システムＬＳＩ設計技術　104－8
(2002. 1. 23)

[招待講演]
SystemC を用いたシステムレベル設計手法

河原林 政道† 長谷川 隆‡ 番場 隆成§ 宮下 晴信¶ 峰 正高*

†NEC エレクトロニクス, 2880 Scott blvd, Santa Clara CA 95050, U.S.A.
‡富士通, 〒197-0833 東京都あきる野市渕上 50
§NEC マイクロシステム, 〒212-8514 神奈川県川崎市幸区塚越 3-484
¶富士ゼロックス, 〒339-0057 埼玉県岩槻市本町 3-1-1
* 富士通キャドテック, 〒222-0033 神奈川県横浜市港北区新横浜 2-3-9
E-mail: †kaba@el.nec.com, ‡thasegaw@jp.fujitsu.com, §t-banba@nmit.nec.co.jp,
¶Harunobu.Miyashita@fujixerox.co.jp, *mine@tkk.ts.fujitsu.co.jp

あらまし C++ベースの新しいシステムレベル記述言語である"SystemC"の機能とその設計試行/事例に関して 4 件紹介する．最初に，SystemC 2.0(第 2 版)の特徴に関して簡単に紹介する．2 番目に，SystemC 2.0 を用いてモデリングした JPEG エンコーダのシミュレーション速度等関して紹介する．3 番目に，「Perl と Verilog-HDL による簡易 CPU バスシステム記述手法」の簡易 CPU バスモデルを SystemC 2.0 で実装したモデリング例を紹介する．最後に，SystemC を用いてシステム LSI を設計した例に関して紹介する．

キーワード C++, システムレベル設計, SystemC

[Invited Talk]
System Level Design Methodology with SystemC

Masamichi KAWARABAYASHI †, Takashi HASEGAWA ‡,
Takanari BANBA §, Harunobu MIYASHITA ¶, and Masataka MINE*

†NEC Electronics, 2880 Scott blvd, M/S SC2401, Santa Clara CA 95050, U.S.A.
‡Fujitsu, 50, Fuchigami, Akiruno, Tokyo 197-0833, Japan
§NEC Micro Systems, 3-484, Tsukagoshi, Saiwai-ku, Kawasaki 212-8514, Japan
¶Fuji Xerox, 3-1-1, Honmachi, Iwatsuki, Saitama 339-0057, Japan
*Fujitsu Cadtech, 2-3-9, Shinyokohama, Kohoku-ku, Yokohama, 222-0033 JAPAN
E-mail: †kaba@el.nec.com, ‡thasegaw@jp.fujitsu.com, §t-banba@nmit.nec.co.jp,
¶Harunobu.Miyashita@fujixerox.co.jp, *mine@tkk.ts.fujitsu.co.jp

Abstract The C++ based and new system level language "SystemC" and its design examples are shown. At first the features of SystemC 2.0 are introduced briefly. Then, the simulation speed of JPEG encoder model in SystemC is shown. The third presenter shows the simple CPU bus model on SystemC 2.0, which was previously developed in "The description technique of the simple CPU bus system with Perl and VerilogHDL". At last the system LSI design example in SystemC is introduced.

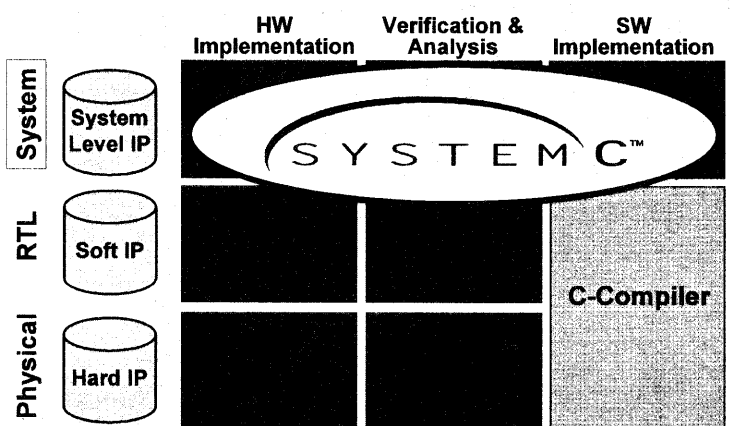
Key words C++, System level design, SystemC

SystemCの最新動向

富士通株式会社
電子デバイス事業本部 第2システムLSI事業部
ソリューション/プロセサ開発部
長谷川 隆
(Vice-Chairman of the Board of Directors, OSCI)

SYSTEMC™

SystemCとはシステムレベルの設計言語



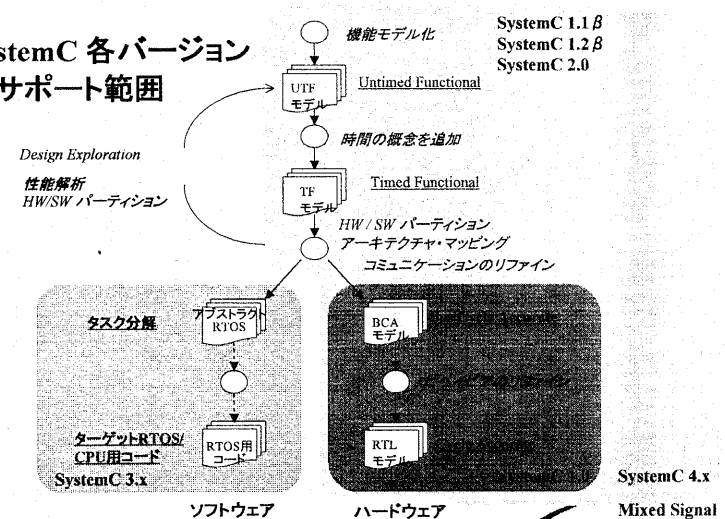
SYSTEMC™

SystemCとは

- C++クラスライブラリで構成されたモデリング言語
  - モジュール, ポート, 信号 (階層表現)
  - プロセス (並列動作モデル)
  - クロック (時間概念)
  - ハードウェア用データ型
    - bit vectors, 4-valued logic, fixed-point types, integers
  - Waiting and watching (プロセスのコントロール)
  - ポートとプロトコルのモデル化 (通信の抽象化)
- シミュレーションカーネルの提供

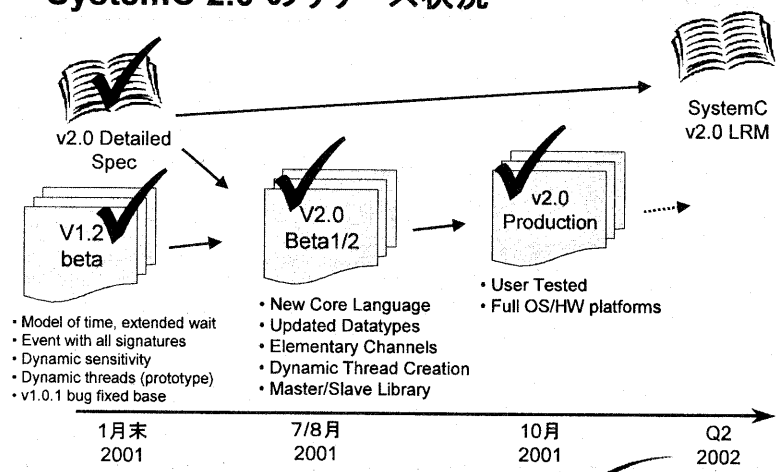
SYSTEMC™

SystemC 各バージョンのサポート範囲



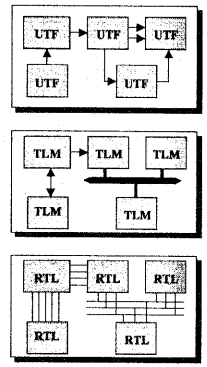
SYSTEMC™

SystemC 2.0 のリリース状況

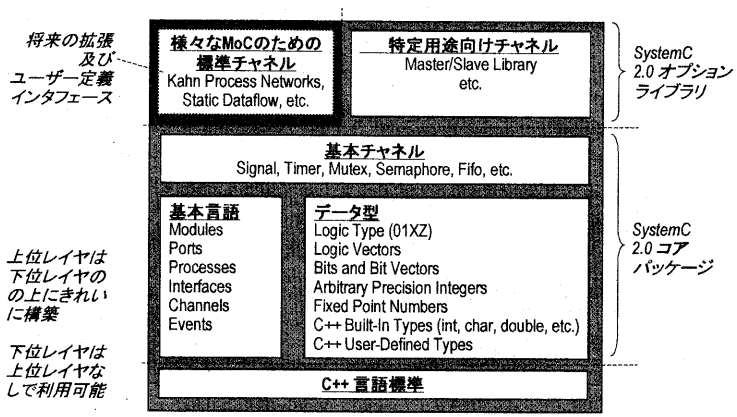


SystemC 2.0: 単一言語で複数の抽象レベルをカバー

- UTF(アンタイムドファンクショナル)レベル
  → 実行可能な仕様
- トランザクションレベル
  → プラットフォーム設計, HW/SW 協調検証
- 端子/信号レベル
  → RTL/動作レベル HW 設計と検証

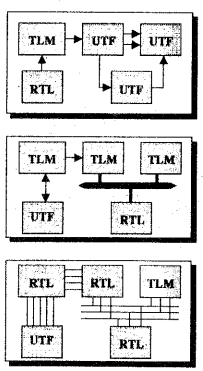


SystemC 2.x 言語アーキテクチャ



段階的詳細化と早期の検証

- 実行可能な仕様をRTLモデルへ一気に持っていくような詳細化する必要はない
- 開発初期に、高速なプラットフォームシミュレーションのための、バスサイクル精度のトランザクションレベルモデル(TLM)を使用
- 協調検証のために異なるシミュレータをつなげる必要はない



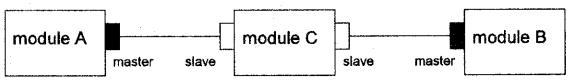
SystemC 2.0を用いた JPEGエンコーダモデリング試行

2002/01/23

NECマイクロシステム
番場 隆成
NECエレクトロニクス
河原林 政道

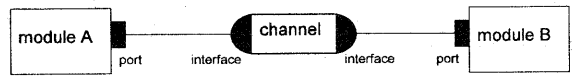
今回のトライアル手順(1)

- SystemC 1.2.1Betaを用いたモデリング
  - ブロック間通信にMaster/Slave機能を利用
    - Remote Procedure Call (RPC)
      あるモジュールから別モジュールの処理の起動
      Master portに対するread/writeアクセス
      ↓
      Slave portに関連付けられた処理が起動
    - 通常のport同様、データの通信が可能



今回のトライアル手順(2)

- SystemC 2.0を用いたモデリング
  - ブロック間通信にChannel機能を利用
    - 通信手段のモデリング機能
    - interface　プロトコル宣言
    - channel　そのchannelが持っているプロトコルの指定と定義



  - ブロック間通信にMaster/Slave機能を利用

Master/SlaveとChannel

Master/Slave

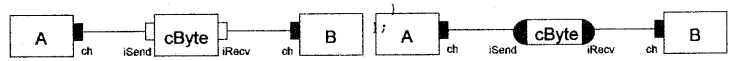
```
SC_MODULE(cByte) {
  sc_inslave<char>  iSend;
  sc_outslave<char> iRecv;
  void send(void) {
    :
  }
  void receive(void) {
    :
  }
  SC_CTOR(cByte) {
    SC_SLAVE(send, iSend);
    SC_SLAVE(receive, iRecv);
  }
};
```

2つのslaveを持つMODULE

Channel

```
class iSend :
  virtual public sc_interface {
  public:
  virtual void send(char) = 0;
};
class iRecv :
  virtual public sc_interface {
  public:
  virtual char receive(void) = 0;
};
class cByte : public sc_channel,
  public iSend, public iRecv {
  void send(char val) {
    :
  }
  char receive(void) {
    :
  }
};
```

2つのinterfaceを持つchannel



Master/SlaveとChannel

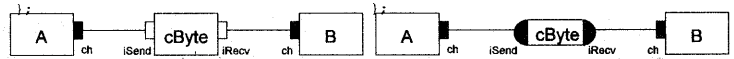
Master/Slave

```
SC_MODULE(A) {
  sc_outmaster<char> ch;
  void main(void) {
    ch = i;
  }
  SC_CTOR(A) {
    SC_THREAD(main);
  }
};
SC_MODULE(B) {
  sc_inmaster<char> ch;
  void main(void) {
    char i = ch;
  }
  SC_CTOR(A) {
    SC_THREAD(main);
  }
};
```

Channel

```
SC_MODULE(A) {
  sc_port<iSend> ch;
  void main(void) {
    ch->send(i);
  }
  SC_CTOR(A) {
    SC_THREAD(main);
  }
};
SC_MODULE(B) {
  sc_port<iRecv> ch;
  void main(void) {
    char i = ch->receive();
  }
  SC_CTOR(B) {
    SC_THREAD(main);
  }
};
```

interface属性を持ったport指定



Master/SlaveとChannel

Master/Slave

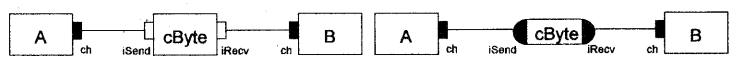
```
int sc_main(int ac, char **av) {
  cByte c("c");
  sc_link_mp<char> iSend, iRecv;
  c.iSend(iSend);
  c.iRecv(iRecv);

  A a("a");
  a.ch(iSend);
  B b("b");
  b.ch(iRecv);

  sc_star(1);

  return 0;
};
```

cByte MODULEを介した間接的接続

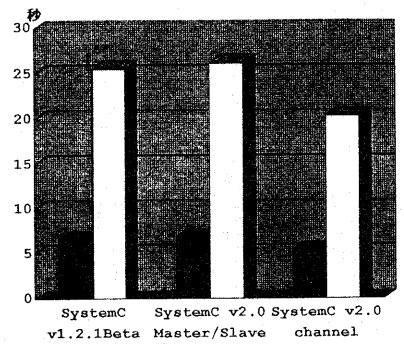
Channel

```
int sc_main(int ac, char **av) {
  cByte c("c");

  A a("a");
  a.ch(c);
  B b("b");
  b.ch(c);

  sc_star(1);

  return 0;
};
```

channelを用いて直接接続



シミュレーション性能



- Master/Slave使用のv1.2.1Betaとv2.0の速度はほぼ同等(v1.2.1Betaが3%速い)
- Channel使用のv2.0の速度は、Master/Slave使用と比べて約20%速い

■ 640x512: "-O2"
□ 1280x1024: "-O2"

UltraSPARC-II 296MHz 2cpu

SystemCへの要望

- 分かりにくいエラーメッセージ
  - コンパイルエラーメッセージ
    ある程度SystemCクラスライブラリの理解が必要？
  - デッドロック時のメッセージ
    適切なメッセージが出ない
  - → 専用のEDAツールの出現を望む
- Windows環境
  - v1.2.1Betaでは動作しない機能があった
  - Windows98/XPのサポート

SystemC 2.0を用いた
簡易CPUバスモデルの設計

富士ゼロックス株式会社
ドキュメント プロダクト カンパニー
CTD&SW開発統括部
CT-PF第二開発部
宮下 晴信

目次

簡易CPUバスモデルについて

「CQ出版社のインターフェース、1997年11月号、Page 207」の

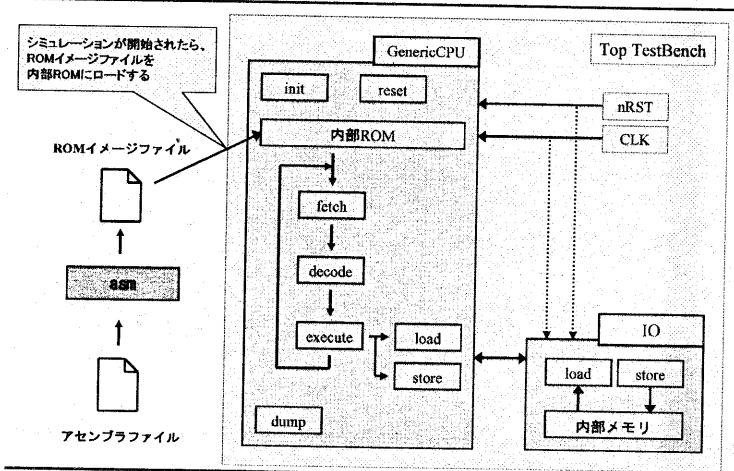
システムオンチップ時代のスケーラブル設計手法、川北浩孝、
第4回、各種設計ノウハウ

**「PerlとVerilog-HDLによる簡易CPUバスシステム記述手法、
原山みや/川北浩孝」**

のVerilog-HDLで記述されたものをベースにSystemC 2.0に書き換えました

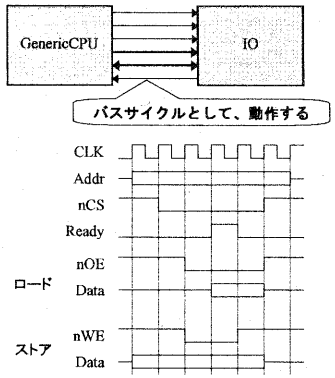
付属のアッセンブラ(asm)およびROMイメージファイル(test.hex)はそのまま使います

システム構成

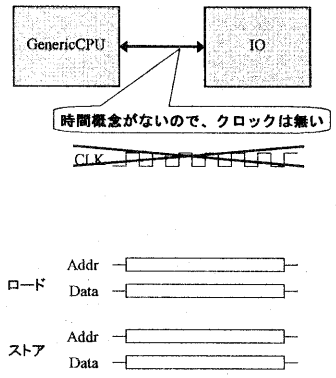


BCAモデルとUTFモデル

BCA



UTF



ソースコード (GenericCPU.hppの共通部)

```
#include <string>

#include "systemc.h"

class GenericCPU : public sc_module {

  private :

    static const int MEM_SIZE = 1024;

    string  hex_file;
    bool    dump;

    int     memory[MEM_SIZE];
    int     Reg[8];
    int     pc, eq, lt, gt;
    int     code, r0, r1, imm;

    void init( void );
    void reset( void );

    int  fetch( int p );
    void decode( int decode );
    void execute( void );
    void dumpreg( void );

    void load(  int addr, int& data );
    void store( int addr, int  data );

    int  readmem( const char *file, int *mem, int size );
    int  analyze_buffer( char *buffer, int *val);
```

```
    void clock_posedge( void );

  public :

    SC_HAS_PROCESS(GenericCPU);

    GenericCPU(sc_module_name name,
               char *_hex_file = "test.hex",
               bool _dump = true)
               : sc_module(name), hex_file(_hex_file), dump(_dump)
    {
      SC_CTHREAD(clock_posedge, CLK.pos());

      watching(nRST.delayed() == false);

      init();
    }
```

コンストラクタに引数が必要な場合は、これを付ける

SC_CTHREADを使う

非同期リセットにwatchingを使う

ソースコード (GenericCPU.hppのポート定義部)

BCA

```
private :

  static const bool LOGIC_0 = false;
  static const bool LOGIC_1 = true;
  static const char *const HiZ_32 =
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

  inline void high_cs( void ){ nCS = LOGIC_1; }
  inline void low_cs( void ) { nCS = LOGIC_0; }
  inline void high_oe( void ){ nOE = LOGIC_1; }
  inline void low_oe( void ) { nOE = LOGIC_0; }
  inline void high_we( void ){ nWE = LOGIC_1; }
  inline void low_we( void ) { nWE = LOGIC_0; }
  inline void hiz_a( void ) { A   = HiZ_32; }
  inline void set_a( const int val ){ A = val; }
  inline void hiz_d( void ) { D   = HiZ_32; }
  inline void set_d( const int val ){ D = val; }
  inline int  get_d( void ) { sc_uint<32> sig; sig = D;  return((int)sig); }

public :

  sc_out_rv<32>     A;
  sc_inout_rv<32>   D;
  sc_out<bool>      nCS;
  sc_out<bool>      nOE;
  sc_out<bool>      nWE;
  sc_in<bool>       READY;
```

パス接続に必要な信号線

UTF

```
#include "sc_mslib.h"
private :
```

sc_inoutmasterを使うので必要

```
public :
```

アドレスを持つため、インデックス付き

```
  sc_inoutmaster<int, sc_indexed<int,1024> > MP;
```


ソースコード (GenericCPU.cppのリセット部)

BCA

```
void GenericCPU::reset( void )
{
  high_cs();
  high_oe();
  high_we();
  hiz_a();
  hiz_d();

  // 内部変数の初期化

  for(int i=0 ; i<8 ; i++)
    Reg[i] = 0;

  pc = 0; eq = 0; lt = 0; gt = 0;
  code = 0; r0 = 0; r1 = 0; imm = 0;

}
```

UTF

```
void GenericCPU::reset( void )
{
  // BCAのように信号の初期化は無い

  // 内部変数の初期化

  for(int i=0 ; i<8 ; i++)
    Reg[i] = 0;

  pc = 0; eq = 0; lt = 0; gt = 0;
  code = 0; r0 = 0; r1 = 0; imm = 0;
}
```

ソースコード (GenericCPU.cppのロード部)

BCA

```
void GenericCPU::load( int addr, int &data )
{
  set_a(addr);

  wait();   low_cs();
  wait();   low_oe();

  while(true) {

    wait();
    if( READY.read() == LOGIC_1 ) {
      data = get_d();                    // データのリード
      break;
    }
  }

  wait();   high_oe();   high_cs();
  wait();   hiz_a();
}
```

バスサイクルでアクセスする

UTF

```
void GenericCPU::load( int addr, int &data )
{
  data = MP[addr];                       // データのリード
}
```

アクセスに必要な時間が、"0"

ソースコード (GenericCPU.cppのストア部)

BCA

```
void GenericCPU::store( int addr, int data )
{
  set_d(data);                           // データのライト
  set_a(addr);

  wait();   low_cs();
  wait();   low_we();

  while(true) {

    wait();
    if( READY.read() == LOGIC_1 ) {
      high_we();
      break;
    }
  }

  wait();   hiz_d();  high_cs();
  wait();   hiz_a();
}
```

バスサイクルでアクセスする

UTF

```
void GenericCPU::store( int addr, int data )
{
  MP[addr] = data;   // データのライト
}
```

アクセスに必要な時間が、"0"

ソースコード (BCA : IO.hpp)

```
#include "systemc.h"

SC_MODULE(IO) {

 private :

  static const int MEM_SIZE = 1024;
  int memory[MEM_SIZE];

  void clock_posedge( void );

 public :

  sc_in_clk          CLK;
  sc_in<bool>        nRST;
  sc_in_rv<32>       A;
  sc_inout_rv<32>    D;
  sc_in<bool>        nCS;
  sc_in<bool>        nOE;
  sc_in<bool>        nWE;
  sc_out<bool>       READY;

  SC_CTOR(IO) {
    SC_CTHREAD(clock_posedge, CLK.pos());

    watching(nRST.delayed() == false);
  }
};
```

SC_CTHREADを使う

```
void IO::clock_posedge( void )
{
  if( nRST.read() == false ) {/* Reset */
    READY = false;
    D = "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
  }
  while(true) {
    wait_until( nCS.delayed() == false);
    wait(); /* Clock */

    if( nOE.read() == false ) { /* Read */
      sc_uint<32> sig_a;
      sig_a = A.read();
      D.write(memory[sig_a]);
    }
    wait(); /* Clock */
    if( nWE.read() == false ) { /* Write */
      sc_uint<32> sig_a, sig_d;
      sig_a = A.read();
      sig_d = D.read();
      memory[sig_a] = sig_d;
    }
    READY.write(true);
    do {
      wait(); /* Clock */
      READY.write(false);
    }
    while ( nOE.read() == false );
    D.write("ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ");
  }
}
```

GenericCPUからのアクセスに対応するバスサイクル

ソースコード (UTF : IO.hpp)

```
#include "systemc.h"
#include "sc_mslib.h"

SC_MODULE(IO) {

 private:
  static const int MEM_SIZE = 1024;
  int memory[MEM_SIZE];

  void io_access( void );

public:

  //  クロックとリセットが無い

  sc_inoutslave<int, sc_indexed<int,1024> > SP;

  SC_CTOR(IO) {
    SC_SLAVE(io_access, SP);
  }
};
```

sc_inoutmasterを使うので必要

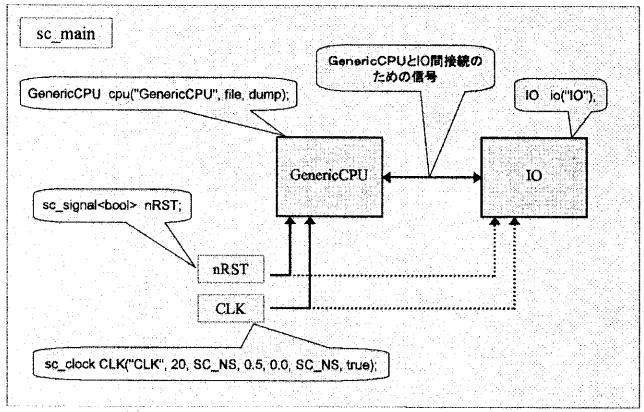
アドレスを持つため、インデックス付き

SC_SLAVEを使う

```
void IO::io_access( void )
{
    int addr = SP.get_address();

    if( SP.input() )                     // Write
      memory[addr] = SP;
    else                                 // Read
      SP = memory[addr];
}
```

アクセスに必要な時間が、"0"

トップテストベンチ(sc_main)



ソースコード (BCA : main.cpp)

```
#include "systemc.h"

#include "GenericCPU.hpp"
#include "IO.hpp"

int sc_main(int argc, char *argv[])
{
  sc_signal<bool>      nRST;
  sc_signal<bool>      READY;
  sc_signal_rv<32>     A;
  sc_signal_rv<32>     D;
  sc_signal<bool>      nCS;
  sc_signal<bool>      nOE;
  sc_signal<bool>      nWE;

  // クロックの宣言

  sc_clock CLK("CLK", 20, SC_NS, 0.5, 0.0, SC_NS, true);

  // GenericCPUの宣言とポート名の割り当て

  GenericCPU           cpu("GenericCPU", file, dump);

    cpu.CLK(CLK);
    cpu.nRST(nRST);
    cpu.A(A);
    cpu.D(D);
    cpu.nCS(nCS);
    cpu.nOE(nOE);

    cpu.nWE(nWE);  cpu.READY(READY);
```

バス接続に必要な信号線の定義、こんなにたくさん

バス接続に必要な信号線のアサイン、こんなにたくさん

```
  // IOの宣言とポート名の割り当て

  IO                   io("IO");

    io.CLK(CLK);
    io.nRST(nRST);
    io.A(A);
    io.D(D);
    io.nCS(nCS);
    io.nOE(nOE);
    io.nWE(nWE);
    io.READY(READY);

// スケジューラを起動し、クロックとリセットを生成する

  nRST = false;
  sc_start(40, SC_NS);

  nRST = true;
  sc_start(-1);

  return 0;     /* this is necessary */
}
```


ソースコード (UTF : main.cpp)

```
#include "systemc.h"

#include "GenericCPU.hpp"
#include "IO.hpp"

int sc_main(int argc, char *argv[])
{
  sc_signal<bool>      nRST;
  sc_link_mp<int>      PORT;
```

GenericCPUとIOを接続するチェネル

```
  // クロックの宣言

  sc_clock CLK("CLK", 20, SC_NS, 0.5, 0.0, SC_NS, true);

  // GenericCPUの宣言とポート名の割り当て

  GenericCPU           cpu("GenericCPU", file, dump);

    cpu.CLK(CLK);
    cpu.nRST(nRST);

    cpu.MP(PORT);
```

チェネル接続のアサインは、たった1つ

```
  // IOの宣言とポート名の割り当て

  IO                   io("IO");

    //  クロックとリセットが無い

    io.SP(PORT);
```

チェネル接続のアサインは、たった1つ

```
// スケジューラを起動し、クロックとリセットを生成する

  nRST = false;
  sc_start(40, SC_NS);

  nRST = true;
  sc_start(-1);

  return 0;     /* this is necessary */
}
```


まとめ

□SystemC 2.0の良いところ

- ◆ フリーなツールである(PC/Linux、Sun/Solaris、PC/Windowsで動作する)
- ◆ C++を知っていれば、ある程度使いこなせる
- ◆ 継承が使える(モデルをオブジェクト化し、再利用できる)
- ◆ 抽象度にあったモデリングができる
  - ➔ Behavior/Interface/Channel を使えば、いろいろな抽象度のモデルが容易にできる

□SystemC 2.0の悪いところ

- ◆ ツールとして、まだ枯れていない(Bugが多い)
- ◆ 利用可能なツールが少ない(市販、フリー共に)
- ◆ 利用可能なモデルが少ない(市販、フリー共に)
- ◆ 参考資料(入門書やノウハウ)が少ない。特に日本語!
- ◆ PC/Windowsでは、市販のC++コンパイラが必要である

SystemCを用いて設計した システムLSI

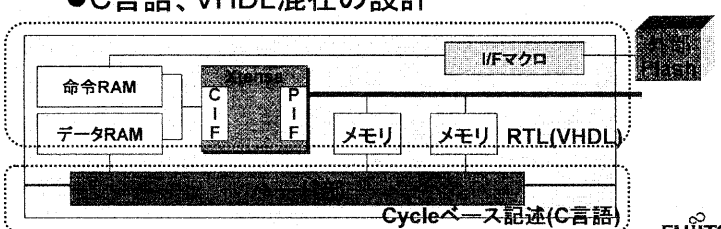
富士通㈱トランスポート事業本部)
基盤技術統括部)システム回路開発部
富士通キャドテック(株)第一開発部
峰　正高

FUJITSU
THE POSSIBILITIES ARE INFINITE

C言語設計の実例(下流工程に適用)

□LSIの概要

- ●機能:SDH/Sonet系のシステム制御用LSI
  - ■ビット処理が主体
- ●規模:1MG Logic + 400KBits RAM
- ●C言語、VHDL混在の設計



FUJITSU
THE POSSIBILITIES ARE INFINITE

設計検証上の課題と対応策(1/2)

□ソフト/ハード協調検証の課題

- ●ソフト/ハード協調検証の高速化

5ms～600ms(平均20ms)の検証
テストパターン1000本程度
20msの検証時間は100Hzのソフトシミュレータで、約5.5時間
1000本のパターンを流すと5500時間＝229日

  - ■高速なCPUコアのシミュレーションモデルが無かった
    - ◆ISSはStand-Aloneの構成で、シミュレーション環境に組み込めるのはRTLモデルのみ
- ●C言語設計でのソフト/ハード協調検証
  - ■使えるツールが無い

FUJITSU
THE POSSIBILITIES ARE INFINITE

設計検証上の課題と対応策(2/2)

□対応策

- ●C言語とVHDL RTLベースの2種のテストベンチを作り、分担して行う
- ●C言語:ソフトとハード論理のデバッグを行う
  - ■ソフトはネイティブコードで実行

自前のソフト/ハード協調検証環境を構築

- ●VHDL RTL : CPUコアなどすべて取り込んだモデルで実施
  - ■ブロック間インタフェースの確認が目的
  - ■ソフトは最小限に:検証専用

FUJITSU
THE POSSIBILITIES ARE INFINITE

言語の選定

□システムモデリング用言語

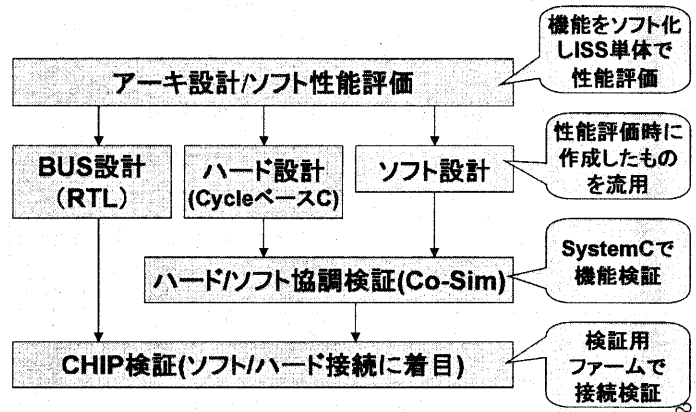
- ●HDL系：VHDL, Verilog
- ●C/C++系：モデリング方法が異なる言語が林立
  - ■HDLのようなC言語：SystemC, Spec-C, SuperLog
  - ■普通のC言語系(ANSI-C)

手軽な設計環境、記述変更はしない

↓

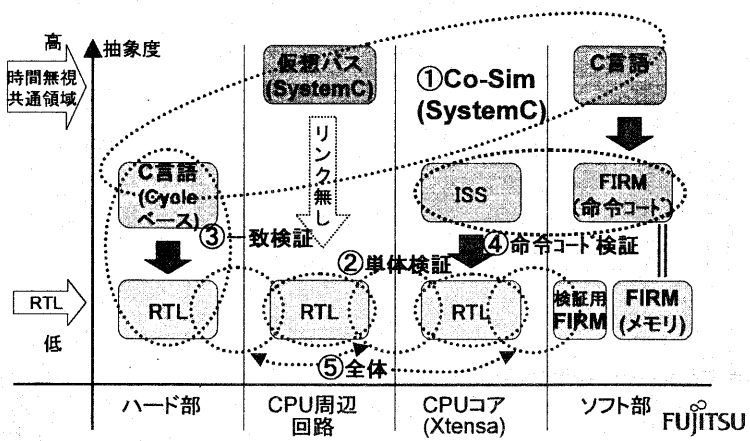
SystemC＋ANSI-C

設計検証フロー



FUJITSU

設計検証の方針

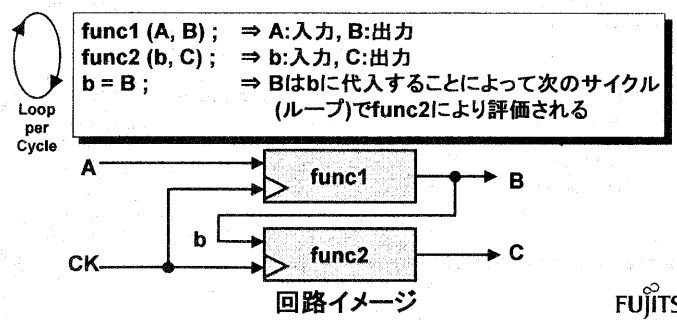


FUJITSU

Cycleベース記述スタイル

□並列動作に見えるような記述

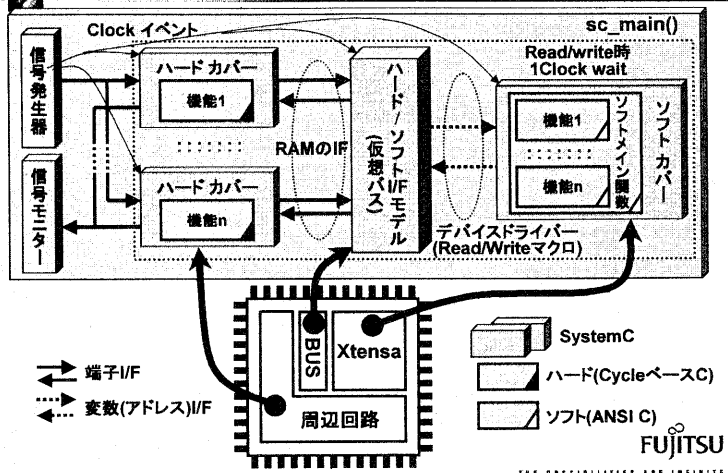
- ●関数呼出しをインスタンス(ブロック)に見せかける
- ●順次処理にならない様に中間変数に代入



回路イメージ

FUJITSU

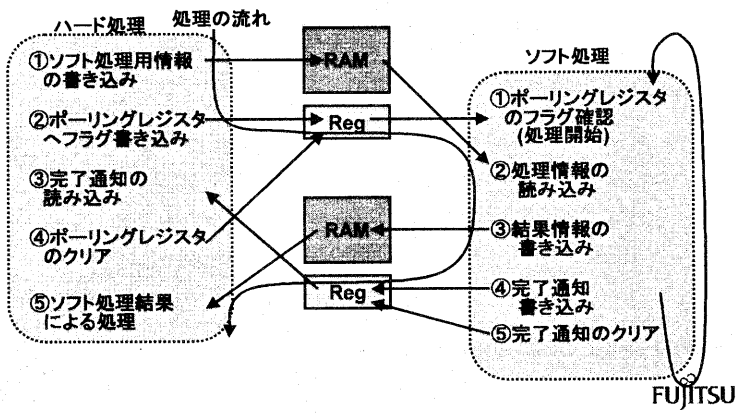
ソフト・ハード協調検証(Co-Sim)構成



FUJITSU
THE POSSIBILITIES ARE INFINITE

設計の条件

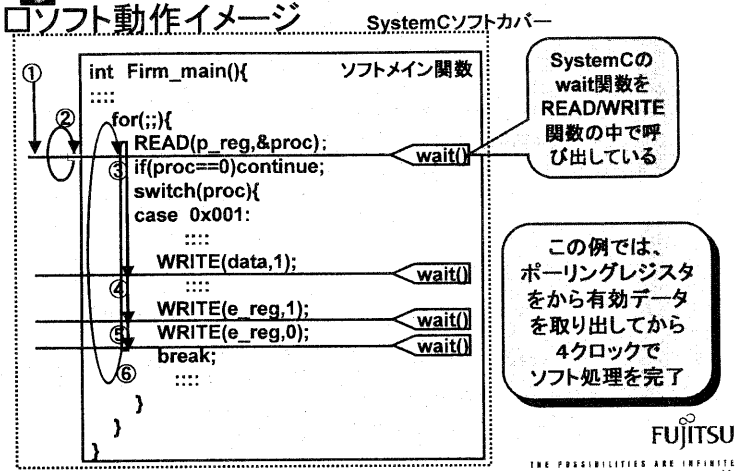
□ハード処理とソフト処理のプロトコルを決める



FUJITSU
THE POSSIBILITIES ARE INFINITE

動作概念

□ソフト動作イメージ



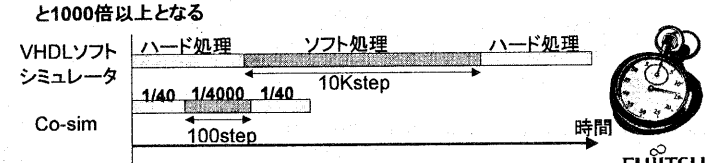
FUJITSU
THE POSSIBILITIES ARE INFINITE

Co-Simシミュレーション速度

	シミュレーション速度	シミュレーション速度比	結果が出るまでの速度比
VHDLソフトシミュレータ	70Hz	1.0	1.0
Co-Sim(SystemC)	3KHz (MAX:61K)	42.8	1000以上(※)

Sun Blade(750MHz)、メモリ5GByte

(※)Co-Simでは、ソフトの速度を無視し、ハードとのデータやり取り時のみWaitを発生させる。Co-simでのソフトのレイテンシは、実際のCPU処理時と比較し100倍以上のため、実際のシミュレーション時間(結果が出る時間)で比較すると1000倍以上となる



FUJITSU
THE POSSIBILITIES ARE INFINITE

C(SystemC)設計で苦労した点

□ビットスライスが書きづらい(解りにくい)

VHDL: `sr_reg1(10)(31 downto 16):=vr_reg1(10)(31 downto16) ;`

Cycle ベースC: `set_slice(sr_reg1[10],31,16,get_slice(vr_reg1[10],31,16)) ;`

□合成前後でのシミュレーション結果が違う

```
uint4  A;  /* 4 bit integer */
uint8  B;  /* 8 bit integer */
A = B ;    ---- ①
A /= 12 ;  ---- ②
```

シミュレーションでは①②の処理は③と一致、合成結果では一致しない

＝ `A = B / 12 ; ---- ③`

※SystemC で、Bit Accurate な記述をすると極端に遅くなる事がわかっていた為、あえてuint4,uint8も 32bitで扱い、高速シミュレーションを実施。一致検証で検出可能だが、イタレーションオーバーヘッドが大きい。

□Co-Sim環境(SystemC)の動作が安定するまでの間、バグの切り分けが困難(VHDLでも同じ)

□ソフトのデバイスドライバー使用法の未徹底によるバグ

●一部、ハードとIFする以外の静的変数にも使用していた

FUJITSU

THE POSSIBILITIES ARE INFINITE

2002/01

13

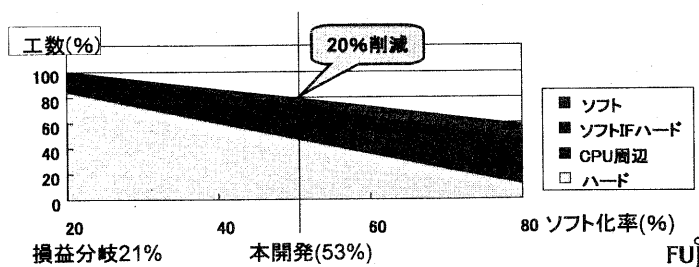
ソフト化の効果

□ソフトでのリカバリ

●仕様変更4件中3件をソフトで対応。

●ハードバグ1件をソフトで対応。

□ソフト化率の変化による工数削減効果



FUJITSU

THE POSSIBILITIES ARE INFINITE

2002/01

14

まとめ

□やる事が多いのに開発効率はあがるのか？

●設計フローの検討、特に検証が鍵

■効率的設計方法(ツール)と最も効果がある検証方式(ツール)をペアで考える

■それぞれの検証が得意とする範囲を明確化し、足らない部分を別の方式でカバーする方法を考える

□設計フローの検討は、システム開発と一体

●EDAツール、モデリング手法などに精通している必要がある

●開発チームにメソドロジ担当が必要

FUJITSU

THE POSSIBILITIES ARE INFINITE

2002/01

15

今後の取り組み

□更に上位への展開

●上位モデル(機能、パフォーマンス)の定型化

●動作合成

■シミュレーションの更なる高速化と人件費削減を期待

◆性能にシビアでないLSIならば、多少の規模増大(製造コストが増大しない範囲)は許される

□パターン分析

●ｱｰｷﾃｸﾁｬとﾒｿﾄﾞﾛｼﾞは、幾つかのパターンに集約

■目的別(アプリケーション別)？性能別？拡張性別？

アーキテクチャ/設計検証環境の半自動化が可能
(プラットフォームの構築)

FUJITSU

THE POSSIBILITIES ARE INFINITE

2002/01

16