Nicarus: A Distributed Verilog Compiler

Jun Wang and Carl Tropper
School of Computer Science
McGill University
Montreal, Quebec, Canada
jwang90, carl@cs.mcgill.ca

Abstract

Software design tools, such as compilers and simulators, are widely used in the integrated circuits (IC) industry. As the circuit complexity grows, better and faster tools are required. One way to speed up a software system is to parallelize it and execute it on multiple CPUs. This idea is particularly appealing when it comes to the compilers for hardware description languages (HDL).

In this paper, we explore the parallelization of a Verilog compiler on a network of computers using Parallel Virtual Machine (PVM). Our design parallelizes the three most important and time-consuming phases of the compilation process while minimizing the communications overhead. Experimental results reveal that the algorithms used for parallelizing the compiler results in a speedup of the compilation process.

1. Motivation

A Hardware Description Language, such as Verilog [5][7][3], is an indispensable tool in the design and verification process of very large scale integrated (VLSI) circuits. Because of its power and flexibility in the modeling, simulation, synthesis and testing of VLSI systems, Verilog has gained widespread acceptance in the hardware design community.

As is the case with (imperative) programming languages such as C, designing hardware with HDLs involves compiling the source design into another form, called a netlist. The netlist is essentially an enumeration of all of the circuit components (e.g. AND,OR,XOR gates). The netlist is then simulated in order to verify the design. A typical design process consists of several iterations of this compilation-simulation cycle.

HDL source files are organized in a hierarchical and modular manner, and the resulting netlist is a network of primitive logical gates. This, to a certain extent, means that the compilation time of HDL sources is proportional to the size of the circuits.

With the complexity of today's integrated circuits and the never-ending challenge of fast design turn-around, it is highly desirable to have the HDL compiler run as fast as possible. In this paper, we present experimental work in parallelizing a Verilog compiler over a network of computers using the message-passing system PVM. Our approach parallelizes three of the most important and time-consuming phases of the compilation process and at the same time minimizes communication overhead. We intend to make use of this compiler in conjunction with a distributed simulator.

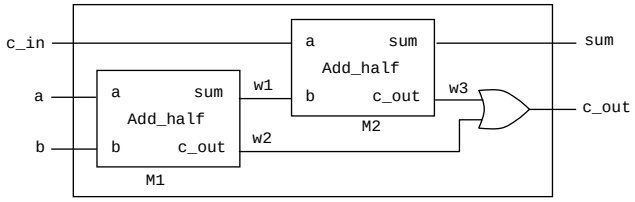
The rest of the paper is organized as follows. Section 2 gives a brief introduction to Verilog and a description of the Icarus Verilog compiler. Section 3 details the design and implementation of Nicarus, our distributed Verilog compiler. In section 4 we present some test results and our analysis. And finally section 5 contains our conclusions.

2. Background and related work

2.1. Verilog

Verilog has become the language of choice for a great number of IC designers. It was created by Cadence and became an IEEE standard in 1995 as *IEEE Std 1364-1995* which was later updated as *IEEE Std 1364-2001* [5].

Verilog provides a rich set of built-in primitives at different abstraction levels, such as logical gates, switches, and user-defined primitives. Primitives are interconnected to form modules, which are in turn interconnected to form a design. The description of the connectivity of primitives and modules is called a structural description. Verilog also supports behavioral descriptions which model the functionality, as opposed to the gate-level connectivity, of a design. Behavioral code is very much like the statements of an imperative programming language. The behavioral descrip-




```
module Add_half(sum, c_out, a, b);
    input a, b;
    output sum, c_out;

    xor  G1(sum, a, b);
    nand G2(c_out_bar, a, b);
    not  G3(c_out, c_out_bar);
endmodule

module Add_full(sum, c_out, a, b, c_in);
    input a, b, c_in;
    output c_out, sum;

    Add_half M1(w1, w2, a, b);
    Add_half M2(sum, w3, w1, c_in);
    or G(c_out, w2, w3);
endmodule

module test_adder(sum, c_out, a, b, c_in);
    input sum, c_out;
    output a, b, c_in;
    reg a, b, c_in;

    initial
        begin
            $monitor($time,,
               "a=%b b=%b c_in=%b c_out=%b sum=%b",
               a, b, c, c_in, c_out, sum);
            #10 a=0; b=0; c_in=0;
            #10 a=1; b=1; c_in=1;
            $finish;
        end
endmodule

module testBench;
    wire w1, w2, w3, w4, w5;

    Add_full   A(w1, w2, w3, w4, w5);
    test_adder T(w1, w2, w3, w4, w5);
endmodule
```

Figure 1. Verilog design of a full adder.

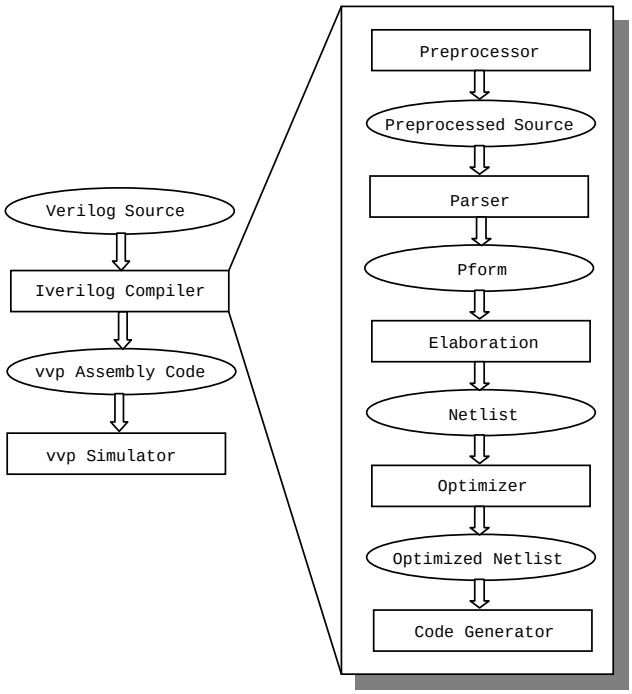


Figure 2. Structure of Icarus Verilog.

tion is processed by a synthesizer in order to create a gate-level implementation of the circuit.

Modules are the basic building blocks of a Verilog design. Each module consists of structural code or behavioral code, or a combination of structural and behavioral code. A module can instantiate other modules. The module instantiation hierarchy usually corresponds to the structure of the physical circuits.

Figure 1 shows an example design of a full adder [3], which will be used throughout the rest of the paper. A full adder is modeled by an `Add_full` module which has two instantiations of module `Add_half`, which models a half adder. The `test_adder` module consists of a behavior begin-end block that provides simulation stimuli to the full adder. The root module, `testBench`, consists of one instantiation of `Add_full` and one of `test_adder`.

2.2. Icarus verilog

Icarus Verilog [8][6] is a set of open-source Verilog development tools that includes a Verilog compiler, *iverilog*, and a logic simulator called the *vvp simulator*. Figure 2 shows the overall structure of Icarus Verilog on the left and the details of the iverilog compiler on the right.

As shown in Figure 2, the compilation process consists of the following five major phases.

- Preprocessing. This phase mainly performs file inclusion and macro substitution. The result can be written

to a file or sent through a pipe to the compiler proper.

- Parsing. The preprocessed source file is parsed and an internal representation of the source, called *pform* (parsed form), is generated.

- Elaboration. This phase transforms the hierarchical pform into a flattened netlist, which is an enumeration of all of the signals and logic gates of the circuit and their interconnections. This phase is carried out in three steps. First, a tree of naming scopes is created. Each module in the source file is a naming scope. Each module instantiation, user-defined function, user-defined task, begin-end block, and fork-join block, creates a sub-scope inside the enclosing module. Secondly, each scope creates all of the signals that it contains. Last, each scope creates all of the primitive gates and the connections between signals and the ports of all of the module instantiations are established.

- Optimization. This phase performs optimizations on the generated netlist, including null circuitry elimination, combinational reduction, and constant propagation.

- Code Generation. Depending on the target type specified in the command-line, one of five code generators is dynamically loaded and the final output is generated. The default output format is the vvp assembly.

3. Parallelizing the Icarus compiler

The intrinsic parallelism of VLSI circuits makes them an attractive target for the study of parallel computing. Our analysis of the Icarus compiler has shown that three of the major phases of the compilation process, i.e., parsing, elaboration, and code generation, can all be executed in parallel with relatively small overhead. When the circuit size reaches a certain point, the benefit of parallel execution will exceed the overhead and produce an overall speedup.

In order to parallelize the compiler, we use one workstation, known as the master, to partition the source file into a number of files such that each file contains only one module definition. These files are then distributed among several slave workstations and the compilation process is executed in parallel. The generated code is also passed to the master to create the final output file.

3.1. Parsing

Modules are the basic building blocks in a Verilog design. The Icarus compiler parses source files on a module by module basis. Since there are no cross-references of names across module boundaries in the parsing stage, it is obvious that parsing can be performed in parallel if the source file is partitioned along module boundaries.

```
partitioning_algorithm() {
    while(there are more modules in the source file) {
        create a file for the next module;
        create a slave to handle the module;
        if(all slaves have been created)
            break;
    }
    // At this point we have created all slaves, the
    // number of which is the smaller of the desired
    // number, which is a command-line argument, and
    // the number of modules in the source.
    while(not all modules have been parsed) {
        if(there are more modules in the source file)
            create a file for the next module;

        if(a slave asks for more modules AND
                                    there are more)
            assign the next module to the slave;
    }
}
```

Figure 3. Algorithm for partitioning source.

We use a technique known as *self-scheduling* [1] to split the source file along module boundaries and distribute the modules to the slaves. First, as soon as a module is split from the source, we spawn a slave to parse that module, until the desired number of slaves have been started, or all of the modules have been assigned, whichever comes first. When a slave finishes parsing one module, it will ask the master for more, and it will be given the next unassigned module until all the modules have been assigned. In other words, if there are M modules and N slaves ($M \geq N$), each of the N slaves is assigned one module, and the rest $M - N$ modules are assigned on a first-come-first-serve basis. In this way, the partitioning is done at the same time that the modules are being parsed and parsing is performed in parallel among the slaves. The partitioning algorithm is shown in Figure 3.

3.2. Elaboration

During parsing, each slave will report to the master the type of the module it has parsed, as well as the type of all of the module instantiations within the module. For example, with the circuit in Figure 1, the slave that parses the module `Add_full` will report to the master that it has parsed `Add_full`, and the module instantiates modules of the type `Add_half`. At the end of the parsing phase, the master will have all the information about the module instantiation relationships among the modules. The master then determines which modules are the root modules and starts the elaboration phase by sending each of the root modules an `ELAB` message.

Upon receiving an `ELAB` message, a slave creates a thread in order to handle the request. The thread follows the same three steps as in Icarus. First, it creates a naming

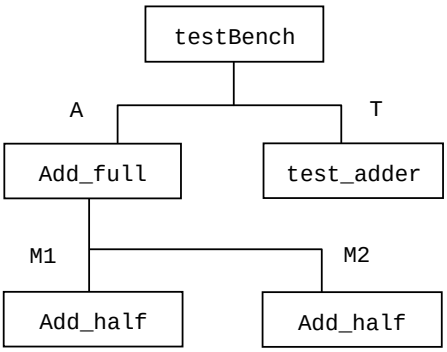


Figure 4. Scope tree of the full adder.

```
elaboration_algorithm() {
    // Upon receiving an ELAB message
    for each module instantiation whose module
          definition is on a different task {
        send an ELAB message to that task;
    }
    create naming scopes;
    create signals within scopes;
    create gates within scopes and make signal
          connections;
    send a REG_PORT message to the master to register
          the module port information;
    wait for all ELAB_END messages from remote tasks;
    send an ELAB_END message back to the requester, with
          signal drive information for all module ports;

    wait for EMIT message;
}
```

Figure 5. Elaboration algorithm.

scope for the module. Then it creates sub-scopes for all the module instantiations, user-defined functions, tasks, begin-end blocks, and fork-join blocks. For each module instantiation, if the corresponding module definition is owned by another task, the slave will send an ELAB message out before it handles local modules. This is done in order to maximize parallelism. Since each ELAB request corresponds to a module instantiation, the requests, hence the threads, form a tree that corresponds to the structural hierarchy of the circuit. The structure of the full adder in Figure 1 is shown in Figure 4. Each node in Figure 4 represents a naming scope. The tree of threads established via the ELAB messages is isomorphic to the scope tree in Figure 4.

After the scope and the sub-scopes are created, a thread then creates all of the signals within their scopes. The next step is to create all of the primitive gates and to establish connectivity among signals. For example, the signals `testBench.A.a` and `testBench.T.a` in Figure 1 are connected together. For a module instantiation that has its module definition on another task, the establishment of connectivity of its ports and outside signals will have to be deferred until after the remote task has finished the ELAB request. Note that during the elaboration process, slaves have no interactions other than sending ELAB requests to each other until the connectivity among modules needs to be established.

One important task related to the establishment of connectivity is the resolution of drive signal for each signal in the circuit. For example, a Verilog register is a drive whereas a net is not. When multiple signals are connected together, we need to determine which one is the drive for the juncture. In the Icarus compiler, this is done in the code generation phase. For our distributed compiler, since the netlist is distributed, the drive signals have to be resolved by collecting the information in a bottom-up manner and then disseminating the information top-down. In order to do this, a thread, after finishing elaborating a module, will send the information about the module ports, such as the number of ports, the width of each port, etc., to the master via an REG_PORT message. It then resolves all signals connected to the ports and sends the information to its requester, i.e., its parent node in the netlist, via an ELAB_END message. The parent node will repeat the process until the root node is reached, at which point, the information will be passed downwards. For example, if modules `test_adder` and `Adder_full` are assigned to different slaves, the two slaves don't know that the signals `testBench.A.a` and `testBench.T.a` are connected to each other. Only when the drive information of these two signals reaches the parent node, `testBench`, can it be determined that the two signals are both connected to `testBench.w3`, and the drive is in fact `testBench.T.a`. Since `testBench` is a root module, the results are final, and the information will be sent downwards so that the slaves handling `test_adder` and `Adder_full` will both know that the drive signal for `testBench.T.a` and `testBench.A.a` is `testBench.T.a`. The algorithm of the elaboration phase is shown in Figure 5.

3.3. Code generation

The code generation phase has a few restrictions. First, the final code is written to a single output file. Secondly, the syntax of the vvp assembly requires that code from different scopes not be interleaved, and code for a child scope not appear before the code for its parent scope. For example, in Figure 4, the scope A for the full adder has two child scopes `M1` and `M2`. In the output file, the code segments for `M1` and `M2` can appear in any order, but they must appear after the code segment for A.

Since all the code has to be written to a single output file, we make all the slaves send their code outputs to the master, which assembles the code in memory and eventually writes to the file. Experiments have shown that it is more efficient to write to a file in a centralized way. We maximize parallelism in the code generation phase by having all the

```
code_generation_algorithm() {
    // Upon receiving an EMIT message
    update signal drive information;
    compute the global ID for each remote child;
    send an EMIT message to each child node with
          the global ID for the child;
    start generating code and send the code to
          the master;
}
```

Figure 6. Code generation algorithm.

tasks perform code generation in parallel. Furthermore, the slaves also send the code output to the master in parallel as the code is being generated. This means the messages carrying the code output from different tasks can reach the master in any random order. In order to sort out the code outputs, we assign each node in the distributed scope tree a global identification number. The scheme to assign the numbers ensures that the number for any scope is smaller than the number for any of its child scopes. Therefore, when the master finally writes the code segments to the output file by the order of the global ID of the segments, we can be assured that the code segment of any scope will appear in the output file before the code segments of its child scopes.

As described above, at the end of the elaboration phase, a node waits for an `ELAB_END` message from each of its remote child nodes. When all the `ELAB_END` messages have been received, the node performs signal drive resolution and send an `ELAB_END` to its parent node, if any. Carried inside each `ELAB_END` message is the number of nodes in the subtree. This number is later used to assign a global ID to each node.

The root node sends its `ELAB_END` message to the master, which in turn sends an `EMIT` message with the global ID for the receiver.

Upon receiving an `EMIT` message, a node first updates its signal drive information. It then immediately sends an `EMIT` message to each of its child nodes with the updated drive information as well as the global ID for the child nodes. Again, this is done to maximize parallelism. After all the `EMIT` messages have been sent out, the node then starts generating code and sending the code to the master. The code generation algorithm is shown is Figure 6.

4. Experimental results

Our distributed Verilog compiler was tested on an Ethernet LAN comprised of a number of Pentium III computers running the FreeBSD operating system. The latency of PVM is measured at 175 μs for our testing environment. The Verilog source used was a reverse-engineered version [4] of the C1355 circuit from the ISCAS-85 benchmark set [2]. The C1355 is a 32-bit single-error correcting circuit with 546 gates, and the reverse-engineered design consists

Table 1. Compilation time.

No. of CPUs	1	2	3	4
Parsing (ms)	92	70	74	84
Elaboration (ms)	1948	1292	1233	1239
Emission (ms)	1916	1802	1728	1733
Total (s)	3.959	3.175	3.041	3.070

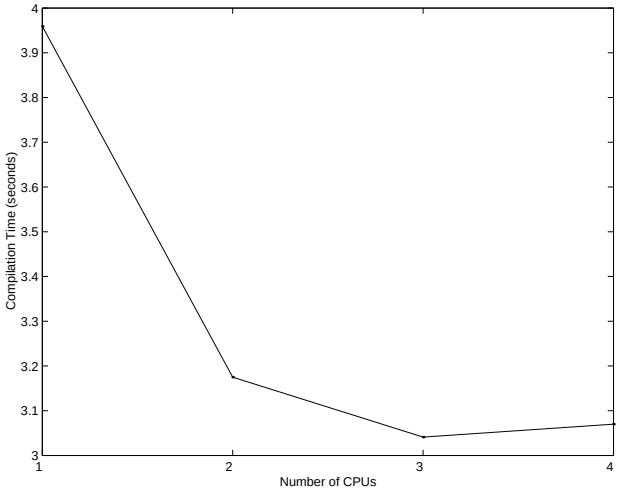


Figure 7. Compilation time vs. number of CPUs.

of four modules. Since all our timing measurements are the elapsed real time obtained through the UNIX system call *gettimeofday*, we replicate the circuit 50 times to create a bigger circuit in order to compensate for the inaccuracy of the measurements.

As the source design has four modules, we tested the compiler with up to 4 CPUs. Table 1 shows the detailed compilation time vs. the number of CPUs used. A chart for the total compilation time is shown in Figure 7.

To compensate for network fluctuations and PVM's round-robin mapping of tasks to CPUs, each data-point is the average of the 10 best results out of 12 measurements. The output file is about 45,000 lines and 5.2 MB in size.

As is shown in Table 1, parsing time is fairly insignificant in this case, accounting for only about 2.5% of the total compilation time. Because of disk I/O, code emission takes roughly the same amount of time as elaboration when 1 CPU was used. When multiple CPUs were used, we reduced the elaboration time by about 35%. However, we did not reduce the emission time very much. This is due to the fact that all the code has to be sent to the master for output to the file, even though the code segments are generated by the slaves in parallel, and apparently disk I/O is a dominant

Table 2. Speedup and number of messages.

No. of CPUs	2	3	4
Speedup	1.247	1.302	1.290
No. of Messages	3038	3445	3446

factor in this phase.

Among the four modules in the source design, two are fairly large and the other two are much smaller. When two CPUs were used, the partitioning algorithm happened to place the two big modules onto different CPUs. This contributes to the significant performance gain over the one-CPU case. It also explains why further performance gain was not as significant when more than two CPUs were used.

We define speedup $SU(n)$ to be the execution time T_1 required for one CPU divided by the execution time T_n when n CPUs are used, i.e. $SU(n) = T_1/T_n$. In our experiments, a speedup of 1.302 is achieved when using three CPUs. The speedup results are presented in Table 2 along with the number of messages sent. As is shown, it takes over 3,000 messages to compile this particular circuit. Over half of all the messages are the ones that carry the generated code to the master. Another significant portion of messages is related to the signal drive resolution. It is thus reasonable to expect to have better results if we use faster networks.

5. Conclusions

HDLs are different from most programming languages in that the compilation output is an enumeration of the hardware components, such as gates and pins, of the circuit. This, combined with the fact that HDL sources are usually hierarchical and modular, makes it feasible to parallelize the compilation process.

We have shown that the parsing, elaboration and code generation phases can all be carried out in parallel with little synchronization. Our experimental results for a small circuit indicate that it is possible to achieve a speedup of the compilation process, which should prove to be valuable for larger circuits. Our future work will focus on the compilation of larger circuits and on further identifying and reducing the sources of overhead, notably the communications cost.

References

[1] S. Brawer. *Introduction to Parallel Programming*. Academic Press, 1989.

[2] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 695–698, June 1985.

[3] M. D. Ciletti. *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*. Prentice Hall, 1999.

[4] M. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse-engineering. *IEEE Design and Test*, 16(3):72–80, July-Sept. 1999.

[5] IEEE Computer Society. *IEEE Std. 1364-2001, IEEE Standard Verilog Hardware Description Language*. 2001.

[6] L. Li, H. Huang, and C. Tropper. DVS: An object-oriented framework for distributed verilog simulation. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 2003.

[7] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language, 5th Edition*. Kluwer Academic Publishers, 2002.

[8] S. Williams. Icarus verilog. *http://icarus.com/eda/verilog/*.