# 一気通貫 ESL[1]ツール Breakfast を用いたソフト/ハードの協調設計環境の構築

## バーチャル・プラットフォーム構築から高位合成まで

プロファウンド・デザイン・テクノロジー（株）
email：tsukamoto@profound-dt.co.jp
URL：http://www.profound-dt.co.jp

[1] ESL：Electronic System Level

# 目次

1. 本資料の目的 ..... 3
2. Breakfast の特徴 ..... 3
3. Breakfast を使った設計の流れ ..... 4
3.1. 例題デザインの仕様 ..... 4
3.2. 設計作業の流れ ..... 5
3.2.1. ブロック図の作成 ..... 5
3.2.2. メモリ・マップの作成 ..... 10
3.2.3. SystemC コードの生成 ..... 11
3.2.4. SystemC TLM シミュレーション ..... 12
3.2.5. バーチャル・プラットフォームのリリース ..... 13
3.2.6. 高位合成 ..... 18
4. まとめ ..... 18
5. （参考）MetaEdit+というツール ..... 19

## 1. 本資料の目的

本資料では、プロファウンド・デザイン・テクノロジー社において現在開発中の ESL ツール Breakfast[2]の概要について説明します。

## 2. Breakfast の特徴

Breakfast は SystemC TLM2.0 をベースとしたバーチャル・プラットフォームの構築から高位合成による RTL(Register Transfer Level)の自動生成までを一気通貫で統合したグラフィカル入力ツールです。Breakfast を使ってバーチャル・プラットフォームのブロック図を作成し、SystemC コードを自動生成します。また、画像処理に特化したブロック図の作成も可能であり、ブロック図から高位合成用の SystemC スケルトン（雛形）コードを生成します。Breakfast を使えば SystemC や高位合成に関する深い知識を有することなく RTL よりも抽象度の高い設計が可能となり、設計効率を改善することができます。もちろんバーチャル・プラットフォームは使わずに、高位合成用のグラフィカル入力ツールとしても使用することが可能です。

また、Breakfast ではメモリ・マップを作成することができます。作成したメモリ・マップから自動生成されるアドレスデコーダ部分の SystemC コードは、バーチャル・プラットフォームと高位合成の両方において同じものを使用します。このためバーチャル・プラットフォームと実際に実装するハードウェアの動作の間に、より一貫性を持たせることが可能となります。

Breakfast では CPU モデルとして QEMU[3]、ハードウェアモデルとして SystemC、高位合成ツールとしては米国 Forte 社の Cynthesizer をサポートしています。また、QEMU と SystemC 間の接続には TLMu と呼ばれる最新の技術を採用しており、GreenSocs の技術やプロセス間通信といった技術は使用していません。

なお、モジュール間 I/F およびバス I/F に関しては TLM レベルからピンレベルへの変換を自動で行います。これは TLM レベルからピンレベルへの変換ライブラリを事前に用意することにより実現しています(Forte 社の技術)。モジュール間 I/F としては vld/busy といった

---

[2] Breakfast は開発コード名です。
[3] フリーの CPU エミュレータ。Google 社の Android 開発プラットフォームにおいても使用されている。また、最近ではザインリンクス社が自社のプラットフォーム Zynq に対応した QEMU ソースコードを公開している。

制御信号を使った一般的なストリームデータ I/F をサポートしています。また、バス I/F に関して現時点では AMBA の AHB バスのみ対応（ただしプロトタイプとして）していますが、必要に応じてライブラリを開発していく予定です。

## 3. Breakfast を使った設計の流れ

### 3.1. 例題デザインの仕様

本資料では、下記のサンプルデザインを例にとり Breakfast の概要について説明します。このデザインのミッションは、「既存 SoC においてソフトウェア処理を行っているいくつかの画像処理を、ハードウェア化することにより高速化する」というものです。デザインに関する概要は以下の通りです。

1．既存の SoC は ARM926 ベースであり、OS として Linux(Debian)が搭載されている。またソフトウェア開発用ボードとして ARM 社の Versatile Platform Basedboard を使用している。
2．高速化したい（ハードウェア化したい）画像処理は以下の通り。
    ・フィルタ処理
    ・ズームアウト処理
    ・誤差拡散処理
    動作周波数は xxxMHz。スループット 1(1 画素/1 クロック)の完全なパイプライン回路として設計。今回はこの部分を高位合成の対象とする。
3．画像データをメモリ（DRAM）に対して読み書きする場合は DMA（読み出し用 DMA と書き込み用 DMA の 2 つ）を使用。ただし SystemC コードは既存のものを使用することとする。今回は高位合成の対象外とする。
4．実際の SoC（実機）ができる前にソフトウェアの先行開発を行う。このための SoC に対するバーチャル・プラットフォームを SystemC により構築し、ソフトウェア開発者に使ってもらう。

設計にあたり最初にやるべきことは、「高速化したい画像処理部分をハードウェア化したら本当に高速化できるのか？」について確認することです。いわゆる性能見積もり、あるいはパフォーマンス見積もりと呼ばれるものです。せっかくハードウェア化して SoC ができても実機で動かしてみると、ソフトウェアで処理していた時とあまり処理速度は変わらない・・・などという結果に終わっては大変です。

しかし現時点では Breakfast には性能見積もり機能は搭載されていません。性能見積もり機能に関しては現在検討中であり、2013 年度中の開発を予定しています。したがって今回の設計においてはこの性能見積もりはなんらかの方法ですでに実施しており、ハードウェア化により高速になるということが確認できているものとします。

### 3.2. 設計作業の流れ

Breakfast を用いた設計作業の流れは以下のようになります。

・ブロック図の作成(ゼロから作成します。既存の SystemC コードを読み込み、ブロック図を自動生成することはできません)
・メモリ・マップの作成
・SystemC スケルトン・コードの生成および SystemC コードの追記
・SystemC TLM(Transaction Level Modeling)シミュレーションの実施。抽象度は LT(Loosely Timed)
・バーチャル・プラットフォームのリリース（抽象度は LT）
・高位合成の実施および RTL シミュレーション。

#### 3.2.1. ブロック図の作成

最初にハードウェアのブロック図を作成します。ブロック図の完成形を図 1 から図 3 に示します。

図 1 は今回のデザインにおけるトップ階層のブロック図です。このブロック図から SystemC コードを Breakfast により自動生成します。なお、自動生成されたトップ階層の SystemC コードに関しては人手によるコードの追記は全く必要ありません。Breakfast は CPU モデル(Versatile のモデルを含む)として QEMU を採用していますが、QEMU と SystemC モデルを接続するには高度な知識を必要とします。しかし、Breakfast を使えばそのような知識を必要とせず、簡単に QEMU の CPU モデルと SystemC を接続することができます。

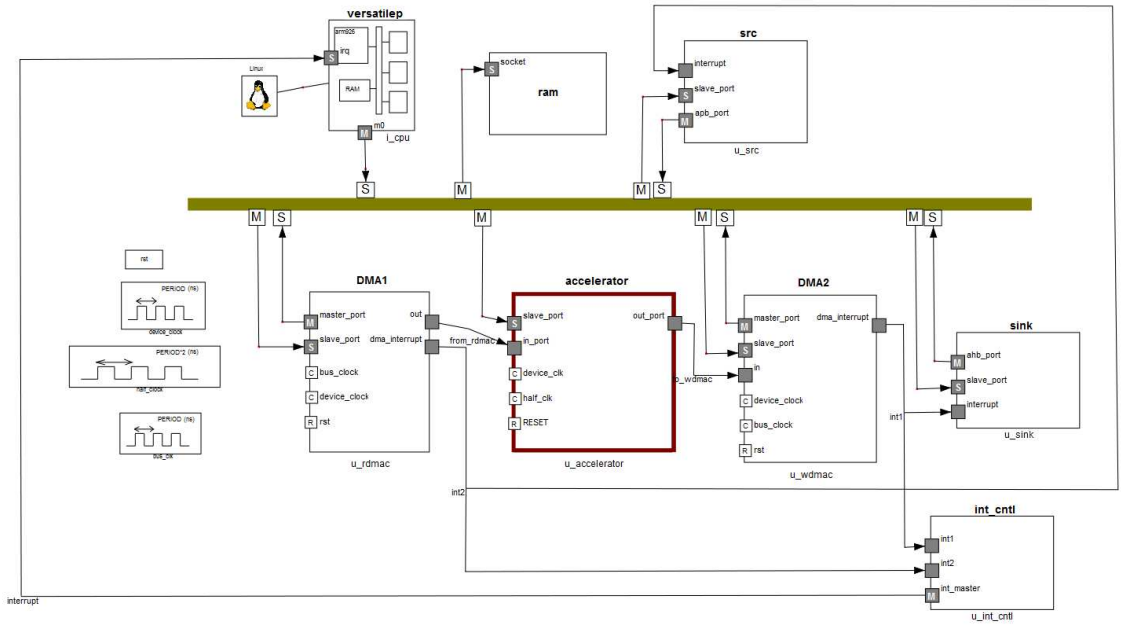


**図 1　バーチャル・プラットフォームのトップ階層**

図 2 は画像処理モジュール(accelerator という名前のモジュール)の下階層（サブモジュール）に関するブロック図を示しています。設計が面倒なラインバッファは１つのシンボルとして配置し、ラインバッファに対しては必要なパラメータを入力するだけでよく、わざわざ人手により設計する必要はありません。また各モジュール間のインタフェース部分に関しても人手による設計は不要です。ラインバッファやモジュール間のインタフェース回路にはバグが潜む可能性が高いので、この部分の設計を自動化することは非常に重要です。

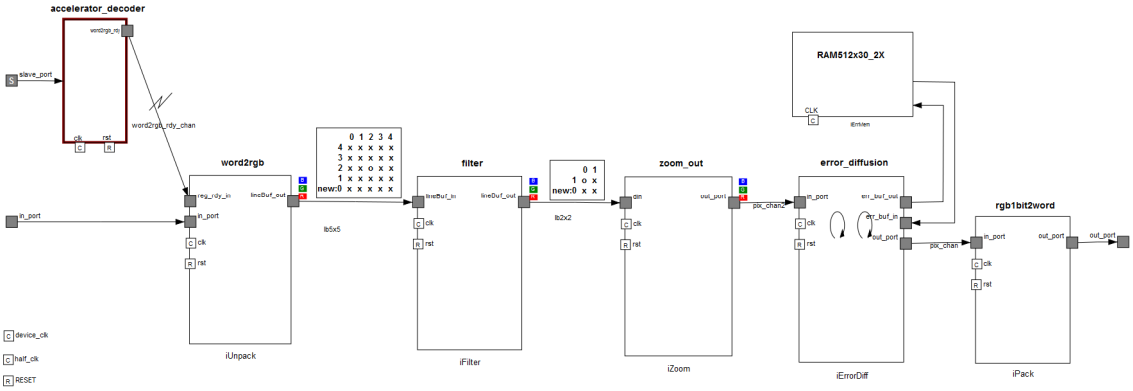


**図 2　accelerator モジュール（画像処理モジュール）の下階層**

図 3 は filter モジュールを設計する場合に作成するブロック図です。filter モジュールは accelerator モジュールのサブモジュールです。filter モジュールは 1 画素ずつデータを読み

込み、(5 画素)x(5 画素)のマトリクスを作成し、そのマトリクスに対して演算を行います。従来の RTL 設計ではマトリクス処理を行う回路の設計はかなり面倒な作業でした。

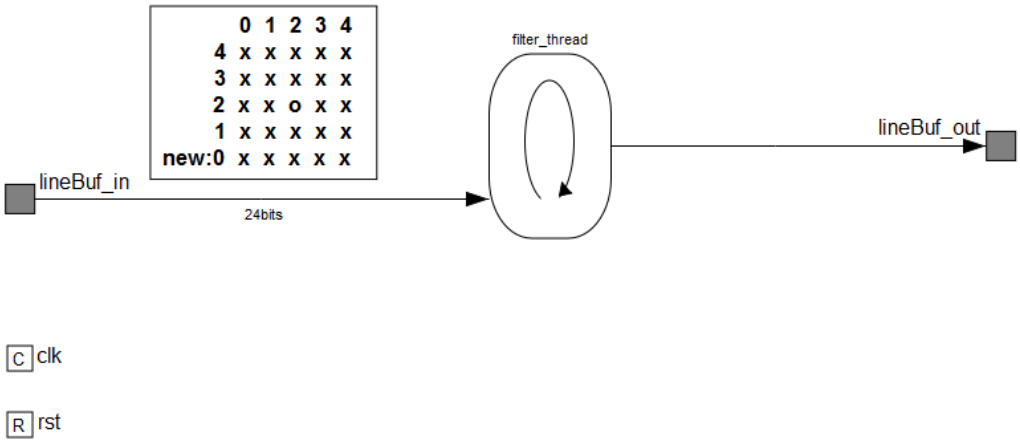


**図 3　filter モジュール内部**

5x5 のマトリクス処理を行うハードウェアを設計するためには、たとえば以下のような点に注意する必要があります。

・ラインバッファには 1 ポートの SRAM を 4 個(画像 1 ラインに対して SRAM を 1 個)使用(2 ポート RAM を使うとより簡単に設計できるが、1 ポート RAM に比べ面積が大きくなってしまう)
・画像端部の扱い
・最初に計算する 5x5 マトリクスの位置

RTL 設計者にとっても、これらを考慮して短期間に設計することは難易度が高くなります。一方、Breakfast と Cynthesizer を組み合わせれば、ラインバッファ設計に必要なパラメータを GUI 上で入力するだけで所望の回路(SystemC および RTL)を得ることができます。なお、ユーザ独自のラインバッファクラスを使用する場合、それに合わせて Breakfast の SystemC 自動生成部分を弊社にて修正するのは容易なことです。

パラメータの設定は Breakfast 上で行ってもよいですし、Cynthesizer 上のツール IFGen(Interface Generator)上で行っても構いません。Breakfast におけるラインバッファ設定の例を図 4 に示します。ラインバッファに対して、マトリクスの x 方向、y 方向サイズ、画素遅延、ライン遅延等を設定すると、その値に応じてラインバッファのシンボルが自動的に変化します。ラインバッファのシンボルがブロック図に表示されるのでブロック図はドキュメントとしてもそのまま利用することができます。これによりユーザはドキュメント用として新たに図を作成する必要がなくなります。

またこの階層でのラインバッファの情報は上階層のブロック図にも Breakfast によって自動的に反映されます。例えば図 2 に表示されているラインバッファのシンボルは、図 2 のブロック図を作成するときに作ったものでありません。下階層（サブモジュール）のブロック図から情報を読み取り、自動的に表示されているものです。

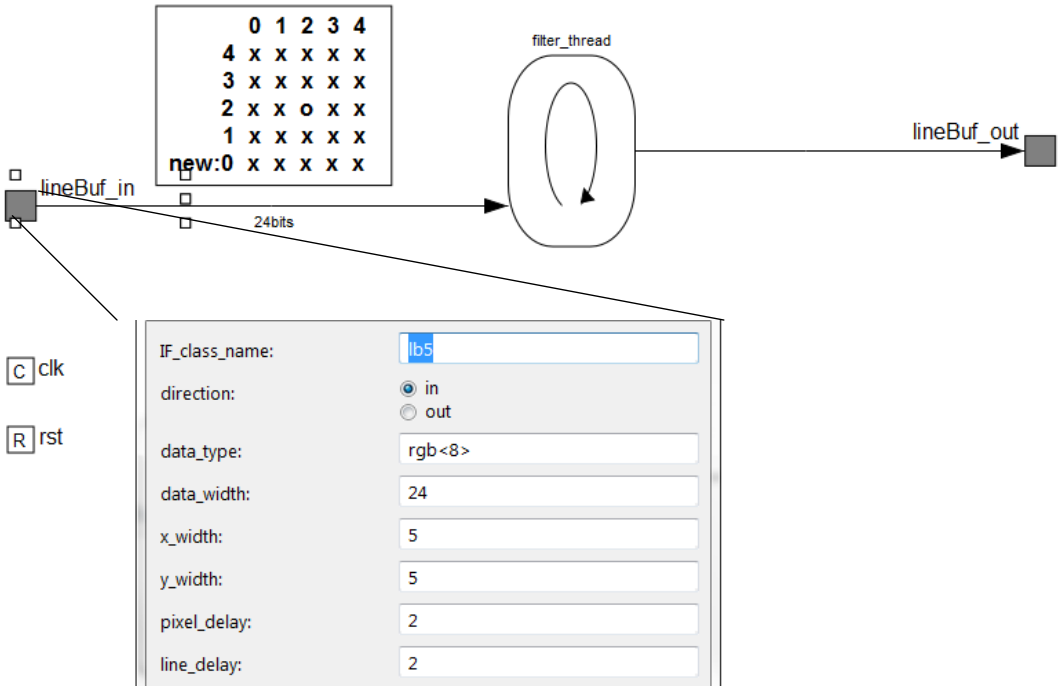


**図 4　ラインバッファに関する設定**

filter モジュールに対して、図 3 に示したブロック図の作成が終わると、Breakfast を使って SystemC のスケルトン・コード（雛形コード）を生成します。生成されるスケルトン・コードの例を図 5 に示します。実際は Forte 社専用のディレクティブも記述されますが、図ではその部分は削除してあります。ユーザはこのスケルトン・コードを参考にしながら必要なコードをこのスケルトン・コードに追加していきます。ここでは図 5 の 36 行目の関数 calc()を完成させていくことになります。これにより、ユーザは画像処理の中身を完成させていくことにのみ集中でき、その周辺の記述については Breakfast が用意してくれます。

なお、このスケルトン・コードはバーチャル・プラットフォーム用と高位合成用とでは同じものを使用します。従来の技術では高位合成用の SystemC をバーチャル・プラットフォーム上にもっていくことは非常に困難でした。しかし Breakfast と Forte 社の技術を組み合わせることにより、高位合成用の SystemC を容易にバーチャル・プラットフォームに組み込むことが可能となります。

```
 1 #include "filter.h"
 2
 3 void filter::thread()
 4 {
 5
 6     {
 7
 8
 9         #include "filter_reset.inc"
10
11         vpf_wait();
12     }
13
14     ParamT p;
15     sc_uint<1>  m_reg_rdy_in;
16     rgb<8> WorkingSet[5][5];
17     rgb<8> m_lineBuf_out ;
18
19     reg_rdy_in.wait_trig( m_reg_rdy_in );
20
21     while(1){
22
23
24
25             p = param_i.read();
26             lineBuf_in.set_size( p.y_size, p.x_size);
27             lineBuf_out.set_size( p.y_size, p.x_size);
28             lineBuf_in.start_tx();
29             lineBuf_out.start_tx();
30             while( !lineBuf_in.y_done() ) {
31
32                 while( !lineBuf_in.x_done() ) {
33
34                     lineBuf_in.get(workingSet);
35
36                     calc( WorkingSet, m_lineBuf_out);
37
38                     lineBuf_out.put( m_lineBuf_out);
39                 }
40                 lineBuf_in.next_y();
41                 lineBuf_out.next_y();
42             }
43             lineBuf_in.end_tx();
44             lineBuf_out.end_tx();
45     }
46 }
47
```

**図 5　SystemC スケルトン・コードの例**

以上のように、accelerator 内の各サブモジュールを設計していきます。

### 3.2.2. メモリ・マップの作成

各ブロック図の作成が終わると、ハードウェアに関するメモリ・マップを設定します。メモリ・マップはまずバスのマスタポートに対して設定します。メモリ・マップの設定例を図 6 に示します。

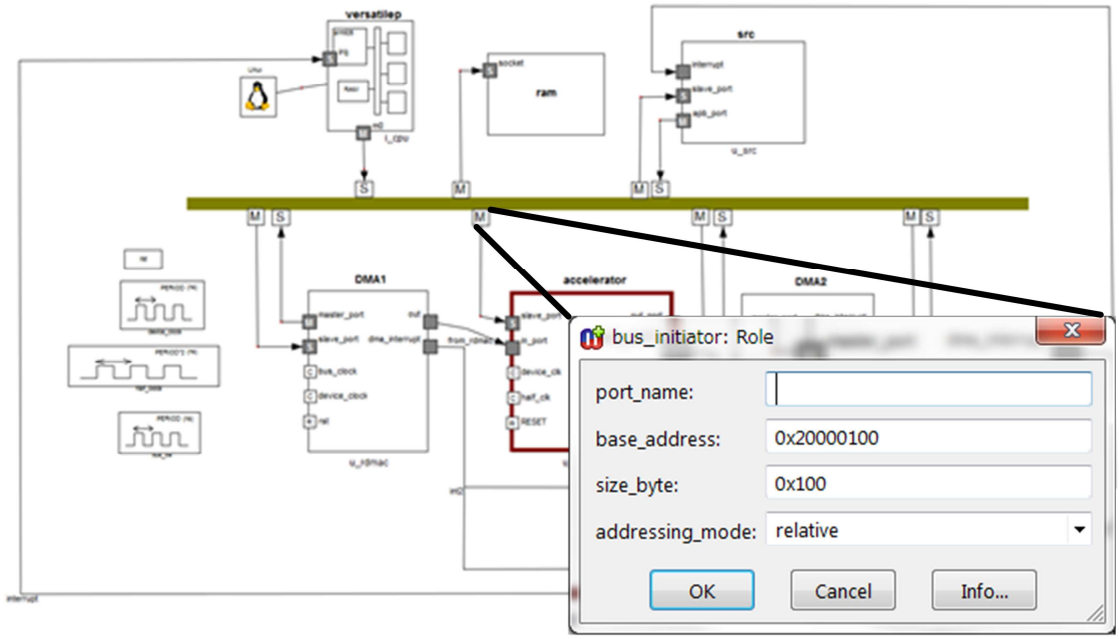


**図 6　メモリ・マップの設定**

バスに対するメモリ・マップの設定が終わると、さらに詳細なメモリ・マップ（レジスタ・マップ）を accelerator モジュール内に対して作成します。

accelerator モジュールには CPU からアクセス可能な複数のレジスタが存在します。各レジスタ（32 ビット）のオフセットアドレスおよびビット・フィールドに関する情報を Breakfast 上で表形式によりメモリ・マップとして作成していきます。この表はアドレスデコーダ回路(図 2 でいうと accelerator_decoder モジュール)の下階層として作成します。メモリ・マップの例を図 7 に示します。

Breakfast は SystemC により記述されたアドレスデコーダをこの表から自動的に生成しま

す。また、生成された SystemC コードはバーチャル・プラットフォームと高位合成の両方において同じものを使用します。これにより、バーチャル・プラットフォームの SystemC 動作と実際のハードウェア（高位合成用の SystemC）の動作の間に、より一貫性を持たせることが可能となります。

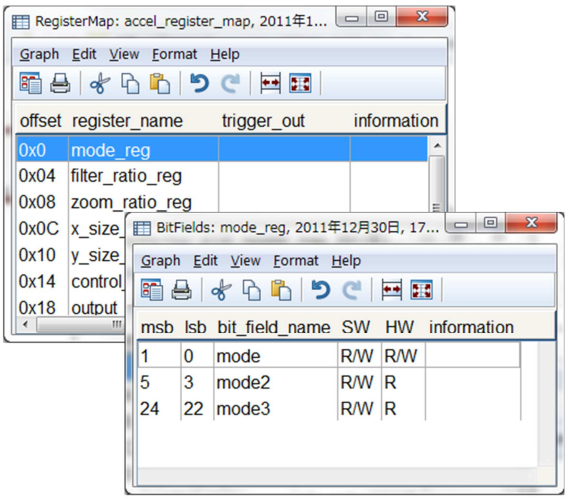


**図 7　詳細メモリ・マップ（レジスタ・マップ）**

accelerator 内の各サブモジュール(filter、zoom_out、error_diffusion など)からアドレスデコーダ回路内のビット・フィールドにアクセスする場合は、このビット・フィールド名を使ってアクセスします。

### 3.2.3.　SystemC コードの生成

各ブロック図とメモリ・マップの作成が終わると、そこから SystemC のスケルトン・コードを自動生成します。SystemC の TLM2.0 や高位合成を意識したコードを書くにはかなりの知識が必要となります。しかし、その多くの部分は決まりきった（固定した）形式をもつので、毎回手で記述することは避けたいところです。

Breakfast ではこの決まりきったコードに関しては自動生成し、変動的な部分をユーザに追記してもらうという方式を採用しています。なお先述した通り、バーチャル・プラットフォームのトップ階層については人手による追記は不要です。また、アドレスデコーダ回路の SystemC コードも Breakfast が自動生成するので、人手によりコードを追記する必要はありません。自動生成された SystemC スケルトン・コードに対して人手による追記が必要な部分は、高位合成対象となる画像処理アリゴリズムのコア部分だけです。

### 3.2.4. SystemC TLM シミュレーション

SystemC コードの作成が終わると TLM シミュレーションを行い、作成した SystemC コードが正しく動くかについて検証します。SystemC のデバッグには gdb などのフリーのツールや市販の SystemC 専用デバッガを使用します。

SystemC TLM シミュレーションの抽象度は LT(Loosely Timed)です。なお、accelerator モジュールの検証には CPU モデルをシミュレーション中に動かすことはありません。CPU モデルの代わりに図 1 中の src モジュールを動かします。一方、バーチャル・プラットフォームを使ってソフトウェア開発者がソフトウェアを開発するときに CPU モデルを動かします。もちろん CPU モデルを動かしながらハードウェア設計（デバッグ）も可能ですが、OS を起動する時間が無駄になります。ハードウェア設計およびデバッグを行うときは、たいていの場合、OS は不要です。CPU モデルの使用の有無については Breakfast 上で簡単に切り替えることができます。

TLM シミュレーションを実施した結果の表示例を図 8 に示します。図 8 の左側の絵が入力画像であり、図 1 中の src モジュールから読み込んだ画像ファイルの内容です。一方、右側の絵は、accelerator モジュールでの画像処理後に書き込まれた図 1 中の ram の内容を図 1 中の sink モジュールが読み取り、表示した結果です。

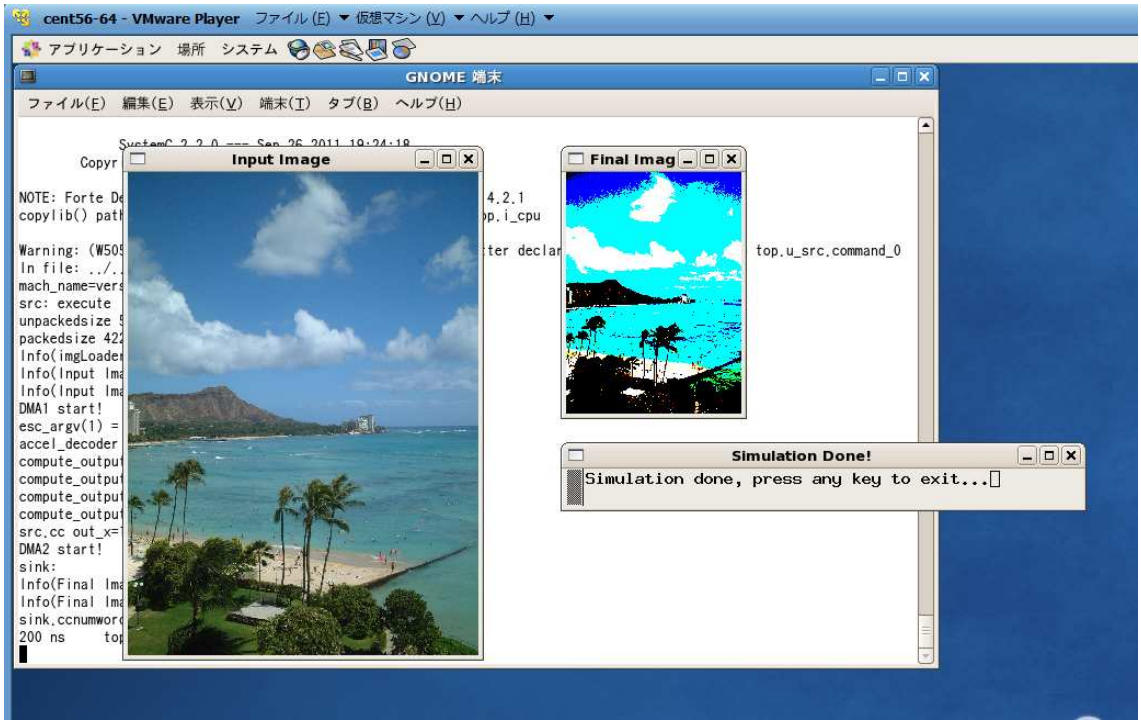


**図 8　TLM シミュレーションの結果**

TLM シミュレーション実行時間は約 3 秒です。
実行環境は以下の通りです。

・入力画像サイズ：320 ピクセル×440 ピクセル
・出力画像サイズ：160 ピクセル×220 ピクセル
・実行マシン　　：Intel Core i7-2600 3.4GHz
・OS　　：CentOS 5.7　64 ビット版（Windows7 上で VMWare Player を使用）

### 3.2.5.　バーチャル・プラットフォームのリリース

accelarator モジュールに対して TLM シミュレーションが終わると、バーチャル・プラットフォームのリリース準備作業に入ります。バーチャル・プラットフォームのリリースのためには CPU モデルを含めた TLM シミュレーションを行います。このためには、accelerator モジュールを制御するためのソフトウェアを準備する必要があります。ここでいう制御とは、accelerator や DMA 内のレジスタにソフトウェア側から値を書き込むことを意味します。今回のデザインでは OS として Linux が想定されており、ソフトウェア側からの制御方法としてたとえば以下のようにいくつかの方法が考えられます（他にもあります）。

・Linux 内のデバイスドライバとして実装する
・mmap()関数を使ってアプリケーションとして実装する。

今回の例では説明を簡単にするため、mmap()を使ってハードウェア（accelerator と DMA）を制御することにします。

図 9 に Linux の mmap()を使った C のソースコードを示します。今回はこのプログラムを”dac”と呼ぶことにします（”dac”に深い意味はありません）。ソースコード中の、

```
write_data(ACCEL_ADDR,3,320);
```

では、物理アドレス ACCEL_ADDR の 3 バイト目に 320 を設定しています。

この C のソースコードに対して ARM 用クロスコンパイルを行い、生成された ELF ファイルをバーチャル・プラットフォーム内の Linux ファイルシステム上に ftp 等を使って配置します。その後、このプログラムをバーチャル・プラットフォーム内の Linux から起動すると、accelerator の SystemC モデルが起動し、画像処理が実施されます。

```
#define ADDR_HW_BASE (unsigned long)0x20000000
...
#define ADDR_DMA1  (unsigned long)0x20000000
#define ADDR_ACCEL (unsigned long)0x21000000
...
#define write_data( addr, offset, data ) *(iomap + ((addr-ADDR_HW_BASE)>>2) + offset )=data;
main()
{
    int  fd;
    volatile unsigned long *iomap;
    int num,from;

    fd = open("/dev/mem",O_RDWR);
    if(fd<0)
    {
        fprintf(stderr,"cannot open /dev/mem¥n");
        return 1;
    }
    num  = 0x60000000UL;
    from = 0x20000000;
    iomap = mmap( 0, num, PROT_READ|PROT_WRITE, MAP_SHARED, fd, from);

    if( iomap < 0){
        printf("error mmap() !¥n");
        exit(1);
    }
    /* Accel */
    write_data( ADDR_ACCEL, 3, 320); //x size
    write_data( ADDR_ACCEL, 4, 440); //y_size
    write_data( ADDR_ACCEL, 1, 50);  //filter_ratio
    write_data( ADDR_ACCEL, 2, 50);  //zoom_ratio
    write_data( ADDR_ACCEL, 0, 0);   //mode
    write_data( ADDR_ACCEL, 5, 1);   //kick

    /* src */
    write_data( ADDR_SRC, 3, 0);        //byte_countの設定(dummy for kick)

    /* sink */
    write_data( ADDR_SINK, 3, 0);        //byte_countの設定(dummy for kick)

    /* RDMA */
    write_data( ADDR_DMA1, 2, ADDR_MEM);   //start_addrの設定
    write_data( ADDR_DMA1, 3, 422400);   //byte_countの設定

    /* WDMA */
    write_data( ADDR_DMA2, 2, ADDR_MEM);   //start_addrの設定
    write_data( ADDR_DMA2, 3, 13200);        //byte_countの設定

    munmap(iomap,num);
    close(fd);
}
```

**図 9　制御ソフトウェア**

つぎにバーチャル・プラットフォームを実際に起動していく様子について説明します。CPU モデルを使用して、バーチャル・プラットを起動している様子を図 10 に示します。今回は CPU モデル(ARM)を使い Linux を起動しています。このための図 10 内の右側の部分に Linux の起動画面が表示されています。なお、CPU 部分のエミュレーションは QEMU が担当しています。

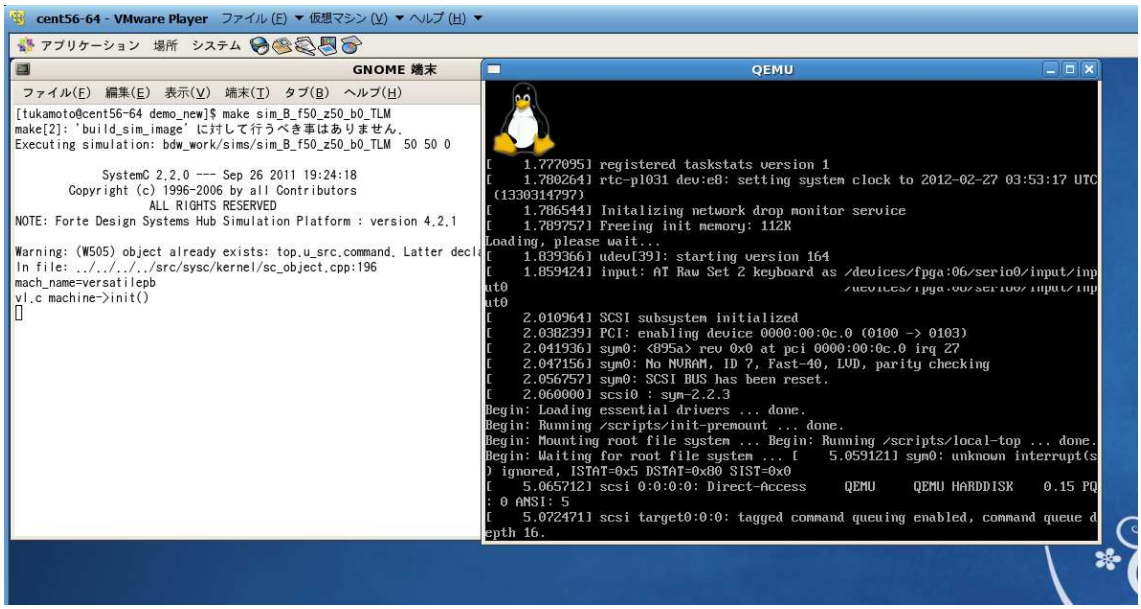


**図 10　CPU モデルを含めたバーチャル・プラットフォームの起動**

次に制御プログラム dac を ARM 用にコンパイルします。これは PC 上でクロスコンパイルを行うことにより実施します。図 11 の左下のウインドウにてクロスコンパイルを実施しています。

クロスコンパイルの結果作成された ELF ファイル”dac”をバーチャル・プラットフォーム上の Linux に転送します。図 11 の右側のウインドウはバーチャル・プラットフォーム上の Linux であり、その中で ftp コマンドを使い、プログラム dac をバーチャル・プラットフォームに転送しています。

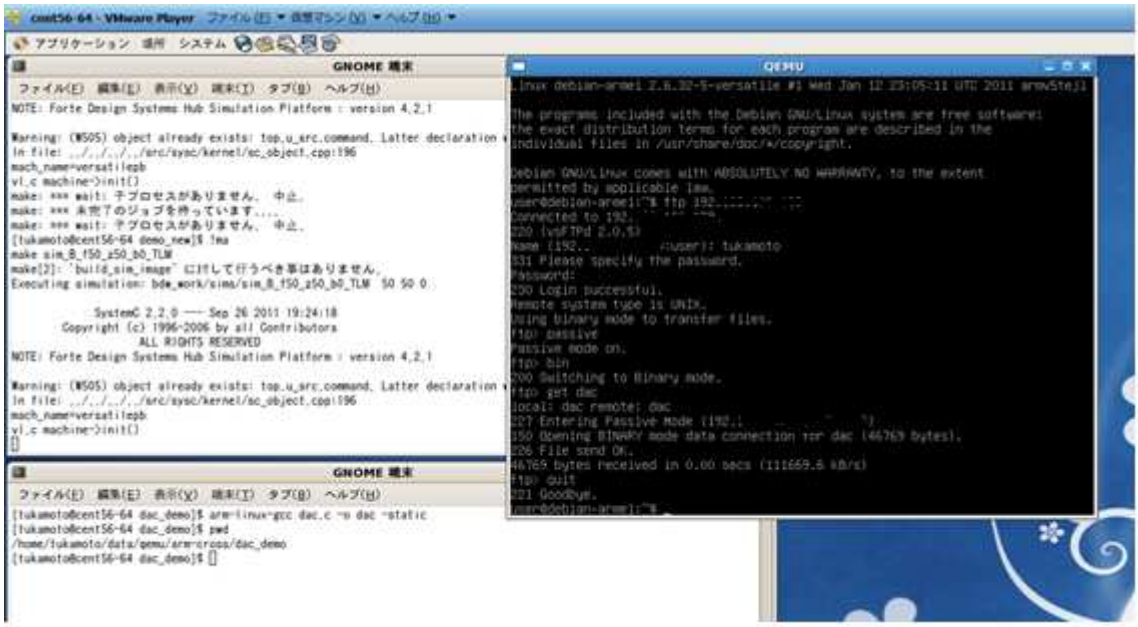

**図 11　PC 上でのクロスコンパイルおよびバーチャル・プラットフォームへの転送**

dac の転送が終わると、バーチャル・プラットフォーム上にて dac を実行します。
図 12 に実行の様子を示します。図 12 の右側のウインドウ（バーチャル・プラットフォーム上の Linux）から dac を実行しています。ただし、プログラム dac は mmap()関数を使っており、この mmap()はルートの権限がないと実行できないため、今回は root アカウントから実行しています。dac の実行時間は約 3 秒です。


**図 12　バーチャル・プラットフォーム上での制御ソフトの実行**

以上のようにバーチャル・プラットフォームが正しく動作していることが確認できたら、バーチャル・プラットフォームをソフトウェア開発者にリリースします。

ソフトウェア開発者はリリースされたバーチャル･プラットフォームを使って OS も含めたソフトウェアの先行開発を行います。この時、ソフトウェアのデバッグには gdb などのデバッガを使用することが可能です。さらには Linux カーネルのデバッグも可能となっています。

### 3.2.6. 高位合成

TLM シミュレーションによる検証が終わり、バーチャル・プラットフォームをソフトウェア開発にリリースした後、ハードウェア設計者は高位合成ツールを使って RTL を自動生成します。

高位合成用の SystemC は TLM シミュレーション時に使用した SystemC コードをそのまま使用します。TLM の抽象度からピンレベルへの抽象度への変換は Cynthesizer が自動で行います。

生成した RTL を検証する場合も、TLM シミュレーションで使用したバーチャル・プラットフォームをそのまま使うことが可能です。ただし、CPU モデルは使用せず、src および sink モジュールを使います。CPU モデルを使用するとシミュレーション速度が非常に遅くなり現実的ではありません。

たとえ CPU モデルを使用しなくても RTL シミュレーションの速度は通常、TLM シミュレーションに比べ格段に遅くなります。RTL シミュレーションの代わりに、サイクル・アキュレート・シミュレーションを実施することも可能です。サイクル・アキュレートな SystemC シミュレーションではクロック以外のイベントは無視し、クロックサイクルごとに RTL 内部の変数の値を評価します。これにより通常の RTL シミュレーションよりも 5～10 倍高速になります。サイクル・アキュレートな SystemC は Cynthesizer により自動的に生成されます。

## 4. まとめ

Breakfast では CPU、バス、ペリフェラル回路をグラフィカル入力ツールにより接続し、SystemC ソースコードを自動生成します。これにより SystemC TLM2.0 に関する深い知識

や QEMU と SystemC を接続する方法についての知識を必要とせずバーチャル・プラットフォームを構築することができます。

画像処理回路に関してラインバッファやモジュール間 I/F は、Breakfast によりいくつかのパラメータを設定しシンボルとして表現するだけでよく、あとは Cynthesizer などの高位合成ツールを使って RTL を自動生成することが可能です。これにより設計効率が大きく向上します。

また、バーチャル・プラットフォーム用 SystemC コードと高位合成用 SystemC コードは共通化でき、バーチャル・プラットフォームの動作と実際のハードウェアの動作との間に、より一貫性を持たせることが可能となります。

## 5. （参考）MetaEdit+というツール

弊社で開発中のグラフィカル入力ツール Breakfast ですが、この種のツールをゼロから開発するとなると膨大な時間と費用がかかります。そこで弊社では開発期間を短縮するために、MetaEdit+というツールを利用しています。MetaEdit+はフィンランドの MetaCase 社が開発・販売しており、日本国内の代理店は富士設備工業（株）が行っています。

簡単に説明すると、MetaEdit+は高機能なお絵かきソフトといったところでしょうか。長方形などのシンボル図形の配置、シンボル間の接続、図形や線の移動、削除などの機能はあらかじめ MetaEdit+に備わっています。ユーザは独自のシンボルを定義し、そのシンボルに対して独自の複数のプロパティを独自に定義することができます。また、図の結線情報にアクセスするための API も用意されており、その API を使えば、作図した図から C のソースコードやドキュメントなどを自動生成するソフトを開発することも可能となります。

MetaEdit+のユーザは大きく以下の二つに分類することができます。
・メタモデラー（meta modeler）：ツール開発者
・モデラー(modeler)：開発されたツールを使う設計者
これらについて、フローチャート作図ツールおよび C ソースコード自動生成ツールの開発を例に説明します。

フローチャート作図ツールを開発する場合、フローチャート作図にて使用する、長方形、ひし形、矢印などのシンボルおよびそれらに対す各種プロパティをあらかじめ定義しておく必要があります。これはメタモデラーの仕事です。またメタモデラーは、作図されたフローチャートのデータに対して、MetaEdit+で用意されている API を使って長方形やひし

形の接続関係を調べ、C のソースコードを自動生成するツールも開発します。一方、モデラーはメタモデラーが用意したシンボルを配置し、各種プロパティに対して値を入力し、シンボル間を接続しながらフローチャートを作図します。また、メタモデラーが事前に開発したソースコード自動生成ツールを使ってモデラーはフローチャートから C ソースコードを自動生成します。

ここではフローチャートの作図ツールを例に挙げましたが、MetaEdit+を適用できるシーンは数えきれないほどあります。要は、何か絵を書いて（抽象化して）そこからソースコードやドキュメントを自動生成したい、という要望があればそれはまさに MetaEdit+の活躍の場であるということです。

ユーザ独自のグラフィカル入力を開発したい、とお考えの方はぜひ一度 MetaEdit+の世界をのぞいてみてはいかがでしょうか。