Vol.2013-ARC-205 No.12
Vol.2013-OS-125 No.12
2013/4/26

C++ をベースとする新しいハードウェア記述の検討

佐藤 真平[1,a)] 吉瀬 謙二[1,b)]

概要：本稿では，RTL モデリングのための新しい言語として ArchHDL を提案する．ハードウェアのレジスタを変数，ワイヤを関数として扱うことで，C++ で RTL モデリングを実現する．ワイヤは，C++11 で定義されたラムダ関数を用いて記述する．ArchHDL によるハードウェアの記述は Verilog HDL に近い．ArchHDL を用いて記述したハードウェアは C++ のプログラムとしてコンパイル可能である．また，論理シミュレーションを高速に行うことができる．

1. はじめに

プロセッサなどのハードウェア設計は，アーキテクチャ設計，論理設計，回路設計，物理設計といったフローで行われる [1]．アーキテクチャ設計と論理設計においては，RTL（Register Transfer Level）のシミュレーションが不可欠である．このために Verilog HDL などのハードウェア記述言語が用いられることが一般的である．

本稿では，RTL の高速なシミュレーションを実現する **ArchHDL** を提案する．ArchHDL は，ハードウェアの RTL モデリングのための新しい言語である．ArchHDL は，ハードウェアの RTL モデリングを Verilog HDL に近い記述で行う．この言語を用いることで，ハードウェアの論理検証を高速に行うことができる．

構成は以下の通りである．2 章で，提案する ArchHDL を理解するために不可欠な C++11 のラムダ関数について述べる．3 章で，ArchHDL を提案する．4 章で，ArchHDL の予備評価を行い，5 章でまとめる．

2. C++ のラムダ関数と使い方

C++11 と呼ばれる C++ ISO 標準により，ラムダ関数の仕様が定義された．GCC では，バージョン 4.5 よりこの ISO 標準の一部がサポートされ，標準ライブラリとしてラムダ関数が利用できるようになった．

我々の提案する ArchHDL では，このラムダ関数を利用する．ここでは，C++11 のラムダ関数の機能を ArchHDL のために必要な範囲で述べる．

図 1 にラムダ関数の定義のみを含む C++ プログラムの例を示す．2 行目のコードは引数に int x と int y をとり，その和を返すラムダ関数の定義である．

```
1 int main() {
2   [=](int x, int y){ return x + y; };
3   return 0;
4 }
```

図 1 ラムダ関数の定義のみを含む C++ プログラムの例

```
 1 #include <functional>
 2
 3 int main() {
 4   int a = 2;
 5   int b = 3;
 6   std::function<int ()> Sum =
 7                       [=](int x, int y) { return x + y; };
 8   int c = Sum(a, b);
 9   return 0;
10 }
```

図 2 ラムダ関数を定義して，それを使う C++ プログラムの例

ラムダ関数の定義は lambda-introducer と呼ばれる [] [*1] の記述からはじまる．ラムダ関数の返り値の型は，return 文より推測できる場合に省略できる．例に示したラムダ関数の返り値の型は int と推測される．

図 2 にラムダ関数を定義して，それを使う C++ のプログラムを示す．ラムダ関数を使うために，1 行目のように functional という標準ライブラリをインクルードする．

6，7 行目で，関数オブジェクト Sum に [=](int x, int y) { return x + y; } というラムダ関数を代入する．このラムダ関数は int 型の値を返す関数であり，その型は std::function<int ()> となる．これにより，Sum は 2 個

[1] 東京工業大学 大学院情報理工学研究科
[a)] satos@arch.cs.titech.ac.jp
[b)] kise@cs.titech.ac.jp

[*1] [] 内にキャプチャと呼ばれるラムダ関数の機能を指定する記述をする必要がある．ArchHDL では [=] のみを使用するため説明は省略する．

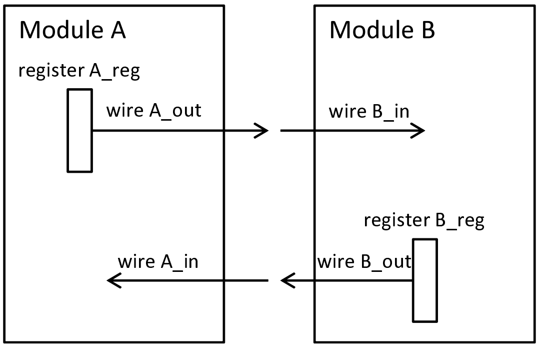

図 3 手続き型言語では記述が難しいモジュール間の接続の例

の int 型の値を引数にとり，その和を返す関数オブジェクトとなる．

8 行目では Sum を呼び出し，返り値を変数 c に代入している．引数には，4, 5 行目にある a, b を与えている．

ArchHDL では，ラムダ関数を図 2 のような使い方をしている．

3. ArchHDL の提案

ArchHDL はハードウェアの RTL モデリングのための言語である．Verilog HDL に近い記述方法を目指している．ユーザはライブラリにより提供される Module クラス，reg クラス，wire クラスおよび C++11 のラムダ関数を用いてハードウェアを記述する．

3.1 ArchHDL の制約

ArchHDL には，次に挙げる制約がある．

- 単一クロックの立ち上がりエッジのみでレジスタの値を更新
- 32 ビットや 64 ビットなどの整数型をベースとした記述

実装を簡潔にするために単一クロックの同期回路で，かつレジスタへの代入はクロックの立ち上がりエッジのみのハードウェアを対象とする．複数のクロックや，クロックの立ち上がりエッジと立ち下がりエッジでの制御はサポートしていない．

演算に用いるデータ型として，C++ 言語の 32 ビットや 64 ビットなどの整数型を使用する．任意のビット幅のレジスタやワイヤはサポートしていない．演算子は C++ が提供する演算子をサポートしており，Verilog HDL で使用可能なビット切り出しやビット連結などの演算はサポートしない．

3.2 ArchHDL の利点

ArchHDL には，次に挙げる利点がある．

- ハードウェアモジュール間の接続の記述が容易
- 論理シミュレーションが高速

ここでは特に重要な前者について述べる．

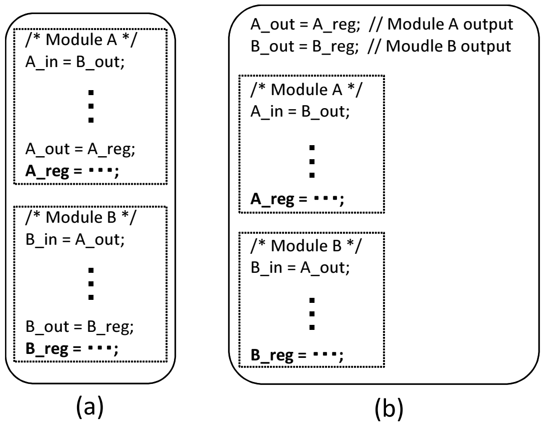

図 4 図 3 のハードウェアの記述例

図 3 に，手続き型言語でハードウェアの挙動を記述しようとすると，記述が難しいモジュール間の接続の例を示す．モジュール A は入力ワイヤ A_in，出力ワイヤ A_out の入出力を持つ．出力ワイヤ A_out はレジスタ A_reg の値を出力する．モジュール B は入力ワイヤ B_in，出力ワイヤ B_out の入出力を持つ．出力ワイヤ B_out はレジスタ B_reg の値を出力する．

モジュール A の出力ワイヤ A_out は モジュール B の入力ワイヤ B_in に接続している．モジュール B の出力ワイヤ B_out は モジュール A の入力ワイヤ A_in に接続している．モジュール A は，あるサイクルに入力として B_reg の値を受け，A_reg の値を出力するというモジュールである．同様に，モジュール B は，あるサイクルに入力として A_reg の値を受け，B_reg の値を出力するというモジュールである．

このハードウェアの挙動をモジュールごとに記述すると，**図 4** (a) のようになる．この記述は，モジュールへの入力，モジュール内の処理，モジュールからの出力，レジスタの更新というまとまりのある記述になっている．レジスタの更新は図中の太字で示した箇所である．

手続き型言語で図 4 (a) のように順番にモジュール A，モジュール B の処理が記述される場合，モジュール A の処理の先頭にある A_in への代入は，1 サイクル前の B_out の状態となり，図 3 のハードウェアとは異なる挙動となる．

このハードウェアの挙動を手続き型言語で記述するには，図 4 (b) のようにレジスタからの読み出しをモジュールごとの処理とは別に記述するといった対策が必要である．しかし，このような記述はモジュールが複雑になればなるほど呼び出し順序に依存関係が生じ，保守性や可読性が損なわれる．

Verilog HDL などのハードウェア記述言語では継続代入やノン・ブロッキング代入のサポートにより呼び出し順序に依らないハードウェアの記述が可能である．ArchHDL では，ライブラリにより提供されるクラス群を使用して

Vol.2013-ARC-205 No.12
Vol.2013-OS-125 No.12
2013/4/26

```
 1 class Counter : public Module {
 2  public:
 3   wire<uint> out;
 4   reg<uint> counter;
 5   void Init() {
 6     out = [=]() { return counter(); };
 7   }
 8   void Always() {
 9     counter <<= (counter() + 1) & 0xff;
10   }
11 };
```

図 5 ArchHDL による 8 ビットカウンタ回路の記述

```
 1 module Counter(CLK, out);
 2     input CLK;
 3     output [7:0] out;
 4
 5     reg [7:0] counter;
 6     assign out = counter;
 7     always @(posedge CLK) begin
 8         counter <= counter + 1;
 9     end
10 endmodule
```

図 6 Verilog HDL による 8 ビットカウンタ回路の記述

ハードウェアを記述すれば，先に例示したようなモジュールの接続も Verilog HDL 同様に記述することができる．

3.3 ArchHDL による RTL モデリング

ArchHDL を用いた RTL モデリングでは，ライブラリにより提供される Module クラス，reg クラス，wire クラスおよび，C++11 のラムダ関数を用いてハードウェアを記述する．

図 5 に，ArchHDL を用いて記述した 8 ビットカウンタ回路のコードを示す．また，**図 6** に，Verilong HDL で記述した 8 ビットカウンタ回路のコードを示す．ArchHDL において，Module クラスを継承して定義されるクラス（Module 子クラス）は，Verilog HDL におけるモジュールに相当する．同様に reg クラス，wire クラスは，それぞれ Verilog HDL におけるレジスタ，ワイヤに相当する．

Module 子クラスは，そのメンバ関数として Init 関数および Always 関数を定義する必要がある．これは，ライブラリにより強制されており，いずれかの関数が必要でない場合でも空の関数を定義する必要がある．

Init 関数には，モジュール内のすべてのワイヤへの継続代入の定義を記述する．これは，Verilog HDL においてモジュール内で定義されている assign 文をすべてこの関数内に記述することに相当する．wire クラスのインスタンスにラムダ関数を代入することでワイヤへの継続代入を記述することができる．図 5 では，6 行目で wire クラスの変数 out に reg クラス の counter の値を返すラムダ関数（[=]()　return counter(); ）を代入している．ここで，reg クラスのオブジェクトは関数として呼び出すことにより，そのサイクルにおける値を取得することができる．このため，先のラムダ関数における counter() によってレジスタの値が取得できる．これは，図 6 の 6 行目に相当する．

Always 関数には，モジュール内のすべてのレジスタへのノン・ブロッキング代入を記述する．ArchHDL では，単一クロックの立ち上がりエッジでの制御のみを対象としている．これは，Verilog HDL における ” always @(posedge clock)“ ブロック内の記述に相当する．reg クラスのインスタンスに <<= 演算子を用いて値を代入することでノン・ブロッキング代入を記述することができる．ArchHDL は，演算子オーバーロードを利用して <<= 演算子を Verilog HDL におけるノン・ブロッキング代入に相当する値の代入として実装している．図 5 では，9 行目で reg クラスの変数 counter に自身の値をインクリメントした値を代入している．これは，図 6 の 8 行目に相当する．

ArchHDL では，データ型として C++ の整数型を使用している．図 5 の例では unsigned int 型を利用しているため，8 ビットカウンタを実現するために値をマスクする必要がある．reg クラスと wire クラスのオブジェクトは関数として呼び出すことにより，そのサイクルにおける値を取得することができる．

3.4 テストベンチの記述

ArchHDL は，C++ 言語をベースとしている．したがって，ユーザは柔軟にテストベンチを記述することが可能である．ここでは，ArchHDL の提供するライブラリを用いたテストベンチ記述の一例を示す．

図 7 は，先に示したカウンタ回路の ArchHDL によるテストベンチの記述例である．インクルードの記述などは省略している．また，14 行目と 16 行目の HALT_CYCLE は定数である．

このテストベンチ記述は，テストモジュール TestTop を作成し，その中でカウンタ回路のインスタンスを生成しテストを行うという設計になっている．そのために，22 行目からの main 関数は簡潔になり，TestTop モジュールのインスタンス生成とライブラリによりステップ実行のみの記述となる．

ArchHDL は，モジュールのインスタンスを生成する際にすべての reg クラスのインスタンス，wire クラスのインスタンスをライブラリで一元管理する設計になっている．そのため，23 行目のようにモジュールのインスタンスを生成したのち，ArchHDL により提供される Step 関数（25 行目）を呼び出せばサイクルごとのシミュレーションができる．

図 7 に挙げたテストベンチの記述は，Verilog HDL で同様の記述ができるという利点がある．**図 8** に Verilog HDL を用いて同様のテストベンチを記述する例を示す．

Vol.2013-ARC-205 No.12
Vol.2013-OS-125 No.12
2013/4/26

```
class TestTop : public Module {
 public:
  reg<uint> HALT;
  reg<uint> cycle;

  wire<uint> cnt_out;
  Counter cnt;

  void Init() {
    cnt_out = cnt.out;
  }
  void Always() {
    cycle <<= cycle() + 1;
    HALT  <<= (cycle() >= HALT_CYCLE);

    if (cycle() > (HALT_CYCLE - 10)) {
      printf("%d %u\n", cycle(), cnt_out());
    }
  }
};

int main() {
  TestTop testtop;
  while (!testtop.HALT()) {
    ArchHDL::Step();
  }
  return 0;
}
```

図 7 ArchHDL を用いたカウンタ回路のためのテストベンチの記述例

```
 1 module TestTop();
 2     reg CLK;
 3     reg  [31:0] cycle;
 4
 5     wire [7:0] ot_cnt;
 6     Counter cnt(CLK, ot_cnt);
 7
 8     initial begin
 9         CLK = 0;
10         cycle = 0;
11         cnt.cnt = 0;
12     end
13
14     always #50 CLK = ~CLK;
15
16     always @(posedge CLK) begin
17         cycle <= cycle + 1;
18         if (cycle > (`HALT_CYCLE - 10))
19             $write("%d %d\n", cycle, ot_cnt);
20         if (cycle >= `HALT_CYCLE) $finish;
21     end
22 endmodule
```

図 8 Verilog HDL を用いたカウンタ回路のためのテストベンチの記述例

ArchHDL を用いた場合と大きく異なる点は 14 行目においてクロックを生成している部分である．それ以外の記述については ArchHDL と Verilog HDL で大きな違いはない．

3.5 ArchHDL の実装

3.5.1 ソフトウェアアーキテクチャ

ArchHDL のライブラリには，Module クラス，wire クラス，reg クラス，これらの 3 個のクラスのインタフェースクラス，Singleton クラスの 7 個のクラスが定義されている．本章では，標準ライブラリのインクルードをのぞくすべてのライブラリのコードを示しながら ArchHDL の実装について述べる．以降の説明では，ユーザが Module クラスを継承して作成したクラスを Module 子クラスと呼ぶ．

```
class RegisterInterface {
 public:
  virtual void Update() = 0;
};

class ModuleInterface {
 public:
  virtual void Init() = 0;
  virtual void Always() = 0;
};

class WireInterface {};

namespace ArchHDL {

class Singleton {
 private:
  std::vector<RegisterInterface*> registers_;
  std::vector<ModuleInterface*> modules_;
  std::vector<WireInterface*> wires_;
 public:
  static Singleton& GetInstance(void) {
    static Singleton singleton;
    return singleton;
  }
  void AddRegister(RegisterInterface* ri) {
    registers_.push_back(ri);
  }
  void AddModule(ModuleInterface* mi) {
    modules_.push_back(mi);
  }
  void AddWire(WireInterface* wi) {
    wires_.push_back(wi);
  }
  void Init() {
    for (uint i = 0; i < modules_.size(); i++) {
      modules_[i]->Init();
    }
  }
  void Exec() {
    for (uint i = 0; i < modules_.size(); i++) {
      modules_[i]->Always();
    }
    for (uint i = 0; i < registers_.size(); i++) {
      registers_[i]->Update();
    }
  }
};

void Step() {
  static bool init = false;
  if (!init) {
    init = true;
    ArchHDL::Singleton::GetInstance().Init();
  }
  ArchHDL::Singleton::GetInstance().Exec();
}

}  // namespace ArchHDL
```

図 9 ArchHDL ライブラリにおける各インタフェースクラスと Singleton クラスと Step 関数の定義

図 9 に RegisterInterface クラス，ModuleInterface クラス，WireInterface クラス，Singleton クラスおよび Step 関数の定義を示す．

ModuleInterface クラス，WireInterface クラス，RegisterInterface クラスはそれぞれ Module クラス，wire クラ

ス，reg クラスのインタフェースクラスである．Singleton クラスが，Module 子クラス，wire クラス，reg クラスのインスタンスをシングルトン・パターンにより一元管理する．これは，ArchHDL のライブラリにおいて核となるクラスである．

Singleton クラスは，メンバ変数として Module クラス，wire クラス，reg クラスのインタフェースクラスのポインタを格納する可変行列をもつ（18 ～ 20 行目）．Module 子クラス，wire クラス，reg クラスのインスタンスが生成される際に，そのインスタンスへのポインタが Singleton クラスに渡される．また，ポインタは Singleton クラスに渡される際にそれぞれのインタフェースクラスに自動でアップキャストされる（26 ～ 34 行目）．

Step 関数（50 ～ 57 行目）は，1 サイクルのシミュレーションを行う関数である．Step 関数を呼び出すと，Singleton クラスの Exec 関数が呼ばれる．ただし，初回 Step 関数の呼び出しのみ Singleton クラスの Init 関数が呼ばれる．Step 関数を繰り返し呼び出すことにより，複数サイクルにわたるシミュレーションが行なわれる．

Init 関数（35 ～ 39 行目）は，保持しているすべての Module 子クラスのインスタンスの Init 関数を呼ぶ（37 行目）．

Exec 関数（40 ～ 47 行目）は，保持しているすべての Module 子クラスのインスタンスの Always 関数を呼び（42 行目），次に保持しているすべての reg クラスのインスタンスの Update 関数を呼ぶ（45 行目）．

Always 関数によりすべてのレジスタについて次のサイクルにおける値が計算される．Update 関数によりレジスタの値が更新される．この Always と Update の処理によりレジスタのノン・ブロッキング代入を実現する．

3.5.2 reg クラスの定義

図 10 に，reg クラスの定義を示す．reg クラスは，扱うデータ型をテンプレート引数にとるテンプレートクラスである．また，インタフェースクラスである RegiterInterface クラスを継承する．

ArchHDL ではレジスタを変数として扱うため，reg クラスはメンバ変数にテンプレート引数で与えられたデータ型の変数 curr_ と next_ を持つ（5，6 行目）．curr_ は，あるサイクルにおけるレジスタの値で，next_はその次のサイクルのレジスタの値である．Module 子クラスの Always 関数の呼び出しにより，next_に値が代入される．reg クラスのメンバ関数 Update を呼ぶことで，next_ の値は curr_ に反映される（15 ～ 20 行目）．これによりレジスタへのノン・ブロッキング代入の挙動を実現する．

reg クラスのオブジェクトへの値の代入をそのメンバ変数 next_ への値の代入とするために，演算子オーバーロードにより <<= 演算子を再定義している（24 ～ 27 行目）．<<= 演算子により代入された値は，変数 next_ に格納され，set_ フラグがセットされる．

```
1 template <typename T>
2 class reg : public RegisterInterface {
3  private:
4   bool set_;
5   T curr_;
6   T next_;
7
8   // copy constructor
9   reg<T>(const reg<T>& other);
10  reg<T>& operator=(const reg<T>& rhs);
11 public:
12  reg(): set_(false), curr_(0), next_(0) {
13    ArchHDL::Singleton::GetInstance().AddRegister(this);
14  }
15  void Update() {
16    if (set_) {
17      curr_ = next_;
18      set_ = false;
19    }
20  }
21  void operator=(T val) {
22    curr_ =  val;
23  }
24  void operator<<=(T val) {
25    set_ = true;
26    next_ = val;
27  }
28  T operator ()() {
29    return curr_;
30  }
31 };
```

図 10 ArchHDL ライブラリにおける reg クラスの定義

ArchHDL では，すべての Module 子クラスのインスタンスの Always 関数を呼び出した後に，すべての reg クラスのインスタンス Update 関数を呼び出す．したがって，Always 関数が呼び出されている間に取得できるレジスタの値 curr_ は Update 関数が呼ばれるまで保持されている．

reg クラスのコンストラクタ（12 ～ 14 行目）では，メンバ変数を初期化し，自身のポインタを Singleton クラスに渡す処理が行われる．テスト記述や初期値設定のために = 演算子による値の代入も定義されている（21 ～ 23 行目）．= 演算子による値の代入は，式が評価された時点で curr_ の値を変更する．reg クラスのインスタンスを関数として呼び出すことで curr_の値を取得することができる（28 ～ 29 行目）．

3.5.3 wire クラスの定義

図 11 に，wire クラスの定義を示す．wire クラスは，テンプレート引数として扱うデータ型をとるテンプレートクラスである．また，インタフェースクラスの WireInterface クラスを継承している．

ArchHDL では，ワイヤは関数として扱うため，wire クラスはメンバ変数にラムダ関数 lambda_ を持つ（4 行目）クラスとなっている．このラムダ関数は，テンプレート引数として与えられたデータ型を返す関数である．

コピーコンストラクタの禁止（7，8 行目）と演算子のオーバーロード（13，14 行目）により，wire クラスへの

Vol.2013-ARC-205 No.12
Vol.2013-OS-125 No.12
2013/4/26

```
1 template <typename T>
2 class wire : public WireInterface {
3  private:
4   std::function<T ()> lambda_;
5
6   // copy constructor
7   wire<T>(const wire<T>& other);
8   wire<T>& operator=(const wire<T>& rhs);
9  public:
10   wire(): lambda_(nullptr) {
11     ArchHDL::Singleton::GetInstance().AddWire(this);
12   }
13   void operator=(std::function<T ()> lambda) {
14     lambda_ = lambda;
15   }
16   T operator()() {
17     return lambda_();
18   }
19 };
```

図 11 ArchHDL ライブラリにおける wire クラスの定義

```
1 class Module : public ModuleInterface {
2  private:
3   // copy constructor
4   Module(const Module& other);
5   Module& operator=(const Module& rhs);
6  public:
7   Module() {
8     ArchHDL::Singleton::GetInstance().AddModule(this);
9   }
10   virtual void Init() {}
11   virtual void Always() {}
12 };
```

図 12 ArchHDL ライブラリにおける Module クラスの定義

= 演算子による代入はラムダ関数に限定される．これにより，wire クラスのオブジェクトは Module 子クラスの Init 関数で記述されるラムダ関数を保持するクラスとなる．

wire クラスのコンストラクタ（10 ～ 12 行目）では，メンバ変数を初期化し，自身のポインタを Singleton クラスに渡す処理が行われる．wire クラスのオブジェクトを関数として呼び出すと，自身の持つラムダ関数を呼び出した結果を返す（16 ～ 18 行目）．これにより，wire クラスのオブジェクトを関数呼び出しすることで，そのサイクルでのワイヤの値が取得できる．

3.5.4 Module クラスの定義

図 12 に，Module クラスの定義を示す．Module クラスは，インタフェースクラスの ModuleInterface クラスを継承するクラスである．ArchHDL でハードウェアを記述する際に，このクラスを継承してモジュールを記述する．

コンストラクタ（7 ～ 9 行目）では，自身のポインタを Singleton クラスに渡す．ModuleInterface クラスにおいて Init 関数と Always 関数が仮想関数として定義されているため，これらの関数を定義する必要がある．Module クラスは，Module 子クラスのインタフェースとして定義しているため，Init 関数と Always 関数として空の関数を定義している（10，11 行目）．

4. 予備評価

ArchHDL で記述したハードウェアのシミュレーション速度について Verilog HDL で記述し Icarus Verilog[2] でコンパイルした場合と比較する．本稿では，オープンソースの Verilog HDL コンパイラである Icarus Verilog を比較対象とする．

予備評価のために，3 章で示した 8 ビットカウンタのインスタンスを 1024 個生成したハードウェアを使用する．ArchHDL では，unsigned int 型のレジスタをマスクして 8 ビットのレジスタを実装している．10 M サイクルのシミュレーション時間を time コマンドにより計測する．

シミュレーション環境は次の通りである．

- CPU：Intel Core i7-3770K
- メモリ：32 GB
- OS：Ubuntu 12.04
- GCC：バージョン 4.6.3，最適化オプション -O3
- Icarus Verilog：バージョン 0.9.5

ArchHDL で記述したハードウェアを GCC でコンパイルした場合のシミュレーション時間は 42.9 秒である．Verilog HDL で記述したハードウェアを Icarus Verilog でコンパイルした場合のシミュレーション時間は 2366 秒である．このとき，ArchHDL の シミュレーション時間は，Verilog HDL に対して 55.2 倍の速度向上である．予備評価の結果から，ArchHDL を用いたシミュレーションが大幅に速度向上することを確認した．

なお，予備評価のために用いたハードウェアは，ある程度の規模があるものの現実的とは言えない．複雑で現実的なハードウェアを用いた評価は今後の課題である．

5. まとめ

ハードウェアの RTL モデリングのための新しい言語として ArchHDL を提案した．ハードウェアのレジスタを変数，ワイヤを関数として扱うことで，C++ で RTL モデリングを実現する．ワイヤは，C++11 で定義されたラムダ関数を用いて記述する．

ArchHDL によるハードウェア記述は， Verilog HDL に近い．手続き型言語でハードウェアの挙動を記述しようとすると，呼び出し順序に依存関係が生じ記述が難しい．そのようなハードウェアの例を用いて，ArchHDL が Verilog HDL のようにハードウェア記述ができることを述べた．

ArchHDL を用いて記述したハードウェアは C++ のプログラムとしてコンパイル可能である．Verilog HDL で記述したハードウェアを Icarus Verilog でコンパイルした場合と比較して，8 ビットカウンタのシミュレーション速度が大幅に向上することを確認した．

今後の課題は，現実的なハードウェアを記述し，シミュ

レーション速度や記述量などから ArchHDL 有用性を示すことである．

謝辞

ArchHDL を開発するにあたって，その設計から実装まで多大な貢献をしていただいた佐野伸太郎氏に感謝致します．本研究の一部は，科学技術振興機構・戦略的創造研究推進事業 (CREST) の「ディペンダブルネットワークオンチッププラットフォームの構築」の支援による．

参考文献

[1] 伊藤則之, 安永守利：プロセッサ設計手法の現状と今後–高性能化を実現する設計フローと CAD システム–，電子情報通信学会論文誌. D, 情報・システム，No. 12, pp. 2004–2030 (2011).
[2] Icarus Verilog: http://iverilog.icarus.com/.