(19)日本国特許庁(JP)　(12)特　許　公　報(B1)　(11)特許番号
特許第3735723号
(P3735723)
(45)発行日　平成18年1月18日(2006.1.18)　(24)登録日　平成17年11月4日(2005.11.4)

(51)Int.Cl.　F I
G06F 17/50 (2006.01)　G06F 17/50 654M
G06F 17/50 654A

請求項の数 17　(全 31 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)
審査請求日　平成17年2月3日(2005.2.3)

早期審査対象出願

(73)特許権者　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】ハードウエ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)【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エアコンポーネントを、ハードウエア記述言語で記述したプロセッサハードウエアファイルと、
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エアファイルに含まれるハードウエアコンポーネントに対応させて前記ハードウエア記述言語で記述したプロセッサソフトウエアファイルに、変換する手順と、
前記プロセッサハードウエアファイルと前記プロセッサソフトウエ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エ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、前記ハードウエア記述言語で記述された複数のプロセッサソフトウエ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エアモジュールに変換して前記プロセッサソフトウエアファイルを生成し、
当該プロセッサソフトウエアファイルは、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エアモジュールを有することを特徴とするハードウエ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エアモジュールは、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エ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、前記ハードウエア記述言語で記述された複数のプロセッサソフトウエ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エアモジュールに変換して前記プロセッサソフトウエアファイルを生成し、
当該プロセッサソフトウエアファイルは、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エアモジュールを有することを特徴とするハードウエ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エアモジュールに変換することを特徴とするハードウエ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エアファイルから、使用されないハードウエアコンポーネントに対応するハードウエア記述言語による記述を削除したプロセッサハードウエ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エアファイルを出力することを特徴とするハードウエ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エアファイルは、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エ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エ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エ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エ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エ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エアコンポーネントを、ハードウエア記述言語で記述したプロセッサハードウエアファイルがあらかじめ生成され、集積回路をハードウエア記述言語で記述したハードウエ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エアファイルに含まれるハードウエアコンポーネントに対応させて前記ハードウエア記述言語で記述したプロセッサソフトウエアファイルに、変換する工程と、
前記プロセッサハードウエアファイルと前記プロセッサソフトウエアファイルを、前記ハードウエ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、前記ハードウエ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、前記ハードウエア記述言語で記述された複数のプロセッサソフトウエ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エアモジュールに変換して前記プロセッサソフトウエ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エアファイルは、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エ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エアファイルに含まれるハードウエアコンポーネントに対応させて前記ハードウエア記述言語で記述したプロセッサソフトウエアファイルに、変換する手順と、
前記プロセッサソフトウエアファイルを、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エアコンポーネントをハードウエア記述言語で記述したプロセッサハードウエアファイルと共に、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エ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、前記ハードウエア記述言語で記述さ

れた複数のプロセッサソフトウエ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エアモジュールに変換して前記プロセッサソフトウエアファイルを生成することを特徴とするハードウエ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、前記ハードウエア記述言語で記述された複数のプロセッサソフトウエ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エアモジュールに変換して前記プロセッサソフトウエアファイルを生成することを特徴とするハードウエ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エアモジュールに変換することを特徴とするハードウエ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エアファイルから、使用されないハードウエアコンポーネントに対応するハードウエア記述言語による記述を削除したプロセッサハードウエ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エアファイルを出力することを特徴とするハードウエア記述言語合成プログラム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、ハードウエ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エ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エ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エア記述言語（Hardware Description Language：ＨＤＬ）によりハードウエア回路を記述し、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、このようなＨＤＬにより記述されたハードウエア設計ファイルをネットリストに変換する論理合成ツールも開発され、実用化されている。
【０００３】
ハードウエアを記述するＨＤＬは、レジスタ・トランスファ・レベル（ＲＴＬ：Register Transfer Level）でハードウエ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、ハードウエア回路の知識をもつハードウエア開発者によってハードウエア回路が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エ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、ソフトウエアツールによって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エアに合成することは、一般的に困難である

。
【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（ＲＴ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、ＲＴ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】
【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エア記述言語で記述されたハードウエア設計ファイルを形成することができるハードウエ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。
【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エアコンポーネントを、ハードウエア記述言語で記述したハードウエアファイルと、（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エアコンポーネントに対応させてハードウエア記述言語で記述したソフトウエ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エア記述言語合成ツールである。そして、このツールは、ハードウエアファイルとソフトウエアファイルとを、ハードウエア設計ファイルとして出力する。
【０００８】
上記第１の側面において、好ましい実施例によれは、複数の前記命令コードにそれぞれ対応して、ハードウエア記述言語で記述された複数のソフトウエ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エアモジュールに変換してソフトウエ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、ハードウエア記述言語で記述されたソフトウエ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エアモジュールに優先的に変換してソフトウエ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エアモジュールは、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エ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。
【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、ハードウエアファイルから、使用されないハードウエ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エアファイルを作成し、当該削除されたハードウエ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エアコンポーネントを削除することができる。

【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エアコンポーネントを、ハードウエア記述言語で記述したプロセッサハードウエアファイルと、

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エアファイルに含まれるハードウエアコンポーネントに対応させて前記ハードウエア記述言語で記述したプロセッサソフトウエアファイルに、変換する手順と、

前記プロセッサハードウエアファイルと前記プロセッサソフトウエ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エア記述言語合成プログラムである。

【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エアコンポーネントを、ハードウエア記述言語で記述したプロセッサハードウエアファイルがあらかじめ生成され、集積回路をハードウエア記述言語で記述したハードウエ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エアファイルに含まれるハードウエアコンポーネントに対応させて前記ハードウエア記述言語で記述したプロセッサソフトウエアファイルに、変換する工程と、

前記プロセッサハードウエアファイルと前記プロセッサソフトウエアファイルを、前記ハードウエ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、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エアコンポーネントに対応させてハードウエア記述言語で記述したプロセッサソフトウエ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エ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でハードウエア記述言語で記述したハードウエ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。

【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。

【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エ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エアコンポーネントについてハードウエア記述言語（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エアとは、その抽象度に違いがなく１対１に対応する。

【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エアファイル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。

【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エアコンポーネントを、ＨＤＬで記述したハードウエアファイルＰ４をあらかじめ準備しておき、ＨＤＬ合成ツールＰ１０が、前述のソフトウエアファイルＰ３と共に、ハードウエアファイルＰ４も出力する。つまり、ハードウエアファイルＰ４は、マイクロプロセッサのソフトウエアの部分を除いたハードウエ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、ソフトウエアファイルＰ３は、そのハードウエアコ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エアファイルＰ３とハードウエアファイル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、ハードウエア技術者がハードウエア回路を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エア設計ファイルＰ３を生成することができるので、ハードウエア技術者でないソフトウエア技術者でも、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記述のハードウエア回路に変換しているので、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エアファイル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エ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エアファイル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エアファイル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エアに必要なハードウエ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エ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う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エア回路のファイル（プロセッサソフトウエアモジュール）を有する。このプロセッサソフトウエアモジュールは、後述するように、ＡＬＬの命令コードをプロセッサハードウエアファイルＰ４のハードウエ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エ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clkin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エアコンポーネントへの制御に対応するものである。

【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して実行される。

【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エアファイルＰ４内にＨＤＬ記述されるハードウエ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エ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、全てのハードウエアコンポーネントが、ＨＤＬ記述されたハードウエア設計ファイルとしてのプロセッサハードウエアファイル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エ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。

【００２７】

図２にもどり、プロセッサハードウエアファイル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、ハードウエア回路をＨＤＬ記述したプロセッサソフトウエ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エアファイルＰ３と、あらかじめ添付されていたプロセッサハードウエアファイルＰ４とをハードウエア設計ファイル（Ｐ３＋Ｐ４）として出力する。

【００２８】

このように、プロセッサハードウエアファイル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エアファイル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エアファイル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エアファイルＰ４がそのまま出力される。

【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エアモジュール（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エ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、ハードウエア回路に対応する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。

【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エアコンポーネントが、プロセッサハードウエアファイル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エアファイルＰ３として出力される。そして、プロセッサソフトウエアファイルＰ３は、ハードウエ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、これらのハードウエ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エアコンポーネントである、入出力端子群clkin、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、ソフトウエアファイル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、前述のハードウエアコンポーネントに供給される。

【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エアファイルP4、対応テーブルTB12、プロセッサソフトウエアファイルP3について詳述する。

【００３３】
［プロセッサハードウエアファイルP4］
図７～図１３は、プロセッサハードウエアファイル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】
ハードウエアファイル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エアコンポーネントをＨＤＬで記述したものである。図７には、ハードウエアファイル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clkin、マスタクリア端子mclrとがあり、入出力ポート定義部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portaout_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。

【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。

【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portaout_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に示すハードウエア回路図が定義される。

【００３９】

図１４に示されたハードウエア回路図は、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portaout_reg[7]を出力し、ゲート回路３０４は信号portaout_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。

【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、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（posegde 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intrise_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intrise_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intrise_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intrise_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タ端子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startclkin_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、プリスケーラ、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、マスタクリア端子mclrがマスタクリアレジスタmclr_sync_regに取り込まれることが記述され、これによりマスタクリア端子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、そのマスタクリアレジスタ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。

【００４５】

以上が、ハードウエアファイルP4の記述例である。そして、ハードウエアファイルP4の最後に、「end else begin」の記述１５６が挿入され、これ以降に、ソフトウエアファイルP3の記述が加えられる。

【００４６】

［対応テーブルTB12］

図１５～図２３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ＡＬＬ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ＨＤＬ記述したファイルとの対応を有する。したがって、図１５～図２４は、命令コードとＨＤＬ記述されたソフトウエ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、ＡＬＬ記述の命令として全部で３５命令を有する。その命令には、１８のバイト対応のレジスタファイル命令群（図１５～図１９）と、４つのビット対応のレジスタファイル命令群（図２０～図２１）と、１３のリテラル（定数）及びコントロール命令群（図２１～図２４）とからなる。

【００４７】

図１５には、バイト対応のレジスタファイル命令２００の一つである命令ＡＤＤＷＦに対応するソフトウエアモジュールのＨＤＬ記述２０２が示されている。この命令ＡＤＤＷＦは、ニーモニック（機械言語）では「000111dfffffff」であり、Wレジスタとレジスタ

ファイルfffffff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ＡＤＤＷＦ命令のＨＤＬ記述２０２は、プログラムカウンタレジスタpc_regが「xxxx」番地の時に（２０２１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（２０２２行）、データソースであるレジスタファイルfffffff番地のデータを入力レジスタin_regに格納し（２０２３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）（２０２２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（２０２６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（２０２７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（２０２８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（２０３０～２０３４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atate_regを「０」にする）（２０３５行）。

【００４８】

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。

【００４９】

このＨＤＬ記述２０２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と（２０２１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、ＨＬＬであるC言語のプログラムをコンパイルしたＡＬＬ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。

【００５０】

図１６は、対応テーブルの命令ＡＮＤＷＦに対応するソフトウエアモジュールのＨＤＬ記述２０４が示されている。この命令ＡＮＤＷＦは、ニーモニック（機械言語）では「000101dfffffff」であり、Wレジスタとレジスタファイルfffffff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をデータディ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）。

【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エ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ィ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、ＤＥＣＦＳＺ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「ｘｘｘｘ」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。

【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、ＩＮＣＦＳＺ、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。

【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fffffff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をデータディスティネーションのレジスタに格納する（２３６７行）。

【００５５】

図２０には、命令ＢＳＦ、命令ＢＴＦＳＣ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ＢＴＦＳＳ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、ＣＬＲＷＤＴ、ＡＮＤＬＷ、ＣＡＬＬ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ＣＡＬＬは、プログラムアドレス「kkkkkkkkkkk」番地のサブルーチンを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「kkkkkkkkkkk」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「kkkkkkkkkkk」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、ソフトウエアファイル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記述するハードウエ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×Ｙ÷Ｐの余り（％）を求める演算を実行するものである。まずヘッダー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】
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ＭＯＶＬＷ、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。
【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エアコンポーネントをＨＤＬ記述したプロセッサハードウエアファイル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エアファイルＰ３として出力する。図１３のソフトウエアファイルＰ

３には、図２５のＡＬＬプログラムの命令「ＭＯＶＬＷ　００」と、命令「ＭＯＶＷＦ０Ａ」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エアファイルＰ４とソフトウエアファイルＰ３とからなるハードウエア設計ファイルが、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エアリソースを割り当てて、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エ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記述やハードウエアコンポーネントを削除して、無駄なハードウエ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、（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エアファイル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エ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。

【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エアファイルや対応テーブルを作成する必要をなくすことができる。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エ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。

【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エアファイル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、全てのハードウエ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、全てのハードウエ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エアコンポーネントを含んだプロセッサハードウエアファイル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。

【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エ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。

【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、所定のハードウエ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、対応するハードウエ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エアファイル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エ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エ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命令コードを対応する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。

【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エアファイルＰ３とプロセッサソフトウエアファイル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。

【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エ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。
【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。
【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エ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エ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エアファイルＰ４を出力する（Ｓ２２Ｂ）。これにより、プロセッサハードウエアファイル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エアファイルＰ４を生成するための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。
【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。

【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。

【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。

【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、演算結果ノードbsfout_nodeに応じて、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。

【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エアファイルＰ３からは、使用されないハードウエ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エアファイル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エアファイルＰ３の最適化により制御回路ユニット３０以外の回路構成が簡単化され、プロセッサソフトウエアファイルＰ４の最適化により制御回路ユニット３０の回路構成が簡単化されることになる。

【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記述のハードウエ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記述のハードウエ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。

【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エアファイルＰ４内にＨＤＬ記述されるハードウエ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エアファイルP4の一例を示す図である。

【図８】プロセッサハードウエアファイルP4の一例を示す図である。

【図９】プロセッサハードウエアファイルP4の一例を示す図である。

【図１０】プロセッサハードウエアファイルP4の一例を示す図である。

【図１１】プロセッサハードウエアファイルP4の一例を示す図である。

【図１２】プロセッサハードウエアファイルP4の一例を示す図である。

【図１３】プロセッサハードウエアファイルP4の一例を示す図である。

【図１４】プロセッサハードウエアファイルP4の一例を示す図である。

【図１５】対応テーブルTB12の具体例を示す図である。

【図１６】対応テーブルTB12の具体例を示す図である。

【図１７】対応テーブルTB12の具体例を示す図である。

【図１８】対応テーブルTB12の具体例を示す図である。

【図１９】対応テーブルTB12の具体例を示す図である。

【図２０】対応テーブルTB12の具体例を示す図である。

【図２１】対応テーブルTB12の具体例を示す図である。

【図２２】対応テーブルTB12の具体例を示す図である。

【図２３】対応テーブルTB12の具体例を示す図である。

【図２４】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。

【図２５】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２６】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２７】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２８】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。

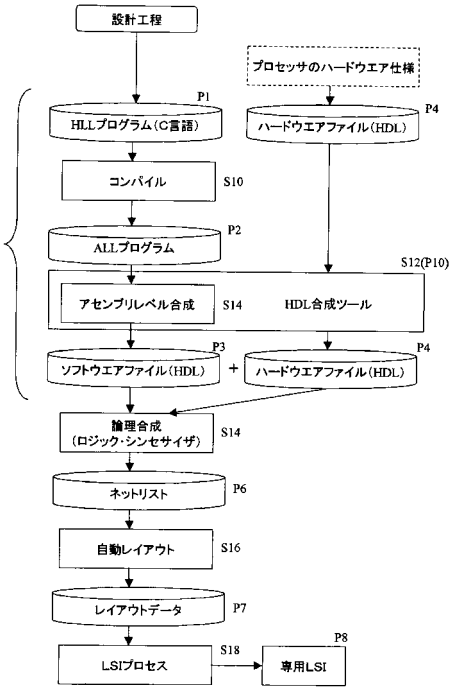
【図２９】図２４のC言語をコンパイルしたＡＬＬプログラムの例を示す図である。

【図３０】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

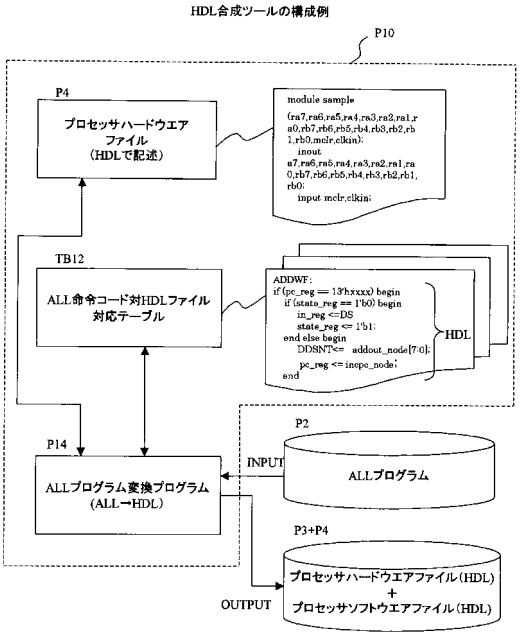
コード）に変換した例を示す図である。
【図３１】図３０に示したＡＬＬプログラムＰ２のニーモニックコードをＨＤＬ合成ツールに与えた結果合成されたＨＤＬ記述ファイルの一例を示す図である。
【図３２】本実施の形態におけるＨＤＬ合成ルールの変形例を示す図である。
【図３３】変形例２におけるＨＤＬ合成手順Ｐ１０のフローチャート図である。
【図３４】複合命令の一例を示す図である。
【図３５】複合命令の別の例を示す図である。
【図３６】複合命令の一例を示す図である。
【符号の説明】
【００９０】
Ｐ１：ＨＬＬ記述プログラム（Ｃ言語）、Ｐ２：ＡＬＬ記述プログラム
Ｐ３：プロセッサソフトウエアファイル（ＨＤＬ）
Ｐ４：プロセッサハードウエアファイル（ＨＤＬ）
Ｐ１０：ＨＤＬ合成ツール

【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エア記述言語で記述されたハードウエア設計ファイルを形成することができるハードウエ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エアコンポーネントを、ハードウエア記述言語で記述したハードウエアファイルと、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エアコンポーネントに対応させてハードエウア記述言語で記述したソフトウエ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エア記述言語合成ツールである。そして、このツールは、ハードウエアファイルとソフトウエアファイルとを、ハードウエア設計ファイルとして出力する。
【選択図】　　図１

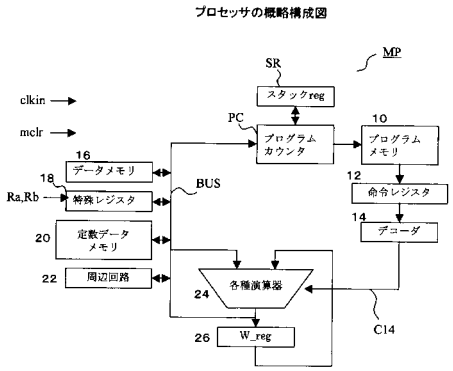
【図１】


設計工程
プロセッサのハードウエア仕様
P1
HLLプログラム（C言語）
P4
ハードウエアファイル（HDL）
コンパイル
S10
P2
ALLプログラム
S12(P10)
アセンブリレベル合成
S14
HDL合成ツール
P3
ソフトウエアファイル（HDL）
+
P4
ハードウエアファイル（HDL）
論理合成
（ロジック・シンセサイザ）
S14
ネットリスト
P6
自動レイアウト
S16
レイアウトデータ
P7
LSIプロセス
S18
P8
専用LSI

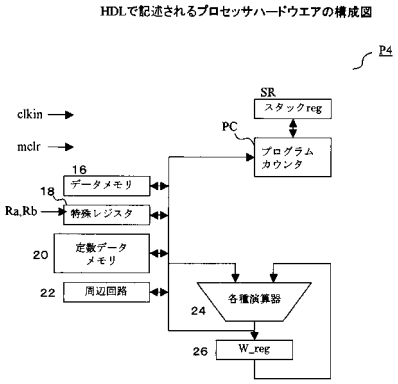
【図２】


HDL合成ツールの構成例
P10
P4
プロセッサハードウエア
ファイル
（HDLで記述）
module sample
(ra7,ra6,ra5,ra4,ra3,ra2,ra1,r
a0,rb7,rb6,rb5,rb4,rb3,rb2,rb
1,rb0,mclr,clkin);
inout
a7,ra6,ra5,ra4,ra3,ra2,ra1,ra
0,rb7,rb6,rb5,rb4,rb3,rb2,rb1,
rb0;
input mclr,clkin;
TB12
ALL命令コード対HDLファイル
対応テーブル
ADDWF:
if (pc_reg == 13'hxxxx) begin
if (state_reg == 1'b0) begin
in_reg <=DS
state_reg <= 1'b1;
end else begin
DDSNT<= addout_node[7:0];
pc_reg <= incpc_node;
end
HDL
P14
ALLプログラム変換プログラム
(ALL→HDL)
P2
INPUT
ALLプログラム
P3+P4
プロセッサハードウエアファイル（HDL）
+
プロセッサソフトウエアファイル（HDL）
OUTPUT

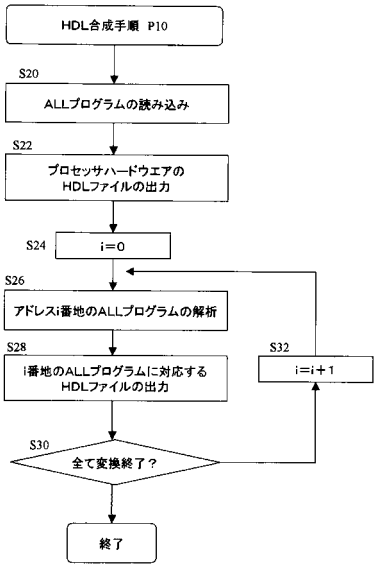
【図３】


プロセッサの概略構成図
MP
SR
スタックreg
clkin
mclr
PC
プログラム
カウンタ
10
プログラム
メモリ
16
データメモリ
18
Ra,Rb
特殊レジスタ
BUS
12
命令レジスタ
14
デコーダ
20
定数データ
メモリ
22
周辺回路
24
各種演算器
26
W_reg
C14

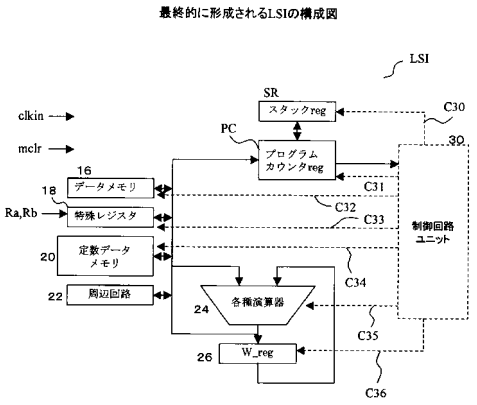
【図４】


HDLで記述されるプロセッサハードウエアの構成図
P4
SR
スタックreg
clkin
mclr
PC
プログラム
カウンタ
16
データメモリ
18
Ra,Rb
特殊レジスタ
20
定数データ
メモリ
22
周辺回路
24
各種演算器
26
W_reg

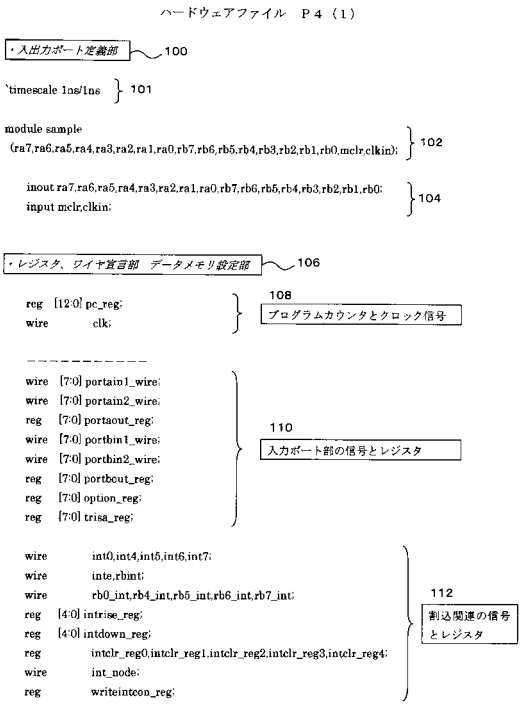
【図５】


HDL合成手順 P10
S20
ALLプログラムの読み込み
S22
プロセッサハードウエアの
HDLファイルの出力
S24
i=0
S26
アドレスi番地のALLプログラムの解析
S28
i番地のALLプログラムに対応する
HDLファイルの出力
S32
i=i+1
S30
全て変換終了？
終了

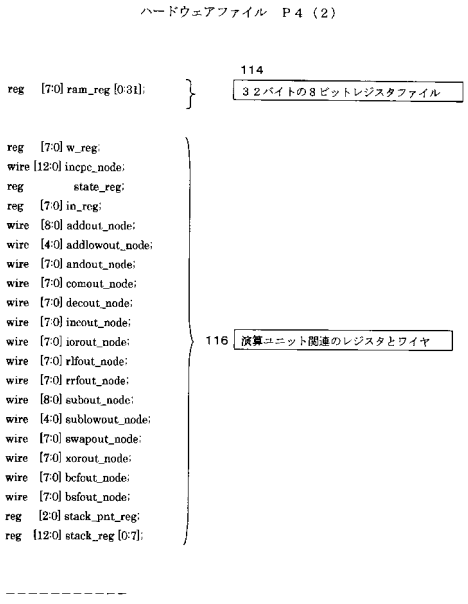
【図６】


最終的に形成されるLSIの構成図
LSI
SR
スタックreg
C30
30
clkin
PC
mclr
プログラム
カウンタreg
16
データメモリ
C31
18
Ra,Rb
特殊レジスタ
C32
C33
制御回路
ユニット
20
定数データ
メモリ
C34
22
周辺回路
24
各種演算器
C35
26
W_reg
C36

【図７】


ハードウェアファイル P４（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,clkin);
102
inout ra7,ra6,ra5,ra4,ra3,ra2,ra1,ra0,rb7,rb6,rb5,rb4,rb3,rb2,rb1,rb0;
input mclr,clkin;
104
・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;
wire clk;
108
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;
110
入力ポート部の信号とレジスタ
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;
112
割込関連の信号
とレジスタ

【図８】


ハードウェアファイル P４（２）
114
reg [7:0] ram_reg [0:31];
３２バイトの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】

ハードウェアファイル　Ｐ４（３）

118

特殊レジスタ、データメモリの配置とメモリアドレス

	bank0	bank1	
00h	INDF	INDF	80h
01h	TMR0	OPTION	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h			87h
08h			88h
09h			89h
0ah	PCLATH	PCLATH	8ah
0bh	INTCON	INTCON	8bh
0ch	データメモリ	データメモリ	8ch
7fh			8fh

【図１０】

ハードウェアファイル　Ｐ４（４）

・入出力切り替え制御部　120

入出力ポートＡ　122

```
assign portain1_wire[7] = (trisa_reg[7] == 1'b1) ? ra7 : 1'bz,
assign portain2_wire[7] = (trisa_reg[7] == 1'b1) ? portain1_wire[7] : portaout_reg[7],
assign ra7 = (trisa_reg[7] == 1'b0) ? portaout_reg[7] : 1'bz,
```

入出力ポートＢ　124

```
assign portbin1_wire[7] = (trisb_reg[7] == 1'b1) ? rb7 : 1'bz,
assign portbin2_wire[7] = (trisb_reg[7] == 1'b1) ? portbin1_wire[7] : portbout_reg[7],
assign rb7 = (trisb_reg[7] == 1'b0) ? portbout_reg[7] : 1'bz,
```

【図１１】

ハードウェアファイル　Ｐ４（５）

・ピン変化割り込み回路　126

```
assign int0 = portbin1_wire[0],
       int4 = portbin1_wire[4],
       int5 = portbin1_wire[5],
       int6 = portbin1_wire[6],
       int7 = portbin1_wire[7];                                   } 128

1301   always @(posedge int0 or posedge intclr_reg0)              }
1302   begin                                                      }
1303       if (intclr_reg0 == 1'b1) begin                         }
1304           intrise_reg[0] <= 1'b0;                            }
1305       end else begin                                         }
1306           intrise_reg[0] <= 1'b1;                            }
1307       end                                                    }
1308   end                                                        }
                                                                  } 130
1309   always @(negedge int0 or posedge intclr_reg0)              }
1310   begin                                                      }
1311       if (intclr_reg0 == 1'b1) begin                         }
1312           intdown_reg[0] <= 1'b0;                            }
1313       end else begin                                         }
1314           intdown_reg[0] <= 1'b1;                            }
1315       end                                                    }
1316   end

1317   assign rb0_int = (option_reg[6] == 1'b1) ? intrise_reg[0] : intdown_reg[0];

   ----------

1401   assign inte = intcon_reg[4] & rb0_int;                     }
1402   assign rbint = intcon_reg[3] & (rb4_int | rb5_int | rb6_int | rb7_int);
                                                                  } 140
1403   assign int_node = intcon_reg[7] & ((intcon_reg[3] & intcon_reg[0])
                                        | (intcon_reg[4] & intcon_reg[1])
                                        | (intcon_reg[5] & intcon_reg[2]));
```

【図１２】

ハードウェアファイル　Ｐ４（６）

・各演算器や制御信号の設定　142

```
assign incpc_node = pc_reg + 13'b0000000000001; //PC+1          144
assign addout_node = {1'b0,w_reg} + {1'b0,in_reg}; //add        }
assign andout_node = w_reg & in_reg; //and                      }
assign comout_node = ~in_reg; //com                              }
assign decout_node = in_reg + 8'b11111111; //dec,decfsz         }
assign incout_node = in_reg + 8'b00000001; //inc,incfsz         }
assign iorout_node = w_reg | in_reg; //ior                      }
assign rlfout_node = {in_reg[6:0],status_reg[0]}; //rlf         }
assign rrfout_node = {status_reg[0],in_reg[7:1]}; //rrf         }
assign subout_node = {1'b0,in_reg} + {1'b0,~w_reg} + 9'b000000001; //sub   } 146
assign swapout_node = {in_reg[3:0],in_reg[7:4]}; //swap          }
assign xorout_node = w_reg ^ in_reg; //xor                      }
assign bcfout_node = in_reg & bit_reg; //bcf,btfsc              }
assign bsfout_node = in_reg | bit_reg; //bsf,btfss              }
assign addlowout_node = {1'b0,w_reg[3:0]} + {1'b0,in_reg[3:0]}; //加算下位４ビット   }
assign sublowout_node = {1'b0,in_reg[3:0]} + {1'b0,~w_reg[3:0]} + 5'b00001; //減算下位４ビット }

assign powerdown_node = sleepflag_reg; //スリープ信号            }
assign startclkin_node = inte | rbint | (~mclr); //スリープ解除信号 } 148
                                                                  }
assign selclk_node = ~(powerdown_node & (~startclkin_node)); //クロック選択信号
```

【図１３】

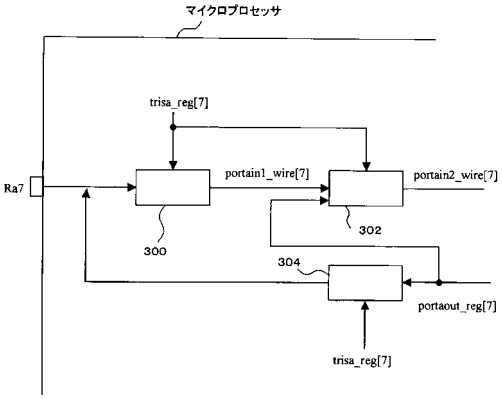
ハードウェアファイル　Ｐ４（７）

```
シンクロナイザー、リセット時の動作設定、タイマ、プリスケーラ、割り込み制御回路                150

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'b0000000000000;
        status_reg[7:4] <= 4'b0001;
        status_reg[3] <= ~sleepflag_reg;
        pclath_reg <= 5'b00000;
        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
------------
        end else begin                                              156
```

【図１４】



【図１５】

対応テーブル　ＴＢ１２（１）

全３５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　ニーモニック：000111dfffffff　クロック数：２
説明：W_reg とレジスタファイル fffffff 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_regf0] <= 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１６】

対応テーブル　ＴＢ１２（２）

②命令：ANDWF　ニーモニック：000101dfffffff　クロック：２
説明：W_reg とレジスタファイル fffffff 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'h0008) begin
2       if (state_reg == 1'b0) begin
3           in_reg <= ram_reg[{state_reg[5],7'b0001111}];
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１７】

対応テーブル　ＴＢ１２（３）

③命令：CLRF　ニーモニック：0000011fffffff　クロック：２
説明：レジスタファイル ffffff 番地をゼロクリアする。
出力 HDL：

```
if (pc_reg == 13'hxxxx) begin
――――――――――
    pc_reg <= incpc_node;
end
```
206

④命令：CLRW　ニーモニック：0000010fffffff　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 <= incpc_node;
end
```
208

⑤命令：COMF　ニーモニック：001001dfffffff　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：000011dfffffff　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：001011dfffffff　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１８】

対応テーブル　ＴＢ１２（４）

⑧命令：INCF　ニーモニック：0010100dfffffff　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：0011111dfffffff　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：000100dfffffff　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：MOVF　ニーモニック：001000dfffffff　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：0000001fffffff　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１９】

対応テーブル　ＴＢ１２（５）

⑬命令：NOP　ニーモニック：00000000000000　クロック：１
説明：何もしない。
出力 HDL：

```
if (pc_reg == 13'hxxxx) begin
    pc_reg <= incpc_node;
end
```
222

⑭命令：RLF　ニーモニック：001101dfffffff　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：001100dfffffff　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：000010dfffffff　クロック：２
説明：レジスタファイル ffffff 番地のデータをから 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：001110dfffffff　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：000110dfffffff　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２０】

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）　234
①命令：BCF　ニーモニック：0100bbbfffffff　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 <= incpc_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：0101bbbfffffff　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：BTFSC　ニーモニック：0110bbbfffffff　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２１】

対応テーブル　ＴＢ１２（７）

④命令：BTFSS　ニーモニック：0111bbbfffffff　クロック：２
説明：レジスタファイル fffffff 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：CLRWDT　ニーモニック：00000001100100　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'b0027) begin
    pc_reg <= incpc_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を呼び出す。
出力 HDL：

```
2501    if (pc_reg == 13'hxxxx) begin
2502        pc_reg <= {pclath_reg[4:3],11'bkkkkkkkkkkk};
2503        stack_reg[stack_pnt_reg] <= incpc_node;
2504        if (stack_pnt_reg == 3'b111) begin
2505            stack_pnt_reg <= 3'b000;
2506        end else begin
2507            stack_pnt_reg <= stack_pnt_reg + 3'b001;
        end
    end
```
250

【図２２】

対応テーブル　ＴＢ１２（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 <= incpc_node;
end
```
256

⑧命令：RETURN　ニーモニック：00000000001000　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_pnt_reg - 3'b001];
2583        if (stack_pnt_reg == 3'b000) begin
2584            stack_pnt_reg <= 3'b111;
2585        end else begin
2586            stack_pnt_reg <= stack_pnt_reg - 3'b001;
        end
    end
```
258

⑨命令：RETFIE　ニーモニック：00000000001001　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２３】

対応テーブル　ＴＢ１２（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：00000001100011　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 <= incpc_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２４】

C言語プログラム　P１

```
//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);     //hgih 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_b();          //X 入力
                output_low(pin_a0);     //a0 を low                 306
                output_high(pin_a0);    //a0 を high
                in2=input_b();          //Y 入力
                output_low(pin_a0);     //a0 を low                 308
                output_high(pin_a0);    //a0 を high
                in3=input_b();          //P 入力
                output_low(pin_a0);     //a0 を high                310
                reg=(long)in1*(long)in2%(long)in3;      //X*Y%P     312
                output_high(pin_a1);    //a1 を high
                output_b(reg);          //演算結果を出力
                output_low(pin_a1);     //a1 を low                 314
        }
}
```

【図２５】

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 main() level
                         19 (28%) worst case
400    402     Stack:    1 locations
*                          404
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 main()  {          //main 関数
....................     int    in1;              //X
003F:  CLRF    04
0040:  MOVLW   1F
0041:  ANDWF   03,F
....................     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２６】

P2(2)

```
0045:  BCF     05.0
....................     output_low(pin_a1);       //hgih 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_b();            //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_b();            //Y 入力
005C:  MOVLW   FF
005D:  BSF     03.5
005E:  MOVWF   06
005F:  BCF     03.5
0060:  MOVF    06,W
```

【図２７】

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_b();            //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    0F,W
007B:  MOVWF   19
007C:  MOVF    17,W
007D:  MOVWF   1B
007E:  MOVF    12,W
007F:  MOVWF   1A
```

【図２８】

P2(4)

```
0080:  MOVF    0F,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    0F,W
008D:  MOVWF   1A
008E:  MOVF    0E,W
008F:  MOVWF   1C
0090:  MOVF    0D,W
0091:  MOVWF   1B
0092:  MOVF    0F,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
```

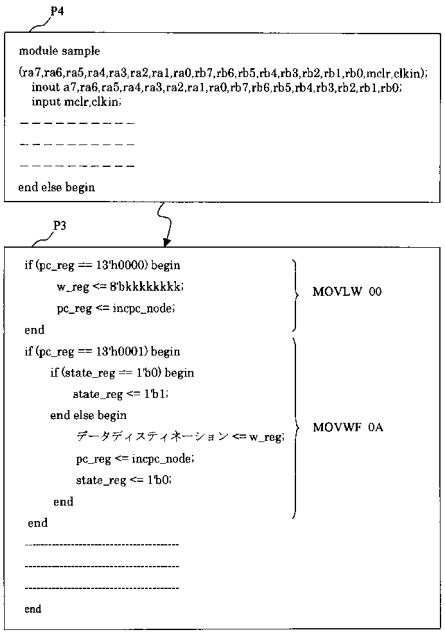
【図３０】

ニーモニックコード

```
:1000000000308A003F28000010308C008F019E01D4
:100010009B0C9A0C031C12281C088F0703189E0ABD
:10002001D089E079E0C8F0C8E0C8D0C8C0B0828C7
:1000300085288D018E018C018F011E08031D232848
:10004001D0803193D2810309F0003109B0D9C0DC7
:10005000 8C0D8F0D1E080F02031D30281D080C0289
:10006000031C39281D088C02031C8F031E088F02F5
:1000700003148D0D8E0D9F0B252800009728840 1F9
:100080001F3083058316051083120510831685 1013
:1009000831285108316051083120514FF30831612
:1000A0008600831206089200831605108312051 03D
:1000B000831605108312051 4FF3083168600831201
:1000C000060893008316051083120510831605 1089
:1000D00083120514FF3083168600831206089400ED
:1000E000831605108312051097011208960 08F01E0
:1000F000130898000F08990017089B0012089A002F
:100100000F089D0013089C0004280E0898000D0895
:1001100097008F01140899000F089A000E089C00A0
:10012000D089B000F089E0014089D0019280C085C
:1001300095008316851083128514003083168600 7F
:10014000831215088600831685108312851 04A28AD
:02015000 63004A
:00000001FF
:PIC16F84
```

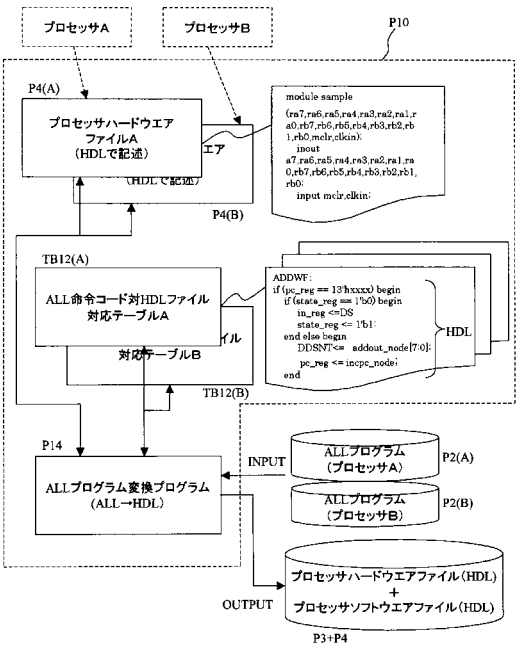
【図３１】

P3+P4

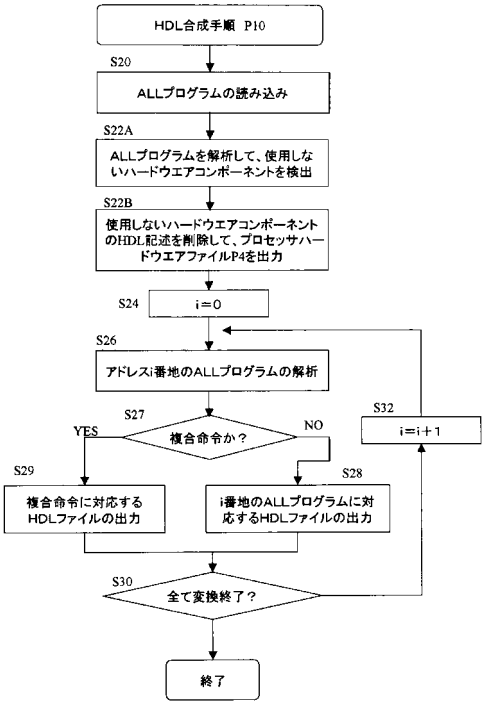


【図３２】

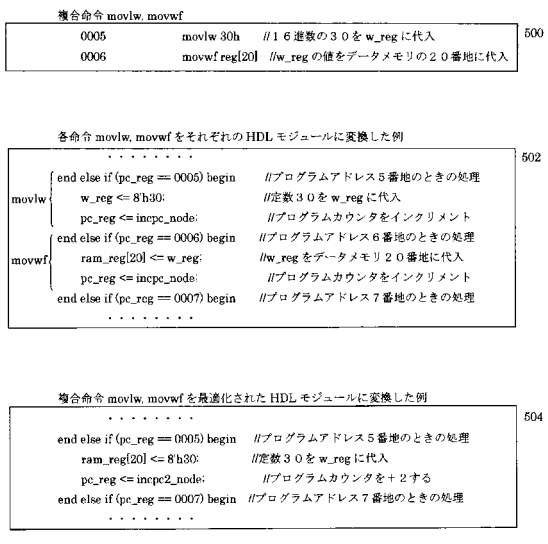
HDL合成ツールの構成例(変形例)



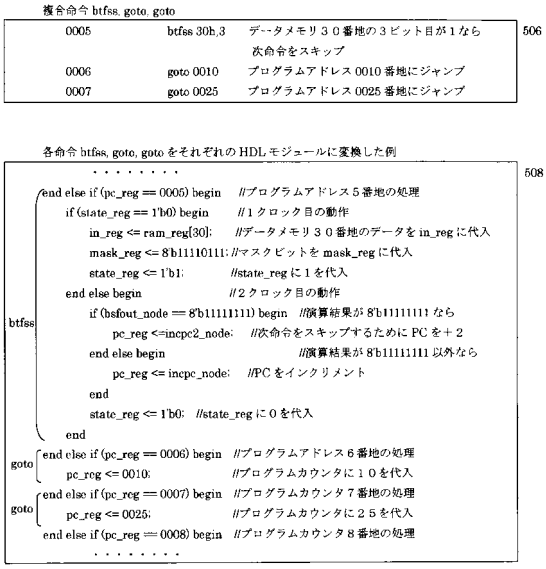
【図３３】



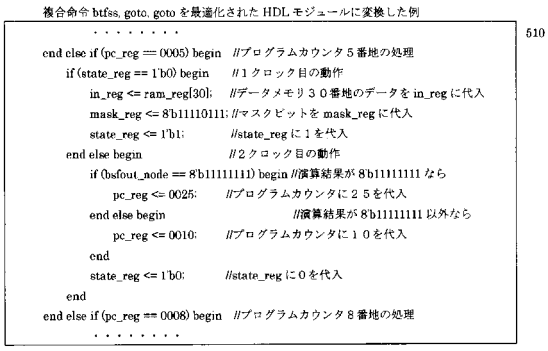
【図３４】


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
500
各命令 movlw, movwf をそれぞれの HDL モジュールに変換した例
・・・・・・・・
movlw
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
movwf
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
・・・・・・・・
502
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
・・・・・・・・
504

【図３５】


複合命令 btfss, goto, goto
0005 btfss 30h,3 データメモリ３０番地の３ビット目が１なら
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
506
各命令 btfss, 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
btfss
if (bsfout_node == 8'b11111111) begin //演算結果が 8'b11111111 なら
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
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
goto
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
goto
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
・・・・・・・・
508

【図３６】


複合命令 btfss, 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
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
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
・・・・・・・・
510

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ヌ７番地 国立大学法人 金沢大学内

審査官 田中 幸雄

(56)参考文献 M. Wazlowski et al., PRISM-II Compiler and Architecture, Proceedings of 1993 IEEE Work shop on FPGAs for Custom Computing Machines, 米国, IEEE, １９９３年 ４月 ５日, p9-16
Xiaoyong Tang et al., Compiler Optimizations in the PACT HDL Behavioral Synthesis Tool for ASICs and FPGAs, Proceedings of 2003 IEEE International SOC Conference, 米国, IEEE, ２００３年 ９月１７日, p189-192

(58)調査した分野(Int.Cl., ＤＢ名)
Ｇ０６Ｆ １７／５０
ＩＥＥＥ Ｘｐｌｏｒｅ