ANSI-Cで記述されたアルゴリズムをFPGAでハードウェアにする

新発想のツール Impulse C を使ったソフトウェアのハードウェア化手法

倉重 克己

本稿では，ANSI-Cで記述されたアルゴリズムをハードウェア化でき，同時にハード/ソフト協調設計（コデザイン）も可能にするImpulse C/CoDeveloperを紹介する．

Impulse C/CoDeveloperは，組み込み設計支援を強く意識したツールであり，既存のWindowsベースの開発環境に自然に入り込む方式を取る．CoDeveloper自身もGNU MinGWベースの簡単な開発デバッグ環境を提供している．ハード/ソフトのインターフェース部を隠ぺい化する手段も提供しており，ソフトウェア・エンジニアがハードウェアも含めてシステムを設計する道を大きく開く．

ハードウェア設計を対象にしていても，主たるユーザ層をソフトウェア・エンジニア，または組み込み設計者においている点が，従来のEDAツールと大きく異なる．もちろん，このことはハードウェア・エンジニアにとっても悪いことではない．

C言語での設計とImpulse C /CoDeveloperの概要

まず，このようなツールが誕生する背景に少し触れておく．現在のソフトウェア設計者，ハードウェア設計者は次のような問題を抱えている．

● ソフトウェア設計者のしごと

世の中のほとんどすべての電気製品にはCPUが入り，今やPCやWS上のプログラミングのほうがマイナになってしまうのではと思われるほど，組み込み設計が重要になっている．

それゆえに，必ずしも標準化して抽象化されたものではないむきだしのハードウェアと対峙しなくてはならない場面が多くなり，ソフトウェア設計者もハードウェア設計に深く関わる必要性が出てきている．

また，ソフトウェア処理のボトルネック部をハードウェア化したいという希望も多くあるが，ソフトウェア設計者の設計環境から，それを可能にする容易な手段がない．

● ハードウェア設計者のしごと

RTL設計で組み合わせ回路のゲートの塊の設計から開放されたと思ったのも束の間，ムーアの法則によって回路規模は拡大を続け，それにともない，果てしない高機能化（複雑化）と短納期を求められ，設計者は今やレジスタの塊と日夜格闘している．

昨今，この問題を解決するため，抽象度をもう一段上げたソフトウェア設計と同様のアルゴリズム水準での手法を可能にする手段がSystemCをはじめとして議論され，導入が検討され始めている．

● ソフトウェア設計者，ハードウェア設計者共通のしごと

伝統的には，システム設計者の判断で設計をハード/ソフトに分割し，それぞれの担当が別々に設計して両者がある程度の完成度になったときから統合デバッグを始めていた．

この手法では，ハード/ソフトのインターフェースや統合スループットの問題が後で判明し，その時点からの手戻りは，現在の多くの複雑なシステムでは現実的な手段ではなくなっている．そのため，最初からのソフト/ハードの協調設計が強く求められている．

こういった問題を解決するためのアプローチには，さまざまな方法が考えられ，現実にC言語ベースのツールが出ている．CoDeveloperは，特に「ソフトウェア設計者がハードウェア設計に入り込める手段を提供すること」を第一の目標として企画された．そのため，C言語から仕様を引いたり加えたといった新しい言語ではなく，ソフトウェア設計者の仕事上の母国語ともいうべき標準のANSI-Cをそのまま使い，C言語の知識をハードウェア設計にそのまま再利用できるようにした．

当然，過去のC言語による設計資産も再活用しやすく，一からの設計ではなく，確立されているアルゴリズムをハードウェア化するといった道も開ける．

図1でLagacy C algorithmとして示している部分が過去の資産である．なお，**図1**で網かけした部分はImpulse Cでの設計をサポートするためのCoDeveloperの要素である．

Impulse Cの仕様

ハードウェアを生成するにあたり，ANSI-Cにハードウェア表現を可能にする並列動作，構造記述などの仕掛けを加えなくてはならない．Impulse Cで採用されている手法はSystemCと同様である．

SystemCが標準ISO C++を使い，クラス・ライブラリを加えて実現しているように，Impulse Cでは標準のANSI-Cに新

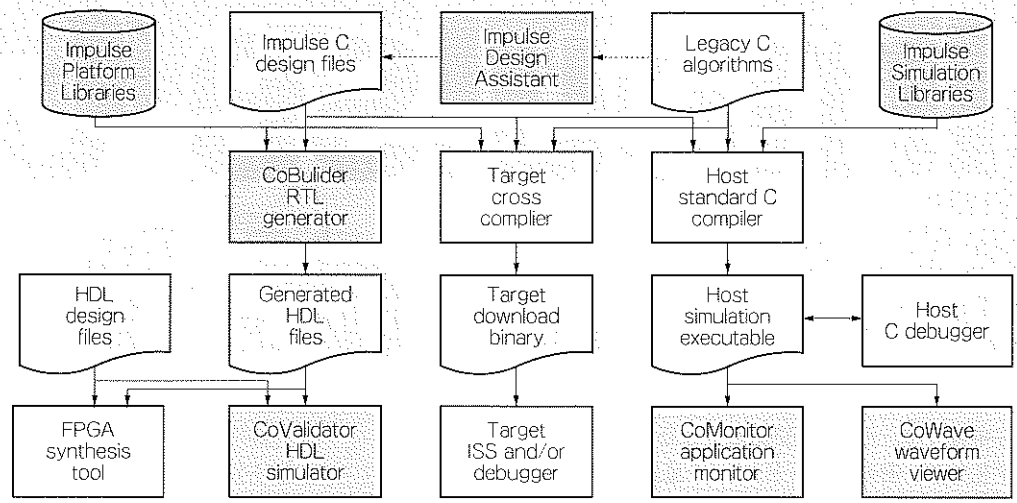


図1
CoDeveloperのツール構成
※網かけの部分はCoDeveloperの要素

表1　Impulse CとSystemCの対比

	Impulse C	SystemC
言語	ANSI-C	ISO C++
ハードウェアサポートのための拡張方法	ANSI-C上で新しい型と組み込み関数定義	C++上で新しいクラス・ライブラリ定義
開発システム	CoDeveloper	各種
ターゲット	おもに組み込み	現在LSI設計が主

しい「型定義」と「組み込み関数定義」のライブラリを加えてサポートしている．その拡張仕様が「Impulse C」と名付けられている．そして，Impulse Cで表現されたハード/ソフト協調設計のシミュレータと動作合成を含む開発環境がCoDeveloperである．**表1**にImpulse CとSystemCの対比を示す．

また，PSP (Platform Support Package) と呼ばれるターゲットのバス仕様を記述するライブラリが完備されている場合には，CoDeveloperの動作合成ツール (CoBuilder) は，単にハードウェア化する部分の動作合成 (VHDL) だけでなく，ハード/ソフトのインターフェースに関わる部分のハードウェア部 (VHDL) とソフトウェア部 (C) も同時に生成する．PSPにより従来避けて通れなかっためんどうなハード/ソフトのインターフェース部が隠ぺいされ，プロセスという抽象的なソフト/ハード共通概念でシステム全体を扱えるようになり，ソフトウェア・エンジニアも容易にハードウェア設計に参加できるようになる．

Impulse Cの仕様，CoDeveloperの動作合成，そしてPSPの考えは米国ロスアラモス国立研究所で研究開発されたStreams-C環境を元にしている．

Impulse Cでのシステム・モデリング

Impulse Cでは，

システム＝独立したいくつかのシーケンシャル・プロセスが並列走行しながら，必要に応じて通信して所定の目的を果たすもの

と考える．

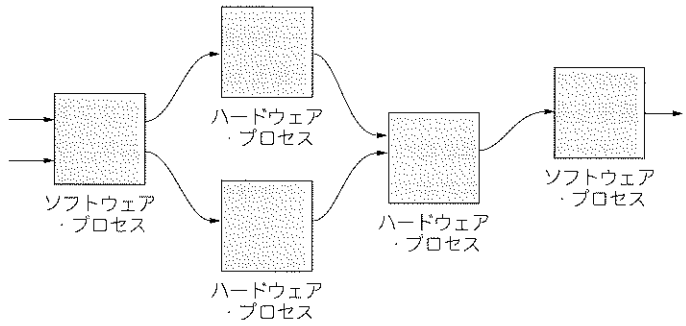


図2　CSPの模式図

この考えをCSP (Communicating Sequential Process) と呼ぶ．Impulse Cでの各プロセスは「シーケンシャル」なので，C言語で記述可能な通常のサブルーチンと本質的に違いはない．ただ，プロセス間の通信にImpulse Cのライブラリを使う (**図2**)．

「並列性の粒度」としては，CSPのプロセスは粗粒度に位置付けられるが，CoDeveloperの動作合成モジュールであるCoBuilderは，ループ内部の処理をパイプライン化するより細かな「並列性の粒度」の制御をサポートする．すなわち，Impulse Cの記述は2段階の「並列性の粒度」を表現できる．

1）プロセスとプロセス間通信要素

各プロセスは，**図3**のようなImpulse Cライブラリで定義されている要素であるストリーム・バッファ〔Stream (FIFO)〕，信号 (Signal) と共有メモリ (Shared Memory) を使って互いに通信する．基本的に，モデリング上の制約は，通信にこれらの要素を使うことに限られ，後は自由な記述が許される．

プロセス間の同期はこれらの通信要素で行われる．特に制御付きのストリーム・バッファ〔Stream (FIFO)〕での同期がImpulse Cの特徴である．ストリーム・バッファ長を適切に取ることで淀みなくデータが流れ，自然な同期が取れ，スルー

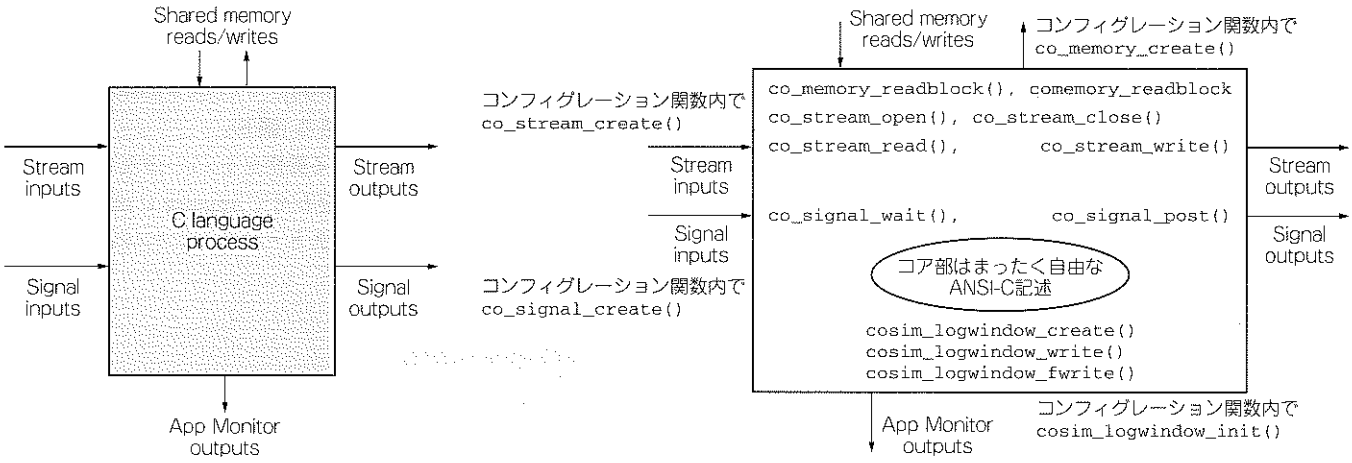


図3　Impulse Cのプロセス

プットを上げることが期待できる手法である．これは，Impulse Cの元になったStreams-Cの名前の由来でもある．

プログラミング・モデルとして，Impulse Cの各プロセスと通信要素は初期化時に定義と同時に生成され，いったん生成されたものが消滅することはない．このようなハードウェアの特性を反映している点は，ソフトウェアにおいて一般のプロセスやスレッドの生成/消滅が自由であることと大いに異なるが，動作シミュレーション時のImpulse Cプロセスは当然スレッドでモデル化される．なお，AppMonitor (CoMonitor) はデバッグのために，プロセス内の変数値をシミュレータに渡し，観測可能にするしかけである．

図3 (b) は，実際に使われるImpulse Cの関数を，**図3 (a)** との対比で示したものである．一般的なI/Oアクセスやインターフェース・ライブラリと似た雰囲気があり，ソフトウェア・エンジニアは関数名から，ある程度のことを想像できるかもしれない．

このような，入出力部がImpulse C関数を経由するサブルーチンを，Impulse Cのプロセスとして定義し，これらを相互につなぐことでシステムを表現する．Impulse Cでのハードウェア化はこのプロセス単位で行われる．

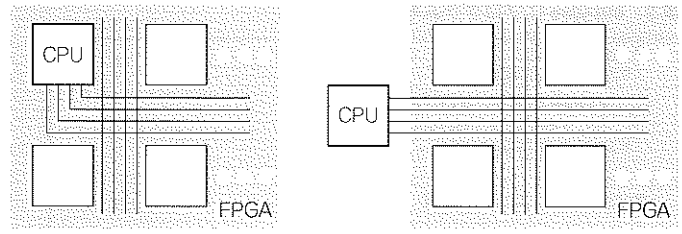
2) コンフィグレーション関数

プロセス間の通信，結合を表現する記述をシステムの構成を表す特別な関数，コンフィグレーション関数で定義する．

コンフィグレーション関数は，システムの最上位でユーザが設計したプロセスと，Impulse Cライブラリで定義してある通信要素をシステム構成に合わせてインスタンス化する．

これは，いってみれば，ハードウェア設計の最上位のネット表現と同じ概念だが，大きな違いは，相互につながれるプロセスはハードウェアとは限らず，ソフトウェアでもありうることであり，この点が重要である．

各プロセスをソフトウェア化するのか，ハードウェア化するのかも，この中で指定されるが，機能検証段階で機能シミュレーションではこの違いは区別されない．ハードウェア化する動作合成時に初めて解釈される．



(a) 組み込みCPUを持つFPGAプラットホーム　(b) CPUとFPGAの両方を搭載している基板プラットホーム

図4　考えられるプラットホーム

3) イニシャライズ関数とプラットホーム指定

イニシャライズ関数は，プログラミング・モデルとしては，コンフィグレーション関数を呼び出し，Impulse Cの各プロセスと通信要素を定義と同時に生成して，実行可能にする．

実装的には，コンフィグレーション関数で構成したシステムを実装へ渡す最終指定をする役割をになう．

すなわち，コンフィグレーション関数を指定して選択するプラットホームへのマッピングを動作合成過程に指示する．Impulse C/CoDeveloperは，設計の実装対象を「プラットホーム」と考え，実装対象のプラットホームの種類を指定する．

「プラットホーム」は，ソフト/ハード両方のプログラミング資源を持つシステムと規定され，現実には「組み込みCPUを搭載可能なFPGA」〔**図4 (a)**〕と，「CPUとFPGAが実装されており，ハード/ソフト両方のプログラミング・リソースが開放されている基板」〔**図4 (b)**〕の2種類が対象になる．

CoDeveloperのGUI環境からのコントロールでは，CoDeveloperのダイアログ経由でのプラットホーム指定が優先される．また，合成HDLだけを取り出すGenericモードもある．

現在，CoDeveloperは，組み込みCPUを持つFPGAプラットホームとして，Nios（Altera社）とMicroBlaze（Xilinx社）をサポートしている．2番目の種類の基板プラットホームは，基板上のバス仕様などを統一した規格でライブラリ化する手順をImpulse社が用意する．これにより基板上のプロセスも抽象化されCoDeveloper上で一貫してサポートが可能となる．

4）Impulse Cのプログラム構成

Impulse CはANSI-Cの枠内の仕様なので，プログラムは当然main関数を持ち，その下に前述した必須のイニシャライズ関数，コンフィグレーション関数，プロセス関数がくる（**表2**）．

5）シミュレーションとデバッグ

Impulse CはANSI-Cの一つのアプリケーションなので，ANSI-C準拠のコンパイラやデバッガなど，各種のツールが使用できる．動作合成前の動作検証段階では，設計全体をWindows上のプログラムとしてシミュレーションできる．したがって，ソフトウェア設計や組み込み設計の現場で慣れ親しんでいるWindows上のVisualStudioやCodeWarrior，GNUのIDE環境を，そのまま活用できる（**図5**）．

プロセスの並列動作を観察する動作シミュレータ（CoMonitor）と，これらの既存環境を組み合わせてImpulse C設計のデバッグに利用できるようプロジェクト・マネージャ（CoManager）はできている．

表2　Impulse Cプログラムを構成する関数

- main関数
  - システムを起動
    - xxxxarch＝Initialize関数();
    - co_execute(xxxxarch);
  - ほかの部分はテスト・ベンチの一部
- Configuration関数
- Initialize関数
- プロセス関数
  - ソフト・プロセス関数には自由なテスト記述が可能
  - プロセス内での内部状態をCoMonitorに渡して観察
- ほかのプロセスとはならない関数
  - テスト・ベンチの一部
  - ソフト・プロセスのサブルーチン

実装の過程

1）動作合成とハード化

コンフィグレーション関数でハード化指定をしたプロセス関数は，CoDeveloperの動作合成ツールであるCoBuilderによってRTLレベルのVHDLに動作合成される．CoBuilderに通す前に，ハードウェア化するプロセスに対しては，必要に応じて多少の書き直しが必要になる．定義したImpulse Cでの入出力はそのままに，ハードウェア化できないC言語の表現を，等価な形でハードウェア化できる表現に書き換えなくてはならない（**図6，表3**）．

CoBuilderの動作合成には，スタンフォード大学，ロスアラモス国立研究所，Impulse社それぞれで開発された三つの強力なオプティマイザが備わっており，品質の良い合成を行う．だが，動作合成の特性を知っておくことで，より少ないリソースで高速に動作させるための表現を追求をすることができる．

Cの#pragmaで動作合成ユニットCoBuilderにループ内のパイプライン化を指示でき，また分割されるパイプライン・ス

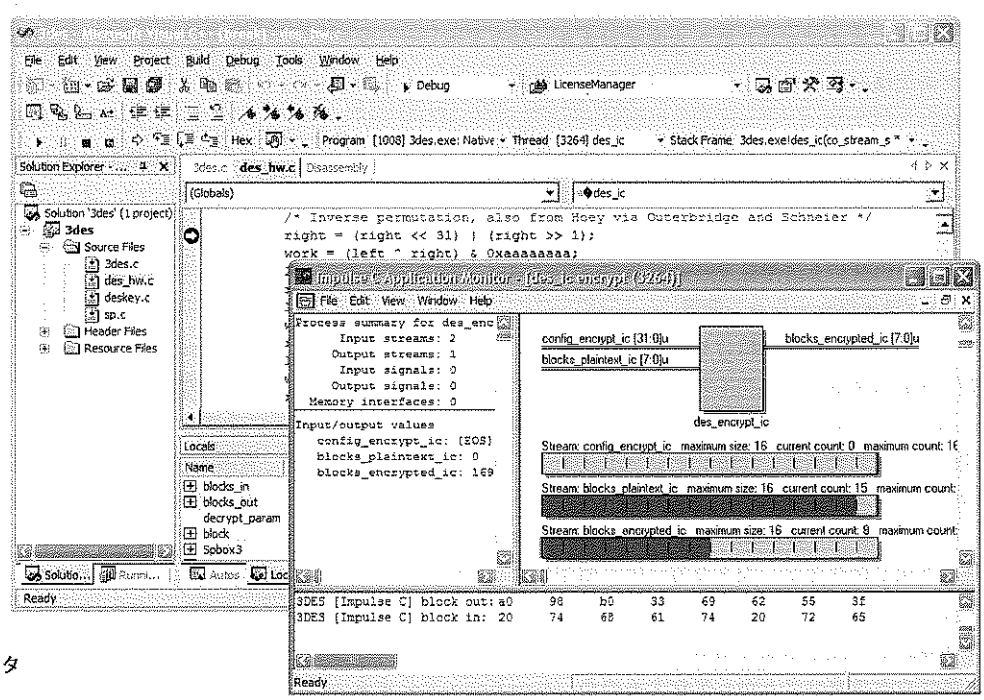


図5
VisualStudio.NET上で動作シミュレータCoMonitorを使ってデバッグ

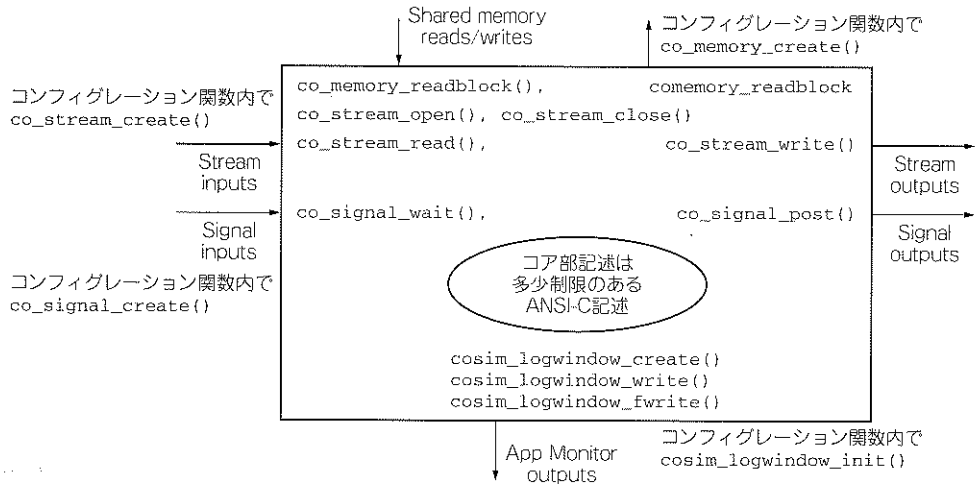


図6
ハードウェア化プロセスのコア部の表現を等価に書き換える

テージの最大遅延の指定も可能で，動作速度の制御ができる．

2) ハード/ソフトのインターフェースとPSP

CSPのプロセスは抽象的なものでハード/ソフトの区別はなく，Impulse Cでもそれを踏襲している．それゆえに，高い次元での設計を可能にし，ソフトウェア・エンジニアにハードウェア設計への道を開くが，実装時には，それまで隠れていたハード/ソフトのインターフェースの問題が顕在化してくる．

たとえば，FIFOでプロセスがつながれている場合を想定しよう．プロセス間が，ソフト-ソフトであればFIFOはメモリ上のデータ構造の問題として解決する．また，ハード-ハード間であれば物理的にFIFOバッファを互いに直結すればよい．

しかし，ハード-ソフト間となると，CPUとハードウェアはCPUバスを経由しての複雑なプロトコルでFIFOをエミュレートする必要がある．このような問題を解決し，プロセスをハードウェアとソフトウェアの区別なく同等に扱えるようにするためのインターフェース・パッケージがPSP(Platform Support Package)である．

PSPによりプロセスは，現実は**図7**中の下図だが，ユーザはインターフェースのない(見えない)**図7**中の上図のような扱いが可能となる．

PSPが用意されていると，CoBuilderはハード化プロセスのアルゴリズムをVHDLに合成するだけでなく，ユーザの設計に合わせたハードウェア側，ソフトウェア側両方のインターフェースを自動生成してくれる(**図8**)．

3) FPGAベンダ・ツールへの設計転送とプロジェクト化

CoBuilderで生成されたRTL-VHDL，インターフェース・モジュールと分離されたソフトウェア部のCコードは，Altera社のQuartusやXilinx社のISEといったFPGAベンダ・ツールのプロジェクトとして登録し，実装の準備を行わなくてはならない．

CoDeveloperは，この過程を半自動化してユーザの負担を軽減している．具体的には，プロジェクト・フォルダの指定をするだけで，QuartusやISEの要求する形で生成したファイル群が指定したプロジェクト・フォルダに転送される(**図9**)．

これにより，ユーザはインターフェースに関して必要となる知識とわずらわしさから解放される．

あとは，プロジェクトに関わる設定，NiosやMicroBlazeのメモリやペリフェラルに関わる設定，ピン配置など，お決まりの設定を行えば完成である．NiosやMicroBlazeが実装できるFPGAを搭載したボードがあれば，CoBuilderの合成出力から約

表3　ハードウェア化のための言語上の制約

● 再帰的な表現の禁止～ハードウェアは自分自身を呼べない
● プロセス内での，Impulse C関数以外の関数呼び出しの禁止 ～ハードウェアにはスタック・ポインタがない インライン関数，マクロはもちろん使用可能
● コンパイル時に確定値を持つポインタ以外のポインタ使用禁止
● 配列以外の複合データ型の禁止

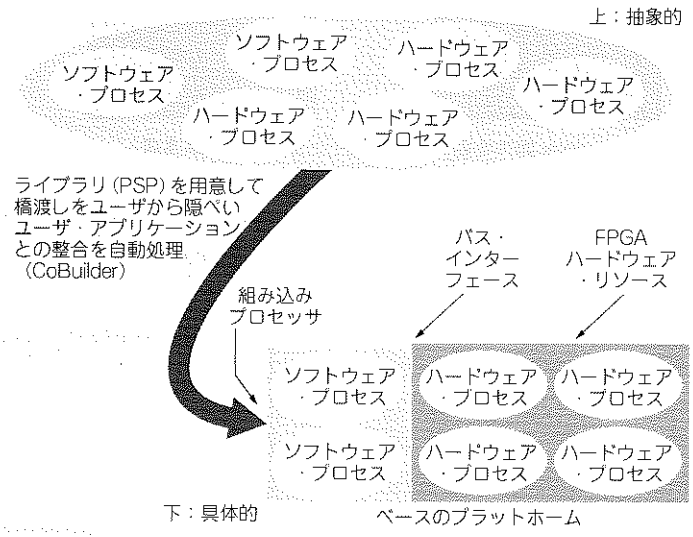


図7　PSPの役割

20分でFPGAコンパイル前まで作業を進めることができる．

4）VHDLモジュール生成ツールとしての活用

CoDeveloperは「プラットホーム」に対するハード/ソフト協調設計（コデザイン）への適用だけでなく，C言語で記述されたアルゴリズムをVHDLのRTLに変換する機能もサポートしている．

プロセス間通信要素の仕様は公開されており，Streamバッファ（FIFO）は対象のプロセスから変換されたVHDLと明確に分けて存在し，生成したモジュールを取り出すのは困難なことではない．ポート仕様がわかっているので，ほかのモジュールとの結合やシミュレーションは容易である．

FPGA設計の場合は，たとえ組み込みCPUのNiosやMicroBlazeを最終的には使わなくても，いったんいっしょにFPGAに組み込むことで，これらのCPUは「組み込まれた」FPGA設計デバッグ環境として機能する．

そこで自由にC言語でテスト・コードを記述し，FPGAをデバッグすることができる．FPGAのハードウェア部が完成した後にCPUを外せば済むことで，デバッグ効率が飛躍的に上が

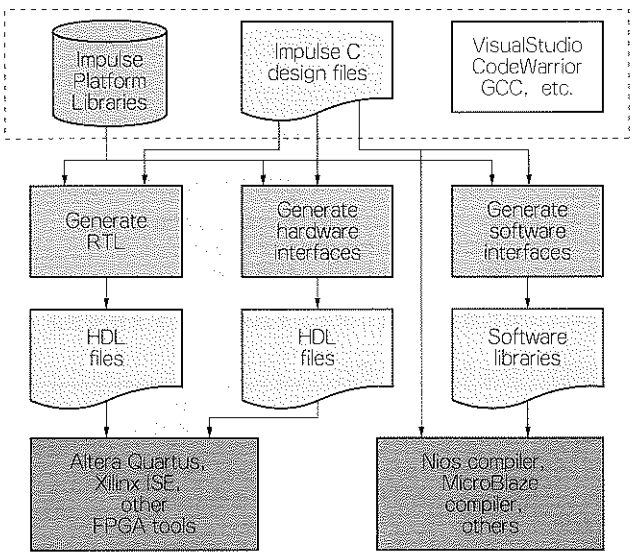


図8　CoBuilder動作合成エンジンのインターフェース生成

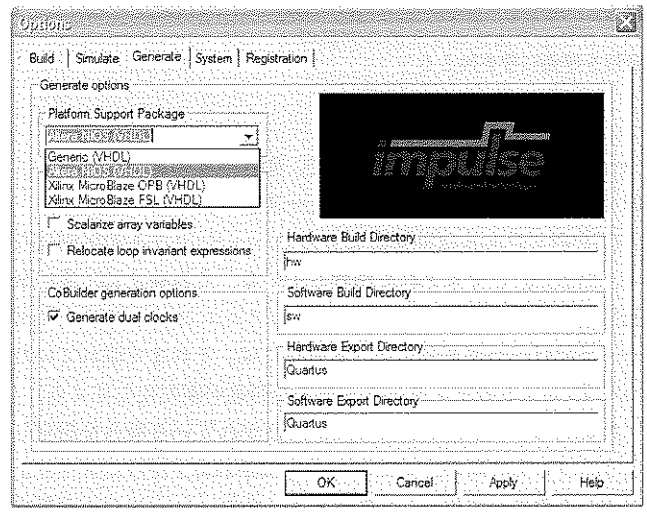


図9　プロジェクトの転送

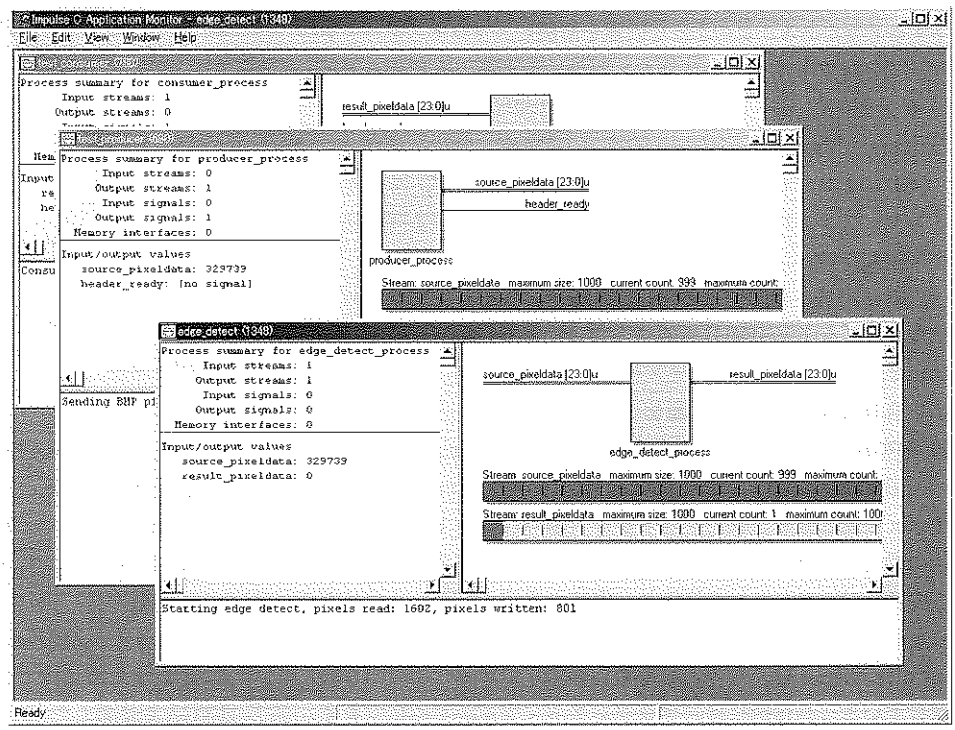


図10　EdgeDetectシミュレーション

ることが期待できる．

次に示す二つの実例は，このような手法によるものである．

Impulse C/CoDeveloper適用例

代表的な応用と思われる，画像処理と暗号システムの例を示す．図4(a)のタイプのプラットホームで，すでにPSPが製品化されている組み込みCPUであるNiosとMicroBlazeを使用した例を1例ずつ紹介する．

● 画像処理への適用例――エッジ検出

Impulse C/CoDeveloperを使用しての，仕様の検討からFPGAへの実装の流れを，実際に起こりそうなストーリ仕立てで実例で説明する．この例はAltera Stratix 1S10で確認された．

1）目的に合ったフィルタ方式とアルゴリズムの選択

エッジ検出画像フィルタとして多くの種類があるが，目的に合ったものはどれか選択しなくてはならない．同時に実現のアルゴリズムを検討しなくてはならない．Impulse Cで記述すると，この段階から実装までが終始一貫して少数のツール上で，慣れ親しんだANSI-Cで実現可能である．Impulse Cの記述は大枠のCSP的な処理フローを決め，コンフィグレーション関数定義と必要なプロセス関数群の一例を作ると，後はそれがテンプレートとなり，アルゴリズム部は図3(b)のコアに対応するので，まったく自由にフィルタの種類とその処理アルゴリズムの検討ができる．

1次微分(グラディエント)フィルタ・システムとそのアルゴリズムがデバッグされ，次にある2次微分(ラプラシアン)によるものを試し比較した．2次微分版は1次微分版で，アルゴリズム・コア以外の部分がテンプレート化されていたので，5分で完了した(図10)．

リスト1～リスト3に，これまでのImpulse Cソースの一部を示し詳細に解説する．以下の説明と併せて読んでもらいたい．

ここで少し，今回試した画像フィルタに触れておく．

画像フィルタの基本は注目している画素(ピクセル)と隣接する8画素を加えた3×3に切り出した9画素の線形演算を基本

リスト1　main関数，コンフィグレーション関数，イニシャライズ関数

```
int main(int argc, char *argv[])
{
    co_architecture my_arch;
    void *param = NULL;
    int c;

    printf("Impulse C is Copyright 2003 Impulse Accelerated Technologies, Inc.¥n");

    my_arch = co_initialize(param);
    co_execute(my_arch);

    printf("¥n¥nApplication complete. Press the Enter key to continue.¥n");
    c = getc(stdin);
    return(0);
}

co_architecture co_initialize(int param)
{
    return(co_architecture_create("edge_detect_arch","Generic_VHDL",config_edge_detect,(void *)param));
}

void config_edge_detect(void *arg)
{
    co_stream source_pixeldata, result_pixeldata;
    co_signal header_ready;
    co_process producer_process;
    co_process edge_detect_process;
    co_process consumer_process;

    IF_SIM(cosim_logwindow_init();)

    source_pixeldata = co_stream_create("source_pixeldata", UINT_TYPE(24), BUFSIZE);
    result_pixeldata = co_stream_create("result_pixeldata", UINT_TYPE(24), BUFSIZE);
    header_ready = co_signal_create("header_ready");

    producer_process = co_process_create("producer_process", (co_function)test_producer, 2,
                                         source_pixeldata, header_ready);
    consumer_process = co_process_create("consumer_process",(co_function)test_consumer, 2,
                                         result_pixeldata, header_ready);
    edge_detect_process = co_process_create("edge_detect_process", (co_function)edge_detect, 2,
                                         source_pixeldata, result_pixeldata);

    co_process_config(edge_detect_process, co_loc, "PE0");    // Assign processes to hardware elements
}
```

いつもペアでmainに記述
システムの起動

アーキテクチャの
インスタンス名

PSP名：
Generic_VHDL
altera_nios
xilinx_microblaze

コンフィグレーション関数名

コンフィグレーション関数へ
渡すパラメータ

CoMonitorの使用を宣言
IF_SIM/IF_NSIMマクロにより
シミュレーションか実装かの切り分け

通信要素名～信号名

通信要素の生成

プロセスのインスタンス名

プロセス関数名

プロセス関数の引き数の数

プロセスの生成

ハード化プロセスの指定

プロセス関数への実引き数の並び

PE0：プロセス要素＝選択FPGAの番号
(FPGA一つの搭載は必然的に0)

リスト2　プロセス関数 EdgeDetect 1 次微分版

```
void edge_detect(co_stream pixels_in, co_stream pixels_out)
{
    IF_SIM(int pixelsread = 0;)
    IF_SIM(int pixelswritten = 0;)
    int currentpos;
    int idx = 0;
    int addpos;
    short pixeldiff1, pixeldiff2, pixeldiff3, pixeldiff4;
    short pixelC, pixelN, pixelS, pixelE, pixelW;
    short pixelNE, pixelNW, pixelSE, pixelSW;
    short pixelMag;
    co_uint8 bytebuffer[BYTEBUFFERSIZE][3];
    co_uint8 nByte;
    co_uint8 nByteMag[3];
    co_uint24 nPixel;
    co_uint2 clr = 0;

    IF_SIM( cosim_logwindow log; )
    IF_SIM ( log = cosim_logwindow_create("edge_detect"); )

    co_stream_open(pixels_in, O_RDONLY, UINT_TYPE(24));
    co_stream_open(pixels_out, O_WRONLY, UINT_TYPE(24));
        ........

    IF_SIM(cosim_logwindow_fwrite(log,
           "Starting edge detect, pixels read: %d, pixels written: %d¥n", pixelsread, pixelswritten);)
    while ( co_stream_read(pixels_in, &nPixel, sizeof(co_uint24)) == co_err_none ) {
        IF_SIM(pixelsread++;)
        bytebuffer[addpos][REDINDEX] = nPixel & REDMASK;
        bytebuffer[addpos][GREENINDEX] = (nPixel & GREENMASK) >> 8;
        bytebuffer[addpos][BLUEINDEX] = (nPixel & BLUEMASK) >> 16;
        addpos++;
        if (addpos == BYTEBUFFERSIZE) addpos = 0;
        currentpos++;
        if (currentpos == BYTEBUFFERSIZE) currentpos = 0;

        for (clr = 0; clr < 3; clr++) {  // Red, Green and Blue
            pixelC  = bytebuffer[B_OFFSETADD(currentpos,0)][clr];
            pixelN  = bytebuffer[B_OFFSETADD(currentpos,WIDTH)][clr];
            pixelS  = bytebuffer[B_OFFSETSUB(currentpos,WIDTH)][clr];
            pixelE  = bytebuffer[B_OFFSETADD(currentpos,1)][clr];
            pixelW  = bytebuffer[B_OFFSETSUB(currentpos,1)][clr];
            pixelNE = bytebuffer[B_OFFSETADD(currentpos,WIDTH+1)][clr];
            pixelNW = bytebuffer[B_OFFSETADD(currentpos,WIDTH-1)][clr];
            pixelSE = bytebuffer[B_OFFSETSUB(currentpos,WIDTH-1)][clr];
            pixelSW = bytebuffer[B_OFFSETSUB(currentpos,WIDTH+1)][clr];

            pixeldiff1 = ABS(pixelSE - pixelNW);
            pixeldiff2 = ABS(pixelNE - pixelSW);
            pixeldiff3 = ABS(pixelS - pixelN);
            pixeldiff4 = ABS(pixelE - pixelW);
            pixelMag = MAX4(pixeldiff1,pixeldiff2,pixeldiff3,pixeldiff4);

            nByteMag[clr] = (co_uint8) pixelMag;
        }
        nPixel = nByteMag[REDINDEX] | (nByteMag[GREENINDEX] << 8) |
                 (nByteMag[BLUEINDEX] << 16);

        co_stream_write(pixels_out, &nPixel, sizeof(co_uint24));
        IF_SIM(pixelswritten++;)
    }

    IF_SIM(cosim_logwindow_fwrite(log,
           "Completed edge detect, pixels read: %d, pixels written: %d¥n", pixelsread, pixelswritten);)
........
    co_stream_close(pixels_in);
    co_stream_close(pixels_out);
}
```


とする．縦横 $m \times n$ の画像とし左下を原点として0番を振り，右へ番号を増やしながら $n-1$ に到達すると，0番の上に一つ進んで n 番として，これを繰り返す．最後の画素番号は $m \times n-1$ で右上となり，3×3の画素位置番号は注目画素を k とすると，**図11**のようになる．

A）1次微分（素朴なグラディエント）フィルタ

注目画素を挟む画素の縦，横，斜め2方向の画素ペアについて，ペア間の計四つの差分を求めそれぞれ絶対値を取る．四つ

リスト3　2次微分(変形8近傍ラプラシアン)のアルゴリズムの核

```
pixelMag = pixelC << 3;
pixelMag -= pixelSE;
pixelMag -= pixelNW;
pixelMag -= pixelNE;
pixelMag -= pixelSW;
pixelMag -= pixelS;
pixelMag -= pixelN;
pixelMag -= pixelE;
pixelMag -= pixelW;
if (pixelMag < 0)
        pixelMag = 0;
```

この画像フィルタリング・アルゴリズムの核を1次微分版と入れ替える
実装時にはプラグマ指定でループ内をパイプライン化した

$k-1$ $+n$	k $+n$	$k+1$ $+n$
$k-1$	k	$k+1$
$k-1$ $-n$	k $-n$	$k+1$ $-n$

図11
3×3画素フィルタリング・ウインドウ

(a) 原画　(b) 1次微分フィルタ結果　(c) 2次微分フィルタ結果

図12　画像フィルタリングの例

のうちの最大を注目画素に対する処理結果とする．これをRGBそれぞれに対して行う．

B) 2次微分(変形8近傍ラプラシアン)フィルタ

一般的な8近傍ラプラシアン・フィルタは，注目画素の値を8倍し，それからその画素を取り巻く8画素の値の合計を引き，それを注目画素に対する処理結果とする．ある目的に周りより明るい点だけを抽出する，結果がプラスになったものだけ取り，ほかは0とする．これをRGBそれぞれに対して行う．

図12(b)，**図12(c)**の結果から2次微分(ラプラシアン)のほうが目的に適切であると判断し，これを選択することにした．

2) 並列化の検討

次に，目的に合った効率的なプロセスの並列化を検討する．フィルタ方式とアルゴリズムの選択には**図13**のモデルを使った．

すなわち，一つのハードウェア・プロセス`edge_detect_process`で，24ビット(8ビット×3)の3原色データのストリームを受信しながらフィルタリングを行った結果の24ビット3原色データを送り出す処理を行うモデルである．

`edge_detect`関数内部を並列化はできない．Impulse Cの取るCSPモデルのプロセス単位はシーケンシャルであり，子プロセスはない．並列化の導入は，`edge_detect`関数の行っている処理を分割して複数のプロセス関数に分けることである．

RGB3原色の各色単位の並列化など，いろいろと考えてシミュレーションを行ったが，検討の結果，`edge_detect`関数の行っていた処理を二つのプロセスに分ける次の処理がハードウェア化された．

「データ入力，3×3のピクセル単位の切り出しに必要なバッファリングと必要なカラム・データの取り出し」を行う`prep_run`関数と，「フィルタリング処理と結果出力」を行う`filter_run`関数が用意され，`edge_detect`関数と置き換えられた．

その結果，**図14**のようなパイプライン処理が実行され，処理スピードの向上が期待できる．

3) 実装

●書き換え

シミュレーション完了を持って，実装に移すのにあたり，若干ソフトウェア側のコードの書き直しが必要になった．ターゲットはStartixが搭載されたAltea社のNios開発キット(1S10)ボードである．このボードの持つCompact Flashディスク機能を使えば手段はあったはずだが，不勉強でWindows上と同じような大きなビットマップ・ファイル形式のテスト・データ，変換データが取り扱えず，検証とデータに別の手段を取る必要があった．

32×32の画像サイズをNiosプログラムの一部として，これを敷き詰めて512×512とした．検証は変換されたデータの一部32×32を文字変換してRS-232-C経由でコンソール上で確認した．

Windows環境で十分な検証が済んでいるからこの程度の検証で安心である．

一般的に，実装に近くなればなるほど，デバッグのためのリ

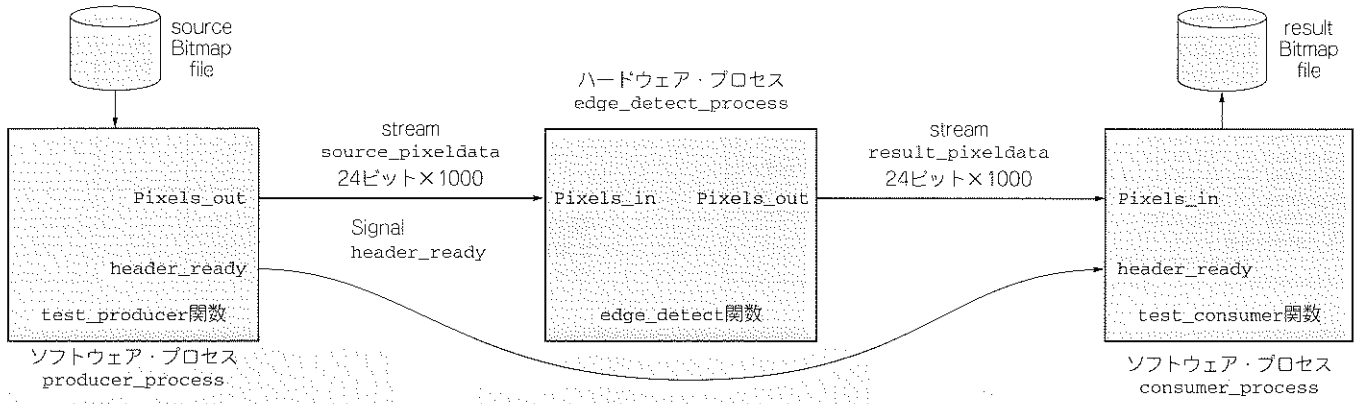


図13　画像フィルタ・シミュレーション・モデル

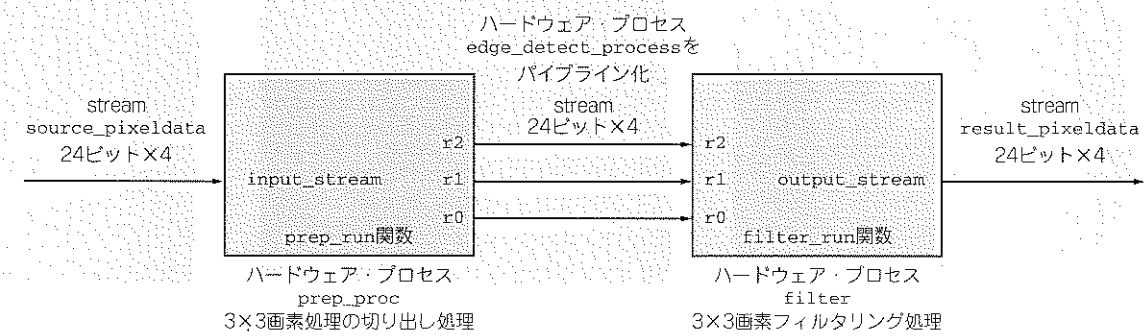


図14
プロセスを分割し，並列化を検討

```
------------------------------------------------------------
;Fitter   Summary                                  ;
------------------------------------------------------------
; Fitter Status ; Successful - Tue Mar 02 12:02:26 2004 ;
; Compiler Setting Name     ; timg              ;
; Top-level Entity Name     ; timg               ;
; Family                    ; Stratix            ;
; Device                    ; EP1S10F780C6       ;
; Total logic elements      ; 4,972 / 10,570  ( 47 % ) ;
; Total pins                ; 108 / 426  ( 25 % )      ;
; Total memory bits         ; 59,780 / 920,448  ( 6 % ) ;
; DSP block 9-bit elements  ; 0 / 48  ( 0 % )          ;
; Total PLLs                ; 0 / 6  ( 0 % )           ;
; Total DLLs                ; 0 / 2  ( 0 % )           ;
------------------------------------------------------------
```

図15　必要としたFPGAのリソース（Niosを含む）

ソースや手段が少なくなるうえに情報が不足しがちでめんどうが増す．デバッグの繰り返し時間も飛躍的に長くなる．抽象度の高い所で，最大限できることをキッチリやる姿勢が大切である．

●動作合成とビット・ファイル生成

CoBuilderは120行程度のハード化C言語ソースを1500行程度のVHDLに1分程度で合成し，NiosのPSPによりインターフェースも生成した．それと同時にNios上で走行するC言語部分も適切に切り分けられた．指定ディレクトリ（**図9**）にQuartusのプロジェクトに合った形で合成されたファイル群も転送された．SOPC BuilderでNios下のバス（Avalon）に結合してシステム構成を完了し，ピン配置などの設定後，30分程度のコンパイル時間でビット・ファイルを得た．Quartusのレポートの一部を**図15**に示す．Niosとペリフェラルを入れて約50%のリソースの使用率である．

動作速度としては，2クロック・サイクルで1ピクセルの処理が実現されている．クロック50MHzでRGB 3×8ビット・フルカラー画素数512×512のフィルタリング処理に約10msを要するが，十分に実用になる水準である．

図16は，Niosが合成されたFPGA上のフィルタから受け取った，32×32画素の処理結果を元に文字変換（明るさが増す方向により大きく複雑な文字を割当）し，コンソール出力したものである．図の縁だけが現れ，正しく処理されている．

ソフトウェア・プロセスとのstream通信によるCPUデータ転送であった`source_pixeldata`と`result_pixeldata`を，Shared Memory通信に変え，DMA転送版も実装して試した．DMA転送版はCPUデータ転送版よりデータ転送速度が約70倍速く，Impulse CでもDMAの効果が確認された．

● 既存のC言語資産の活用――3DES暗号の例

既存のANSI-Cソースを再利用してハードウェア化する例として，現在もっとも使われている暗号システムの一つである3DES暗号アルゴリズムのハードウェア化の例を取り上げる．

この例は，Xilinx Virtex IIデバイス上で動作を確認した．

データ通信で使われる暗号は高速で流れて来る平文ストリーム入力を，暗号化して同率のレートで送出しなくてはならないので高速処理が必要になる．これは逆も同様である．Impulse Cのストリームは，この処理の入出力に適したものと思われる．

1) 採用した3DESアルゴリズム

Qualcomm社のエンジニアであるPhil Karn氏が公開し，パブリック・ドメインになっている3DESのソース・コードを使用する(http://www.ka9q.net/)．これは，並列動作向けに開発されたものでなく，通常のCPUでの実行を想定したものである．

2) Impulse Cのモデル

ハードウェア化するため，Impulse Cのプロセス関数に適合させる変更をする必要がある．データ・ブロックとキーの送受信にストリームの読み込み/書き込みを使用したストリーム・ベースの通信にするための最低限の変更を加え，核のアルゴリズムはまったく変更を加えなかった．

CoDeveloperの統合マネージャであるCoManegerには，Impulse Cの通信要素をつなぐポート記述を補助する機能――DesignAssistant機能(**図17**)があり，旧資産をImpulse C記述へ変更する助けとなる．

「元のCコードによるCPU走行」と，Impulse Cによりハードウェア化した「専用エンジン」との同一データによる比較をするため，この二つのバージョンが一つのシミュレーション対象になるようなシステムをImpulse Cで記述(**図18**)した．

前例の画像フィルタと同様に，大きなテキスト・ファイルを使って大量のデータを扱うシミュレーション(**図19**)で暗号変換の動作検証をした．

3) 実装

実装は，同一FPGA上で，ハードウェア化した3DES専用エンジン版と，元のコードをMicroBlaze上のプログラムとして走行させた際，両者が同一のデータを同一量だけ処理することで，処理の速度を比較した(**図20**)．

前例の画像フィルタと同様に，シミュレーションで十分に動作検証はできているので，実装には決まった1ブロック・データを1000回繰り返し処理することで，ハードウェアが安定しているかどうかのテストで済ませた．

そのため，**図20**に適合するようにソフトウェア側に変更を加えた．この変更をシミュレーションで確認した後にCoBuilder

図16　フィルタリング結果の出力

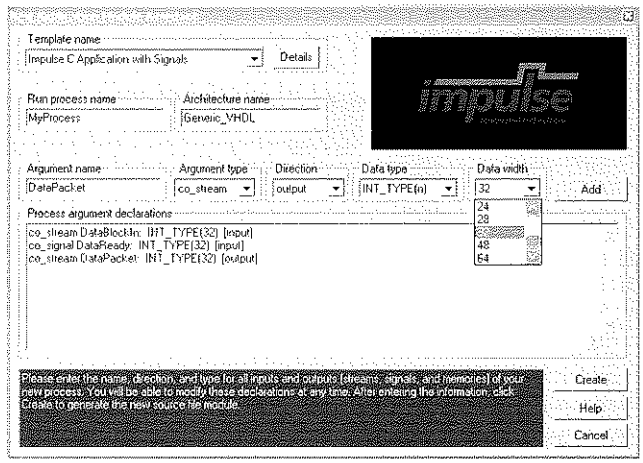


図17　DesignAssistant機能

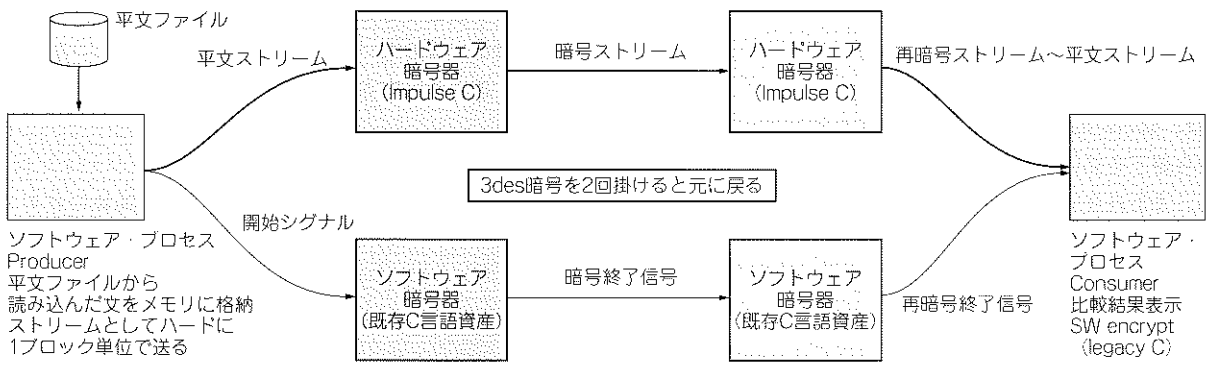


図18　3DES機能検証に使用したシミュレーション・モデル

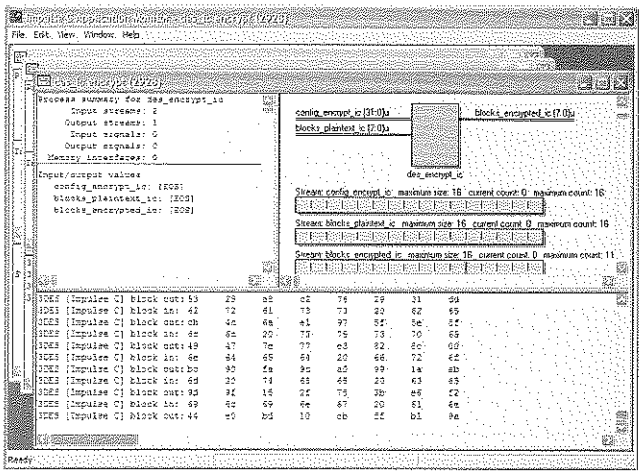


図19　3DES機能シミュレーション

```
Total Number 4 input LUTs: 7,823 out of  10,240    76%
  Number used as logic:                5,972
  Number used as a route-thru:            61
  Number used for Dual Port RAMs:        320
    (Two LUTs used per Dual Port RAM)
  Number used for 32x1 RAMs:           1,280
    (Two LUTs used per 32x1 RAM)
  Number used as 16x1 RAMs:                8
  Number used as Shift registers:        182
```

図21　必要としたFPGAのリソース(MicroBlazeを含む)

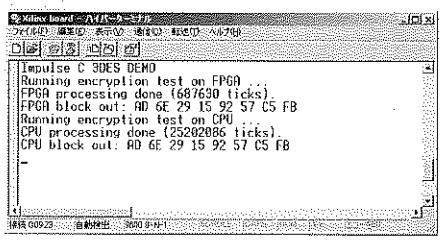


図22
3DES暗号処理
レポート出力

に渡した．一般に暗号アルゴリズムはシフト，ビット・スワップ，XORの塊のような構造を持ち，計算量が多く，展開はロジックの爆発ともいうべき大きな回路となる．この3DESの例もCoBuilderにより，180行のC言語ソース・コードが15分程度で約6000行のVHDLに展開された．

ISE上でのコンパイル時間は約90分を要した．使用したVirtex II XC2V1000のリソースは，MicroBlazeやそのペリフェラルを含めて**図21**に示すようになった．

図22は，MicroBlazeがVirtex上に合成された3DES暗号エンジンから終了を受けて，経過時間と受け取った暗号化データを表示し，また自身もソフトウェア版を走らせて経過時間と暗号化の結果を報告したものである．ticksの単位は走行クロック100MHzの周期なので10nsを意味する．

これを見ると，

ハードエンジン版	6.9 ms
ソフトウェア実行	252.0 ms

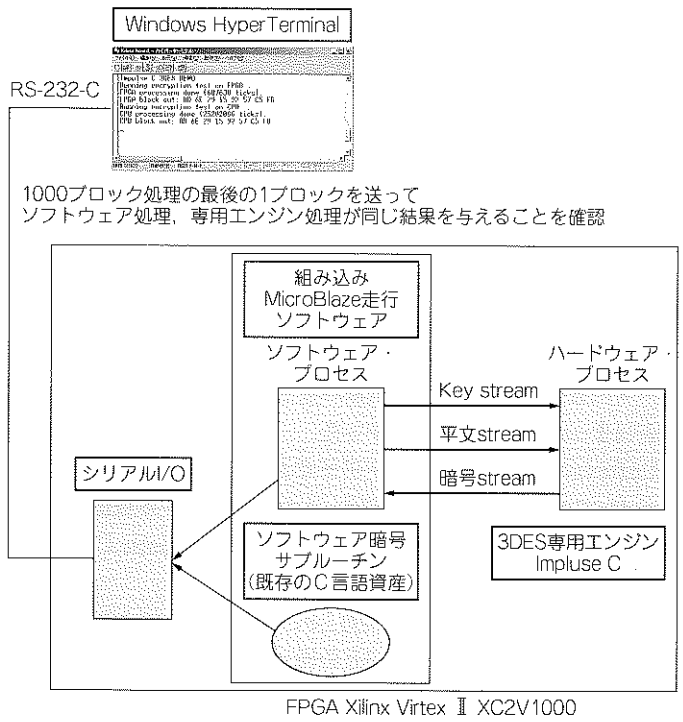


図20　3DES暗号実装検証モデル

となっており，約37倍ハード・エンジンのほうが速いということがわかる．この時間にはエンジンとCPU間のデータ転送など含まれるので，純粋にはもっと速く，40倍以上であろう．

これは一例だが，既存のCコードをハードウェア化するだけで40倍の高速化が実現できるというのは驚きである．

おわりに

今回，実験に使用したPCの仕様は，Celeron 1.6GHz，メモリ384Mバイトの環境である．

ソフトウェア設計でアセンブラが必要な場面はどのくらいあるのだろうか？　多くのCコンパイラはコンパイル結果のアセンブラ出力をする機能があるが，これを見て手を入れる人はどのくらいいるのだろうか？

現在，これらはかなり特殊なケースとなっていると思う．ハードウェア設計でも似たようなことが起こるのだろうか？

筆者は，案外近いうちに普通になるのではないかと想像する．多分，違う意見の方も大勢いると思う．議論してみたいものである．

PSPのような手段が普通になれば，スーパ・エンジニア1人がそれを用意すれば，ほかの多くの人は抽象の世界で済むはずだ．

くらしげ・かつみ　インターリンク　http://wwwilink.co.jp/