東京工業大学工学部

学士論文

Research on High-speed Logic Simulation for Computer Architectures

指導教員　Kenji KISE　Associate Professor

平成 25 年 7 月

提出者

学科　Department of Computer Science

学籍番号　09_06410

氏名　Tatsuya KANEKO

指導教員　印	
学科長　認定印	

2013　学士論文内容梗概

Research on High-speed Logic Simulation for Computer Architectures

指導教員　Kenji KISE　Associate Professor
Department of Computer Science
09_06410　Tatsuya KANEKO

The VLSI chips such as high performance processors or SoCs with processor elements are designed in the flow of architectural design, logic design, circuit design, and physical design. In the architectural design and logical design, Register Transfer Level (RTL) simulation is essential for logical verification. Hardware description languages such as Verilog HDL or VHDL are often used for the RTL modeling and simulation. But the logic simulation speed of Verilog HDL is slow.

ArchHDL which is a new language for hardware RTL modeling on C++ was proposed to solve this problem. ArchHDL treats a register as a variable and a wire as a function. The hardware description in ArchHDL is similar to that in Verilog HDL.

The architectural simulation speed with ArchHDL is much faster than that with Icarus Verilog, which is a verilog simulator of open source. It is often faster than VCS of Synopsys, Inc. VCS is one of the fastest verilog simulators and proprietary software.

In this thesis, I aim to speed up the logical simulation with ArchHDL. I propose and implement three methods as follows: (1) removal of conditional branch for data update, (2) storing register values to the continuous memory location and (3) the parallelization of the execution of multiple instances.

I evaluate the simulation time with 4096 of the counters circuit as a micro benchmark and a stencil-computation circuit as a realistic hardware as an evaluation. The evaluation results as the micro benchmark show that the elapsed time of the proposed methods is 5.23 times faster than that of the original ArchHDL. The evaluation results as the realistic hardware show that the elapsed time of the proposed methods is 1.95 times faster than that of the original ArchHDL. The results show that ArchHDL applied the proposed methods can perform faster than VCS in the all evaluations in this thesis.

Contents

Chapter 1

Introduction

1.1 Background and research objective

The VLSI chips such as high performance processors or SoCs with processor elements are designed in the flow of architectural design, logic design, circuit design, and physical design. In the architectural design and logical design, Register Transfer Level (RTL) simulation is essential for logical verification. Hardware description languages such as Verilog HDL [1] or VHDL [2] are often used for the RTL modeling and simulation.

ArchHDL was proposed as a new language for hardware RTL modeling [3]. The hardware description in ArchHDL is similar to that in Verilog HDL.

The architectural simulation speed with ArchHDL is much faster than that with Icarus Verilog [4], which is a verilog simulator of open source. It is also faster than that with NC-Verilog of Cadence, Inc. NC-Verilog is a verilog simulator and proprietary software. However, it is not faster than that with VCS of Synopsys, Inc. [5] in some hardware simulation. VCS is one of the fastest verilog simulators and proprietary software.

In this thesis, I aim to speed up the logical simulation with ArchHDL. I propose and implement three methods as follows: (1) removal of conditional branch for data update, (2) storing register values to the continuous memory location and (3) the parallelization of the execution of multiple instances.

In comparison with Icarus Verilog, NC-Verilog and VCS, I show the usefulness of the methods in this thesis.

1.2 Outline of this thesis

The rest of this thesis is organized as follows. Section 2 provides the overview of ArchHDL. In section 3, I propose the three methods of ArchHDL. In section 4, I evaluate the simulation speed of ArchHDL in comparison with the various verilog simulators. In section 5, I present a conclusion of this thesis.

Chapter 2

Overview of ArchHDL

2.1 The Lambda Function of C++11

In ArchHDL, *lambda function* is used for hardware modeling. The specification of lambda function is defined in the C++ ISO standard named C++11 and lambda function is supported as a C++ standard library from GCC version 4.5 released in 2010. In this section, we explain a little about the C++11 lambda function.

Simply to say, a lambda function is an anonymous function or a function without its name. Fig. 2.1 shows a sample program which has a definition of lambda function. The definition of the lambda function is in the line 2 of the code. The function takes two int values of x and y as arguments and returns the sum of these arguments. Note that this program includes just a definition of lambda function for explanation. Therefore, the function in the line 2 has no effect for this program.

The description of lambda function starts from *lambda-introducer* [] and a lambda introducer may contain a *lambda-capture* like "=". Note that a statement start with "[=]" is a lambda function. The return type of a lambda function is automatically deduced from the return expression in compile time. In the case of this example, the return type is expected as int.

Fig. 2.2 shows a sample program using lambda function. In the line 4 and 5, a lambda `function [=](int x, int y) { return x + y; }` is assigned to a function object *Sum*. The type of the lambda function is `std::function<int ()>`, and *Sum* is declared with this type. Thus, *Sum* becomes a function which takes two int arguments and returns an int value.

In the line 6, the function *Sum* is called. The return value of *Sum* is assigned to a variable *c*. The variable *a* and the variable *b* are defined in the line 2 and 3. They are two arguments of the function *Sum*. This program returns a value of 5 computed by

```
1 void sample() {
2   [=](int x, int y){ return x + y; };
3 }
```

Fig. 2.1 A sample C++ program which includes just a definition of lambda function. The definition of the lambda function is in the line 2. The function takes two int values of x and y as arguments and returns the sum of these arguments.

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

Fig. 2.2 A sample C++ program which includes a definition and a use of lambda function. Sum is a function object. A lambda function defined in the line 5 is assigned to Sum. Sum is used in the line 6 with two integer arguments. This program returns a value of 5.

Sum.

2.2 RTL modeling on ArchHDL

The hardware description in ArchHDL is similar to that of Verilog HDL. The hardware RTL modeling on ArchHDL is done with *Module* class, *reg* class, *wire* class and the lambda function of C++11.

Fig. 2.4 shows a code of 8-bit counter in Verilog HDL. The code for the same 8-bit counter in ArchHDL is shown in Fig. 2.3. In the code in ArchHDL, a class which declared as a subclass of the *Module* class corresponds to a module of Verilog HDL. In the following, we call this class "*Module child* class". Similarly, the *reg* class and the *wire* class correspond to a register and a wire of Verilog HDL respectively.

In a *Module child* class, A *Init* function and An *Always* function are declared for a wire assignment and a register assignment.

In the *Init* function, users write the assignment of all wire in a module. To realize the continuous assignment in C++, ArchHDL uses the lambda function and defines all wire as a function. The description in *Init* function corresponds to the assign

```
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 };
```

Fig. 2.3 A description of 8-bit counter in ArchHDL.

```
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
```

Fig. 2.4 A description of 8-bit counter in Verilog HDL.

statement of Verilog HDL. In the line 6 of Fig. 2.3, the lambda function (`[=]() { return counter(); }`) which returns a value of *reg counter* is assigned to the *wire out*. Note that, the value of the *reg* class object can be get by calling the object as a function. Therefore, the function call *counter()* returns the value of *reg counter*. This statement equals to the line 6 of Fig. 2.4.

In the *Always* function, users write the assignment of all register in a module. In ArchHDL, the assignment of a value to a register is allowed only at the time of a positive edge of a single clock. ArchHDL realizes the non-blocking assignment to a register using <<= operator. In the ArchHDL library, the <<= operator is overloaded as the non-blocking assignment implementation. Therefore, statements in the *Always* function is corresponds to statements in the `always@(posedge clock)` block in Verilog HDL. In the line 9 of Fig. 2.3, the *reg counter* is assigned an incremented value of itself. This statement equals to the line 8 of Fig. 2.4.

ArchHDL uses the integer type of C++ as a data type of registers and wires. In

```
1  class TestTop : public Module {
2   public:
3    reg<uint> HALT;
4    reg<uint> cycle;
5
6    wire<uint> cnt_out;
7    Counter cnt;
8
9    void Init() {
10     cnt_out = cnt.out;
11   }
12   void Always() {
13     cycle <<= cycle() + 1;
14     HALT  <<= (cycle() >= HALT_CYCLE);
15
16     if (cycle() > (HALT_CYCLE - 10)) {
17       printf("%d %u\n", cycle(), cnt_out());
18     }
19   }
20 };
21
22 int main() {
23   TestTop testtop;
24   while (!testtop.HALT()) {
25     ArchHDL::Step();
26   }
27   return 0;
28 }
```

Fig. 2.5 A sample description of a test bench for the 8-bit counter in ArchHDL

the example shown in Fig. 2.3, we used *unsigned int* as the data type for the register and the wire. To implement the 8-bit counter in ArchHDL, the value of *unsigned int* is masked by `0xff` as in the line 9 of Fig. 2.3.

2.3 Test bench description of ArchHDL

ArchHDL is implemented using C++. Users are able to describe a test bench flexibly to the extent possible in C++. In this section, we show an example of a test bench in ArchHDL which is similar to a test bench in Verilog HDL.

Fig. 2.5 is an example of a test bench for the 8-bit counter shown in Fig. 2.3 using ArchHDL. The description of includes and definitions are omitted. The variable *HALT_CYCLE* used in the line 14 and 16 is a constant number.

This test bench is designed as making the test module *TestTop* to test the 8-bit

```
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
```

Fig. 2.6 A sample description of a test bench for the 8-bit counter in Verilog HDL

counter in it. In this way, the description of the *main* function (in the line 22 to 28) becomes simple. Only the creation of the *TestTop* module instance and the call of *Step* function are written in the main function.

The *reg* class instances and the *wire* class instances in a *Module child* class are managed in the ArchHDL library. The pointers of them are passed to ArchHDL library when the instance of the *Module child* class is created. Thus, the simulation of the 8-bit counter can be carried out only with the call of *Step* function which defined in the ArchHDL library after the creation of *TestTop* module.

Fig. 2.6 is an example of a test bench for the 8-bit counter in Verilog HDL. ArchHDL is able to write a test bench in similar description in Verilog HDL. The major difference between the test bench in ArchHDL and Verilog HDL is the description of clock generation which is denoted in the line 14 of Fig. 2.6. Otherwise there is no significant difference in the description in ArchHDL and Verilog HDL.

2.4 Implementation of ArchHDL

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

Fig. 2.7 shows the definition of *RegisterInterface* class, *ModuleInterface* class, *WireInterface* class, *Singleton* class and Step function.

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

As member variables, *Singleton* class has three dynamic arrays which keep pointers of *Module* class, *wire* class and *reg* class (denoted in the line from 18 to 20). When the instance of a *Module child* class, a *wire* class or a *reg* class is created, the pointer to its class 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 26 to 34).

The *Step* function is the function to do one cycle simulation of implemented hardware. In the *Step* function, the *Init* function and the *Exec* function in *Singleton* class are called. The multicycle simulation can be carried by repeated call of the *Step* function.

The *Init* function of *Singleton* class (denoted in the line from 35 to 39) calls the *Init* function of all *Module child* class instance which kept in *Singleton* class. Note that, the *Init* function in *Singleton* class is only called at the first call of the *Step* function.

In the *Exec* function (denoted in the line from 40 to 47), at first *Always* functions of All *Module chile* class instance held in *Singleton* class are called (denoted in the line 42). Next, *Update* functions of All *reg* class instance held in *Singleton* class are called (denoted in the line 45).

A value at next cycle of all register is computed by calling the *Always* function. The value of registers are updated to the new value by calling the *Update* function. The process of the *Always* function and the *Update* function implements the non-blocking assignment of Verilog HDL.

```
1  class RegisterInterface {
2   public:
3    virtual void Update() = 0;
4  };
5
6  class ModuleInterface {
7   public:
8    virtual void Init() = 0;
9    virtual void Always() = 0;
10 };
11
12 class WireInterface {};
13
14 namespace ArchHDL {
15
16 class Singleton {
17  private:
18   std::vector<RegisterInterface*> registers_;
19   std::vector<ModuleInterface*> modules_;
20   std::vector<WireInterface*> wires_;
21  public:
22   static Singleton& GetInstance(void) {
23     static Singleton singleton;
24     return singleton;
25   }
26   void AddRegister(RegisterInterface* ri) {
27     registers_.push_back(ri);
28   }
29   void AddModule(ModuleInterface* mi) {
30     modules_.push_back(mi);
31   }
32   void AddWire(WireInterface* wi) {
33     wires_.push_back(wi);
34   }
35   void Init() {
36     for (uint i = 0; i < modules_.size(); i++) {
37       modules_[i]->Init();
38     }
39   }
40   void Exec() {
41     for (uint i = 0; i < modules_.size(); i++) {
42       modules_[i]->Always();
43     }
44     for (uint i = 0; i < registers_.size(); i++) {
45       registers_[i]->Update();
46     }
47   }
48 };
49
50 void Step() {
51   static bool init = false;
52   if (!init) {
53     init = true;
54     ArchHDL::Singleton::GetInstance().Init();
55   }
56   ArchHDL::Singleton::GetInstance().Exec();
57 }
58
59 }  // namespace ArchHDL
```

Fig. 2.7 The source code of each interface class, *Singleton* class and *Step* function in the ArchHDL library.

```
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 };
```

Fig. 2.8 The source code of *reg* class in the ArchHDL library.

2.4.1 Definition of reg class

Fig. 2.8 shows the definition of *reg* class. This class is a template class which takes a data type to use in the class as the template argument. The *RegisterInterface* class is inherited as the interface class.

ArchHDL deals a register as a variable. Therefore, the *reg* class has two variables *curr_* and *next_* which data type is given by the template arguments. The value of *curr_* is a value at one cycle, and the value of *next_* is a value at the next cycle. A value is assigned to the variable *next_* by calling the *Always* function. The value of the variable *next_* is assigned to the variable *curr_* by calling the *Update* function

```
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 };
```

Fig. 2.9 The source code of *wire* class in the ArchHDL library.

which is a member method of *reg* class. In this way, the non-blocking assignment to the register is carried out.

To assign a value to the variable *next_* in the *reg* class object, <<= operator is used. We redefine the <<= operator using operator overload. The value assigned to the *reg* class object by the <<= operator is stored to the variable *next_*. At the same time of assignment, the *set_* flag is set.

After calling the *Always* functions of all *Module* class instance, the *Update* functions of all *reg* class instance are called. Thus, the 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 and give the pointer of itself to *Singleton* class. The assignment to *reg* class object by = operator is also defined for the description of test bench or the setting of initial value. The value assigned to the *reg* class object by the = operator updates the variable *next_* immediately. The value of *reg* class is given by calling the object as a function.

2.4.2 Definition of wire class

Fig. 2.9 shows the definition of *wire* class. This class is a template class which takes a data type to use in the class as the template argument. The *WireInterface* class is

```
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 };
```

Fig. 2.10 The source code of *Module* class in the ArchHDL library.

inherited as the interface class.

ArchHDL deals wire as a function. Therefore, *wire* class has a variable *lambda_* to hold a lambda function. The data type of return value of this lambda function is the data type given by the template argument.

The assignment to the *wire* class object is limited to the assignment of the lambda function by disallowing the copy constructor and overloading the = operator. In this way, the *wire* class becomes to be the class which held a lambda function described in the *Init* function of *Module child* class.

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

2.4.3 Definition of Module class

Fig. 2.10 is the definition of *Module* class. This class inherits *ModuleInterface* class. We use *Module* class as the parent class to describe a module in ArchHDL.

The constructor of *Module* class gives the pointer of itself to *Singleton* class. The *Init* function and the *Always* function are declared as virtual functions in *Module-Interface*. Therefore, the empty *Init* function and the empty *Always* function are also defined in *Module* class.

Chapter 3

The Proposal of Optimization for ArchHDL

3.1 A Direction for High-speed Simulation

As a direction for high-speed simulation, I propose both optimizations in sequential programming and parallelization.

3.2 Proposed methods with sequential program

3.2.1 Removal of conditional branch for data update

In the implementation shown in Fig. 2.8, ArchHDL gives non-blocking assignment and blocking assignment as methods for updating the value of the *reg* class instance.

As to blocking assignment, the member variable *curr_* of the *reg* class instance must be assigned to the value if the blocking assignment is carried out to the *reg* class instance.

On the other hand, as to non-blocking assignment, the member variable *set_* of the *reg* class instance is copied to true and the member variable *next_* is assigned to the value if the non-blocking assignment is carried out to the *reg* class instance. Only when the member variable *set_* is true, the member variable *next_* is copied to the value of the member variable *curr_* in the *Update* method, which is a member method of *reg* class. It shows the value of the register is updated to the new value before the cycle in which the non-blocking assignment is carried out to the *reg* class instance.

In the implementation shown in Fig. 2.8, it is not necessary to perform the pro-

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

Fig. 3.1 The source code of *reg* class which is removed the conditional branch

cess of assignment if the values of the *next_* and *curr_* of the *reg* class instance are same. Therefore the unnecessary assignment is avoided by using the variable *set_*. This implementation may be effective if the *reg* class instance is rarely updated.

In my proposed method, the value of the *next_* is always assigned to the variable *curr_* every cycle. It eliminates the overhead of the *if* branch. This implementation is effective if the *reg* class instance is updated frequently.

Fig. 3.1 shows the implementation of the proposed method. It is removed the variable *set_* from the implementation shown in Fig. 2.8.

3.2.2 Storing register values to the continuous memory location

Fig. 3.2 shows the process of the *reg* class instance with ArchHDL. It denotes Fig. 2.7 in the line from 44 to 46. The *reg* class instances are painted gray and metadata of the class, the variable *next_* and the variable *curr_* are represented from the left. The

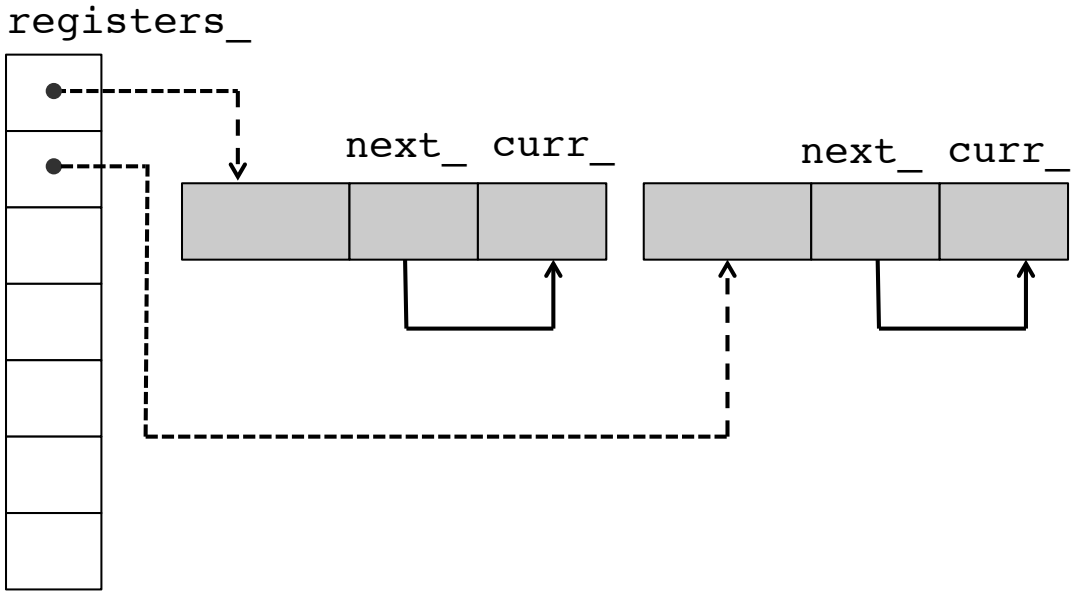

Fig. 3.2 The process of the *reg* class instance with ArchHDL

big frame on the left denotes *registers_* using `std::vector` in the line 18 of Fig. 2.7. Solid arrows represent a copy. Dotted arrows represent a pointer reference.

To simulate the non-blocking assignment in ArchHDL, it follows the values of the *registers_* and obtains a pointer to the *reg* class instances and the *Update* method of the *reg* class instances is called.

If I implement the removal of the conditional branch for data update, which is shown in Section 3.2.1, the value of the *next_* is always assigned to the variable *curr_* every cycle in *reg::Update* method.

The two overheads of this assignment and function call make the speed of the ArchHDL slow down.

Fig. 3.3 shows the proposed method. Fig. 3.3 shows the values of the *reg* class instances stored as an array. It is changed into holding each pointer that all of the *reg* class instances have the value of the next cycle and the value of the current cycle in the proposed method. The *reg* class instances are painted gray and metadata of the class, the `&next_` and the `&curr_` are represented from the left. The `&next_` and the `&curr_` are pointers of the value of *next_* and *curr_*. The two big frames on the bottom denote the arrays gathered the value of *next_* and *curr_*. They are named *next collections* and *curr collections* here. Solid arrows represent an assignment. Dotted arrows represent a pointer reference.

It is necessary to examine the address to which the *reg* class instances are al-

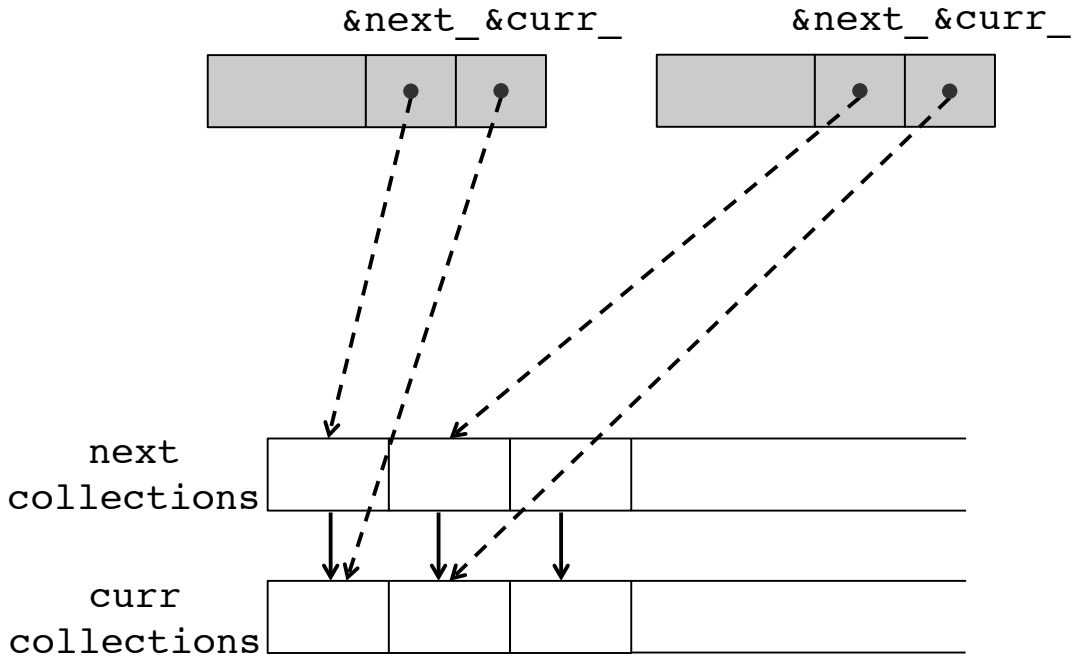

Fig. 3.3 The values of the *reg* class instances stored as an array

located in the implementation of original ArchHDL before the process of assigning the value of the *next_* to the variable *curr_* is executed before the next cycle. But if the implementation of the *reg* class is the removal of the conditional branch for data update, *Update* method is a simple assignment as shown in Fig. 3.3. Memory access can be carried out continuously because the memory allocation of the variable *next_* and *curr_* is continuous. The memory allocation is discrete in the implementation of original ArchHDL. The overhead of a function call is eliminated because it is unnecessary to call the *Update* method. In the above reasoning, this implementation is effective.

As to the implementation of the proposed method, two large arrays of the type of `unsigned int` are allocated as the *next collections* and the *curr collections*. The constructor of the *reg* class allocates each region to the *next collections* and the *curr collections* depending on the type of the template arguments. The allocated region is multiples of 4 bytes to speed up reference. The `&next_` and the `&curr_` are the addresses of the variable *next_* and *curr_*. The `reg::Update` method is not called but the next collections are copied to the curr collections.

```
1  void Exec() {
2  #pragma omp parallel num_threads(8)
3    {
4  #pragma omp for
5      for (uint i = 0; i < modules_.size(); i++) {
6        modules_[i]->Always();
7      }
8  #pragma omp for
9      for (uint i = 0; i < registers_.size(); i++) {
10       registers_[i]->Update();
11     }
12   }
13 }
```

Fig. 3.4 The source code is parallelized in the *for* statements of *Exec* method with OpenMP in 8 threads.

3.3 The parallelization of the execution of multiple instances

I have proposed the methods with sequential programming. I propose the parallelization of the execution of multiple instances in this section.

As shown from 40 to 47 lines in Fig. 2.7, each of the *Module* and *reg* class instances calls *Module::Always* and *reg::Update* method every cycle.

Module::Always method can be described freely by the user in the line from 41 to 43 in Fig. 2.7. Therefore there is no guarantee that *Module::Always* method can be executed separately for each of the *Module* class instances. However, it is assumed that it can execute independently in this thesis. Its proof is the research task from now on. It is possible to parallelize the execution of the *Module::Always* method.

Fig. 3.2 shows the update of the register by solid arrows. It is possible to execute independently for each instance. It is possible to parallelize the execution of the *reg::Update* method.

In my proposed method, I parallelize *Module::Always* and *reg::Update* method in the line from 41 to 46 of Fig. 2.7. I use OpenMP [6] to parallelize it.

Fig. 3.4 shows that the source code is parallelized in the *for* statements of *Exec* method with OpenMP in 8 threads. Fig. 3.4 shows parallelized code of the line from 40 to 47 in Fig. 2.7. The number of threads can be changed by the environment. The line 2 of Fig. 3.4 shows parallelization in 8 threads. The line 4 and the line 8 are

OpenMP directives to parallelize the execution of the *for* statements.

Generally, it is important for parallelization to assign the tasks equally to each thread. Several methods such as *static* to determine statically and *dynamic* to determine dynamic exist as a way to balance the burden of the *for* statement in OpenMP. If you specify the *dynamic* as a way to schedule, it has the advantage that the tasks are assigned to each thread almost equally. But It has the disadvantage of large overhead. If each of the tasks of each register and each module is not very different from the hardware description with ArchHDL, it is almost effective to specify *static*.

The chunk size can be specified as the option of the OpenMP. If you specify the *static* as a way to schedule and no chunk size, the chunk size is similar to a value dividing the number of loop iterations by the number of threads.

I specify the *static* as a way to schedule and no chunk size in the evaluation of this paper. It is the default setting.

Chapter 4

Evaluation

In this chapter, I evaluate the elapsed time of logic simulation of ArchHDL and the elapsed time is compared with that of logic simulation with Icarus Verilog, NC-Verilog and VCS.

Table 4.1 shows the simulation environment. I use the two computers of the same specification for the evaluation. One computer is used to evaluate the simulation time with Icarus Verilog and ArchHDL. The other is used to evaluate that with NC-Verilog and VCS. The two computers are the same specification for such as hardware, CPU, memory and so on. However, I use different operating systems due to the limitations of the software.

I describe the reason for using different operating systems. NC-Verilog and VCS support only RPM-based Linux distributions. I use CentOS 5.9, which is a RPM-based Linux distribution for this evaluation. However, the version of GCC includes CentOS 5.9 is 4.1.2. Section 2.1 shows that the version of GCC requires 4.5 or greater for using the lambda function in ArchHDL. I use the Ubuntu 12.04 because the version of GCC is 4.6.3. I use `-O2` as optimization option of GCC. Icarus Verilog can run on both computers. I use it on Ubuntu 12.04 in this evaluation. The version of Icarus Verilog on Ubuntu 12.04 is 0.9.5. The version of NC-Verilog is 06.20-s004. The version of VCS is vcsC-2009.06.

Because the number of cores in CPU is 4 and each core executes 2 threads simultaneously, I evaluate the parallelization with OpenMP in 8 threads.

In the evaluation, I use two micro benchmarks and the stencil-computation circuit [7] as a realistic hardware. I have written hardware description for ArchHDL and Verilog HDL by myself and verified that the outputs of both the hardware description are same.

I describe the name of the labels, which are used in the evaluation. Original

Table. 4.1 The simulation environment

	Icarus Verilog, ArchHDL	NC-Verilog, VCS
OS	Ubuntu 12.04	CentOS 5.9
CPU	Core i7-3770K 3.50GHz	Core i7-3770K 3.50GHz
Memory	16 GB	16 GB

```
1 unsigned int xor() {
2   static unsigned int y = 2463534242;
3   y ^= (y << 13);
4   y ^= (y >> 17);
5   return (y ^= (y << 5));
6 }
```

Fig. 4.1 The algorithm of random number generation based on XORSHIFT RNG on C language

ArchHDL is named **ArchHDL**. ArchHDL applied the first optimization is named **NO SET**, which is described in Section 3.2.1. ArchHDL applied the second optimization is named **MEM MAP**, which is described in Section 3.2.2. ArchHDL applied the third optimization is named **PARA**, which is described in Section 3.3.

4.1 Evaluation by Micro Benchmark

I evaluate the simulation time with the counter circuits and the random number generator circuits by XORSHIFT RNG (Random Number generator) as micro benchmarks.

The counter circuit is a circuit that adds 1 per cycle as shown in Fig. 2.3. I can specify the number of the counters in order to increase the scale of hardware. The circuit by XORSHIFT RNG is implemented as the random number generator based on XORSHIFT RNG by a hardware description. XORSHIFT RNG uses only the *exclusive or* and a *bit shift* to generate the random numbers. Fig. 4.1 shows the algorithm of random number generator based on XORSHIFT RNG in C language.

Fig. 4.2 shows the speed up ratio normalized by Icarus Verilog compared with the elapsed time of 4096 of the counter circuit. The vertical axis is the speed up ratio normalized by Icarus Verilog.

The simulation time of ArchHDL which includes **ArchHDL**, **NO SET**, **MEM**

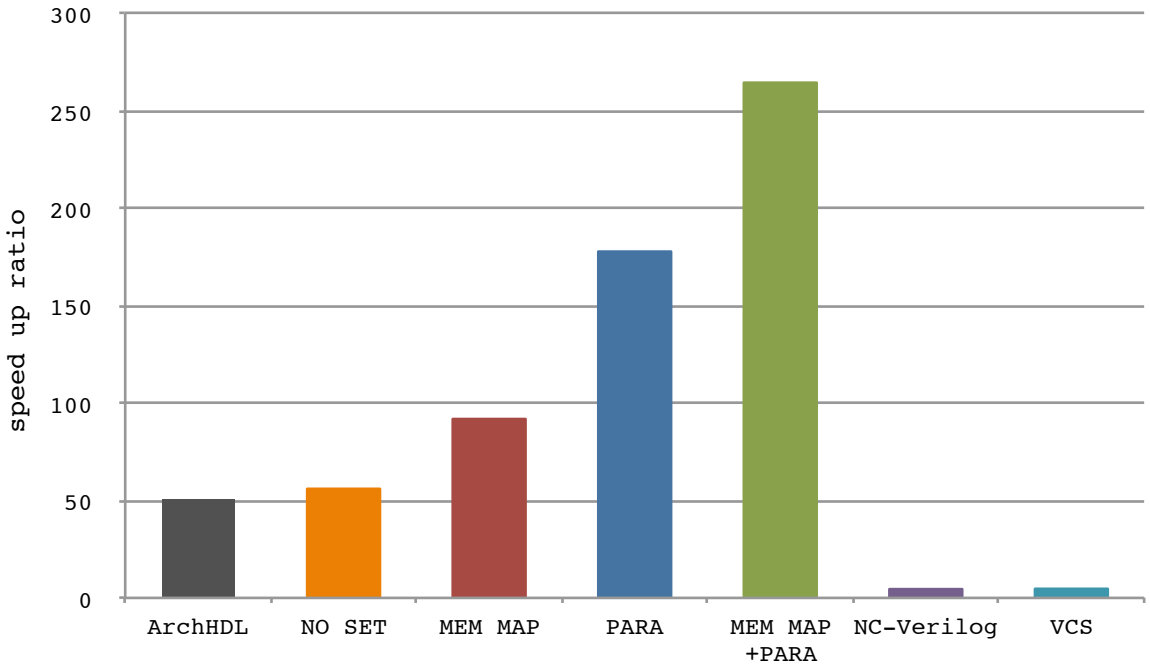

Fig. 4.2 Speed up ratio normalized by Icarus Verilog compared with the elapsed time of 4096 of the counter circuit

MAP, **PARA** and **MEM MAP + PARA** is much faster than that of NC-Verilog and VCS. The evaluation result shows that the elapsed time of **MEM MAP + PARA** is 58.8 times faster than that of NC-Verilog and 56.7 times faster than that of VCS.

The proposed methods in this paper are effective as compared with the original ArchHDL. The elapsed time of **MEM MAP + PARA** is 5.23 times faster than that of the original ArchHDL.

Fig. 4.3 shows the speed up ratio compared with the elapsed time of the counter circuits with Icarus Verilog in ArchHDL applying the proposed methods. The vertical axis is the speed up ratio normalized by Icarus Verilog. The horizontal axis is the number of counters.

The speed up ratio compared to Icarus Verilog is almost unchanged if the number of the counters is changed because **MEM MAP** is executed in sequential program. The elapsed time of **PARA** and **MEM MAP + PARA** which are executed in parallel is faster than that of **MEM MAP** if the number of counters is 1024 or more. The proposed methods with sequential program are effective in parallelized version because the simulation time of **MEM MAP + PARA** is always faster than that of **PARA**. Because the number of counters can be regarded as the scale of hardware,

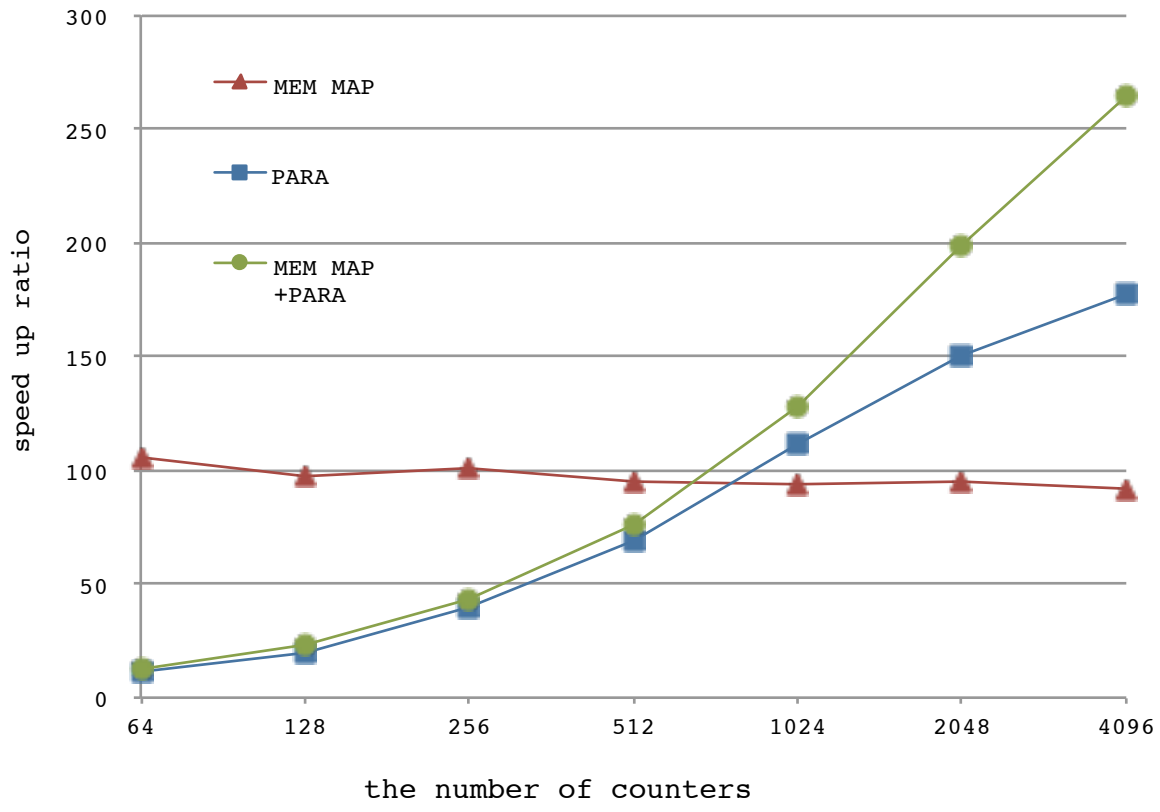

Fig. 4.3 Speed up ratio compared with the elapsed time of the counter circuits with Icarus Verilog in ArchHDL applying the proposed methods

the parallelization is effective if the hardware is large-scale.

Fig. 4.4 shows speed up ratio normalized by Icarus Verilog compared with the elapsed time of 512 of the random number generator circuits by XORSHIFT RNG. The number of trials is about 524 thousand times. This circuit includes the 512 random number generators with different initial values.

The simulation time of ArchHDL is much faster than that of NC-Verilog and VCS. The simulation time of **MEM MAP + PARA** is 32.2 times faster than that of NC-Verilog and 11.3 times faster than that of VCS.

The proposed methods in this paper are effective as compared with the original ArchHDL. The simulation time of **MEM MAP + PARA** is 2.78 times faster than that of the original ArchHDL.

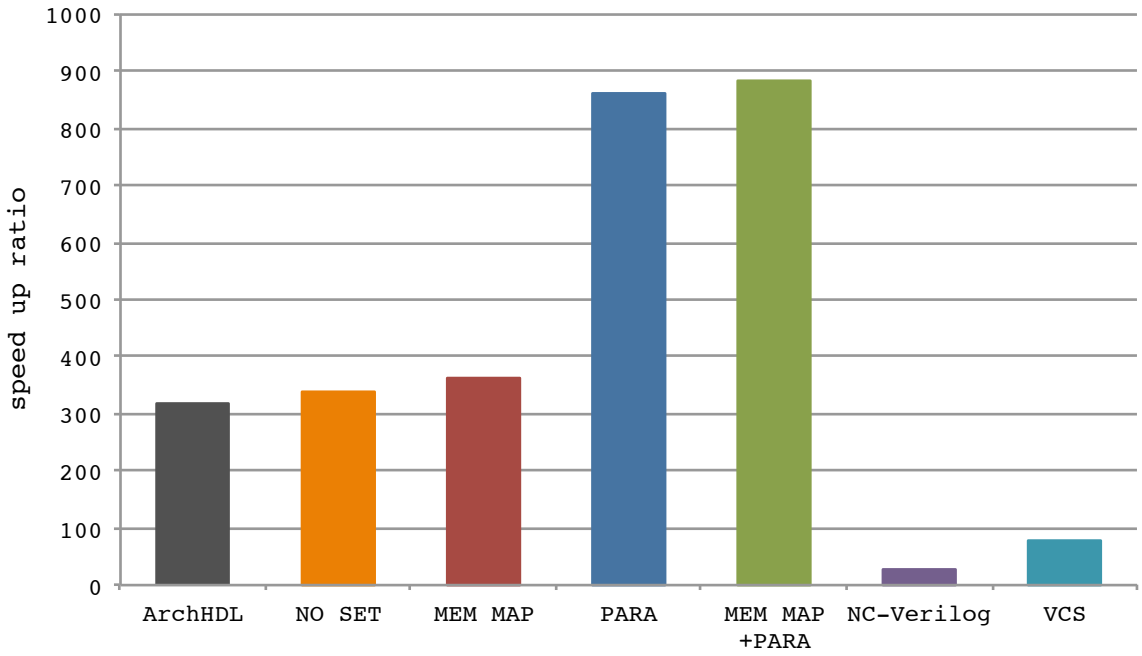

Fig. 4.4 Speed up ratio normalized by Icarus Verilog compared with the elapsed time of 512 of the random number generator circuits by XORSHIFT RNG

4.2 Evaluation by Stencil-Computation Circuit

Fig. 4.5 shows speed up ratio normalized by Icarus Verilog compared with the elapsed time of a stencil-computation circuit. The speed up ratio as elapsed time in Icarus Verilog is 1 is denoted in the vertical axis.

The simulation time of the original ArchHDL is faster than that of NC-Verilog. The simulation time of the original ArchHDL and **NO SET** is not faster than that of VCS. But the simulation time of **MEM MAP + PARA** is 1.83 times faster than that of VCS.

Update method is called 325,469,175 times in a stencil-computation circuit. The number of the value of the reg non-updated is 5,145,760 times. That is, the number of the value of the reg non-updated is only 1.58% of the entire *Update* method call. Therefore the simulation time of **NO SET** is faster than that of the original ArchHDL. The simulation time of **MEM MAP** is 1.31 times faster than that of the original ArchHDL.

The parallelization is effective because this hardware with ArchHDL has the 133

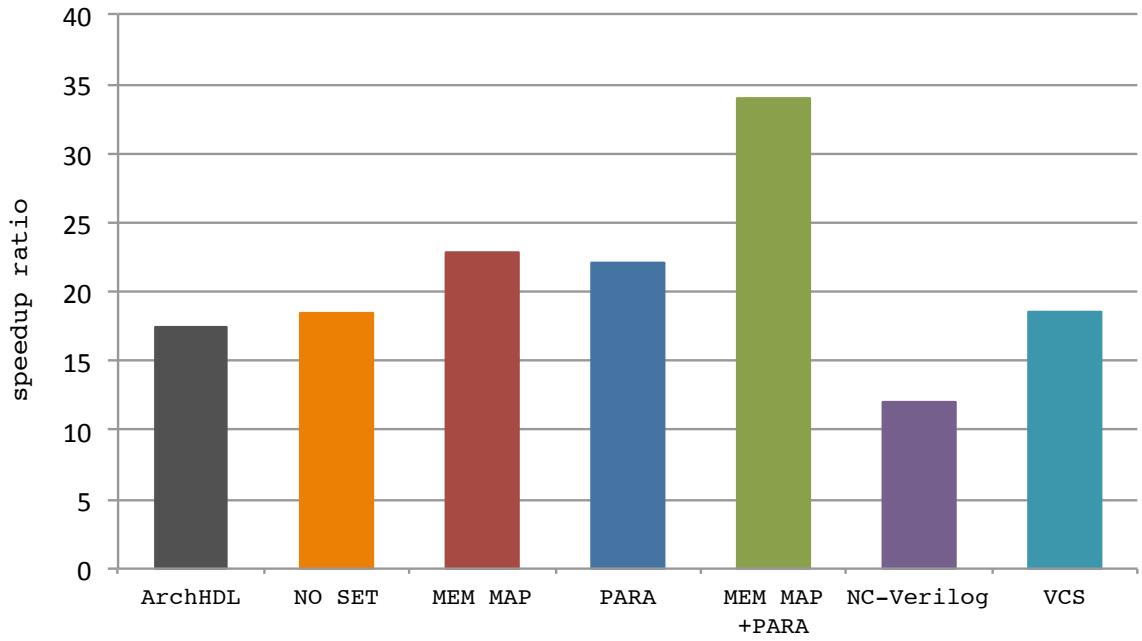

Fig. 4.5 Speed up ratio normalized by Icarus Verilog compared with the elapsed time of a stencil-computation circuit

Module class instances and the 991 *reg* class instances. The simulation time of **MEM MAP + PARA** is 1.95 times faster than that of the original ArchHDL.

Chapter 5

Conclusion

I propose, implement and evaluate the methods for high-speed simulation of ArchHDL, which is proposed as a new language for hardware RTL modeling. ArchHDL treats registers as variables and wires as functions, which realizes an RTL modeling on C++.

The three methods I propose and implement in this thesis are as follows: (1) removal of the conditional branch for data update, (2) storing register values to the continuous memory location and (3) the parallelization of the execution of multiple instances.

In the proposed methods, I compare the elapsed time of ArchHDL with that of Icarus Verilog, NC-Verilog and VCS and evaluate the simulation time by using the hardware descriptions of 4096 of the counters circuit and a random number generator circuits by XORSHIFT RNG as a micro benchmark and a stencil-computation circuit as a realistic hardware.

With 4096 of the counters circuit, the elapsed time of ArchHDL with the proposed methods is 58.8 times faster than that of NC-Verilog and 56.7 times faster than that of VCS. With the random number generator circuits by XORSHIFT RNG, it is 32.2 times faster than that of NC-Verilog and 11.3 times faster than that of VCS. With a stencil-computation circuit, it is 2.82 times faster than that of NC-Verilog and 1.83 times faster than that of VCS.

With 4096 of the counters circuit, the elapsed time of ArchHDL with the proposed methods is 5.23 times faster than that of the original ArchHDL. With the random number generator circuits by XORSHIFT RNG, it is 2.78 times faster. With a stencil-computation circuit, it is 1.95 times faster.

These results show that ArchHDL applied the three proposed methods can perform faster than VCS in all evaluations in this thesis.

Acknowledgment

I would like to express the deepest appreciation to Associate Prof. Kenji KISE. He has been my supervisor. His constant support, guidance, and encouragement have been essential for me to complete my thesis. I also would like to thank all the members at Kise Laboratory. I would particularly like to thank Mr. Shimpei SATO. He proposed ArchHDL and gived insightful comments and suggestions. Discussions with Mr. Shinya TAKAMAEDA-YAMAZAKI and Mr. Ryosuke SASAKAWA have been insightful. I would like to thank them.

I would like to thank Ms. Yukiko ASOH. She corrected the English written by me.

I would like to thank Mr. Masaru IRITANI. I received generous support from him.

I would like to thank Mr. Shintaro SANO. He had made a significant contribution to the development of ArchHDL.

Bibliography

[1] : IEEE Standard Verilog Hardware Description Language, *IEEE Std 1364-2001*, pp. 1–856 (2001).

[2] : IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)*, pp. 1–112 (2004).

[3] 佐藤真平，吉瀬謙二：C++ をベースとする新しいハードウェア記述の検討，情報処理学会研究報告. 計算機アーキテクチャ研究会報告 (ARC-205)，pp. 1–7 (2013).

[4] : Icarus Verilog. http://iverilog.icarus.com.

[5] : Synopsys VCS. http://www.synopsys.com/VCS.

[6] : OpenMP. http://openmp.org.

[7] 小林諒平，高前田（山崎）伸也，吉瀬謙二：多数の小容量 FPGA を用いたスケーラブルなステンシル計算機の開発，先進的計算基盤システムシンポジウム論文集，pp. 179–187 (2013).