ArchHDL: A Novel Hardware RTL Design Environment in C++

Shimpei Sato[1](✉) and Kenji Kise[2]

[1] Japan Advanced Institute of Science and Technology, Nomi, Ishikawa, Japan
[2] Tokyo Institute of Technology, Tokyo, Japan
shimpei.sato@jaist.ac.jp, kise@cs.titech.ac.jp

Abstract. LSIs are designed in four stages including architectural design, logic design, circuit design, and physical design. In the architectural design and the logic design, designers describe a hardware in RTL. However, they generally use different languages. Typically a general purpose programming language such as C or C++ and a hardware description language such as Verilog HDL or VHDL are used in the architectural design and the logic design, respectively. In this paper, we propose a new hardware description environment for the architectural design and logic design which aims to describe and verify a hardware in one language. The environment consists of (1) a new 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. 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 implements non-blocking assignment in C++. Using these features, designers are able to write a hardware in a Verilog HDL-like style. The source code of ArchHDL is able to convert to Verilog HDL by the translation tool and is able to synthesize for an FPGA or an ASIC. We implemented a many-core processor in ArchHDL. The simulation speed for the processor by ArchHDL achieves about 4.5 times faster than the simulation speed by Synopsys VCS. We also convert the code to Verilog HDL and estimated the hardware resources on an FPGA. To implement the 48-node many-core processor, it needs 71 % of entire resources of Virtex-7.

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. For these RTL modeling and simulation, some hardware description languages such as Verilog HDL and VHDL are often used.

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

K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 53–64, 2015.
DOI: 10.1007/978-3-319-16214-0_5

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.

We have proposed *ArchHDL*[11] as a new hardware description language for RTL modeling and high-speed architectural simulation. In this paper, we propose a new hardware description environment for architectural design and logic design which aims to write and verify a hardware in one language. 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.

ArchHDL is a hardware description language for hardware RTL modeling based on C++. The key features of this language are that (1) a combinational circuit description is handled as a function call and (2) non-blocking assignment support in C++ by the ArchHDL library. The goal of the proposed environment is to attain following three points.

– Easy hardware modeling in RTL
– High-speed simulation compared to Verilog HDL simulation

In the previous work, we found that RTL simulation speed using ArchHDL is much faster than a Verilog HDL simulation using Icarus Verilog[6] which is a well known free software. However, the simulation was performed with a simple hardware like 8-bit counter circuit, and it was not a practical evaluation to confirm the usefulness of ArchHDL. In this paper, we implement a many-core processor as a practical hardware in ArchHDL and evaluated the simulation speed. The source code of ArchHDL is transformed into Verilog HDL by the translation tool and the Verilog HDL code is synthesized for an FPGA. We convert the many-core processor code to Verilog HDL using the translation tool and measure the hardware resources on an FPGA.

2 A New Hardware Description Environment

We propose a new hardware design environment which consists of a new hardware description language called ArchHDL[11] and source code translation tool from ArchHDL to Verilog HDL. In this section, the hardware description in ArchHDL and development of the translation tool are delivered.

2.1 Concept of ArchHDL

ArchHDL is a hardware description language based on C++. It provides a C++ library to describe hardware in RTL. The library includes definitions of *Module* class, *reg* class, *wire* class, and functions for simulation.

The key features of this language are: (1) implementing combinational circuits as functions and (2) supporting non-blocking assignment to registers. 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

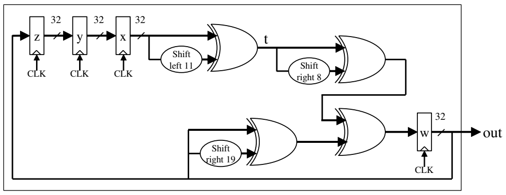
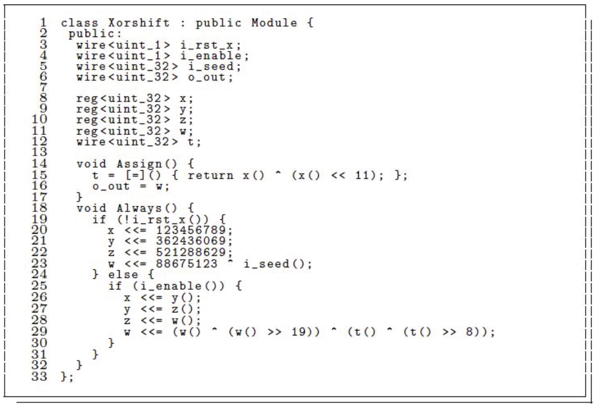


Fig. 1. A block diagram of sample circuit used for explanation of hardware description in ArchHDL. (Xorshift random value generator)



```
class Xorshift : public Module {
 public:
  wire<uint_1> i_rst_x;
  wire<uint_1> i_enable;
  wire<uint_32> i_seed;
  wire<uint_32> o_out;

  reg<uint_32> x;
  reg<uint_32> y;
  reg<uint_32> z;
  reg<uint_32> w;
  wire<uint_32> t;

  void Assign() {
    t = [=]() { return x() ^ (x() << 11); };
    o_out = w;
  }
  void Always() {
    if (!i_rst_x()) {
      x <<= 123456789;
      y <<= 362436069;
      z <<= 521288629;
      w <<= 88675123 ^ i_seed();
    } else {
      if (i_enable()) {
        x <<= y();
        y <<= z();
        z <<= w();
        w <<= (w() ^ (w() >> 19)) ^ (t() ^ (t() >> 8));
      }
    }
  }
};
```

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

assignment, which is generally supported in hardware description languages such as Verilog HDL or VHDL, is not supported in general purpose programming languages. ArchHDL implements non-blocking assignment by the library in C++.

ArchHDL realizes an RTL hardware description with Verilog HDL-like style by these features and defined classes in the library. A source code described in ArchHDL is able to compile with general C++ compilers, then the simulation of the hardware is delivered by the execution of the binary file. The simulation speed is faster than the simulation using Verilog HDL simulator.

2.2 Hardware RTL Modeling in ArchHDL

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

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

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

An instance of *wire* class and *reg* class can be regarded as a *wire* and *reg* in Verilog HDL respectively. These classes are implemented as a template class in the ArchHDL library. Therefore, users must specify the data type into the angled bracket when they declare an instance of *wire* or *reg*. In Fig. 2, *uint_1* and *uint_32* are used as data types for *wire* and *reg*. The number at the end of these data types represents the bit width of the data, and is intended to simplify the analysis by the source code translation tool. These data types are defined in the library. However, these are actually implemented as an alias of *unsigned int*. 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 and *reg* class instance. The *wire* class and the *reg* class are implemented as functional objects in the library, so a value of these instance can be referred by a function call of the class instance.

A module description in ArchHDL does not employ ports. 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 the Fig. 2, an instance name starting from "i_" is an input port and an instance name starting from “o_” is an output port. These rules are also applied to an instance which declared using array.

Combinational circuits are defined by assigning a functional object to a *wire* class instance. All of assignment statements to instances are written in *Assign* function as denoted from the 14th line to the 17th line in the Fig. 2. In the case of this example, combinational circuits which are able to describe in the Verilog HDL assign statement 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 of a *reg* class instance. Statements of non-blocking assignment are described using “`<<=`” operator, and they are written in *Always* function as indicated from the 18th line to the 32nd line in the Fig. 2. The *Always* function is equivalent to “always @(posedge CLK)“ in Verilog HDL.

2.3 Testbench in ArchHDL

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

PortConnect function is used to connect ports of Xorshift module as denoted from the 15th line to the 20th line. The role of this function is same to *Assign* function mentioned above. However, it is necessary to describe *PortConnect function* and *Assign* function separately to simplify the analysis of the translation tool. From the 24th line to the 28th line, register initializations are described in *Initial* function. This function is equivalent to the initial block in Verilog HDL.

2.4 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. Designers are able to declare modules, registers, or wires using an array in ArchHDL. 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 a flexible description to the testbench. 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 the testbench description is equal to typical software simulators. Furthermore, the simulation speed is faster than Verilog HDL simulation which described in 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) the described hardware may use only one clock signal, (2) the assignment of value to a register is done only in a positive edge of the clock and (3) the designers describe registers and wires using C++ integer type like 32-bit int or 64-bit int.

ArchHDL supports a hardware which allows assigning a value to a register in a positive edge of a single clock. Therefore it does not support to use multiple clocks and to assign a value in a negative edge of a clock. Although SFL[9] has similar constraint, it was used for variety of hardware descriptions[5,7].

ArchHDL uses C++ integer type like 32-bit int or 64-bit int for the 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 the bit selection operation and the bit concatenation operation which are supported in Verilog HDL.

The ArchHDL library is implemented with about 200 lines of 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 with the progress of this work.

2.5 Translation Tool from ArchHDL to Verilog HDL

We are developing a code translator from ArchHDL to Verilog HDL. It is able to automatically generate a Verilog HDL code from ArchHDL.

As shown in some examples of hardware description in ArchHDL, its description style becomes Verilog HDL-like description style by using classes such as reg class or wire class provided by the library. Especially, designers are expected to describe the same statement about an assignment and an arithmetic expression in ArchHDL and Verilog HDL. Thus, the code translation from ArchHDL to Verilog HDL can be delivered without any optimization.

Fig. 4 shows the translation flow from ArchHDL code to Verilog HDL code. The tool receives ArchHDL code as an input and outputs the Verilog HDL. The translation is delivered as follows: (1) code scanning, (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.

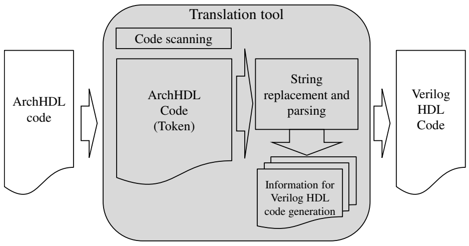


Fig. 4. Translation flow from ArchHDL code to Verilog HDL code

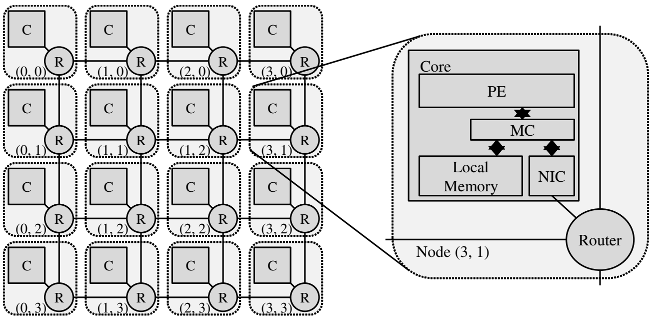


Fig. 5. M-Core Architecture used as a practical hardware for evaluation

The main restrictions of description in ArchHDL which can be converted to Verilog HDL are as follows: (1) using only reg class or 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 is not supported in some Verilog HDL simulator. We think that we can describe a practical hardware sufficiently in ArchHDL under such restrictions.

3 Experimental Results

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

3.1 A Sample Hardware for Practical Evaluation

We implemented M-Core Architecture[12], a many-core processor employs scratchpad memories, as a sample hardware for evaluation. Fig. 5 shows the M-Core Architecture. Each node of the 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[10]. Each node is connected to the mesh network via the router. The router architecture is the conventional Input-buffered Virtual Channel router[3] with five-stages 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 send from a PE to its NIC via memory-mapped IO. NIC reads data from the local memory using these information. After that, packets are generated and injected into the network. When a packet arrives a node, the information of the packet is used to write the received data to the local memory.

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. 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 and Intel Compiler are used. The compiler optimization option is -Ofast for both compilers. The detailed computer environment for simulation is as follows

- CPU: Intel Xeon E5-2687W
- Memory: 32 GB
- OS: Ubuntu 13.04 x86_64
- GCC (g++): version 4.7.3, optimization -Ofast
- Intel Compiler (ICPC): version 14.0.1 optimization -Ofast
- Synopsys VCS: version H-2013.06

In the simulation, various number of cores (from 2×2 to 10×10) of the many-cores processor are used. 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.

ArchHDL simulation can be parallelized using OpenMP. This parallelization is supported by the ArchHDL library and users do not need to change their source code. The parallelized simulation accuracy is same as that of before parallelization. For the parallelized simulation, we use a computer which has the 8 physical cores (16 logical cores using SMT) CPU.

Fig. 6 illustrates the simulation results in case of using 1 thread. The X-axis indicates the node counts 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

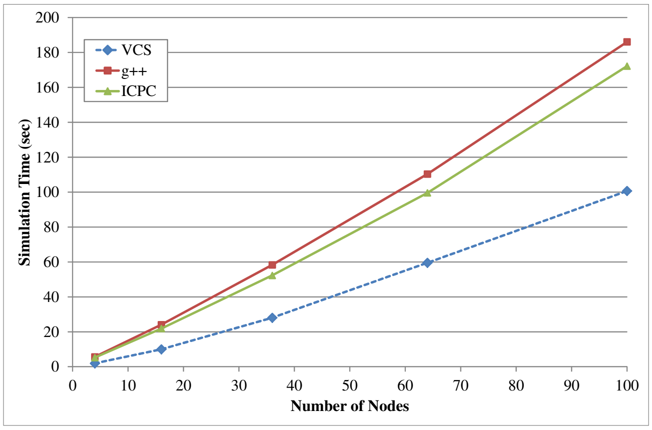


Fig. 6. The simulation time comparison using just single-threaded

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).

Fig. 7 shows the simulation results in case of parallelized simulation (with 8-thread and 16-thread). "VCS" and "ICPC 1 thread" in Fig. 7 are same as ones illustrated in Fig. 6.

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.

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 1 shows the hardware resources of the processor which are delivered during place and route. The processor occupied 54,509 slices which is 71% of

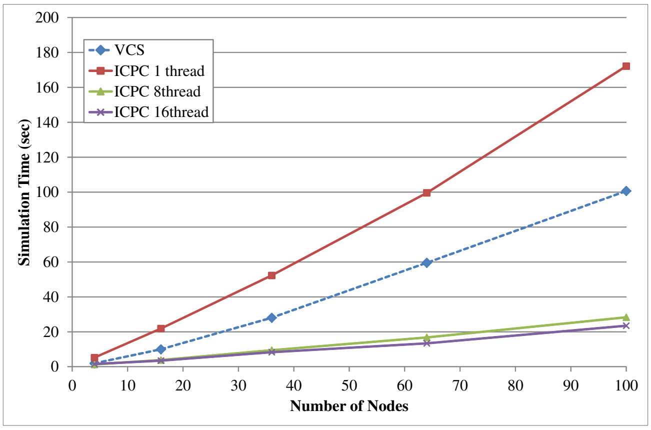


Fig. 7. The parallelized simulation time in ArchHDL. Note that the simulation time of VCS is single-thread.

Table 1. Hardware Resource of a 64-node many-core processor on Virtex-7

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

entire resources, and runs at 114.9 MHz according to the synthesis report. 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.

From the result of hardware resources, we found that the scale of the hardware generated by the converted Verilog HDL code by the translation tool is appropriate.

4 Related Works

Chisel[2], SystemC[1], and MyHDL[4] 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 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[8] 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.

SFL[9] 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, the development and maintenance of PARTHENON system had stopped.

5 Conclusion

In this paper, we propose a new hardware description environment for architectural design and logic design which aim to write and verify a hardware in one language. 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) Easy hardware modeling in RTL, (2) Realizing the environment which is able to verify both architectural design and logic design in one description, and (2) High-speed simulation compared to Verilog HDL simulation.

We evaluate our proposed hardware description environment in two aspects: (1) The simulation speed of 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 resource utilization of the 48-node many-core processor on Virtex-7 was 54,509 in occupied slices. From the result, we found that the scale of the hardware generated by the converted Verilog HDL code by the translation tool is appropriate.

As future works, we consider about that: (1) The detailed verification of the converted code from ArchHDL to Verilog HDL, (2) To develop a source code translation tool from Verilog HDL to ArchHDL to obtain the first RTL simulation, and (3) Implement the hardware described in ArchHDL into FPGAs and confirm its behavior.

References

1. IEEE standard for Standard SystemC Language Reference Manual. IEEE std. 1666–2011 (2011)
2. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: constructing hardware in a scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference, DAC 2012, pp. 1216–1225. ACM, New York (2012). http://doi.acm.org/10.1145/2228360.2228584
3. Dally, W., Towles, B.: Principles and Practices of Interconnection Networks. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science (2004)
4. Decaluwe, J.: Myhdl: a python-based hardware description language. Linux J. **2004**(127), 5 (2004). http://dl.acm.org/citation.cfm?id=1029015.1029020
5. Hayasaka, H., Haramiishi, H., Shimizu, N.: The design of pci bus interface. In: Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC 2003, pp. 579–580 (2003)
6. Icarus Verilog Web page: http://iverilog.icarus.com
7. Kon, C., Shimizu, N.: The design of an i8080a instruction compatible processor with extended memory address. In: Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC 2003, pp. 571–572 (2003)
8. MyHDL Web page: http://www.myhdl.org/doku.php/performance
9. Parthenon Web page: http://www.kecl.ntt.co.jp/parthenon/
10. Patterson, D., Hennessy, J.: Computer Organization and Design: The Hardware/software Interface. Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann (2012)
11. Sato, S., Kise, K.: ArchHDL: a new hardware description language for high-speed architectural evaluation. In: Proceedings of IEEE 7th International Symposium on Embedded Multicore SoCs (MCSoC 2013), pp. 107–112, September 2013
12. Uehara, K., Sato, S., Miyoshi, T., Kise, K.: A study of an infrastructure for research and development of many-core processors. In: Proceedings of International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 414–419, December 2009