IEICE TRANS. INF. & SYST., VOL.E101–D, NO.2 FEBRUARY 2018

PAPER Special Section on Reconfigurable Systems

ArchHDL: A Novel Hardware RTL Modeling and High-Speed Simulation Environment

Shimpei SATO[†a)], **Ryohei KOBAYASHI**[††b)], ***and*** **Kenji KISE**[†c)], ***Members***

SUMMARY LSIs are generally designed through four stages including architectural design, logic design, circuit design, and physical design. In architectural design and logic design, designers describe their target hardware in RTL. However, they generally use different languages for each phase. Typically a general purpose programming language such as C or C++ and a hardware description language such as Verilog HDL or VHDL are used for architectural design and logic design, respectively. That is time-consuming way for designing a hardware and more efficient design environment is required. In this paper, we propose a new hardware modeling and high-speed simulation environment for architectural design and logic design. Our environment realizes writing and verifying hardware by one language. The environment consists of (1) a new hardware description language called ArchHDL, which enables to simulate hardware faster than Verilog HDL simulation, and (2) a source code translation tool from ArchHDL code to Verilog HDL code. ArchHDL is a new language for hardware RTL modeling based on C++. The key features of this language are that (1) designers describe a combinational circuit as a function and (2) the ArchHDL library realizes non-blocking assignment in C++. Using these features, designers are able to write a hardware transparently from abstracted level description to RTL description in Verilog HDL-like style. Source codes in ArchHDL is converted to Verilog HDL codes by the translation tool and they are used to synthesize for FPGAs or ASICs. As the evaluation of our environment, we implemented a practical many-core processor in ArchHDL and measured the simulation speed on an Intel CPU and an Intel Xeon Phi processor. The simulation speed for the Intel CPU by ArchHDL achieves about 4.5 times faster than the simulation speed by Synopsys VCS. We also confirmed that the RTL simulation by ArchHDL is efficiently parallelized on the Intel Xeon Phi processor. We convert the ArchHDL code to a Verilog HDL code and estimated the hardware utilization on an FPGA. To implement a 48-node many-core processor, 71% of entire resources of a Virtex-7 FPGA are consumed.

key words: *hardware description language, RTL modeling, RTL simulation*

1. Introduction

VLSI chips such as high performance processors and SoCs with many hardware elements are designed in the flow of (1) architectural design, (2) logic design, (3) circuit design, and (4) physical design. In architectural design and logic design, simulations in register transfer level (RTL) are indispensable for efficient debugging and logical verification.

Manuscript received May 2, 2017.
Manuscript revised September 8, 2017.
Manuscript publicized November 17, 2017.

[†]The authors are with Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
[††]The author is with University of Tsukuba, Tsukuba-shi, 305–8571 Japan.
a) E-mail: satos@ict.e.titech.ac.jp
b) E-mail: kobayashi@cs.tsukuba.ac.jp
c) E-mail: kise@cs.titech.ac.jp

DOI: 10.1587/transinf.2017RCP0012

Architectural design and logic design are also important for hardware FPGA implementation. Available hardware resources in FPGAs are increasing, and requirements to implement a large scale hardware are also increasing. So, the elapsed time of an RTL simulation for such large scale design is becoming very long even if using a fast RTL simulator. Therefore, CAD systems which realize high-speed RTL simulation are strongly required.

In this paper, we propose **ArchHDL** that is a novel hardware RTL modeling and high-speed simulation environment. This environment comprises of (1) a hardware description language called ArchHDL which enables to simulate a hardware faster than Verilog HDL simulation and (2) a source code translation tool from ArchHDL to Verilog HDL. In this environment, designers implement their target hardware with a provided ArchHDL library based on C++. The simulation is carried by executing an ArchHDL code which implements a target hardware as a C++ program For the following hardware design flow, the source code written in ArchHDL is converted to a Verilog HDL code by the translation tool.

In ArchHDL, designers write and verify a hardware using C++ language. Designers generally use a general-purpose programming language such as C or C++ to describe a hardware in the architectural design phase. In logic design phase, they use a hardware description language such as Verilog HDL or VHDL to describe a hardware in RTL. Designers typically have to describe their target hardware twice by using different languages and it is time-consuming way to design a hardware. ArchHDL realizes transparent and efficient hardware design from the architectural design phase to the logic design phase by using one programming language.

Our environment realizes high-speed hardware RTL simulation. Typical Verilog HDL simulator takes a long time to simulate a hardware because it can simulate a detailed situation including some delays. Such detailed simulation is necessary for hardware design. However, requirement for high-speed logic level simulation in RTL is also thought to be high. The RTL simulation of a hardware written in ArchHDL is realized by executing the program compiled with a typical C++ compiler. Then, it achieves high-speed simulation compared to Verilog HDL simulation. Additionally, the ArchHDL library supports parallel execution of the simulation. This realizes more high-speed simulation of a large scale hardware which uses many modules such as many core processors by parallel execution.

This paper is based on our previous works [1]–[3], where we proposed ArchHDL and showed preliminary evaluation results. In this paper, we show the following contributions:

- Providing a detailed description of ArchHDL with its library implementation.
- High-speed RTL simulation of a practical hardware model by ArchHDL.
- Efficiently parallelized RTL simulation on Intel Xeon Phi processor by ArchHDL.

2. A Novel Hardware RTL Modeling Environment

2.1 Concept of ArchHDL

ArchHDL is a hardware description language based on C++. It provides a C++ library for hardware RTL modeling. The characteristics of ArchHDL are:

1. Verilog HDL-like coding style.
2. Description of a combinational circuit as a function using lambda expression of C++11.
3. Supporting non-blocking assignment.
4. Supporting user-defined data types and object-oriented programming style.
5. Cycle based simulation (not event driven).
6. Parallel simulation using OpenMP without decreasing the simulation accuracy.
7. Simple library (only about 200 lines in total).

The ArchHDL library includes definitions of the *Module* class, the *reg* class, the *wire* class, and functions for simulation. Using these classes and supported non-blocking assignment, hardware designers can write a hardware in Verilog HDL-like style. More abstracted description such as functional level is allowed because a hardware written in ArchHDL is just a C++ program. It realizes transparent design of hardware from abstracted level description to RTL level description.

To describe a combinational circuit as a function, the lambda expression which is newly added to the C++ standard library called C++11 is used. Non-blocking assignment, which is generally supported in hardware description languages such as Verilog HDL or VHDL, is not supported in general purpose programming languages. ArchHDL supports non-blocking assignment by the ArchHDL library which realized by using C++ operation overload.

The simulation of a hardware is carried by the execution of the source code written in ArchHDL compiled with general C++ compilers. The simulation speed is faster than the simulation using Verilog HDL simulator.

2.2 Hardware RTL Modeling in ArchHDL

Figure 1 is a block diagram of the sample circuit used for explanation of ArchHDL hardware modeling in this section. The circuit is a 32-bit pseudo random value generator using

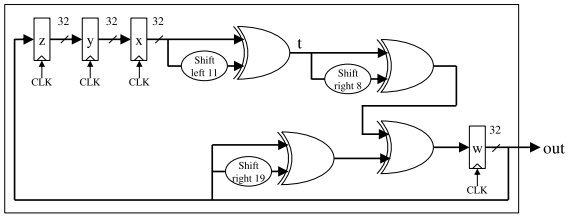

Fig. 1 A block diagram of sample circuit used for explanation of hardware description in ArchHDL. (Xorshift random value generator)

```
1  class Xorshift : public Module {
2   public:
3    wire<uint_1> i_rst_x;
4    wire<uint_1> i_enable;
5    wire<uint_32> i_seed;
6    wire<uint_32> o_out;
7
8    reg<uint_32> x;
9    reg<uint_32> y;
10   reg<uint_32> z;
11   reg<uint_32> w;
12   wire<uint_32> t;
13
14   void Assign() {
15     t = [=]() { return x() ^ (x() << 11); };
16     o_out = w;
17   }
18   void Always() {
19     if (!i_rst_x()) {
20       x <<= 123456789;
21       y <<= 362436069;
22       z <<= 521288629;
23       w <<= 88675123 ^ i_seed();
24     } else {
25       if (i_enable()) {
26         x <<= y();
27         y <<= z();
28         z <<= w();
29         w <<= (w() ^ (w() >> 19)) ^ (t() ^ (t() >> 8));
30       }
31     }
32   }
33 };
```

Fig. 2 A description of Xorshift pseudo random value generator in ArchHDL.

Xorshift algorithm [4]. It employs four registers and generates random value by XOR and shift operations. Initialization mechanisms of register values and input of seed, which are contained in the sample description we present later, are omitted in this figure.

Figure 2 is a sample description of Xorshift random value generator in ArchHDL. Descriptions about inclusion of libraries are omitted.

The *Xorshift* class is declared as a subclass of the *Module* class and it represents the hardware module. A hardware module is declared as a subclass of the *Module* class which is defined in the ArchHDL library. Behavior of hardware modules is mainly described by using the *reg* class, the *wire* class, the *Assign* function, and the *Always* function, which are also defined in the library. A class defined by inheriting the *Module* class corresponds to a *module* in Verilog HDL.

Five wires and Four registers are declared in the module. An instance of the *wire* class and the *reg* class can be regarded as a *wire* and *reg* in Verilog HDL respectively. These classes are implemented as template classes in the ArchHDL library. Therefore, users must specify a data type into the angled bracket when they declare an instance of the *wire* class or the *reg* class. In Fig. 2, *uint_1* and *uint_32* are used as data types for *wire* and *reg*. The *wire* class and the *reg* class are implemented as functional objects in the library, so a value of these instances can be referred by a function call of the class instance.

The number at the end of these data types represents

the bit width of the data. The number is just used to specify the bitwidth when translating the source code by the translation tool, and these are actually implemented as an alias of *unsigned int* in the library. Therefore, a value declared with these data type is not masked by the represented bit width. Also, user defined structure is able to use as a data type for a *wire* class or *reg* class instance.

A module description in ArchHDL does not employ ports which are usually supported in hardware description languages. Therefore, the description about connections between modules is implemented by referring directly to *wire* or *reg* class instances declared in a module. To simplify the analysis by the translation tool, some naming rules are applied. As indicated from the 3rd line to the 6th line in Fig. 2, an instance name starts from "i_" is an input port and an instance name starts from "o_" is an output port.

Combinational circuits are defined by assigning a functional object to a *wire* class instance. All of assignment statements to *wire* instances should be written in *Assign* function as denoted from the 14th line to the 17th line in Fig. 2. In the case of this example, combinational circuits that are able to describe in Verilog HDL assign statements are only used. However, using the lambda expression of C++11 denoted in the 15th line, designers are able to define various combinational circuits in ArchHDL.

The ArchHDL library supports non-blocking assignment to *reg* class instance. Statements of non-blocking assignment are described using "<<=" operator, and they should be written in a *Always* function as indicated from the 18th line to the 32nd line in Fig. 2. The *Always* function is equivalent to "always @(posedge CLK)" in Verilog HDL.

2.3 Testbench in ArchHDL

Figure 3 shows a sample testbench in ArchHDL for the random value generator shown in Fig. 2. Descriptions of inclusion of libraries are omitted. This code is to display generated random values for 30 cycles. The seed value for the random value generator is set as 1.

```
1  #include "common/xorshift.h"
2
3  class TestTop : public Module {
4   public:
5    static const int HALT_CYCLE = 30;
6
7    reg<uint_1> HALT;
8    reg<int> cycle;
9
10   wire<uint_1> rst_x;
11   reg<unsigned int> seed;
12   wire<unsigned int> rand;
13   Xorshift xorshift;
14
15   void PortConnect() {
16     xorshift.i_rst_x = rst_x;
17     xorshift.i_enable = rst_x;
18     xorshift.i_seed = seed;
19     rand = xorshift.o_out;
20   }
21   void Assign() {
22     rst_x = [=]() { return (cycle() < 1) ? 0 : 1; };
23   }
24   void Initial() {
25     HALT = 0;
26     cycle = 0;
27   }
28   void Always() {
29     cycle <<= cycle() + 1;
30     HALT  <<= (cycle() >= HALT_CYCLE);
31
32     if (!rst_x()) {
33       seed <<= 1;
34     } else {
35       printf("%08x\n", rand());
36     }
37   }
38 };
39
40 int main() {
41   TestTop testtop;
42   do {
43     ArchHDL::Step();
44   } while (!testtop.HALT());
45   return 0;
46 }
```

Fig. 3 A sample testbench for Xorshift in ArchHDL.

In the main function from the 40th line, the *TestTop* module is generated and then the *Step* function, which provided by the library, is called in the do-while loop. At the time of the *TestTop* instance is generated, all of *reg*, *wire*, and *Module* class instances are stored in a data area prepared in the ArchHDL library. The *Step* function calls the *Always* function of all *Module* class instances stored in the library data area. Therefore, the call of the function simulates the hardware behavior in one cycle.

The *PortConnect* function is used to connect ports of the Xorshift module as denoted from the 15th line to the 20th line. The role of this function is same to the *Assign* function mentioned above. However, it is necessary to describe the *PortConnect function* and the *Assign* function separately to simplify the analysis of the translation tool. From the 24th line to the 27th line, register initializations are described in the *Initial* function. The role of this function is equivalent to initial block in Verilog HDL.

2.4 ArchHDL Library Implementation

2.4.1 Software Architecture

Seven classes are defined in the ArchHDL library. They are the *Module* class, the *ModuleInterface* class, the *wire* class, the *WireInterface* class, the *RegInterface* class, the *reg* class and the *Singleton* class. In this subsection, we explain about the implementation of the ArchHDL library while showing its source code.

Figure 4 shows the definition of the *RegisterInterface* class, the *ModuleInterface* class, the *WireInterface* class, the *Singleton* class and the *Step* function.

The *ModuleInterface* class, the *WireInterface* class and the *RegisterInterface* class are interface classes of *Module* class, *wire* class and *reg* class respectively. ArchHDL adopts the singleton pattern, and *Singleton* class consolidates instances of *Module* child class, *wire* class and *reg* class. This class is the most important class in the ArchHDL library.

The *Singleton* class has three dynamic arrays as the member variables, which keep pointers to instances of *Module* class, *wire* class and *reg* class (denoted in the line from 23 to 25). When an instance of *Module* child class, *wire* class or *reg* class is created, a pointer to the instance is passed to the instance of *Singleton* class. At that time, the pointer is upcasted to its interface class automatically (denoted in the line from 32 to 40).

The *Step* function is the function to do one cycle simulation of implemented hardware. In *Step* function, an *Assign* function, an *Initial* function and an *Exec* function in the *Singleton* class are called. Multi-cycle simulation can be carried by iterative call of the *Step* function.

The *Assign* function in the *Singleton* class (denoted in the line from 41 to 53) calls *PortConnect* functions and *As-*

```
 1  class ModuleInterface {
 2   public:
 3    virtual void PortConnect() = 0;
 4    virtual void Assign() = 0;
 5    virtual void Initial() = 0;
 6    virtual void Always() = 0;
 7  };
 8
 9  class RegisterInterface {
10   public:
11    virtual void Update() = 0;
12  };
13
14  class WireInterface {
15   public:
16    virtual int Assign() = 0;
17  };
18
19  namespace ArchHDL {
20
21  class Singleton {
22   private:
23    std::vector<ModuleInterface*> modules_;
24    std::vector<WireInterface*> wires_;
25    std::vector<RegisterInterface*> update_registers_;
26
27   public:
28    static Singleton& GetInstance(void) {
29      static Singleton singleton;
30      return singleton;
31    }
32    void AddModule(ModuleInterface* mi) {
33      modules_.push_back(mi);
34    }
35    void AddRegister(RegisterInterface* ri) {
36      update_registers_.push_back(ri);
37    }
38    void AddWire(WireInterface* wi) {
39      wires_.push_back(wi);
40    }
41    void Assign() {
42      for (auto module : modules_) {
43        module->PortConnect();
44        module->Assign();
45      }
46      int cond;
47      do {
48        cond = 0;
49        for (auto wire : wires_) {
50          cond += wire->Assign();
51        }
52      } while (cond);
53    }
54    void Initial() {
55      for (auto module : modules_) {
56        module->Initial();
57      }
58    }
59    void Exec() {
60      update_registers_.clear();
61      for (auto module : modules_) {
62        module->Always();
63      }
64      for (auto reg : update_registers_) {
65        reg->Update();
66      }
67    }
68  };
69
70  void Step() {
71    static bool first_step = true;
72    if (first_step) {
73      first_step = false;
74      ArchHDL::Singleton::GetInstance().Assign();
75      ArchHDL::Singleton::GetInstance().Initial();
76    }
77    ArchHDL::Singleton::GetInstance().Exec();
78  }
79
80  }  // namespace ArchHDL
```

Fig. 4 The source code of each interface class, *Singleton* class and *Step* function in the ArchHDL library.

sign functions of all *Module* child class instances which are held in the *Singleton* class. By this process, all of wires in target hardware are connected by assignment of lambda expression. Note that, the *Assign* function in the *Singleton* class is only called at the first call of the *Step* function.

The *Initial* function of the *Singleton* class (denoted in the line from 54 to 58) calls *Initial* functions of all *Module* child class instances which are held in the *Singleton* class. By this process, initial values are assigned to registers in the module. Note that, the *Initial* function in the *Singleton* class is only called at the first call of the *Step* function.

In *Exec* function (denoted in the line from 59 to 67), at first *Always* functions of all *Module* child class instances held in *Singleton* class are called (denoted in the line 62). Next, *Update* functions for *reg* class instances, in which their value is changed by the *Always* function, are called (denoted in the line 65). By this process, "always @(posedge CLK)" block for one clock cycle is simulated.

Values for the next cycle of all registers are computed by calling *Always* function, but it is not allowed to refer in the current cycle. The values of registers are updated to

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

Fig. 5 The source code of *reg* class in the ArchHDL library.

the new values by calling *Update* function. The process of *Always* function and *Update* function realizes non-blocking assignment of Verilog HDL.

2.4.2 Definition of *reg* Class

Figure 5 shows the definition of *reg* class. This class is a template class which takes a data type to use in its class instance as the template argument. The *RegisterInterface* class is inherited as the interface class.

The *reg* class has two variables, *curr_* and *next_*, which data type is given by the template argument. Value of the *curr_* is a current value in a certain cycle, and value of the *next_* is a value for the next cycle. A value is assigned to the variable *next_* by calling the *Always* function. A value of the variable *next_* is assigned to the variable *curr_* by calling *Update* function which is a member method of *reg* class. In this way, non-blocking assignment to a register is implemented.

To assign a value to the variable *next_* in *reg* class object, "<<=" operator is used. We redefine the "<<=" operator by using operator overload. If values of *next_* and *curr_* are different, the pointer to the *reg* class instance is stored to the array that keeps pointers to the registers to be updated.

After calling *Always* functions of all *Module* class instance, the *Update* functions of *reg* class instances are called. Thus, a value of the variable *curr_* in *reg* class is kept while the function call of the *Always* functions.

The constructor of *reg* class initializes the member variables. Blocking assignment to *reg* class object by "=" operator is also defined for description of test bench or setting of initial value. A value assigned to a *reg* class object by the "=" operator updates the variable *next_* immediately. A reference to a *reg* class value is given by calling the instance as a function.

2.4.3 Definition of *wire* Class

Figure 6 shows the definition of *wire* class. This class is a template class which takes a data type to use in its class instance as the template argument. The *WireInterface* class is inherited as the interface class.

```
template <typename T>
class wire : WireInterface {
 private:
  std::function<T ()> lambda_;
  const wire<T>* prev_;

  // disallow copy
  wire<T>(const wire<T>& other);
 public:
  wire(): lambda_(nullptr), prev_(nullptr) {
    ArchHDL::Singleton::GetInstance().AddWire(this);
  }
  void operator=(std::function<T ()> lambda) {
    lambda_ = lambda;
  }
  void operator=(const wire<T>& rhs) {
    std::function<T ()> lambda = rhs.GetLambda();
    if (lambda == nullptr) {
      prev_ = &rhs;
    } else {
      lambda_ = lambda;
    }
  }
  void operator=(const reg<T>& rhs) {
    lambda_ = rhs.GetLambda();
  }
  int Assign() {
    if (lambda_ == nullptr) {
      assert(prev_ != nullptr);
      std::function<T ()> lambda = prev_->GetLambda();
      if (lambda == nullptr) {
        return 1;
      } else {
        lambda_ = lambda;
      }
    }
    return 0;
  }
  T operator()() {
    return lambda_();
  }
  std::function<T ()> GetLambda() const {
    return lambda_;
  }
};
```

Fig. 6 The source code of *wire* class in the ArchHDL library.

The *wire* class has a variable *lambda_* to hold a lambda function. Data type of return value of this lambda function is a data type given by template argument. Assignment to a *wire* class object is limited to assignments from a lambda function, a *wire* class instance, and a *reg* class instance.

The constructor of the *wire* class initializes the member variables and pass a pointer to itself to the *Singleton* class. Calling a object of *wire* class as a function returns a return value of the lambda function evaluation. Thus, a value of a wire in a certain cycle can be get from function call of the *wire* class instance.

2.4.4 Definition of Module Class

Figure 7 is the definition of the *Module* class. This class inherits the *ModuleInterface* class. We use the *Module* class as the parent class to describe a hardware module in ArchHDL.

The constructor of the *Module* class gives a pointer to itself to the *Singleton* class. The *PortConnect* function, the *Assign* function, the *Initial* function, and the *Always* function are declared as virtual functions in the *ModuleInterface*. Therefore, they are also declared as virtual functions in the *Module* class.

2.5 Parallelization Using OpenMP

Figure 8 is the *Exec* function of the *Singleton* class. There are two loops in the function. Both of them are able to execute in parallel because they do not have loop dependency. We use OpenMP for parallelization. This parallelization is supported by the ArchHDL library and users do not need to change their source code. The accuracy of the simulation does not change by parallelization.

In Fig. 8, the *modules_* holds all pointers to module

```
class Module : public ModuleInterface {
 private:
  // copy constructor
  Module(const Module& other);
  Module& operator=(const Module& rhs);
 public:
  Module() {
    ArchHDL::Singleton::GetInstance().AddModule(this);
  }
  virtual void PortConnect(){}
  virtual void Assign(){}
  virtual void Initial(){}
  virtual void Always(){}
};
```

Fig. 7 The source code of the *Module* class in the ArchHDL library.

```
void Exec() {
  update_registers_[thread_num].clear();
#pragma omp for
  for (size_t i = 0; i < modules_size; ++i) {
    modules_[i]->Always();
  }
  for (auto reg : update_registers_[thread_num]) {
    reg->Update();
  }
}
```

Fig. 8 Parallelized *Exec* function of ArchHDL.

classes of a target hardware. First for-loop calls the *Always* function of every modules in parallel by using OpenMP, which calculate register value for the next cycle. At this time, if the *next_* has different value to the *curr_* in a *reg* class instance, the pointer to the instance is added to the vector *update_registers_*. The adding process sometimes occurs simultaneously in the *Always* function call of different threads. Therefore, we prepare the array for each thread to prevent the conflict. The second for-loop calls the *Update* function of registers in each *update_registers_* of threads.

We do not set any specific thread scheduling type for OpenMP in our library. Therefore, thread scheduling depends on a default setting of a compiler and typically it will set as static.

2.6 Translation Tool from ArchHDL to Verilog HDL

We are developing a code translator from ArchHDL code to Verilog HDL code. As shown in some examples of hardware description in ArchHDL, ArchHDL is able to describe a hardware in Verilog HDL-like description style by using classes such as the *reg* class and the *wire* class provided by the library. The code translation from ArchHDL to Verilog HDL is realized by simple code parser and replacement function. Especially, designers are expected to describe the same statement about assignments and arithmetic expressions in ArchHDL and Verilog HDL. Thus, the code translator does not optimize the code during translation.

Figure 9 shows the translation flow from ArchHDL code to Verilog HDL code. The tool receives an ArchHDL code as an input and outputs a Verilog HDL code. The translation is delivered as follows: (1) code scanning and (2) information generation for Verilog HDL code by string replacement and parsing.

In the parsing process, statements which are not able to describe in Verilog HDL syntax are analyzed. Basically it is not necessary to parse statements written in ArchHDL in detail, because they must be the same statements in Verilog HDL.

The main restrictions of description in ArchHDL which can be converted to Verilog HDL are as follows: (1) using only the *reg* class and the *wire* class as a data type and (2) using up to 2-dimensional array for instance declaration.

In particular, the reason of the limitation for the array depends on that the Verilog HDL syntax limitation and the multidimensional array are not supported in some Verilog HDL simulator. We think that we can describe a practical hardware sufficiently in ArchHDL under such restrictions.

2.7 Advantages and Disadvantages of ArchHDL over Verilog HDL

The advantages of ArchHDL are (1) the intuitive module description by object-oriented programming and (2) the flexible testbench description using C++ standard environment.

Hardware resources on LSIs or FPGAs are increasing, and opportunities to describe a hardware which implements a lot of the same module like many-core processors are also increasing. In ArchHDL, designers are able to declare modules, registers, or wires using array. Therefore, they also able to use *for* statements to describe the behavior of such hardware intuitively.

Architectural verification needs plenty of simulations using various parameters, and requires flexible description to its testbenches. The testbench description in ArchHDL is able to use the random value generators, variable-length array and so on which are included in C++ standard library. Therefore, the flexibility of testbench description is equal to typical software simulators. Furthermore, the simulation speed is faster than Verilog HDL simulation which described in the same abstraction level.

To simplify the implementation of ArchHDL library and the source code translation tool, the current ArchHDL has some restrictions compared with the description capability of Verilog HDL. The main restrictions are (1) described hardware may use only one clock signal, (2) the assignment of values to registers is done only in a positive edge of the clock, and (3) designers describe registers and wires using C++ integer type like 32-bit *int* or 64-bit *int*.

ArchHDL supports hardware which allows assigning values to registers at positive edges of a single clock. Therefore, it does not support to use multiple clocks and to assign values at negative edges of the clock. Although SFL [5] has similar constraint, it was used for variety of hardware descriptions [6], [7].

ArchHDL uses C++ integer type like 32-bit *int* or 64-bit *int* for a data type of registers and wires. The declaration of registers and wires of any bit width are not supported. ArchHDL supports C++ common operators, and does not support bit selection operation and bit concatenation operation which are supported in Verilog HDL.

The ArchHDL library is implemented with about 200 lines of a source code and it is simple. Users are able to expand the library to introduce other clock signals or data types. We think that these restrictions are caused by an initial stage implementation of the library. Therefore, they will be eliminated by the progress of this work.

3. Experimental Results

We evaluate our proposed hardware modeling environment which consists of ArchHDL and the source code translation tool in two aspects: (1) The simulation speed and parallelization efficiency of a hardware described in ArchHDL, and (2) The FPGA resource utilization of a hardware synthesized from Verilog HDL code generated by the source code translation tool.

3.1 A Sample Hardware for Practical Evaluation

We implemented M-Core Architecture [8], a many-core processor employs scratchpad memories, as a sample hardware for evaluation. Figure 10 shows the construction of M-Core Architecture. Each node of M-Core Architecture is composed of a core and a router. The core consists of a Processing Element (PE), a memory controller (MC), a local memory, and a Network Interface Controller (NIC). The PE is a 32-bit five-stage pipelined MIPS processor [9]. Each node is connected to the mesh network via the router. The router architecture is a conventional Input-buffered Virtual Channel router [10] with five-stage pipeline, four virtual channels per input port, and 4-flit buffer per virtual channel.

For a data transfer between cores, DMA transfer is used. DMA command is sent from a PE to its NIC via memory-mapped IO. The NIC loads sending data from the local memory using information of the DMA command. After that, packets are generated and injected into the network. When the packets arrives at a destination node, information in the packet is used to store the received data to the local memory.

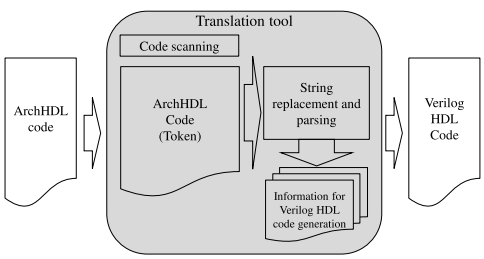

Fig. 9 Translation flow from ArchHDL code to Verilog HDL code.

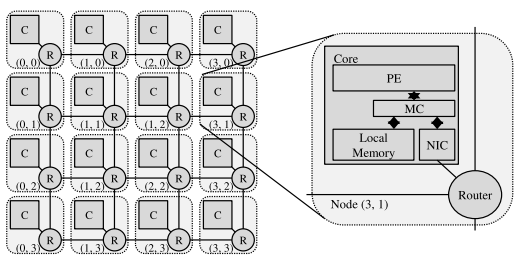

Fig. 10 M-Core Architecture used as a practical hardware for evaluation.

Table 1 Hardware Configuration

	Xeon CPU	Xeon Phi
Model	E5-2687W	31S1P
Architecture	SandyBridge	Knights Corner
Frequency	3.1GHz	1.1GHz
Threads/core	2	4
Cores	8	57
Total Threads	16	228
Memory	64GB	8GB
OS	Ubuntu 14.04.2	MPSS 3.4.3

3.2 Evaluation of Simulation Speed

We implement the many-core processor introduced in the previous section in ArchHDL and measure the simulation time while running an application on the processor. The simulation time is compared with the time of Verilog HDL simulation using Synopsys VCS version H-2013.06. The Verilog HDL code used in the VCS simulation is automatically generated from the ArchHDL code by our translation tool. For ArchHDL source code compilation, GCC 4.7.3 (g++) and Intel Compiler 14.0.1 (ICPC) are used. The compiler optimization option is Ofast for both compilers. The detailed computer environment for simulation is concluded in Table 1.

Firstly, we show the simulation speed of ArchHDL compared with that of Verilog HDL simulation on Intel Xeon CPU. For the simulation, the number of nodes of the many-cores processor are varied (from 2×2 to 10×10). The many-core processor executes an application that every cores communicate with each other in every 100 cycles. The total execution cycles of the simulation is 100,000 cycles. The simulation is performed 10 times for each configuration, and the average simulation time of them is used as the final result. The parallelized simulation accuracy is the same as single-thread execution of the simulation. For the parallelized simulation, we use a computer with Xeon CPU, which has the 8 physical cores (16 logical cores using SMT).

Figure 11 illustrates the simulation results under single thread. The X-axis indicates the model size of the sample many-core processor and the Y-axis represents the simulation time in second.

We can see that the simulation time increases when the number of nodes increases. The simulation by VCS is fastest (maximum 2.9 times faster than GCC and maximum 2.7 times faster than Intel Compiler). However, the speedup of the VCS simulation compared to other simulations (GCC and Intel Compiler) decreases when the number of nodes increases, In particular, the speedup is maximum when the number of nodes is 4 (2×2 network) and minimum when the number of nodes is 100 (10×10 network).

Figure 12 shows the simulation results in case of parallelized simulation (with 8-thread and 16-thread). "VCS" and "ICPC 1 thread" in Fig. 12 are same as ones illustrated in Fig. 11.

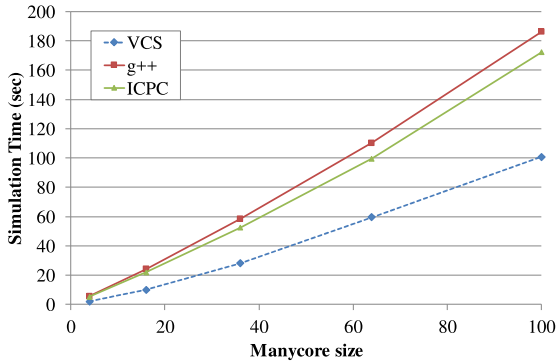

Fig. 11 Single-threaded simulation time on Xeon CPU. VCS, g++, and ICPC represent simulation speed by Verilog HDL simulator of Synopsys, GNU compiler (GCC) for C++, and Intel compiler for C++, respectively.

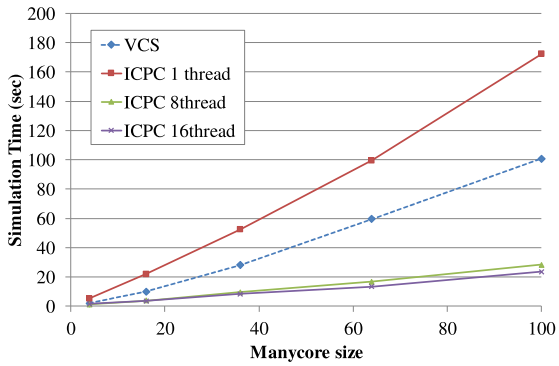

Fig. 12 Multi-threaded simulation time on Xeon CPU. Note that the simulation time of VCS is single-thread. VCS and ICPC represent simulation speed by Verilog HDL simulator on Synopsys, and Intel compiler for C++, respectively.

We can see that every parallelized simulation is faster than the VCS simulation. The parallelized simulation is 4.5 times faster in maximum than the VCS simulation when using 16-thread. The simulation results also show that the speedup of the parallelized simulation compared to the VCS simulation increases when the number of nodes increases.

Second, we show the parallelization efficiency of the ArchHDL simulation using Intel Xeon Phi processor. Figure 13 shows the speedup ratios depending on the number of used physical cores in Xeon Phi and Fig. 14 shows the speedup ratios of Xeon Phi 57 cores (total 228 threads) against Xeon CPU 8 cores (total 16 threads).

From Fig. 13, we can see that the performance of parallel execution on Xeon Phi is higher when the model size of simulation is larger and 57-core execution is about 48 times faster than the 1-core execution in 16×16 size. Also from Fig. 14, we can see that Xeon Phi 57-core is slower than Xeon CPU 8-core when the size is 4×4 and 8×8, but on the biggest size (16×16) Xeon Phi is 1.4 times faster than Xeon CPU.

When the number of nodes is divisible by the number of threads, the chunk size for multi-threaded simulations fits to the multiple of node size. Thus, load balance of such simulation becomes almost uniformly and the multi-threaded

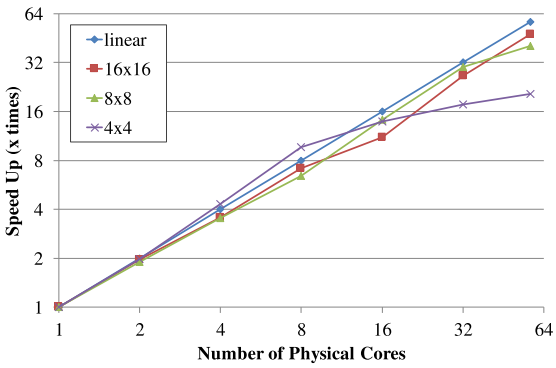

Fig. 13 Speedup of many-core on Xeon Phi

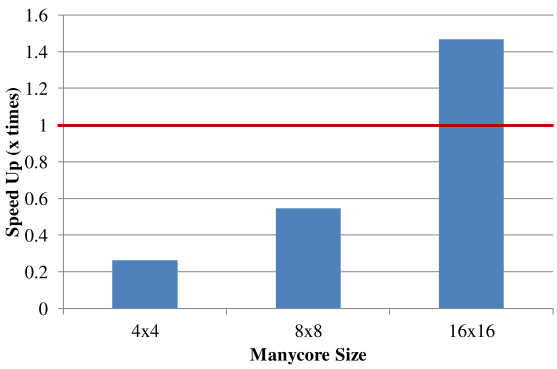

Fig. 14 Speedup of Xeon Phi (57-core) against Xeon CPU(8-core)

simulation performs well.

3.3 FPGA Resource Utilization of the Many-Core Processor Synthesized from Converted Verilog HDL Code

Here, we estimate the FPGA resource utilization of M-Core architecture. The target device is Xilinx Virtex-7 XC7VX485 on the evaluation kit VC707.

The number of nodes to implement is 48 (8×6). The register files on the Processing Element of each node is implemented using LUT-RAM. The local memory in each core and the input buffers in each router are implemented using Block-RAM. Parameters of the local memory on each core are 32-bit data width, 32 KB total size, and 3 port (2 read, 1 write). Parameters of the input buffer are 38 bit data width, 16 entry (4 entry for each 4 virtual channels in a port), and 2 port (1 read, 1 write). The number of router port except the edge node of the processor is 5. Therefore, five Block-RAMs are used for input buffers in a router.

Table 2 shows the hardware utilization of the processor on the Virtex-7 FPGA. The result is obtained from a synthesis report of place and route. According to the synthesis report, the processor occupied 54,509 slices which is 71% of entire resources, and runs at 114.9 MHz. The number of slices for each element in a node are about 460 slices for Processing Element, about 25 slices for Memory Controller, about 280 slices for Network Interface Controller, and about 400 slices for router.

Table 2 Hardware utilization of a 48-node many-core processor on the Virtex-7.

	Slice	Reg	LUT	BRAM (RAMB36E1))
Used	54,509	79,641	158,103	1,007
Utilization	71%	13%	52%	97%

From the result of hardware utilization, we found that the size of the hardware generated from the converted Verilog HDL code by the translation tool is practicable. For example, Heracles [11], that is an RTL based many-core simulation environment, implements a similar many-core processor to our processor. Its node mainly consists of a 7-stage pipelined 32-bit MIPS processor core, instruction/data caches, and virtual channel NoC router. In [11], in spite of using of hand-written Verilog HDL design methodology and a Virtex-6 LX550T FPGA which has 10% more slices than our targeted FPGA, the authors could implement an at most 25-node many-core processor because of its core complexity, whose node has a 32 KB local memory. Compared with that, our processor core is simpler but ArchHDL are able to generate a many-core processor with about twice core counts instead. Given the above, it is possible to conclude that the hardware size of our many-core processor generated from our proposed ArchHDL is practicable.

4. Related Works

Chisel [12], SystemC [13], MyHDL [14], and PyMTL [15] are hardware description languages that are able to compile as a program of general-purpose programming language. Although hardware designers are able to describe hardware in RTL in these languages, the hardware description style in those languages is very different from the style of Verilog HDL.

SystemC [13] is designed based on C++ and it is implemented as a C++ class library. The hardware described in SystemC is able to compile and execute as a C++ program. Hardware designers describe hardware using classes and macros which are provided in the SystemC library. While most of the HDLs like Verilog HDL, VHDL and proposed ArchHDL support the RTL of design, SystemC originally supports the design at a higher abstraction level to model large hardware systems.

MyHDL is designed based on Python. The hardware described in MyHDL is compiled and is executed as a Python program. The project provides a tool to convert a source code in MyHDL to Verilog HDL and VHDL for hardware synthesis. It is reported that the architectural simulation speed of MyHDL is about 3 times faster [16] than the speed of Verilog HDL compiled by Icarus Verilog. Although this project is unique using Python, MyHDL has not been used extensively.

Chisel is designed based on Scala. The hardware description in Chisel is converted to C++ code for high-speed simulation, and also is converted to Verilog HDL code for ASIC synthesis. It is reported that the simulation speed of Chisel C++ simulation is 7.8 times faster against Synopsys

VCS.

PyMTL is a Python based hardware design framework and function level, cycle level and register transfer level simulations can be done within the same framework. In cycle level and RTL level simulation, it uses a JIT compiler to generate a simulation binary. It converts Python code into Verilog HDL and further converts the Verilog files into C++ simulation code by using Verilator [17]. In this way, PyMTL offers high coding usability thanks to Python while it suppresses the simulation speed degradation within 4 to 6 times compared to a tuned C++ simulation.

SFL [5] is a unique language and PARTHENON is a high-level CAD tool for SFL developed by NTT(Nippon Telegraph and Telephone Corporation). In SFL, the designer does not describe a clock signal explicitly and the system assumes the existence of the global clock implicitly. This strategy is the same as ArchHDL. Although SFL is an attractive language, development and maintenance of PARTHENON system had stopped.

A number of works have studied parallel RTL simulation and some works used GPUs or many-core.

In [18], [19], authors investigate the parallel RTL simulation of SystemC [13] using GPUs. SystemC is a hardware description language implemented as a C++ class library. They are based on discrete event-driven simulation and [19] propose a method whose aim is to reduce synchronization events. On the other hand, [20] converts Verilog files to GPU source code. The simulation method is based on Chandy-Misra-Bryant (CMB) algorithm, which is asynchronous parallel simulation protocol. Although these works achieve high speedup (more than 10x - 100x), the simulated circuits are simple such as AES or 8b/10b decoding/encoding and do not evaluate the methods with practical hardware designs. We use complex and large hardware designs and therefore our simulation environment is practicable.

In [21], authors studied parallel simulation of SystemC using Intel Single-chip Cloud Computer (SCC) [22] which has 48 cores connected by a 2D mesh network. The simulation algorithm is also based on CMB. They used several sizes of Hermes Multi-Processor System [23] models running MPEG applications for evaluation. It showed that simulation on 48 SCC cores speedup up to 30x compared to that on 1 SCC core. However, they did not compare the simulation time on SCC with that on general multi-core systems. We compared the simulation time on Xeon Phi to that on Xeon CPU and showed that the simulation on Xeon Phi is truly faster than Xeon CPU.

5. Conclusion

In this paper, we propose a new hardware RTL modeling and high-speed simulation environment for architectural design and logic design. The environment comprises of (1) a hardware description language called ArchHDL and (2) a source code translation tool from ArchHDL to Verilog HDL. The goals of the proposed environment are to attain following: (1) transparent and efficient hardware modeling, (2) realizing the environment which is able to verify both architectural design and logic design in one description, and (3) high-speed simulation compared to the Verilog HDL simulation.

We evaluate our proposed environment in two aspects: (1) the simulation speed and parallelization efficiency of a hardware described by ArchHDL and (2) the amount of resources usage of hardware when synthesizing Verilog HDL code generated from ArchHDL code by the translation tool. For practical evaluation, we implemented a many-core processor in ArchHDL and also converted the source code to Verilog HDL by the translator. The simulation speed of ArchHDL was about 4.5 times faster than the simulation speed using Synopsys VCS which is one of the fastest Verilog HDL simulator. The parallelization efficiency of the simulation on the Intel Xeon Phi processor was high in the case of a large scale hardware simulation. The resource utilization of the 48-node many-core processor on Virtex-7 was 71% on the Virtex-7. From the result, we found that the scale of the hardware generated by the converted Verilog HDL code by the translation tool is applicable.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 16H02794.

References

[1] S. Sato and K. Kise, “ArchHDL: A New Hardware Description Language for High-Speed Architectural Evaluation,” Proc. IEEE 7th International Symposium on Embedded Multicore SoCs (MCSoC ’13), pp.107–112, Sept. 2013.
[2] S. Sato and K. Kise, “ArchHDL: A Novel Hardware RTL Development Environment in C++,” Proc. 11th International Symposium on Applied Reconfigurable Computing (ARC ’15), pp.53–64, May 2015.
[3] T. Misono, R. Kobayashi, S. Sato, and K. Kise, “Effective Parallel Simulation of ArchHDL under Manycore Environment,” Proc. 3rd International Symposium on Computing and Networking -Across Practical Development and Theoretical Research- (CANDAR ’15), Dec. 2015.
[4] G. Marsaglia, “Xorshift RNGs,” J. Statistical Software, vol.8, no.14, pp.1–6, 2003.
[5] Parthenon Web page. http://www.kecl.ntt.co.jp/parthenon/.
[6] C. Kon and N. Shimizu, “The design of an i8080a instruction compatible processor with extended memory address,” Proc. 2003 Asia and South Pacific Design Automation Conference (ASP-DAC ’03), pp.571–572, 2003.
[7] H. Hayasaka, H. Haramiishi, and N. Shimizu, “The design of PCI bus interface,” Proc. 2003 Asia and South Pacific Design Automation Conference (ASP-DAC ’03), pp.579–580, 2003.
[8] K. Uehara, S. Sato, T. Miyoshi, and K. Kise, “A study of an infrastructure for research and development of many-core processors,” Proc. International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp.414–419, Dec. 2009.
[9] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/software Interface, Morgan Kaufmann Series in Computer Graphics, Morgan Kaufmann, 2012.
[10] W. Dally and B. Towles, Principles and Practices of Interconnection

Networks, The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2004.
[11] M.A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: A Tool for Fast RTL-based Design Space Exploration of Multicore Processors," Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13), pp.125–134, 2013.
[12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," Proc. 49th Annual Design Automation Conference (DAC '12), New York, NY, USA, pp.1216–1225, ACM, 2012.
[13] "IEEE standard for Standard SystemC Language Reference Manual," IEEE std. 1666-2011, 2011.
[14] J. Decaluwe, "MyHDL: a python-based hardware description language," Linux J., vol.2004, no.127, pp.5–, Nov. 2004.
[15] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," Proc. 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '14), pp.280–292, Dec. 2014.
[16] MyHDL Web page. http://www.myhdl.org/doku.php/performance.
[17] Verilator Web page. https://www.veripool.org/.
[18] M. Nanjundappa, H.D. Patel, B.A. Jose, and S.K. Shukla, "SCGPSim: A fast SystemC simulator on GPUs," Proc. 15th Asia and South Pacific Design Automation Conference (ASP-DAC '10), pp.149–154, Jan. 2010.
[19] S. Vinco, V. Bertacco, D. Chatterjee, and F. Fummi, "SAGA: SystemC acceleration on GPU architectures," Proc. 2012 Design Automation Conference (DAC '12), pp.115–120, June 2012.
[20] H. Qian and Y. Deng, "Accelerating RTL simulation with GPUs," Proc. 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '11), pp.687–693, Nov. 2011.
[21] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker, "Adaptive algorithm and tool flow for accelerating systemc on many-core architectures," Proc. 17th Euromicro Conference on Digital System Design, pp.137–145, Aug. 2014.
[22] J. Howard, S. Dighe, S.R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V.K. De, and R.V.D. Wijngaart, "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," IEEE J. Solid-State Circuits, vol.46, no.1, pp.173–183, Jan. 2011.
[23] E.A. Carara, R.P. de Oliveira, N.L.V. Calazans, and F.G. Moraes, "HeMPS - a framework for NoC-based MPSoC generation," Proc. 2009 IEEE International Symposium on Circuits and Systems, pp.1345–1348, May 2009.

Shimpei Sato received the B.E., M.E., and D.E. degrees in engineering from Tokyo Institute of Technology, Tokyo, Japan, in 2007, 2009, and 2014, respectively. He is currently an Assistant Professor with the Department of Information and Communications Engineering of Tokyo Institute of Technology. From 2014 to 2016, he worked in High performance computing area as a post doctoral researcher, where he investigated an application performance analysis/tuning method. His current research interests include approximate computing realization by architecture design and circuit design and their applications. He is a member of IEEE, ACM, and IPSJ.

Ryohei Kobayashi received the M.E degree and the Ph.D. degree from Tokyo Institute of Technology, Japan in 2013 and 2016. He is currently an assistant professor of Center for Computational Sciences, University of Tsukuba, Japan. His research interests include FPGA systems for high performance computing. He is a member of IPSJ.

Kenji Kise received the B.E. degree from Nagoya University in 1995, the M.E. degree and the Ph.D. degree from the University of Tokyo in 1997 and 2000 respectively. He is currently an associate professor of School of Computing, Tokyo Institute of Technology. His research interests include computer architecture and parallel processing. He is a member of ACM, IEEE, and IPSJ.