

⑮命令: RRF ニーモニック: 001100dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータを右シフトし、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 226

⑯命令: SUBWF ニーモニック: 000010dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータから W_reg を減算し、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 228

⑰命令: SWAPF ニーモニック: 001110dfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータの上位 4 ビットと下位 4 ビットを入れ替え、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 230

⑱命令: XORWF ニーモニック: 000110dfffff クロック: 2
 説明: W_reg とレジスタファイル ffffff 番地のデータの排他的論理和をとり、d で指定された格納先に保存する。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 232

【 図 2 0 】

対応テーブル TB12 (6)

2. ビット対応のレジスタファイル命令 (4 種類)
 ①命令: BCF ニーモニック: 0100bbbfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータの bbb ビットを 0 にする。
 出力 HDL:

```
2361 if (pc_reg == 13'hxxxx) begin
2362   if (state_reg == 1'b0) begin
2363     in_reg <= データソース;
2364     bit_reg <= マスク;
2365     state_reg <= 1'b1;
2366   end else begin
2367     データディスティネーション <= bcfout_node
2368     pc_reg <= inipc_node;
2369     state_reg <= 1'b0;
  end
end
```

 236

bbb	マスク
000	8'b11111110
001	8'b11111101
010	8'b11111011
011	8'b11110111
100	8'b11101111
101	8'b11011111
110	8'b10111111
111	8'b01111111

235

②命令: BSF ニーモニック: 0101bbbfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータの bbb ビットを 1 にする。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 238

③命令: BTFSF ニーモニック: 0110bbbfffff クロック: 2
 説明: レジスタファイル ffffff 番地のデータの bbb ビットが 0 なら次命令をスキップする。
 出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
  -----
  end
```

 240

【 図 2 1 】

対応テーブル TB 1 2 (7)

④命令: BTFFSS ニーモニック: 0111bbbffff クロック: 2
説明: レジスタファイルffff番地のデータのbbbビットが1なら次命令をスキップする。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 242

3. リテラルおよびコントロール命令 (13種類) 243

①命令: ADDLW ニーモニック: 111110kkkkkkkk クロック: 2
説明: W_reg と 8 ビットリテラルデータ kkkkkkkk を加算し、W_reg に保存する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 244

②命令: CLRWDI ニーモニック: 0000001100100 クロック: 1
説明: ウォッチドッグタイマをクリアする。しかし、現在のところウォッチドッグタイマを実装していないのでこの命令は NOP 命令と同様の動作をする。
出力 HDL:

```
if (pc_reg == 13'h0027) begin
    pc_reg <= inpc_node;
    status_reg[3] <= 1'b1;
end
```

 246

③命令: ANDLW ニーモニック: 111001kkkkkkkk クロック: 2
説明: W_reg と 8 ビットリテラルデータ kkkkkkkk の論理積をとり、W_reg に保存する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 248

④命令: CALL ニーモニック: 100kkkkkkkkkk クロック: 1
説明: プログラムアドレス kkkkkkkkkk 番地のサブルーチン呼び出す。
出力 HDL:

```
2501 if (pc_reg == 13'hxxxx) begin
2502     pc_reg <= (pclath_reg[4:3], 11'bkkkkkkkkkk);
2503     stack_reg[stack_ptr_reg] <= inpc_node;
2504     if (stack_ptr_reg == 3'b111) begin
2505         stack_ptr_reg <= 3'b000;
2506     end else begin
2507         stack_ptr_reg <= stack_ptr_reg + 3'b001;
    end
end
```

 250

【 図 2 2 】

対応テーブル TB 1 2 (8)

⑤命令: GOTO ニーモニック: 101kkkkkkkkkk クロック: 1
説明: プログラムアドレス kkkkkkkkkk 番地にジャンプする。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    pc_reg <= (pclath_reg[4:3], 11'bkkkkkkkkkk);
end
```

 252

⑥命令: IORLW ニーモニック: 111000kkkkkkkk クロック: 2
説明: W_reg と 8 ビットリテラルデータ kkkkkkkk の論理和をとり、W_reg に保存する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 254

⑦命令: MOVLW ニーモニック: 110000kkkkkkkk クロック: 1
説明: 8 ビットリテラルデータ kkkkkkkk を W_reg に移動する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    w_reg <= 8'bkkkkkkkk;
    pc_reg <= inpc_node;
end
```

 256

⑧命令: RETURN ニーモニック: 0000000001000 クロック: 1
説明: サブルーチンから無条件復帰する。
出力 HDL:

```
2581 if (pc_reg == 13'hxxxx) begin
2582     pc_reg <= stack_reg[stack_ptr_reg - 3'b001];
2583     if (stack_ptr_reg == 3'b000) begin
2584         stack_ptr_reg <= 3'b111;
2585     end else begin
2586         stack_ptr_reg <= stack_ptr_reg - 3'b001;
    end
end
```

 258

⑨命令: RETFIE ニーモニック: 0000000001001 クロック: 1
説明: 割り込みルーチン復帰する。復帰時に割り込みを許可する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 260

【 図 2 3 】

対応テーブル TB 1 2 (9)

⑩命令: RETLW ニーモニック: 110100kkkkkkkk クロック: 1
説明: サブルーチンから復帰する。復帰時に 8 ビットリテラルデータを W_reg に保存する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 262

⑪命令: SLEEP ニーモニック: 000000011000111 クロック: 1
説明: スリープモードにする。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    sleepflag_reg <= 1'b1;
    pc_reg <= inpc_node;
    status_reg[3] <= 1'b0;
end
```

 264

⑫命令: SUBLW ニーモニック: 111100kkkkkkkk クロック: 2
説明: 8 ビットリテラルデータ kkkkkkkk から W_reg を減算し、W_reg に保存する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 266

⑬命令: XORLW ニーモニック: 111010kkkkkkkk クロック: 2
説明: W_reg と 8 ビットリテラルデータ kkkkkkkk の排他的論理和をとり、W_reg に保存する。
出力 HDL:

```
if (pc_reg == 13'hxxxx) begin
    -----
end
```

 268

【 図 2 4 】

C言語プログラム P 1

```
//乗剰余算 X*Y%P を計算する

#include <16F84.h> //ヘッダーファイルの定義 300

void main() { //main 関数
    int in1; //X
    int in2; //Y
    int in3; //P
    int reg; //X*Y%P の演算結果格納 } 302

    output_low(pin_a0); //high で X,Y,P を順次入力
    output_low(pin_a1); //high で X*Y%P の演算結果を出力 } 304

    while(true){ //無限ループ 305
        output_high(pin_a0); //a0 を high
        in1=input_b0; //X 入力
        output_low(pin_a0); //a0 を low
        output_high(pin_a0); //a0 を high
        in2=input_b0; //Y 入力
        output_low(pin_a0); //a0 を low
        output_high(pin_a0); //a0 を high
        in3=input_b0; //P 入力
        output_low(pin_a0); //a0 を high
        reg=(long)in1*(long)in2%(long)in3; //X*Y%P
        output_high(pin_a1); //a1 を high
        output_b(reg); //演算結果を出力
        output_low(pin_a1); //a1 を low
    }
}
```

【 図 2 5 】

```

ALL プログラム P2(1)
CCS PCM C Compiler, Version 3.188. 22996

Filename: sample.LST

ROM used: 169 (17%)
Largest free fragment is 169
RAM used: 14 (21%) at main0 level
19 (28%) worst case
Stack: 1 locations
400
402
*
0000: MOVLW 00
0001: MOVWF 0A
0002: GOTO 03F
0003: NOP
..... //乗剰余算 X*Y%P を計算する
.....
..... #include <16F84.h> //ヘッダーファイルの定義
..... ##### Standard Header file for the PIC16F84 device #####
..... #device PIC16F84
..... #list
.....
..... void main0 { //main 関数
.....     int in1; //X
0003F: CLRF 04
0040: MOVLW 1F
0041: ANDWF 03F
.....     int in2; //Y
.....     int in3; //P
.....     int reg; //X*Y%P の演算結果格納
.....
.....     output_low(pin_a0); //high で X,Y,P を順次入力
0042: BSF 03.5
0043: BCF 05.0
0044: BCF 03.5

```

【 図 2 6 】

```

P2(2)
0045: BCF 05.0
.....     output_low(pin_a1); //high で X*Y%P の演算結果を出力
0046: BSF 03.5
0047: BCF 05.1
0048: BCF 03.5
0049: BCF 05.1
.....
.....     while(true){ //無限ループ
.....         output_high(pin_a0); //a0 を high
004A: BSF 03.5
004B: BCF 05.0
004C: BCF 03.5
004D: BSF 05.0
.....         in1=input_b0; //X 入力
004E: MOVLW FF
004F: BSF 03.5
0050: MOVWF 06
0051: BCF 03.5
0052: MOVF 06,W
0053: MOVWF 12
.....         output_low(pin_a0); //a0 を low
0054: BSF 03.5
0055: BCF 05.0
0056: BCF 03.5
0057: BCF 05.0
.....         output_high(pin_a0); //a0 を high
0058: BSF 03.5
0059: BCF 05.0
005A: BCF 03.5
005B: BSF 05.0
.....         in2=input_b0; //Y 入力
005C: MOVLW FF
005D: BSF 03.5
005E: MOVWF 06
005F: BCF 03.5
0060: MOVF 06,W

```

【 図 2 7 】

```

P2(3)
0061: MOVWF 13
.....     output_low(pin_a0); //a0 を low
0062: BSF 03.5
0063: BCF 05.0
0064: BCF 03.5
0065: BCF 05.0
.....     output_high(pin_a0); //a0 を high
0066: BSF 03.5
0067: BCF 05.0
0068: BCF 03.5
0069: BSF 05.0
.....     in3=input_b0; //P 入力
006A: MOVLW FF
006B: BSF 03.5
006C: MOVWF 06
006D: BCF 03.5
006E: MOVF 06,W
006F: MOVWF 14
.....     output_low(pin_a0); //a0 を high
0070: BSF 03.5
0071: BCF 05.0
0072: BCF 03.5
0073: BCF 05.0
.....     reg=(long)in1*(long)in2%(long)in3; //X*Y%P
0074: CLRF 17
0075: MOVF 12,W
0076: MOVWF 16
0077: CLRF 0F
0078: MOVF 13,W
0079: MOVWF 18
007A: MOVF 0E,W
007B: MOVWF 19
007C: MOVF 17,W
007D: MOVWF 1B
007E: MOVF 12,W
007F: MOVWF 1A

```

【 図 2 8 】

```

P2(4)
0080: MOVF 0E,W
0081: MOVWF 1D
0082: MOVF 13,W
0083: MOVWF 1C
0084: GOTO 004
0085: MOVF 0E,W
0086: MOVWF 18
0087: MOVF 0D,W
0088: MOVWF 17
0089: CLRF 0F
008A: MOVF 14,W
008B: MOVWF 19
008C: MOVF 0E,W
008D: MOVWF 1A
008E: MOVF 0E,W
008F: MOVWF 1C
0090: MOVF 0D,W
0091: MOVWF 1B
0092: MOVF 0E,W
0093: MOVWF 1E
0094: MOVF 14,W
0095: MOVWF 1D
0096: GOTO 019
0097: MOVF 0C,W
0098: MOVWF 15
.....     output_high(pin_a1); //a1 を high
0099: BSF 03.5
009A: BCF 05.1
009B: BCF 03.5
009C: BSF 05.1
.....     output_b(reg); //演算結果を出力
009D: MOVLW 00
009E: BSF 03.5
009F: MOVWF 06
00A0: BCF 03.5
00A1: MOVF 15,W

```

【図 29】

```

P2(5)
00A2: MOVWF 06
..... output_low(pin_a1): //a1 を low
00A3: BSF 03.5
00A4: BCF 05.1
00A5: BCF 03.5
00A6: BCF 05.1
..... }
00A7: GOTO 04A
..... }
.....
00A8: SLEEP

```

【図 30】

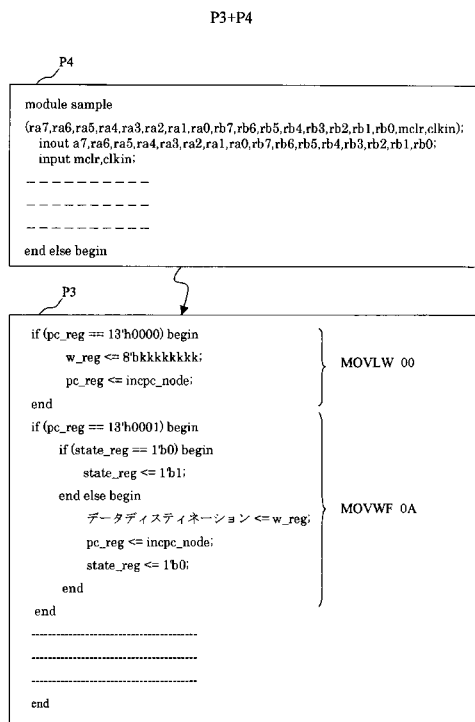
ニーモニックコード

```

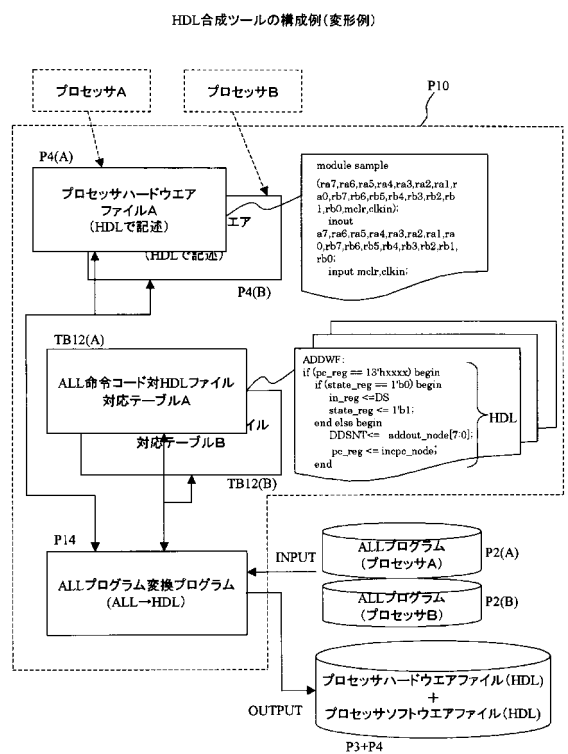
:10000000308A003F28000010308C008F019E01D4
:100010009B0C9A0C031C12281C088F0703189E0ABD
:100020001D089E079E0C8FC8E0C8D0C8C0B0828C7
:1000300085288D018E018C018F011E08031D232848
:100040001D0803193D2810309F0003109B0D9C0DC7
:100050008C0D8F0D1E080F02031D30281D080C0289
:10006000031C39281D088C02031C8F031E088F02F5
:1000700003148D0D8E0D9F0B252800097288401F9
:100080001F30830583160510831205108316851013
:10009000831285108316051083120514FF30831612
:1000A000860083120608920083160510831205103D
:1000B0008316051083120514FF3083168600831201
:1000C0000608930083160510831205108316051089
:1000D00083120514FF3083168600831206089400ED
:1000E00083160510831205109701120896008F01E0
:1000F00013089800F08990017089B0012089A002F
:10010000F089D0013089C0004280E0898000D0895
:1001100097008F01140899000F089A000E089C00A0
:10012000D089B000F089E0014089D0019280C085C
:10013000950083168510831285140030831686007F
:1001400083121508860083168510831285104A28AD
:0201500063004A
:00000001FF
:PIC16F84

```

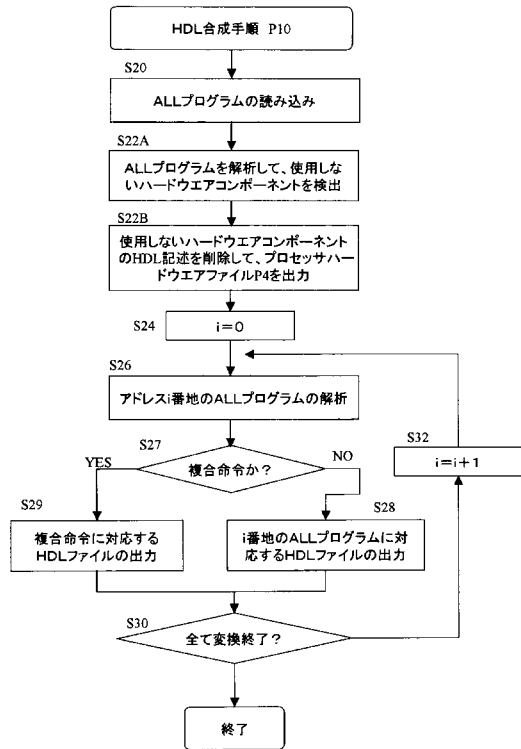
【図 31】



【図 32】



【 図 3 3 】



【 図 3 4 】

```

複合命令 movlw, movwf
0005    movlw 30h    //16進数の30をw_regに代入
0006    movwf reg[20] //w_regの値をデータメモリの20番地に代入
  
```

```

各命令 movlw, movwf をそれぞれの HDL モジュールに変換した例
.....
end else if (pc_reg == 0005) begin //プログラムアドレス5番地のときの処理
  w_reg <= 8'h30; //定数30をw_regに代入
  pc_reg <= incpc_node; //プログラムカウンタをインクリメント
end else if (pc_reg == 0006) begin //プログラムアドレス6番地のときの処理
  ram_reg[20] <= w_reg; //w_regをデータメモリ20番地に代入
  pc_reg <= incpc_node; //プログラムカウンタをインクリメント
end else if (pc_reg == 0007) begin //プログラムアドレス7番地のときの処理
  .....
  
```

```

複合命令 movlw, movwf を最適化された HDL モジュールに変換した例
.....
end else if (pc_reg == 0005) begin //プログラムアドレス5番地のときの処理
  ram_reg[20] <= 8'h30; //定数30をw_regに代入
  pc_reg <= incpc2_node; //プログラムカウンタを+2する
end else if (pc_reg == 0007) begin //プログラムアドレス7番地のときの処理
  .....
  
```

【 図 3 5 】

```

複合命令 btfs, goto, goto
0005    btfs 30h,3   データメモリ30番地の3ビット目が1ならば
                                次命令をスキップ
0006    goto 0010   プログラムアドレス0010番地にジャンプ
0007    goto 0025   プログラムアドレス0025番地にジャンプ
  
```

```

各命令 btfs, goto, goto をそれぞれの HDL モジュールに変換した例
.....
end else if (pc_reg == 0005) begin //プログラムアドレス5番地の処理
  if (state_reg == 1'b0) begin //1クロック目の動作
    in_reg <= ram_reg[30]; //データメモリ30番地のデータをin_regに代入
    mask_reg <= 8'b11110111; //マスクビットをmask_regに代入
    state_reg <= 1'b1; //state_regに1を代入
  end else begin //2クロック目の動作
    if (bsfout_node == 8'b11111111) begin //演算結果が8'b11111111ならば
      pc_reg <= incpc2_node; //次命令をスキップするためにPCを+2
    end else begin //演算結果が8'b11111111以外ならば
      pc_reg <= incpc_node; //PCをインクリメント
    end
    state_reg <= 1'b0; //state_regに0を代入
  end
end else if (pc_reg == 0006) begin //プログラムアドレス6番地の処理
  pc_reg <= 0010; //プログラムカウンタに10を代入
end else if (pc_reg == 0007) begin //プログラムカウンタ7番地の処理
  pc_reg <= 0025; //プログラムカウンタに25を代入
end else if (pc_reg == 0008) begin //プログラムカウンタ8番地の処理
  .....
  
```

【 図 3 6 】

```

複合命令 btfs, goto, goto を最適化された HDL モジュールに変換した例
.....
end else if (pc_reg == 0005) begin //プログラムカウンタ5番地の処理
  if (state_reg == 1'b0) begin //1クロック目の動作
    in_reg <= ram_reg[30]; //データメモリ30番地のデータをin_regに代入
    mask_reg <= 8'b11110111; //マスクビットをmask_regに代入
    state_reg <= 1'b1; //state_regに1を代入
  end else begin //2クロック目の動作
    if (bsfout_node == 8'b11111111) begin //演算結果が8'b11111111ならば
      pc_reg <= 0025; //プログラムカウンタに25を代入
    end else begin //演算結果が8'b11111111以外ならば
      pc_reg <= 0010; //プログラムカウンタに10を代入
    end
    state_reg <= 1'b0; //state_regに0を代入
  end
end else if (pc_reg == 0008) begin //プログラムカウンタ8番地の処理
  .....
  
```

【手続補正書】

【提出日】平成17年8月5日(2005.8.5)

【手続補正1】

【補正対象書類名】特許請求の範囲

【補正対象項目名】全文

【補正方法】変更

【補正の内容】

【特許請求の範囲】

【請求項1】

少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリ及びプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルと、

アセンブリレベル合成プログラムとを有し、

前記アセンブリレベル合成プログラムは、

高位レベル言語で記述された高位レベル言語プログラムを当該マイクロプロセッサに対応するコンパイラにより変換した前記マイクロプロセッサが実行可能なアセンブリレベル言語プログラムの命令コードを、解析する手順と、

前記解析したアセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する手順と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルとを出力する手順とをコンピュータに実行させることを特徴とするハードウェア記述言語合成プログラム。

【請求項2】

請求項1において、

更に、複数の前記命令コードにそれぞれ対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成し、

当該プロセッサソフトウェアファイルは、前記アセンブリレベル言語プログラムの命令コードに対応する前記プロセッサソフトウェアモジュールを有することを特徴とするハードウェア記述言語合成プログラム。

【請求項3】

請求項2において、

前記プロセッサソフトウェアモジュールは、前記プログラムカウンタのカウント値が前記アセンブリレベル言語プログラムのプログラムアドレスと一致する時に、当該プログラムアドレスの命令コードに対応する所定の動作を行う制御回路の記述、を有することを特徴とするハードウェア記述言語合成プログラム。

【請求項4】

請求項1において、

更に、複数の前記命令コードそれぞれに対応して、及び複数の一連の命令コードからなる複合命令コードに対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コード及び複合命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成し、

当該プロセッサソフトウェアファイルは、前記アセンブリレベル言語プログラムの命令

コード及び複合命令コードに対応する前記プロセッサソフトウェアモジュールを有することを特徴とするハードウェア記述言語合成プログラム。

【請求項 5】

請求項 4 において、

前記変換手順は、前記複合命令コードを個別の命令コードに優先して、当該複合命令コードに対応するプロセッサソフトウェアモジュールに変換することを特徴とするハードウェア記述言語合成プログラム。

【請求項 6】

請求項 1 において、

前記アセンブリレベル合成プログラムは、更に、

前記アセンブリレベル言語プログラムの命令コードを解析して、前記プロセッサハードウェアファイルから、使用されないハードウェアコンポーネントに対応するハードウェア記述言語による記述を削除したプロセッサハードウェアファイルを作成する手順を有し、

前記出力手順は、当該削除されたプロセッサハードウェアファイルを出力することを特徴とするハードウェア記述言語合成プログラム。

【請求項 7】

請求項 1 において、

前記プロセッサソフトウェアファイルは、前記プログラムカウンタのカウントが前記アセンブリレベル言語プログラムのプログラムアドレスと一致する時に、当該プログラムアドレスの命令コードに対応する所定の動作を行う制御回路の記述、を有することを特徴とするハードウェア記述言語合成プログラム。

【請求項 8】

請求項 2 または 4 において、

前記プロセッサハードウェアファイルと前記対応テーブルとを、複数のマイクロプロセッサに対応して、複数組有することを特徴とするハードウェア記述言語合成プログラム。

【請求項 9】

請求項 2 または 4 において、

前記プロセッサハードウェアファイルと前記対応テーブルとを、仮想的マイクロプロセッサに対応して有すると共に、

更に、前記仮想マイクロプロセッサに対応して前記高位レベル言語プログラムを前記アセンブリレベル言語プログラムに変換する仮想コンパイラプログラムを有することを特徴とするハードウェア記述言語合成プログラム。

【請求項 10】

少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリとプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルがあらかじめ生成され、集積回路をハードウェア記述言語で記述したハードウェア記述言語ファイルを生成する集積回路の設計方法において、

高位レベル言語で記述された高位レベル言語プログラムを、当該マイクロプロセッサに対応するコンパイラにより前記マイクロプロセッサが実行可能なアセンブリレベル言語プログラムの命令コードに変換する工程と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する工程と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルを、前記ハードウェア記述言語ファイルとして出力する工程とを有する集積回路の設計方法。

【請求項 11】

請求項 10 において、

更に、前記ハードウェア記述言語ファイルを、論理合成ツールにより、前記集積回路のネットリストに変換する工程を有することを特徴とする集積回路の設計方法。

【請求項 12】

請求項 10 において、

更に、複数の前記命令コードにそれぞれ対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルがあらかじめ生成され、

前記変換工程は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成する工程を含み、

当該プロセッサソフトウェアファイルは、前記アセンブリレベル言語プログラムの命令コードに対応する前記プロセッサソフトウェアモジュールを複数有することを特徴とする集積回路の設計方法。

【請求項 13】

高位レベル言語で記述された高位レベル言語プログラムを、所定のマイクロプロセッサに対応するコンパイラにより変換した前記マイクロプロセッサが実行可能なアセンブリレベル言語プログラムの命令コードを、解析する手順と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する手順と、

前記プロセッサソフトウェアファイルを、前記マイクロプロセッサの少なくともプログラムメモリとプログラムデコーダを除くハードウェアコンポーネントをハードウェア記述言語で記述したプロセッサハードウェアファイルと共に、出力する手順とをコンピュータに実行させることを特徴とするハードウェア記述言語合成プログラム。

【請求項 14】

請求項 13 において、

更に、複数の前記命令コードにそれぞれ対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成することを特徴とするハードウェア記述言語合成プログラム。

【請求項 15】

請求項 13 において、

更に、複数の前記命令コードそれぞれに対応して、及び複数の一連の命令コードからなる複合命令コードに対応して、前記ハードウェア記述言語で記述された複数のプロセッサソフトウェアモジュールを有する対応テーブルを有し、

前記変換手順は、当該対応テーブルを参照して、前記アセンブリレベル言語プログラムの命令コード及び複合命令コードを、それぞれ対応するプロセッサソフトウェアモジュールに変換して前記プロセッサソフトウェアファイルを生成することを特徴とするハードウェア記述言語合成プログラム。

【請求項 16】

請求項 15 において、

前記変換手順は、前記複合命令コードを個別の命令コードに優先して、当該複合命令コードに対応するプロセッサソフトウェアモジュールに変換することを特徴とするハードウェア記述言語合成プログラム。

【請求項 17】

請求項 13 において、

前記アセンブリレベル合成プログラムは、更に、

前記アセンブリレベル言語プログラムの命令コードを解析して、前記プロセッサハード

ウェアファイルから、使用されないハードウェアコンポーネントに対応するハードウェア記述言語による記述を削除したプロセッサハードウェアファイルを作成する手順を有し、

前記出力手順は、当該削除されたプロセッサハードウェアファイルを出力することを特徴とするハードウェア記述言語合成プログラム。

【手続補正 2】

【補正対象書類名】明細書

【補正対象項目名】0007

【補正方法】変更

【補正の内容】

【0007】

上記目的を達成するために、本発明の第1の側面によれば、(1)マイクロプロセッサのプログラムメモリとプログラムデコーダなどを除くハードウェアコンポーネントを、ハードウェア記述言語で記述したハードウェアファイルと、(2)高位レベル言語プログラムをコンパイルして得られたアセンブリレベル言語プログラムを解析して、その命令コードをハードウェアコンポーネントに対応させてハードウェア記述言語で記述したソフトウェアファイルに変換するアセンブリレベル合成プログラムとを有するハードウェア記述言語合成ツールである。そして、このツールは、ハードウェアファイルとソフトウェアファイルとを、ハードウェア設計ファイルとして出力する。

【手続補正 3】

【補正対象書類名】明細書

【補正対象項目名】0011

【補正方法】変更

【補正の内容】

【0011】

上記の目的を達成するために、本発明の第2の側面によれば、少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリ及びプログラムデコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルと、

アセンブリレベル合成プログラムとを有し、

前記アセンブリレベル合成プログラムは、

高位レベル言語で記述された高位レベル言語プログラムを当該マイクロプロセッサに対応するコンパイラにより変換した前記マイクロプロセッサが実行可能なアセンブリレベル言語プログラムの命令コードを、解析する手順と、

前記解析したアセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する手順と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルとを出力する手順とをコンピュータに実行させることを特徴とするハードウェア記述言語合成プログラムである。

【手続補正 4】

【補正対象書類名】明細書

【補正対象項目名】0012

【補正方法】変更

【補正の内容】

【0012】

更に、上記目的を達成するために、本発明の第3の側面によれば、少なくとも入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタと、プログラムメモリと、プログラムデコーダとを有するマイクロプロセッサの、前記プログラムメモリとプログラム

デコーダを除く前記入出力端子と、レジスタ群と、演算ユニットと、プログラムカウンタとを含むハードウェアコンポーネントを、ハードウェア記述言語で記述したプロセッサハードウェアファイルがあらかじめ生成され、集積回路をハードウェア記述言語で記述したハードウェア記述言語ファイルを生成する集積回路の設計方法において、

高位レベル言語で記述された高位レベル言語プログラムを、当該マイクロプロセッサに対応するコンパイラにより前記マイクロプロセッサが実行可能なアセンブリレベル言語プログラムの命令コードに変換する工程と、

前記アセンブリレベル言語プログラムの命令コードを、前記プロセッサハードウェアファイルに含まれるハードウェアコンポーネントに対応させて前記ハードウェア記述言語で記述したプロセッサソフトウェアファイルに、変換する工程と、

前記プロセッサハードウェアファイルと前記プロセッサソフトウェアファイルを、前記ハードウェア記述言語ファイルとして出力する工程とを有することを特徴とする。

【**手続補正5**】

【**補正対象書類名**】明細書

【**補正対象項目名**】0013

【**補正方法**】変更

【**補正の内容**】

【**0013**】

上記発明の側面によれば、高位レベル言語プログラムを、コンパイラによりマイクロプロセッサが実行可能なアセンブリレベル言語プログラムの命令コードに変換し、そのアセンブリレベル言語プログラムの命令コードをマイクロプロセッサのハードウェアコンポーネントに対応させてハードウェア記述言語で記述したプロセッサソフトウェアファイルに変換する。アセンブリレベル言語プログラムの命令コードは、例えばあるレジスタにある定数を加算してあるレジスタに格納するというように、マイクロプロセッサのハードウェアコンポーネントに対応しているので、かかるアセンブリレベル言語プログラムであれば、1対1でハードウェア記述言語で記述したハードウェア設計ファイルに置き換えることができ、従来例のような検証やシミュレーションを行う必要がなく、実用的である。

フロントページの続き

(72)発明者 秋田 純一

石川県金沢市角間町又7番地 国立大学法人 金沢大学内

Fターム(参考) 5B046 AA08 BA03