社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

仮想マルチプロセッサモデルに基づく高速 SoC プロトタイピング手法

吉田　浩章[†,††]　藤田　昌宏[†,††]

† 東京大学 大規模システム設計教育研究センター (VDEC)
〒 113–8656 東京都文京区本郷 7–3–1
†† 科学技術振興機構 戦略的創造研究推進事業 CREST
E-mail: †hiroaki@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

あらまし　SoC では高性能・高電力効率を両立するため，複数のプロセッサに加えて多数の特定用途アクセラレータを搭載するヘテロジニアス構成となっている．このような多数のアクセラレータを持つ SoC の設計手法として，本稿では仮想マルチプロセッサモデルに基づく高速 SoC プロトタイピング手法を提案する．最初からシステム構成を決定するのではなく，十分に柔軟性のあるモデルから始め，性能・電力効率見積もりによって無駄な部分を特定し，その部分の専用化を段階的に進めていく事で高性能と高電力効率を両立可能なハードウェアを設計可能である．また，我々が開発を進めている SoC 設計環境 Cyneum についても説明する．

キーワード　システムオンチップ (SoC)，仮想マルチプロセッサモデル，高位合成，高電力効率

Rapid SoC Prototyping Based on Virtual Multi-Processor Model

Hiroaki YOSHIDA[†,††] and Masahiro FUJITA[†,††]

† VLSI Design and Education Center(VDEC), University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo, 113–8656, Japan
†† CREST, Japan Science and Technology Agency
E-mail: †hiroaki@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

Abstract　To meet both high performance and high energy efficiency, System-on-Chip (SoC) has a heterogenous architecture including many application-specific accelerators. As a design methodology for such heterogeneous SoCs, we propose a rapid SoC prototyping method based on virtual multi-processor model. Starting from a flexible, homogenous multi-processor model, the proposed method iteratively improves the energy efficiency by identifying inefficient portions of the design and replacing them with specialized hardware. Also, we present a SoC design environment Cyneum which we have been developing recently.

Key words　System-on-Chip (SoC), virtual multi-processor model, high-level synthesis, high energy effeciency

1　は じ め に

近年ますますその重要性を増している組み込み機器では非常に高い性能と電力効率を両立することが常に求められている．携帯電話やデジタルカメラに代表される携帯メディア処理デバイスはデスクトップ PC よりも遥かに高い処理性能を数十分の 1 の消費電力で実現しなくてはならない．例えば，典型的なデスクトップ PC のピーク性能は数 GOPS(Giga Operations Per Second) であるのに対して，3G 携帯電話のレシーバは 35~40GOPS の処理能力が必要であり，次世代の携帯電話で主流となるとされている OFDM レシーバでは 210~290GOPS の処理能力が必要になるとされている．携帯機器では電池容量や熱の制約から消費電力を 1W 程度に抑える必要があり，OFDM レシーバでこれを実現するためには 3~5pJ/op という非常に高い電力効率が求められる．このような高い性能と電力効率を両立させるために，ASIC を用いて特定用途向けの専用ハードウェア化することが一般的となっている．例えば，90nm テクノロジを用いた OFDM レシーバの ASIC 実装では 5pJ/op という高い電力効率を実現可能である [1]．一方で，効率的な組み込みプロセッサでは 250pJ/op(ASIC の 50 倍) であり，典型的なラップトップ PC に至っては 20nJ/op(ASIC の 4000 倍) という低い電力効率である．このように ASIC は高性能と高電力効率を両立可能な唯一の手段であり，これが他方式との決定的な差別化要因となっている．

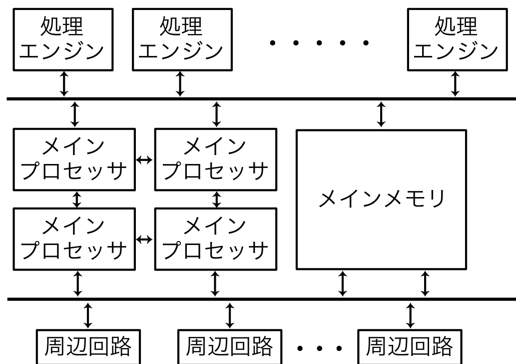


図 1: 民生用携帯機器 SoC の基本構成例 [2]

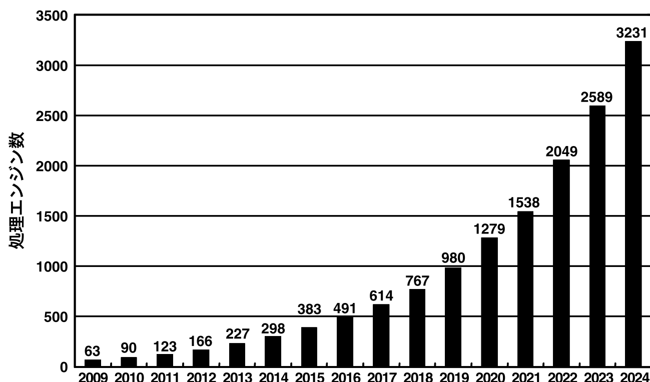
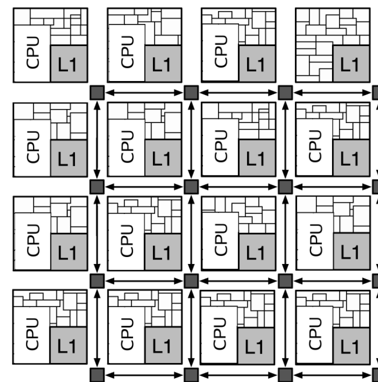


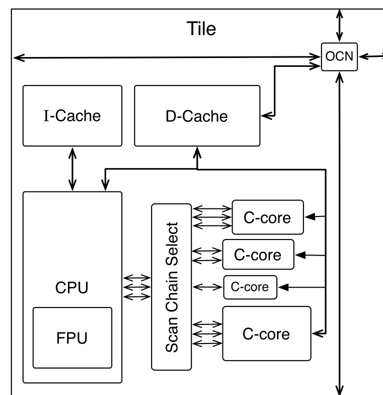
図 2: 民生用携帯機器 SoC 内の処理エンジン数の動向予測 [2]

半導体技術の劇的な進歩に伴う大規模化により，1 チップ上に様々な機能を搭載したシステムオンチップ (SoC) の実現が可能になっている．一般的な携帯機器向けの SoC は図 1 に示すように複数の組み込みプロセッサや大規模なメインメモリ，周辺回路と多数の処理エンジンからなる．基本的な処理はプロセッサで実行し，高い性能が必要とされる機能については処理エンジンと呼ばれる特定用途専用ハードウェアを用いて実現することで高い効率を実現することができる．最新の ITRS ロードマップ [2] によると，この処理エンジンの数は 2024 年には 3000 個を超えると予測されている (図 2)．このように，SoC では柔軟性のあるプロセッサと多数の高効率な専用ハードウェアからなるヘテロジニアス化が更に進むことが予想される．

一方で汎用プロセッサの分野においては，チップの大規模化に伴って多数のプロセッサコアを 1 チップに搭載するチップマルチコア化が進んでいる．しかしながら，上で述べた通りプロセッサの電力効率は専用ハードウェアに比べて大きく劣っており，そのため 1 チップに多数のコアを搭載しても熱や電力の問題から全てのコアを同時に動作させることができない「利用率の壁」(Utilization Wall) という問題が顕著になりつつある．この問題に対して，最近になり活発な研究が行われている．Conservation cores [3] はチップマルチプロセッサ (CMP) 内の各プロセッサに図 3 に示すような c-core と呼ばれる専用ハードウェアを接続し，よく使われる機能に対してはこの c-core を用いることで電力効率を改善することが可能である．実験結果では実用的なアプリケーションの実行全体で 2 倍程度の電力効率の改善が可能であったとの報告がある．また, [4] では実際に



(a) 全体構成



(b) 各コアの構成

図 3: Conservation core システム [3]

H.264 エンコーダを CMP システムで実装し，電力効率を非効率化している要因について解析を行っている．CMP 実装と ASIC 実装の比較では，700 倍もの電力効率の差を確認しており，主要因は CMP の汎用性によるものであるとしている．命令メモリのフェッチやメモリのキャッシュ，データパスのビット幅や演算器の種類や数などがその例である．論文では結論として，プロセッサに加えて特定用途向けの専用ハードウェアを追加することで同等の電力効率を達成可能であるとしている．このように，様々なグループが CMP の効率化の研究を行った結論としてヘテロジニアス化が有効であると考えている．これは本質的に SoC と同じ方向性であり，ヘテロジニアス化が進んでいくという上述の主張を異なる側面から支援しているものである．

このようなヘテロジニアス構成を持つ SoC 設計では，多数の専用ハードウェアの設計を含む大規模なシステムな設計を短期間に行わなければならない．最近では SystemC 言語に代表されるシステムレベル設計記述言語を用いた設計手法が主流になりつつある．一般的にシステムレベル設計記述言語を用いた設計手法では，あらかじめプラットフォームと呼ばれる何らかのシステム構成を想定して，各構成要素および通信部分を記述する．しかしながら，効率的なシステムを設計する場合には様々な構成を探索する必要があり，このようなプラットフォームを想定した記述では，柔軟に構成を変更することは難しく大幅な構成の変更をする場合には，記述を大きく変更しなくてはならない．これは，主にシステムレベル設計記述言語が動作と構造を一体に記述しているためである．このため，動作と構造を分

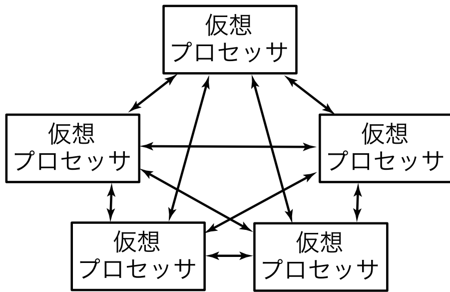


図 4: 仮想マルチプロセッサ (VMP) モデル

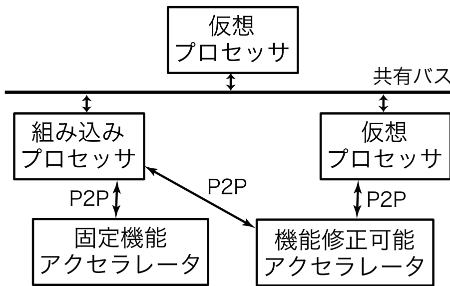


図 5: 仮想マルチプロセッサモデルの段階的詳細化

離して記述し，容易に様々な構成を探索する記述方法が重要になっていると思われる．

またシステムの複雑化に伴い，設計誤りによって設計・製造やり直しを複数回行うことになり，開発コストが増大してしまうという問題が顕著になってきている．これは主に専用ハードウェアの機能が固定であることが要因であり，プロセッサを用いることで解決可能であるが，今度は性能と効率の両立ができなくなってしまう．設計誤りの修正に必要な規模は一般的に小さいため，適切なプログラマビリティを選択することが可能であれば性能と効率を両立することが可能である．

このような考察に基づいて，本稿では SoC を効率的に設計可能な手法を提案する．基本的な流れとしては，まず仮想マルチプロセッサモデルと呼ばれる十分な柔軟性を持つ構成を想定し，通常の CMP 向けソフトウェア開発と同じように初期設計を行う．次に性能・電力効率の解析結果に基づき無駄な部分を特定し，その部分の専用化を進めることによって具体的な実装を段階的に決定する．このようにすることで，初期的な構成に依存することなく，無駄のない高効率なハードウェアを設計することが可能である．各構成要素の実装に際しては，様々なプログラマビリティも制御可能であり，必要とされる性能・効率の制約下で適切なプログラマビリティを持つハードウェアを生成可能である．以下，第 2 節で提案手法を説明し，第 3 節でこの提案手法をツールとして実装した SoC 設計環境 Cyneum について説明する．

2 提 案 手 法

2.1 仮想マルチプロセッサモデル

本提案手法では仮想マルチプロセッサ (VMP) モデルと呼ばれる構成に対して CMP 向けソフトウェア設計と同様のプログラミングを行い，またモデルの各要素の具体的な実装を段階的に

```
fifo_t f1, f2, f3;
par {
    proc p1 {
        complex_t samples1[2048];
        while(1) {
            ADC(&samples1, 2048);
            fifo_put(&f1, samples1);
        }
    }
    proc p2 {
        complex_t samples2[2048];
        while(1) {
            fifo_get(&f1, samples2);
            FFT(samples2);
            fifo_put(&f2, samples2);
        }
    }
    proc p3 {
        complex_t samples3[2048];
        while(1) {
            fifo_get(&f2, samples3);
            demodulation(samples3);
            fifo_put(&f3, samples3);
        }
    }
}
```

図 6: 簡略化 OFDM レシーバの構造記述例

決定していくことで設計を行う．VMP モデルの例を図 4 に示す．初期的な VMP モデルは複数の仮想プロセッサとその間を接続する通信路からなる．仮想プロセッサは典型的な命令セットからなり，どの命令も 1 サイクルで実行する．第 3 節で説明するように，我々の実装では LLVM [5] の仮想機械を仮想プロセッサとしている．また初期的なネットワークは全てのプロセッサ対をポイントツーポイント (P2P) で接続する完全接続型ネットワークである．

提案手法では段階的に仮想プロセッサの実装および部分ネットワークの実装を決定していく詳細化を行う．部分的に詳細化された VMP モデルの例を図 5 に示す．初期的な VMP モデルは最大限に柔軟性を持たせた構成となっておりプログラミングは容易であるが，一方で無駄が多く非常に効率が悪い．詳細化は部分的な専用化を進める事で，無駄を省き効率を向上させる手続きと見なす事ができる．

2.2 プログラミングモデル

第 1 節で述べたように，様々な構成の探索を容易にするためには設計の際に動作と構造を分離して記述できる事が望ましい．提案手法では，動作に関しては従来のソフトウェアと同様に C/C++言語などで記述する．構造に関しては C 言語に最小の拡張を施した言語を用いて記述する．簡略化した OFDM レシーバの構造の記述例を図 6 に示す．この C 言語拡張は SoC-C 言語 [6] に基づいている．SoC-C 言語は SoC 向けの設計記述言語であり，構造の変更の容易性を高める事に主眼を置いている

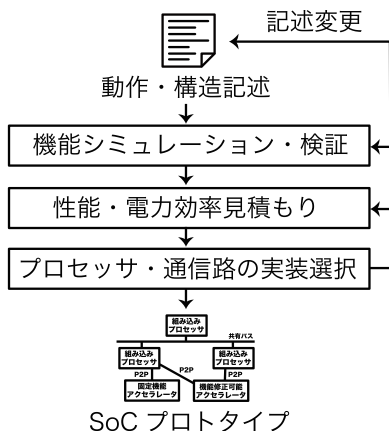


図 7: 提案手法における設計の流れ

点で本稿の目的と合致している．par 構文は指定されたブロック内の各プロセス (proc) を並列に実行する事を表しており，全てのプロセスの実行が終了した後にブロック全体の実行が終了する．この構文は設計における並列性を記述するための機構であり，レース条件やデッドロックの回避は設計者の責任となる．proc 構文は 1 つのプロセスを表しており，指定されたブロックの内部は逐次的に実行される．現在はプロセス間通信として FIFO による通信のみが利用可能である．fifo_put 関数は与えられたデータを指定されたチャネル上に送信し，fifo_get 関数は指定されたチャネルからデータを受信する．チャネルが一杯である，もしくは空である場合にはブロックする．ADC や FFT，demodulation といった動作に関する記述は通常の C 言語などで記述されており，構造記述を変更した場合でも影響はない．このようにして動作と構造を分離して記述する事で，様々な構成を容易に探索する事が可能である．

2.3 設計の流れ

提案手法における設計の流れを図 7 に示す．設計は第 2.2 節で説明した設計記述言語で記述する．初期的には各プロセッサが典型的なプロセッサであり，任意のプロセッサ間の通信が可能であるような VMP モデルを対象にして設計を行う．この段階では，設計記述は通常の CMP 向けソフトウェアと同様にコンパイルされ，汎用の PC 上で動作・機能検証することができる．次に各仮想プロセッサ上での性能や電力効率の見積もりを行う．実装が決定していない段階では，性能は仮想プロセッサの命令ステップ数で見積もり，また電力効率は演算器の使用効率やメモリのアクセス頻度，通信路上のデータ転送量に基づいて見積もりを行う．もしくは，可能な各実装方式に対してコンパイルや合成を実際に行い，設計空間の探索を行う事も可能である．また，通信路に関しても実行時の通信量や頻度に基づいて，性能や電力効率の見積もりを行う．

次の段階では，各仮想プロセッサと通信路の具体的な実装を決定する．仮想プロセッサの実装としては，組み込み向けプロセッサ，コンフィギャラブルプロセッサ・ASIP，固定機能専用アクセラレータ，製造後機能修正可能アクセラレータなどの中から設計者が選択する．プロセッサに関してはそのアーキテクチャ向けのコンパイラ，固定機能専用アクセラレータに関しては高位合成ツール，製造後機能修正可能アクセラレータに関しては次の節で説明する手法を用いて実装を行う．通信路についてもバスや P2P，ネットワークオンチップ (NoC) などの実装方式から選択する．上述の通り仮想プロセッサや未実装の通信に関しても性能・効率見積もりは可能であるため，部分的な詳細化の段階においても見積もりは可能である．設計の各段階において，見積もりの結果に応じて構成を変更する．第 2.2 節で説明したように，提案した設計記述言語では動作と構造を分離して記述するため，構成の変更は容易である．

3 SoC 設計環境 Cyneum

3.1 概　　要

我々は本稿で提案している設計手法の一実装として Cyneum という SoC 設計環境の開発を進めている．本節では Cyneum の概要について説明する．第 2.2 節で説明した設計記述言語で記述された設計を入力として，LLVM コンパイラインフラストラクチャ[5] を用いて SSA 形式の 3 番地コード形式の命令列からなる中間言語に変換する．この中間言語は以下で説明するようにプロセッサ向けコンパイラや高位合成，性能・効率見積もりといった多くの部分で共通に用いられる．Cyneum では仮想プロセッサの実装方式として以下を使用可能である．

- ARM や MIPS などの組み込み向けプロセッサ
- 高位合成によって生成される専用アクセラレータ
- 第 3.3 節で説明する製造後機能修正可能アクセラレータ

これらのコンパイル・合成手法については次節以降で説明する．通信路の実装設計については現時点では P2P による FIFO のみが使用可能である．性能・電力効率見積もり手法については第 3.4 節で説明する．

3.2 コンパイラ・合成

LLVM によって変換された中間言語は，LLVM が提供するバックエンドを用いて ARM や MIPS 向けのコードに変換することが可能である．また，Cyneum 上には典型的な高位合成アルゴリズムが実装してあり，中間言語からコントロールデータフローグラフを生成し，スケジューリング・バインディングを経て専用ハードウェアの RTL を生成する事が可能となっている．このようにコンパイラや合成，見積もり手法の間で中間言語を共有する事で，仮想プロセッサの段階から一貫した見積もりが可能となっている．

3.3 製造後機能修正可能アクセラレータ [7]〜[9]

近年の SoC 開発コストの増大と開発期間の短縮に伴い，高位合成を利用した設計手法の導入が進んでいる．高位合成によって生成されたハードウェアアクセラレータは高性能と高効率の両立が求められる様々な分野において利用されている．一方で製造故障や設計誤りによる製造後修正がますます重要となっており，製造後修正を可能とするプログラマブルハードウェアアクセラレータが注目されている [10], [11]．プログラマブルアクセラレータ (以下，アクセラレータ) の基本構成を図 8 に示す．アクセラレータは機能ユニット (FU)，FU 間を接続する配線部分，およびプログラマブル制御回路からなる．各 FU はあらかじめ決められた 1 つ以上の種類の演算を行うことが可能であり，

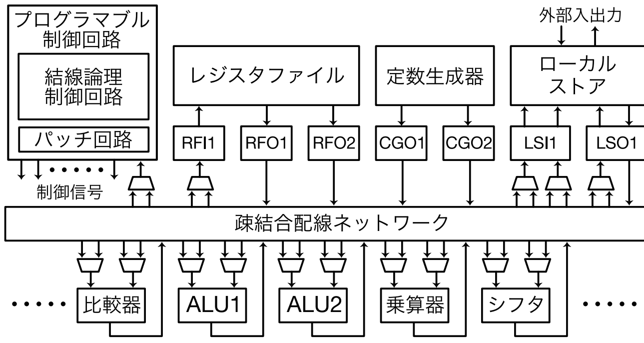


図 8: プログラマブルアクセラレータの基本構成

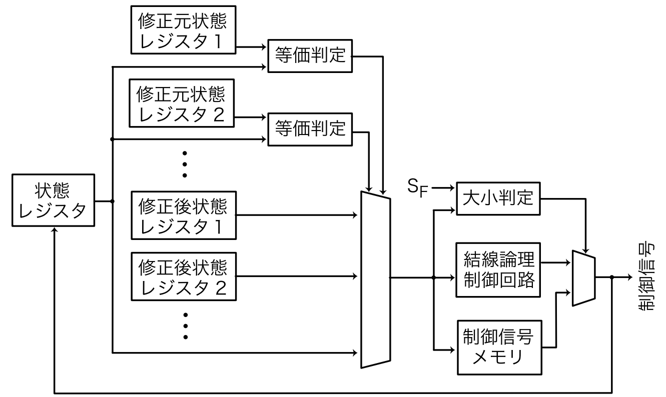


図 9: パッチ回路の基本構成

制御信号によって演算の種類を決定する．典型的な FU としては，ALU や乗算器，比較器，バレルシフタなどがある．レジスタファイルやローカルストア，定数生成器などのメモリ素子では，メモリ素子の書き込みポートや読み込みポートをそれぞれ FU とみなす．このようにすることで，合成およびコンパイルの際にメモリ素子へのアクセスを算術演算と同様に扱うことが可能である．各 FU の入力は他の FU の出力もしくはマルチプレクサの出力と接続される．同様に各 FU の出力も他の FU の入力やマルチプレクサの入力に接続される．マルチプレクサの入力は FU の出力と接続されている．マルチプレクサの制御信号によって各 FU は入力信号を選択する．レジスタファイル (RF) はレジスタの集合であり，複数の書き込みポート (RFI) および読み込みポート (RFO) を持つ．レジスタは局所変数値の格納に使用される．各 RF ポートの制御信号によって，どのレジスタにアクセスするかを決定する．定数生成器はあらかじめ決められた定数を出力することが可能であり，レジスタファイル同様，読み込みポート (CGO) への制御信号に基づいて定数を生成する．ローカルストアは主に配列やグローバル変数値を格納する RAM であり，また外部とのデータのやり取りにも使用される．ローカルストアも書き込みポート (LSI) と読み込みポート (LSO) を持つが，他のメモリ素子とは異なり，ポートはアドレスとデータの 2 つの信号線を持つ．ローカルストアの書き込みポートは書き込みイネーブルの制御入力を持つ．

プログラマブル制御回路 (以下，制御回路) は結線論理制御回路とパッチ回路からなり，現在の状態に基づいて FU やマルチプレクサへの制御信号を生成する．制御回路はデータパスが生成した 1 ビットの制御信号に基づいて次状態を決定する．パッチ回路は，結線論理制御回路のいくつかの状態に対して生成する制御信号を修正することができる．修正されない状態に対しては，結線論理制御回路が生成する制御信号がそのまま出力される．図 9 に示すようにパッチ回路は状態パッチ部分と制御信号パッチ部分からなる．ここで，結線論理制御回路に実装されている状態を $s_1, ...s_n$，パッチ回路に実装されている状態を $s_{n+1}, ..., s_{n+m}$ とする．状態パッチ部分では，結線論理制御回路のいくつかの状態をパッチ回路の状態に変換する．また，制御信号パッチ部分は，パッチ回路内の各状態 $s_{n+1}, ..., s_{n+m}$ の制御信号を生成する．

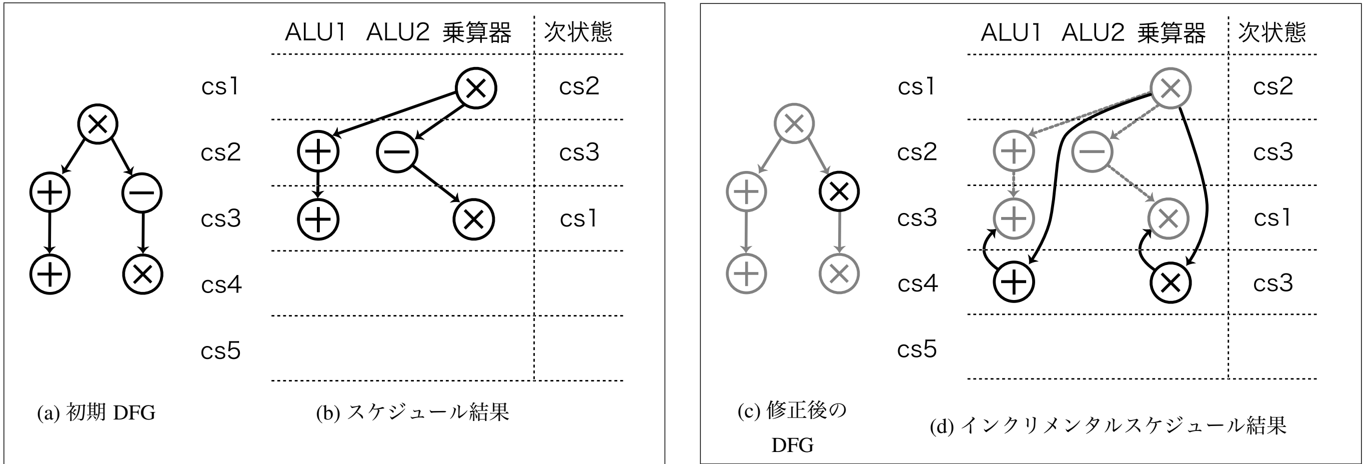
パッチ回路の動作原理を図 10 を用いて説明する．ここでデータパスは 2 つの ALU と 1 つの乗算器を備えている．図 10(a) に示す初期設計のデータフローグラフ (DFG) のスケジュール結果が図 10(b) であるとする．このスケジュールでは 3 つの状態 cs1, cs2, cs3 があり，状態遷移は cs1→cs2→cs3→cs1→ ⋯ の順で繰り返される．これらは結線論理制御回路で実装されている．次に機能修正後のデータフローグラフを図 10(c) に示す．この機能修正では減算が乗算に修正されている．修正された演算に対応する状態は cs2 であるため，cs2 に含まれている加算とともに再スケジュールを行う必要があり，新しい状態 cs4 を導入した再スケジュール結果は図 10(d) になる．この cs4 のスケジュール情報はパッチ回路内の制御信号メモリに格納され，また図 10(e) に示すように修正元状態レジスタ 1 を cs2 に，修正後状態レジスタ 1 を cs4 にする．修正元状態レジスタ 2 は使用しないため，到達しない状態を設定しておくことで，状態変換を無効化する．こうすることで，状態遷移は cs1→cs4→cs3→cs1→ ⋯ となり，機能修正が実現できる．

3.4 性能・電力効率見積もり

上述の通り，入力設計記述は LLVM [5] を用いて中間言語に変換されている．仮想プロセッサの実装が決定していない段階においては，LLVM 仮想機械上で中間言語を実際に実行し，そのステップ数やメモリアクセス量などから性能や電力効率の見積もりを行う．仮想プロセッサが具体的なプロセッサに実装された場合には，コンパイラによって実際にコードを生成することでより正確な見積もりを行う事が可能である．また，固定機能アクセラレータや製造後機能修正可能アクセラレータで実装する場合でも，実際に合成・解析を行うことで見積もりを行うことが可能である．

4 まとめと今後の課題

SoC では高性能・高電力効率を両立するため，複数のプロセッサに加えて多数の特定用途アクセラレータを搭載するヘテロジニアス構成となっており，そのアクセラレータの数は今後も増加していくと考えられている．マルチコア化が進むプロセッサの分野においても，熱や電力の問題から全てのコアを同時に動作させることができない問題が顕著となってきており，問題解決にはヘテロジニアス構成が有効であるとされている．このような多数のアクセラレータを持つ SoC の設計手法として，本稿



(a) 初期 DFG　(b) スケジュール結果　(c) 修正後の DFG　(d) インクリメンタルスケジュール結果

レジスタ番号	変換元状態	変換後状態
1	cs2	cs4
2	—	—

(e) 状態変換表

図 10: パッチ回路の動作例

では仮想マルチプロセッサモデルに基づく高速 SoC プロトタイピング手法を提案した．十分に柔軟性のあるモデルから，性能・電力効率見積もりによって無駄な部分を特定し，その部分の専用化を段階的に進める事で高性能と高電力効率を両立可能なハードウェアを設計可能である．また，我々が開発を進めている SoC 設計環境 Cyneum についても説明した．

今後は以下の課題に取り組む予定である．

実設計での評価: 実用的な例題を用いて提案手法を適用する事で高性能・高電力効率を両立するハードウェアを高い設計生産性で設計可能である事を示す．

再利用性の向上: 実際の開発プロジェクトでは一から設計をやり直すのではなく，過去の設計に対する機能追加や改善によって設計を進める事が一般的である．この点に関しては，提案手法を改良してインクリメンタルな設計を可能にする事で解決する事が可能であると考えている．

文　献

[1] W. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield, “Efficient embedded computing,” *Computer*, vol. 41, no. 7, pp. 27 –32, 2008.

[2] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors 2009 Update*, 2009.

[3] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: Reducing the energy of mature computations,” in *Proc. of ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2010, pp. 205–218.

[4] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *Proc. ACM Int. Symp. Computer Architecture (ISCA)*, Jun. 2010, pp. 37–47.

[5] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, May 2004, p. 75.

[6] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin, “SoC-C: Efficient programming abstractions for heterogeneous multicore systems on chip,” in *Proc. of ACM Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Oct. 2008, pp. 95–104.

[7] 吉田 浩章, 藤田 昌宏, “潜在的多様性を考慮したプログラマブルハードウェアの高位合成手法,” 電子情報通信学会技術研究報告, vol. 109, no. 462, pp. 67–72, 2010 年 3 月.

[8] 吉田 浩章, 藤田 昌宏, “製造後機能修正可能な高電力効率アクセラレータの高位設計手法,” 情報処理学会 DA シンポジウム 2010 論文集, pp. 45–50, 2010 年 9 月.

[9] 吉田 浩章, 藤田 昌宏, “動的パッチ読み出し機構を備えた製造後機能修正可能アクセラレータ,” 情報処理学会研究報告, vol. 2010-SLDM-146, no. 6, pp. 31-36, 2010 年 10 月.

[10] M. Reshadi and D. Gajski, “A cycle-accurate compilation algorithm for custom pipelined datapaths,” in *Proc. IEEE/ACM Int. Symp. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2005, pp. 21–16.

[11] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, “Bridging the computation gap between programmable processors and hardwired accelerators,” in *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2009, pp. 313–322.