SystemCを使ったシステム設計

デバイスの記事　システムの記事

第5回 インターフェース接続

長谷川裕恭

C++をベースとするシステム・レベル言語「SystemC」に関する連載の第5回である．今回は，副スレーブを含む複雑なマスタ・スレーブ通信の記述例を紹介する．また，インターフェースとチャネルの概念を利用して，複雑なマスタ・スレーブ記述を一つの関数として処理する方法についても解説する．

（編集部）

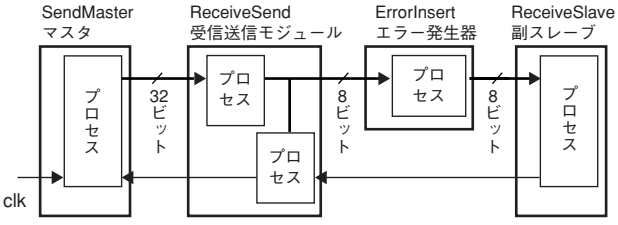
今回は，前回説明したマスタ・スレーブ記述について，もう少し複雑な例を紹介していきます．

●副スレーブを含むマスタ・スレーブ通信

前回説明したのは，マスタが32ビット幅の信号を送信し，スレーブが受信後にACK（アクノリッジ）を返すという単純なものでした．今回はスレーブ（ReceiveSend；受信送信モジュール）が32ビット幅の信号を受けるだけでなく，さらに副スレーブに8ビットごとに送信していきます（**図1**）．

スレーブは，単にデータを受け取り，処理結果をマスタに返すだけでなく，副スレーブから見ればマスタの役割を果たします．副スレーブでは8ビットの受信ごとにACK（'1'）を返します．データ転送にはパリティ・ビットを付加し，エラーがあれば'0'を返します．もし，副スレーブからエラーが返ってこなければ，スレーブは同じデータを再送します．

スレーブと副スレーブの間では，わざとエラーを発生させるモジュールを通過させ，このエラー再送を動作させます．



〔図1〕複雑なマスタ・スレーブ通信

スレーブ（ReceiveSend；受信送信モジュール）は，32ビット幅の信号を受け，さらに副スレーブに8ビットごとに送信していく．スレーブは，単にデータを受け取って処理結果をマスタへ返すだけでなく，副スレーブから見ればマスタの役割を果たす．副スレーブでは8ビットの受信ごとにACK（'1'）を返す．データ転送にはパリティ・ビットを付加し，エラーがあれば'0'を返す．もし副スレーブからエラーが返って来れば，スレーブは同じデータを再送する．スレーブと副スレーブの間に，わざとエラーを発生させるモジュールを入れて，このエラー再送を動作させる．

●マスタはクロック同期で記述

前回，解説したように，アンタイムド（untimed）の記述と言えども，マスタとなるモジュールはクロック同期で記述したほうがよいでしょう．すべてがアンタイムドだと，非同期ループの問題を解決できないことがあります．

リスト1がマスタの記述です．前回の単純なマスタ・スレーブ通信の記述からクロック同期に変更されています（**リスト1**の①）．

マスタからは，クロックごとに32ビットのデータを送信していきます．もし，スレーブから受信エラーが返る（ReceiveAckに'0'が返る）と，同じデータを再送します（**リスト1**の③）．

ただし，今回，マスタとスレーブの間にはエラー発生回路を挿入していないので，実際に再送されることはありません．送信されるデータは，**リスト1**の④でパリティを計算し，出力ポートSendData32bitに出力しています（**リスト1**の⑥）．マスタは，8個のデータを送付すると，**リスト1**の⑦のsc_stopによってシミュレーションを停止します．

●アンタイムドでは起動信号ごとにプロセス文を記述

図1のReceiveSendは，マスタからの送信を受けると同

〔リスト1〕複雑なマスタ・スレーブ通信の記述例（マスタ）

ヘッダ・ファイルSendMaster.h

```
#ifndef __SENDMASTER_H
#define __SENDMASTER_H
#include <systemc.h>
SC_MODULE( SendMaster ) {
  sc_in_clk           clk;
  sc_in <bool>        ReceiveAck;
  sc_out<sc_uint<33> > SendData32bit;
  int                 Index;
  void MasterProcess();
  SC_CTOR( SendMaster ) {
    SC_THREAD( MasterProcess );
      sensitive << clk.pos();
  }
};
#endif
```

①マスタをクロック同期に変更

②クロックの立ち上がりで動作

ソース・ファイルSendMaster.cpp

```
#include "SendMaster.h"
void SendMaster::MasterProcess() {
  sc_uint<32> SendData_tmp;
  bool        Parity;
  Index = 0;
  cout.setf(ios::showbase);
  cout.setf(ios::hex,ios::basefield);
  while(true) {
    wait();
    if(ReceiveAck == 1 ) {
       Index++;
    } else {
      cout << "Parity Error. ReSend same data.¥n";
    }
    if(Index < 8 ) {
      switch(Index) {
          case 0 : SendData_tmp = 0x12345678; break;
          case 1 : SendData_tmp = 0x23456789; break;
          case 2 : SendData_tmp = 0x3456789a; break;
          case 3 : SendData_tmp = 0x456789ab; break;
          case 4 : SendData_tmp = 0x56789abc; break;
          case 5 : SendData_tmp = 0x6789abcd; break;
          case 6 : SendData_tmp = 0x789abcde; break;
          case 7 : SendData_tmp = 0x89abcdef; break;
          default : SendData_tmp = 0x00000000;
      }
      Parity = 0;
      for( int i=0; i<32;i++) {
        Parity ^= SendData_tmp[i];
      }
      cout << "¥n********** No." << Index << " ************¥n";
      cout << "SendData  = " << SendData_tmp.to_int() << ";¥n";

      SendData32bit = (Parity,SendData_tmp);
    } else {
      sc_stop();
    }
  }
}
```

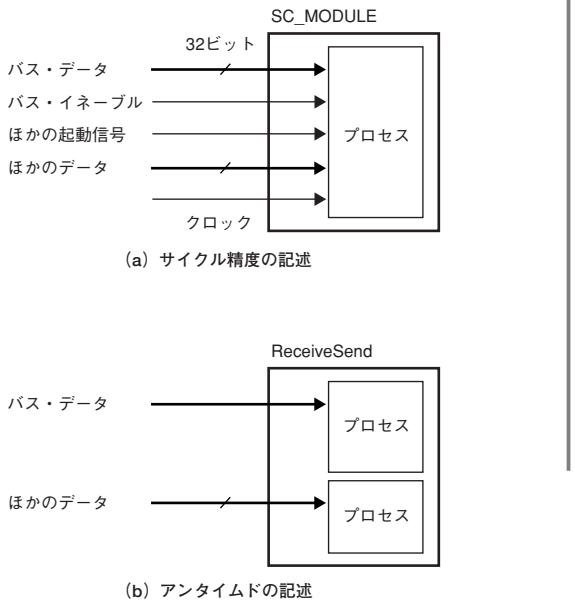
③ACKが‘1’ならカウント・アップ，‘0’なら同じデータを再送

④パリティを計算

⑤Integerに変換，sc_uintのままでは16進表示しない

⑥出力ポートへの書き込み

⑦シミュレーションの実行を停止させる



〔図2〕アンタイムドでは起動信号ごとにプロセス文を記述していく

サイクル精度の記述の場合，SC_MODULEの中は一つのプロセス文にまとめたほうがシミュレーション速度が早くなる．また，複数のプロセス文を持つ記述はビヘイビア合成ツールが苦手としている．アンタイムド記述の場合，起動信号ごとにプロセス文を分けると起動信号に対するデコードが必要なくなる．また，SC_THREADを利用する場合，複数の起動信号を記述できない

時に，スレーブからのACKを受信します．したがって，モジュールの外部から2系統の起動信号を受けることになります．アンタイムドの記述では，起動信号ごとにプロセス文を記述していきます（図2）．

サイクル精度（cycle accurate）の記述の場合，SC_MODULEの中は一つのプロセス文にまとめたほうがよいでしょう．そのほうがシミュレーション速度が早くなります．

サイクル精度の記述は，ビヘイビア合成ツールによってRTL（register transfer level）記述，あるいはゲート・レベルのネットリストに変換できます．ただし，現在のビヘイビア合成ツールは，複数のプロセス文を持つ記述の処理を苦手としています．

これに対して，アンタイムドの記述にビヘイビア合成を適用することはできません．アンタイムドのSystemC記述に対応したビヘイビア合成ツールが登場するまでには，まだ数年はかかると思われます．したがって，アンタイムドの記述では，一つのモジュールに一つのプロセス文という条件にこだわる必要はありません．また，アンタイムドの場合，二つの起動信号によって一つのプロセス文を起動す