第2章

導入記述例

本章では，簡単なデータ変換関数を用いて，SystemC検証とビヘイビア合成による設計フロー，および記述スタイルを紹介します．細かい構文などの説明は次章以降で行います．ここでは，まず，SystemCによる設計手法の全体像をつかむようにしてください．

設計フローでは，まず，ハードウェア化する関数に対して，入出力に`sc_fifo`を用いたTLM（Transaction Level Modeling）モデルを作成し，テストベンチを用いて検証します．期待どおりの動作を確認したら，データの入出力をタイミング精度で記述したBCA（Bus Cycle Accurate）モデルに変更して動作を確認し，設計制約を与えてビヘイビア合成を実行します．RTLデザインが生成されたら，再度，RTLシミュレーションを実行します．性能目標を満足するRTLデザインが得られたら，従来の設計フローに進みます．

本章では，SystemC/C++を用いたオブジェクト指向設計の紹介として，クラス（`class`）文を用いた構造体を作成します．構造体クラス内にメンバ変数および変数を扱うメンバ関数を宣言することで，記述を効率化できます．ANSI-C言語に対するC++を使用するメリットの一つは，オブジェクト指向設計を行えることにあります．SystemCはC++に基づいているため，ハードウェア設計に対してオブジェクト指向を適用することができます．

2-1 アルゴリズム記述例

ソフトウェアによるアルゴリズム検証，もしくは，ハードウェアとソフトウェアの協調検証によって，ハードウェアとして実現するC/C++アルゴリズムを選択します．アルゴリズムには，画像処理や暗号化，ドライバ，通信などの多くのアルゴリズムがありますが，本章の導入例として，*RGB*データを*YUV*に変換する単純な関数を用いることにします．**リスト2-1**に変換関数を示します．*R*，*G*，*B*それぞれのデータを，輝度信号*Y*，色差信号*U*（またはC_r），*V*（またはC_b）に変換します．実際の設計では，より複雑な関数がハードウェア化の対象となりますが，基本的な記述スタイルや設計フローを説明するために，この簡単な関数を用いることにします．

浮動小数点演算はビヘイビア合成では扱えないか，扱えても必要なハードウェアの面積が大きくなるので，この変換関数では，データ型として整数を用います[注2-1]．

リスト 2-1　ハードウェア化する変換関数

```
void calc_rgb2yuv(int r, int g, int b,
  int &y, int &u int &v)
{
  // y (255,0)
  // u, v offset by 128 (127,-128)
  //
  // y = r*0.299 + g*0.587 + b*0.114;
  // u = r*-0.169 + g*-0.332 + b*0.5 + 128;
  // v = r*0.5 + g*-0.418 + b*-0.082 + 128;
  y = ( r*77  + g*151 + b*29  ) / 256;
  u = ( r*-43 - g*85  + b*128 ) / 256 + 128;
  v = ( r*128 - g*107 - b*21  ) / 256 + 128;
}
```

2-2　データ型と構造体の定義

C++ではデータ型としてintやshortなどを用いていますが，ハードウェア化する際には，ビット幅を明確にしたほうが，最終的なハードウェアが小さくなります．変換関数は，最大値が255で*Y*，*U*，*V*は正の整数に正規化しているので，8ビットの符号なしの整数を表すデータ型sc_uint<8>を用いることにします．

RGBデータについては，*R*，*G*，*B*をそれぞれ個別のデータとして扱うより，一つのデータとして扱ったほうが便利なので，class構文を用いて構造体を定義します．C++の構造体では，データ・メンバとそれに関連した関数を同じ構造体の中にカプセル化することができます．このようなデータ構造をオブジェクトと呼びます．SystemCはC++上に構築されているので，ハードウェア設計に対してオブジェクト指向の設計スタイルを取り入れることができます．

ここでは*RGB*および*YUV*に対して，それぞれのデータをクラスに宣言し，データに関連する関数も同時に宣言します[注2-2]．**リスト 2-2**に示した記述例では，データ・メンバの読み書きを行う関数と，データ・メンバを個別に加算して結果を最大値にクリップする関数を宣言しています．ここで宣言した加算演算子をオーバロード演算子と呼び，C++コンパイラは演算子の引き数のデータ型を調べ，データ型が一致した演算子があれば，その演算子を自動的に呼び出します．ここでは加算演算だけを定義していますが，同じように乗算，減算，比較などの演算子のオーバロード記述も可能です（詳細は，「4.9 演算子のオーバロード」を参照）．SystemCでは，ポートに構造体を使用する場合，等号==，sc_trace，coutのオーバロード演算の定義が義務付けられているので，それらも宣言しています．

なお，ヘッダ・ファイルの複数回の読み込みは避けなければなりません．そのため，#ifndefを用いて環境変数がすでに定義されているかどうかを調べ，環境変数が定義されていれば，再度，読み込まないようにします．

注2-1：固定小数点に対してはsc_fixedやsc_ufixedを用いることができ，通常，ビヘイビア合成可能である．

注2-2：*RGB*と*YUV*に共通なクラスを定義して，typedefによって異なる型名にすることもできるし，共通クラスを継承して，異なるクラスとして作成することもできる．

リスト 2-2　データ型と構造体クラスを定義したファイル（class.h）例

```
#ifndef _CLASS_H_
#define _CLASS_H_

#include <systemc.h>

class rgb8_t
{
 public:
  sc_uint<8> r;
  sc_uint<8> g;
  sc_uint<8> b;

  // コンストラクタ宣言
  rgb8_t() {}

  rgb8_t(const int cr, const int cg, const int cb) :
    r(cr), g(cg), b(cb) {}

  // メンバ関数の宣言
  rgb8_t read() {
     rgb8_t obj;
     obj.r = r; obj.g = g; obj.b = b;
     return obj;
  }

  void write(const rgb8_t& obj) {
    r = obj.r; g = obj.g; b = obj.b;
  }

  void write(const int& a1, const int& a2, const int& a3) {
    r = a1; g = a2; b = a3;
  }

  void operator=(const rgb8_t& obj) {
    write(obj);
  }

  // オーバロード演算子の宣言
  bool operator==( const rgb8_t& obj )
  {
    if ( r == obj.r && g == obj.g && b == obj.b)
      return ( true );
    else
      return ( false );
  }

  friend rgb8_t operator+ (rgb8_t& a, rgb8_t& b) {
    rgb8_t obj;
    obj.r = ((a.r + b.r) > 255) ? (sc_uint<8>)255 : (sc_uint<8>)(a.r + b.r);
    obj.g = ((a.g + b.g) > 255) ? (sc_uint<8>)255 : (sc_uint<8>)(a.g + b.g);
    obj.b = ((a.b + b.b) > 255) ? (sc_uint<8>)255 : (sc_uint<8>)(a.b + b.b);
    return obj;
  }

};

// sc_traceによる波形表示の定義
inline void sc_trace(sc_trace_file *tf, const rgb8_t& o, const sc_string& n) {
  sc_trace(tf, o.r, n + "_r");
  sc_trace(tf, o.g, n + "_g");
  sc_trace(tf, o.b, n + "_b");
}

// ostreamによる出力表示の定義
inline ostream& operator << ( ostream& os, const rgb8_t& o ) {
```

リスト 2-2　データ型と構造体クラスを定義したファイル (class.h) 例 (つづき)

```
  os << o.r << " " << o.g << " " << o.b;
  return os;
}


class yuv8_t
{
public:
  sc_uint<8> y;
  sc_uint<8> u;
  sc_uint<8> v;

  // コンストラクタ宣言
  yuv8_t() {};

  yuv8_t(const int cy, const int cu, const int cv) :
    y(cy), u(cu), v(cv) {}

  // メンバ関数の宣言
  yuv8_t read() {
    yuv8_t obj;
    obj.y = y; obj.u = u; obj.v = v;
    return obj;
  }

  void write(const yuv8_t& obj) {
    y = obj.y; u = obj.u; v = obj.v;
  }

  void operator=(const yuv8_t& obj) {
    write(obj);
  }

  // オーバロード演算子の宣言
  bool operator==( const yuv8_t& obj ) {
    if ( y == obj.y && u == obj.u && v == obj.v)
      return ( true );
    else
      return ( false );
  }

  friend yuv8_t operator + (yuv8_t& a, yuv8_t& b) {
   yuv8_t obj;
    obj.y = ((a.y + b.y) > 255) ? (sc_uint<8>)255 : (sc_uint<8>)(a.y + b.y);
    obj.u = ((a.u + b.u) > 255) ? (sc_uint<8>)255 : (sc_uint<8>)(a.u + b.u);
    obj.v = ((a.v + b.v) > 255) ? (sc_uint<8>)255 : (sc_uint<8>)(a.v + b.v);
    return obj;
  }

};

// sc_traceによる波形表示の定義
inline void sc_trace(sc_trace_file *tf, const yuv8_t& o, const sc_string& n) {
  sc_trace(tf, o.y, n + "_y");
  sc_trace(tf, o.u, n + "_u");
  sc_trace(tf, o.v, n + "_v");
}

// ostreamによる出力表示の定義
inline ostream& operator << ( ostream& os, const yuv8_t& o ) {
  os << o.y << " " << o.u << " " << o.v;
  return os;
}

#endif
```