AN INVESTIGATION OF THE SIMULATION PERFORMANCE OF VERILOG FOR LARGE CIRCUITS

By

CHEN, TING-CHANG

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

2003

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2005

AN INVESTIGATION OF THE SIMULATION PERFORMANCE OF VERILOG FOR LARGE CIRCUITS

Thesis Approved:

Dr. Louis G. Johnson

Thesis Advisor

Dr. Weili Zhang

Dr. Yumin Zhang

Dr. A. Gordon Emslie

Dean of the Graduate Collage

ACKNOWLEGMENTS

I sincerely thank my advisor Dr. Louis G. Johnson, who directed me to finish this thesis, and also my committee members Dr. Weili Zhang and Dr. Yumin Zhang.

I have too many words to my parents, Mr. Chen, Wen-Liang and Ms. Shieh, Shu-Ju. Thank you for giving me all I have. I can never say enough about your devotion to me.

I also want to say thanks to all my friends in Stillwater. Thank you for your kind support during these days studying here.

Finally, thanks to my country Taiwan, which is a magnificent place and always welcomes everyone to visit.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE & INSTRUCTION

CHAPTER I INTRODUCTION AND LITERATURE

Background and the problem

There is no efficient way so far for predicting the simulation time of a large circuit, which is still the bottleneck of the whole developing process while the time-to-market for a large circuit is demanded shorter and shorter whit a requirement of more and more complex circuits. If the time for simulating a large circuit can be predicted in advance, it will benefit the design process. The method of prediction for simulation time also can be used to judge if a developing tool is suitable, and selecting a good tool is very important to the developing process. Unfortunately, there is a great discrepancy on performance tests for different software; therefore it adds the difficulty to the selection of a tool. The lack of standards and varieties of testing methods allows the simulation software companies to make claims that are difficult to verify. In this study, a method will be proposal not only for predicting the simulation time but also for the performance comparison empirically. The literature on the performance of simulation software are categorized and reviewed as background knowledge of simulators. This is necessary for understanding the specification of a simulator. The Cadence® NC-Verilog and Verilog-XL are applied as tools in this study. The difference between these two tools will be figured out and also the performance is compared.

Some methods for speeding up simulation

Some literatures and studies about accelerating the simulation or improving the bottleneck of design process are picked up and categorized into four classes. They are simulating methods and verification, hardware supplement, project partitioning, and benchmark selection. The users of commercial tools can follow some of these methods to improve their design process while the remaining is for the software engineers, who develop the simulating tools. Some experiences shared in the internet are helpful and an example is raised in this study.

Simulating methods and verification

There are many simulating methods and two main streams are selected here for comparison: the cycle-base and event-driven methods. The cycle-base method evaluates the whole circuit in every time unit. It's slow because that some useless messages are calculated. For the long delay gates it will lower the performance very obviously. From the other aspects, this method is good for simulating the sequence logic circuits, the concurrency and can avoid some glitch problems naturally. For event-driven method, all of the events will be collected into a list called event-wheel or timing-wheel. This method is very sensitive to glitches. One glitch in a bad timing could take over the whole simulation. The feedback circuit is also a great concern in this method. Z. Wang proposed a method called LECSIM (LEvelized event driven Compiled SIMulator) can improve the simulation by integrating the event driven interpretive simulation and levelized compiled

simulation [1]. The LECSIM runs about 8-77 times faster compared to traditional unit-dealy event-driven interpretive simulator. The great characteristic of this improvement is its combining the levelized, event driven and complied together to achieve the performance issue. Peter M. Maurer borrowed the idea from object-oriented programming skill and invented a technique named EVCF, Event-Driven Conditional-Free [2]. The EVCF is similar to state machine by eliminating conditional test. In other words, it just performs the assignment and ‘gotos’. The EVCF is 7-60 faster times than Conventional Event-Driven algorithm. Robert S. French applied static simulation technique, which contains two innovations [3]. First, a general event graph replaces an event-driven simulation by capturing its semantics. Second, a technique called partial evaluation schedules the events as well as possible by using statically available information. This method can apply to general models: synchronous, asynchronous designs and the certain-delay-time. According to his test, their system speeded up the simulation time by two times while its overhead is only 4% of which is in VCS code.

Since it is impossible to have an exhausted test for a large circuit, the verification is helpful. The verification can help rather than replace the simulation. It can be a direction to simulation and avoiding some unnecessary simulation. For example, Gary York offered an integrated environment for HDL (Hardware Description Language) verification, which performs implementation and design verification. Many verification skills are presented there [4].

Hardware supplement

All the software codes are performed by hardware eventually. If we can have specially designed hardware for specific function, the performance of the job will be better than while it is finished in general purpose machine. Charlie Burns offered the architecture by applying pipeline method on event driven [5]. Because the special designed machine is very expensive, there exists a tradeoff of performance and cost by moving a part of the test instead of the whole one onto the special designed hardware, and Matthias Bauer introduced an idea [6]. The drawback of this method is the management overheads for the great communication between the software simulator and hardware which takes care the special function. Young-Il Kim brought it further [7]. He translated the test bench into equivalent hardware to prevent the cost of communication and accelerated the simulating speed. It could compress the simulation time by a factor of 1000 compared to the conventional hardware accelerating simulation.

Project partitioning

For a large project, it is impossible to be done by a single person. The art of dividing a project into small modules is an important issue. The complication is proportional to simulation time by exponential rate, and John Willoughby from Cadence® offered a tool called Synergy [8]. This tool can help to solve all these problems for partitioning a project and to handle the time constraints, connecting single levels, and changing of the upper level code. From their test, it showed the memory

requirement and simulation time were both significantly reduced. Monte Becker proposed an idea by separating the most common part of the whole project [9]. The Bus Functional Models (BFM) is separated from the other modules and simulate in a less detailed way and it is a good suggestion for design a project. Kunle Olukotun gave a concept on how to build a project from the ground to top by hierarchical method [10]. The design levels are categorized into four areas: algorithm, architecture, register-transfer, and logic. He used the HLL to design algorithm and architecture level, and made sure the algorithm and architecture are both correct since this language is faster. Then the HDL was applied for the remaining two levels. There are different focuses for different levels in the design process. While you are working in the register-transfer level, you do not need to worry if the algorithm and architecture are correct by applying this idea. Additionally, the study combined different levels for simulation simultaneously. For some less important portions of the circuit, the system can switched to higher level and returned the result for lower level's simulation in order to save simulation time.

Benchmark selection

A good benchmark can shorten the simulation time. There are different kinds of benchmarks for variety purposes of simulations. Kunle Olukotun gave three kinds of benchmark: statistics-driven simulation, in which the input is a set of statistics gathered from an existing machine or a more complete simulator; trace-driven, which has a sequence of events primarily used for detailed architecture performance prediction; and execution-driven, which has more design detail to model the data transformations for

accurately simulating the behavior of the system [10]. Roland E. Wunderlich proposed a SMARTS (Sampling Microarchitecture Simulation) framework which applies the statistical method to select an appropriate benchmark subset that could speedup the simulation [11]. In one words, the input of simulation is one of the key points for the speed of simulation.

Experience shared in internet

There are many experiences shared in internet from forums. Sometimes it is more helpful than the periodic pressing from the view of a commercial tool user. Because the Cadence® tool is used in this study, the Cadence web site was searched and a good experience was shared by Kathleen Meade [12]. The author offered some suggestions and command-line options that affect the simulating performance of NC-Verilog. There are as follows:

Install the latest software release: The latest version of software will have better performance than preceding version.

Use the NC-Verilog build-in profiler: This function can give you a whole view for your simulation, and you can find the bottle neck of your design by applying “ncverilog +ncprofile <other_option>”.

Disable timing for improved performance: For some logic circuit without timing concern, you can switch the SDF (standard delay format) off to avoid the timing checking for improving performance. The command is “ncverilog +delay_mode_distributed

+notimingcheck +noneg_tchk". There are also some other methods described in the electronic manual of Cadence®.

Optimize use of design access options: The more options you have, the slower the simulation is. All options added to the simulation consume the resources of the system. It is best to use as few simulating options as possible. If you need to have some options, try to select one with less of an impact. You need to take all the options into your account all the time, and understand all the options.

CHAPTER II METHODOLOGY AND FINDINGS

The hardship in software testing

There are many obstacles for testing the performance of software. The lack of literatures and publications could explain the hard situation [14]. In the real world, the performance test seems to compare the apples to oranges. For examples, it is hard to find two identical systems with the same hardware, operation system and performing the same task. Even though they have the same hardware and software system for the same task, if there exist different configurations or input data, it could have a great difference on performance. But it does not mean that it is worthless for researching this topic. Under some terms, the result could be a very valuable reference.

The tools for this study

This study uses Cadence® tools to evaluate the running time of shift registers for predicting simulation time of a similar large circuit. The counter circuits were used in the experiment at first, and one big problem of measuring the simulation time was found. Since not every bit in a counter changes every clock cycle, it is hard to control the measuring on the same base for running time. For example, when the value of the counter changes from 2 to 3, in the binary system, it just changes the less significant bit of its

content (010 to 011) and from 3 to 4 all three bits are alternated (011 to 100). The change is in certain order different from every clock cycle. According to our investigating, the NC-Verilog integrated the verification skill and it just evaluates the time of changing bits. Therefore we change the counters to shift-registers, which alter their contents every clock cycle. The in-site test, which we test the software in the exact system used to develop the project, is named in this study. The specifications of our equipments are: Sun Enterprise 4000 for machine, Solar 9 for OS, Memory 1.5 GB shared by 6 processes, NC-Verilog of version 04.10-p002, and Verilog-XL of version 2.8.s008. The behavioral codes and structural codes, which are synthesized by the synthesizer from behavioral codes directly, are tested for NC-Verilog and Verilog-XL. By measuring the time of fixed clock cycles to varying hardware volume and fixed hardware volume to varying clock cycles, the time of simulations are recorded and equations are derived for predicting approximate simulation time of a similar design. From the equations, the relations of simulation time for behavior and structural codes of NC-Verilog and Verilog-XL can be known empirically. No matter how it is described in the manufacture's brochure, this test method does mean something for people who use this system to design a project. This method also can apply for the people who want to know the performance of their system. The difference between NC-Verilog and Verilog-XL will be introduced first.

The differences between the Cadence® NC-Verilog and Verilog-XL

The NC-Verilog is the latest product for replacing Verilog-XL of Cadence, and this section as follow comes from reference [13]. These main differences can be

attributed to compilation, backannotation, interactive debugging, simulation, and races and event orders.

Simulators: Cadence addressed that NC-Verilog is an average 8 times faster in RTL simulation while there is about 6 times faster at gate level compared to Verilog-XL. NC-Verilog has some utilities that help transform Verilog-XL into NC-Verilog. NC-Verilog is compliant with the IEEE 1364 standard where the IEEE 1364 standard borrowed many definitions from Verilog-XL. The NC-Verilog applies a new architecture that is totally different from Verilog-XL. For example, the NC-Verilog merged the behavioral and gate level simulating engines into one and there are two in Verilog-XL. The performance of NC-Verilog improves significantly.

Compilation: NC-Verilog supports the compiler directive ‘undefineall, which gives more flexibility to programmers by freeing all the macros defined by ‘define. The format is “‘ifdef <name>, ‘undefineall, ‘endif”. This function can avoid duplicate of defining a name. In Verilog-XL, you must define all output ports to specific blocks. This is not a restriction in NC-Verilog. In other words, you can use the “assign” function to assign the output ports in NC-Verilog but it does not work in Verilog-XL.

Backannotation: The NC-Verilog released some restrictions from Verilog-XL for SDF backannotation. You do not need to match the reference bits of vectors in the SDF file to corresponding specific block exactly and you can just select part of which in NC-Verilog. Verilog-XL treats a $setuphold check as two checks, while the NC-Verilog does not.

Interactive debugging: Unlike Verilog-XL simulation, whose interactive mode commands are a subset of the Verilog language, the NC-Verilog interactive simulation supports the industry standard TCL command language.

Simulation: Verilog-XL supports only 6 delays (0, 1 and X states), and NC-Verilog supports 12 delays (0, 1, X and Z states). NC-Verilog can handle the recursive via the stack scheme whereas Verilog-XL does not have this function. NC-Verilog treats initialization from the time zero and it has output from the moment it starts to simulate the circuit. But Verilog-XL does not work by this way. For the first several cycles, these two simulators could draw different values for the same code. NC-Verilog is better than Verilog-XL for handling the glitch.

Races and Event Orders: According to the IEEE standard 1364, the simulator can handle the active events in any order. Since the engines of simulators for NC-Verilog and Verilog-XL are different, the output of the same circuit could have different results due to the active events they arrange.

The testing codes

All the codes and instructions are put in the appendix. In the [Code & Instruction 1: The behavioral code of shift register], the Hardware_Volume will be replaced by real number, which represents the bit number of shift register while it is active. The value in the for-loop should be taken considerably. If the ascending order is used (from 1 to k), the synthesizer will not generate the right structural code for the shift register. And the maximum Hardware_Volume value in this system that the synthesizer can synthesize is

999 (the number is from 0 to 999, which stands for 1000 bits). If a value is greater than 999 filled, there causes an error during the synthesizing. Due to the tool's limitation we just can write the structural code by hand when the Hardware_Volume is greater than 1000 for shift register. The [Code & Instruction 2: The translation of behavioral code to structural code] of appendix is applied for translating the behavioral codes to structural codes.

The structural code generated by synthesizer is in the [Code & Instruction 3: The structural code of shift register]. A part of the repeating codes has been omitted since they are trivial. This code is fetched from the synthesizing result for the Hardware_Volume of 999.

By checking the codes, we can confirm the structural and behavioral codes matched. The df001 is the register module of our library of alf ami350hxsc3.alf. In this study, the same test bench is used to NC-Verilog and Verilog-XL for structural and behavioral codes. The test bench is presented in the [Code & Instruction 4: The test bench of shift register].

The Hardware_Volume in the [Code & Instruction 4] is the same variable as the behavioral code in the [Code & Instruction 1]. Different timescales of test benches will have different results of simulation time. In here the timescale 1ns/10ps is employed. According to our test bench, two number stand for one clock cycle. For example, if the 1000 is replaced for the variable Clock_time_2, it means 500 clock cycles will be simulated. For the $dispaly instruction, it will output the calculating result to the console. The more it displays, the longer the simulation time needs. From our investigation for the NC-Verilog, it combines the verification skill which just calculates the value needed for

displaying to console in the simulation. If the $dispaly instruction is taken away for NC-Verilog simulation, no matter what Hardware_Volume is, it will simulate in a very fast speed by calculating nothing. In the same situation, the Verilog-XL still calculates them but saves the time for I/O to display. In this study, the highest significant bit of the shift register is shown on the console for debugging and performance tradeoff. Therefore taking unnecessary messages away from being displayed is always the best rule of thumb needed to be kept in mind.

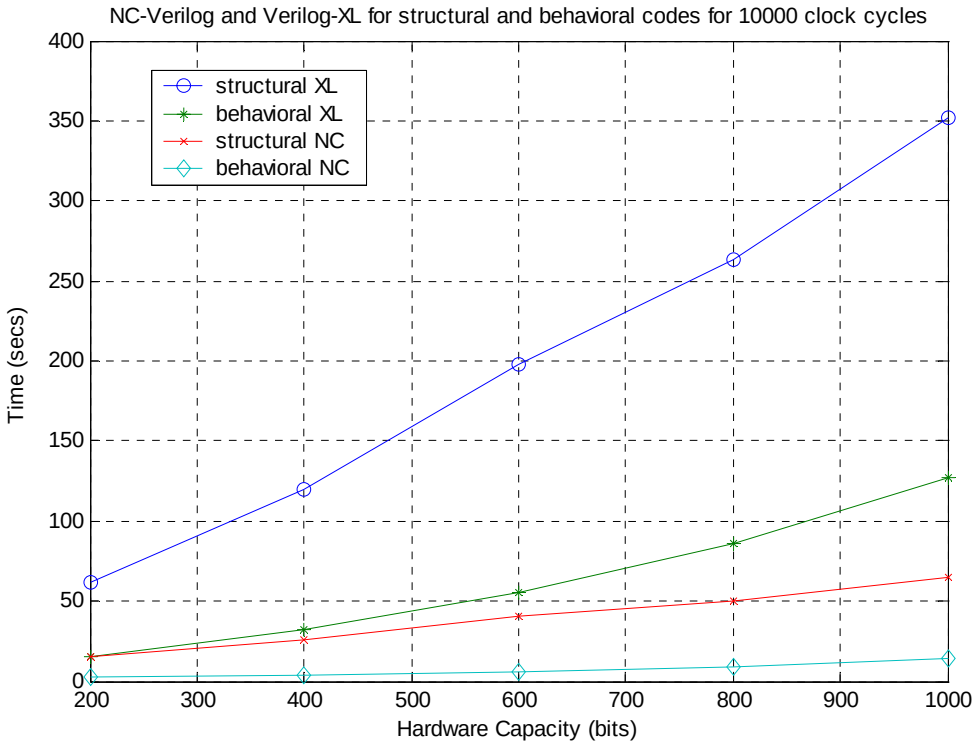
The relationship between different tools

The relationships of NC-Verilog and Verilog-XL for behavioral and structural codes are investigated here. This test employed 10,000 clock cycles. The [Code & Instruction 5: The simulation instructions] of appendix is used.

The –v command is used to direct the library ami350hxsc.d.lib. The +ncstatus command is applied for displaying the consuming time of NC-Verilog simulation. The <target file> stands for the file of structural or behavioral code, and the <test bench> is obviously for test bench. The result is in the [Table 1] and the [Figure 1].

The shift register test at 10,000 clock cycles					
the bit-capacity for shift register	200	400	600	800	1000
behavioral code of VerilogXL (secs)	14.9	32	54.9	86.5	127.2
structural code of VerilogXL (secs)	62.2	119.8	197.8	263.1	352.4
behavioral code of NC-Verilog (secs)	2.9	4.1	6.3	9.2	14.1
structural code of NC-Verilog (secs)	15.1	26.1	41	50.3	64.5

[Table 1: The shift register test at 10,000 clock cycles]



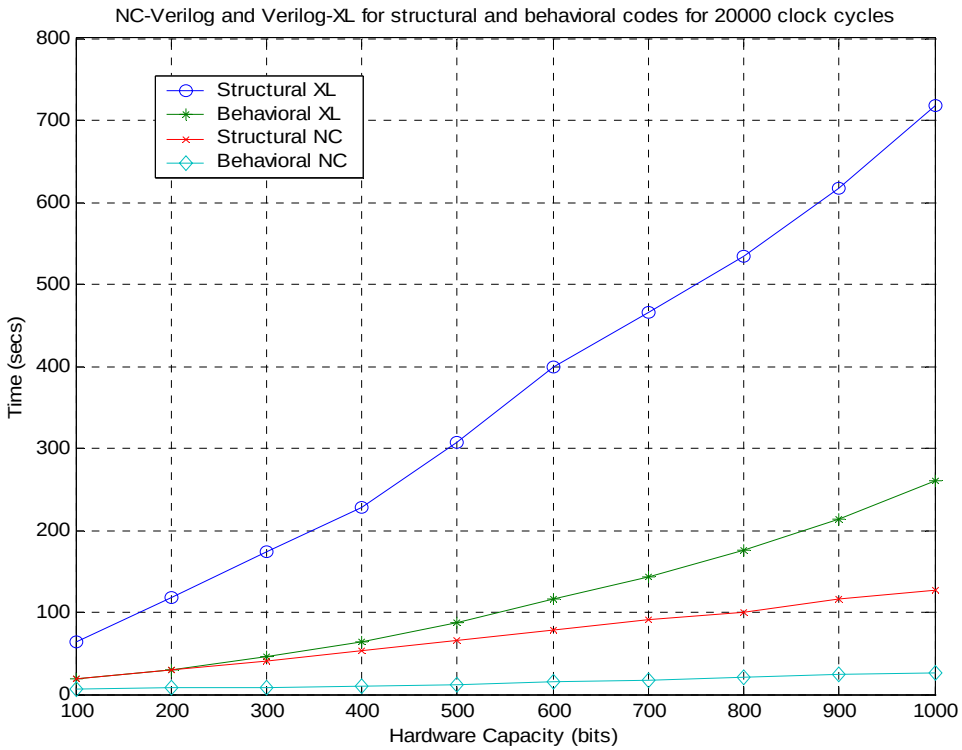
[Figure 1: NC-Verilog and Verilog-XL for structural and behavioral codes for 10,000 clock cycles]

Because the simulation time here is short (4.1 seconds for behavioral NC-Verilog at Hardware_volume 400), it means a small difference of time measuring will cause a big error. One second difference in the behavioral NC-Verilog test could be a 25% error for 4.1 seconds of simulation time. Because of the synthesizer problems, the hardware capacity cannot exceed 1000 bits in this test mentioned above. In order to reduce the measuring errors, the test will be repeated by increasing the clock cycles to 20,000 and having more measuring points.

The shift register test at 20,000 clock cycle					
the bit-capacity for shift register	100	200	300	400	500
behavioral code of VerilogXL(secs)	18.9	30.5	45.3	63.6	87.2
structural code of VerilogXL(secs)	64.2	118.7	173.1	228.8	308
behavioral code of NC-Verilog(secs)	6.4	7.5	8.6	10.7	12.5
structural code of NC-Verilog(secs)	19	30.2	41.4	53.5	65.1
the bit-capacity for shift register	600	700	800	900	1000
behavioral code of VerilogXL(secs)	115.7	143.8	174.8	214.1	261.2
structural code of VerilogXL(secs)	398.9	466.2	534.6	617.9	717.8
behavioral code of NC-Verilog(secs)	14.7	17.2	20	23.6	26.9
structural code of NC-Verilog(secs)	77.7	90.2	100.8	115.5	126.9

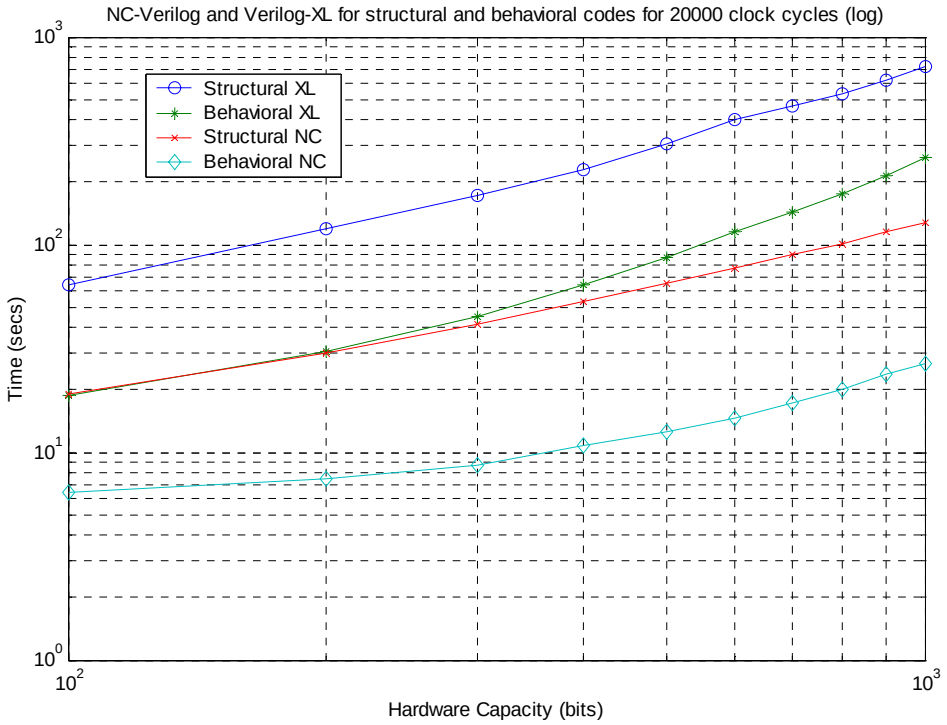
[Table 2: The shift register test at 20,000 clock cycles]

The values in [Table 2] are plotted on [Figure 2] as follow.



[Figure2: NC-Verilog and Verilog-XL for structural and behavioral codes for 20,000 clock cycles]

By comparing the [Figure 2] to the [Figure 1], the trends of the curves are very similar. It means the increasing of clock cycles for this test has a linear relationship to the simulation time. In order to have a closer look into the [Figure 2] for their trends, the coordinator system will be changed to logarithm system in the [Figure 3] below.



[Figure3: NC-Verilog and Verilog-XL for structural and behavioral codes for 20,000 clock cycles (log)]

By inspecting the [Figure 3], except the behavioral XL result, the remaining three have the same increasing rate after 500 bits of hardware capacity. It makes sense because the percentage of overheads in the small hardware capacity of structural NC-Verilog code is higher than it is in large hardware capacity. Because this test will be used to estimate the “approximate” simulation time for large design, the behavioral code of Verilog-XL will be assumed to have the same increasing rate as the remaining three, and the behavioral NC-Verilog code simulation time will be used as base for normalization to

define a parameter called “Tool-Factor”. By dividing all the rows to row “behavioral code of NC-Verilog” in the [Table 2] respectively, it turns out a multiplying factor table shown in the [Table 3] below.

The multiplying factor					
the bit-capacity for shift register	100	200	300	400	500
behavioral code of VerilogXL(secs)	2.95	4.07	5.27	5.94	6.98
structural code of VerilogXL(secs)	10.03	15.83	20.13	21.38	24.64
behavioral code of NC-Verilog(secs)	1	1	1	1	1
structural code of NC-Verilog(secs)	2.97	4.03	4.81	5	5.21
the bit-capacity for shift register	600	700	800	900	1000
behavioral code of VerilogXL(secs)	7.87	8.36	8.74	9.07	9.71
structural code of VerilogXL(secs)	27.14	27.1	26.73	26.18	26.68
behavioral code of NC-Verilog(secs)	1	1	1	1	1
structural code of NC-Verilog(secs)	5.29	5.24	5.04	4.89	4.72

[Table 3: The multiplying factor]

The average value of each row will be found on the [Table 4]. With this table, if the simulation time of behavioral NC-Verilog code is found then the remaining simulation times are known for similar style programming codes.

The Tool-Factor	
behavioral code of VerilogXL	6.896
structural code of VerilogXL	22.584
behavioral code of NC-Verilog	1
structural code of NC-Verilog	4.72

[Table 4: The Tool-Factor]

According to Cadence® [13], the NC-Verilog is an average of 8 times faster than Verilog-XL for behavioral codes, where there is about 6 times for structural codes. This result consists with our test in the [Table 4]. It is about 7x for behavioral codes and 5x (22.584 / 4.72) for structural codes.

The equation for prediction

The module for predicting the simulation time will be conducted. First of all, the fixed clock cycle to different hardware capacities are tested to [Table 5] for behavioral NC-Verilog codes.

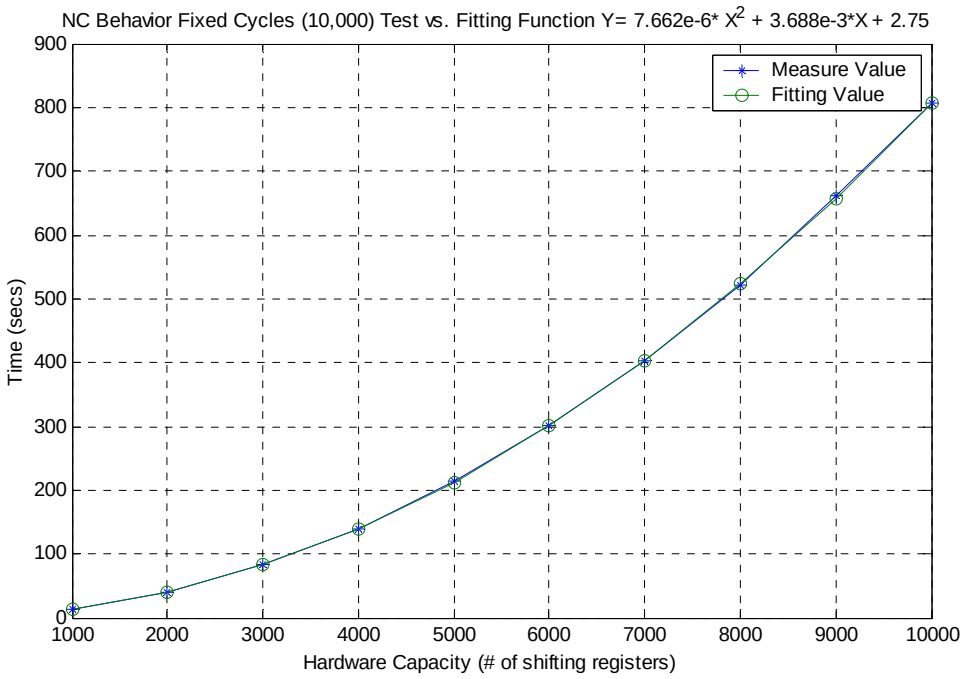
Shift register on fixed clock cycles (10,000 clock cycles) for NC-Verilog					
the bit-capacity for shift register	1000	2000	3000	4000	5000
behavioral simulation time (secs)	14.1	40.8	82.4	139.7	213.2
the bit-capacity for shift register	6000	7000	8000	9000	10000
behavioral simulation time (secs)	300.7	403.4	521.9	661.4	805.8

[Table 5: Shift register on fixed clock cycles (10,000) for behavioral NC-Verilog]

A fitting equation is derived from [Table 5]:

$$\text{Time} = 7.662*10^{-6}* \text{Hardware\_Volume}^{2} + 3.688*10^{-3}* \text{Hardware\_Volume} + 2.75 \quad (1)$$

The fitting equation and the original data plotted are put onto [Figure 4].



[Figure 4: Behavioral NC code at fixed cycles (10,000) test vs. fitting function]

This is a perfect match and it presents the test of having a quadratic increasing rate. When the hardware volume increases for the same clock cycle, the simulation time swells up by a square rate for the style in the [Code & Instruction 1] associated with the test bench [Code & Instruction 4]. The next step, different numbers of clock cycles will be tested for different fixed hardware capacity of 2000 ([Table 6]), 4000 ([Table 7]), 6000 ([Table 8]), 8000 ([Table 9]), and 10000 ([Table 10]) bits respectively.

Shift register on fixed hardware capacity 2,000 bits for NC-Verilog					
# of clock cycles	1000	2000	3000	4000	5000
behavioral simulation time (secs)	5.1	8.9	12.9	16.8	20.5
# of clock cycles	6000	7000	8000	9000	10000
behavioral simulation time (secs)	24.5	28.5	32.5	36.5	40.1

[Table 6: Shift register on fixed hardware capacity 2,000 bits for NC-Verilog]

Shift register on fixed hardware capacity 4,000 bits for NC-Verilog					
# of clock cycle	1000	2000	3000	4000	5000
behavioral simulation time (secs)	14.8	28.4	42.2	56	69.4
# of clock cycle	6000	7000	8000	9000	10000
behavioral simulation time (secs)	83.4	97	110.6	124.5	137.9

[Table 7: Shift register on fixed hardware capacity 4,000 bits for NC-Verilog]

Shift register on fixed hardware capacity 6,000 bits for NC-Verilog					
# of clock cycle	1000	2000	3000	4000	5000
behavioral simulation time (secs)	30.8	60.6	90.4	120.3	149.9
# of clock cycle	6000	7000	8000	9000	10000
behavioral simulation time (secs)	179.8	209.6	239.2	268.8	299.4

[Table 8: Shift register on fixed hardware capacity 6,000 bits for NC-Verilog]

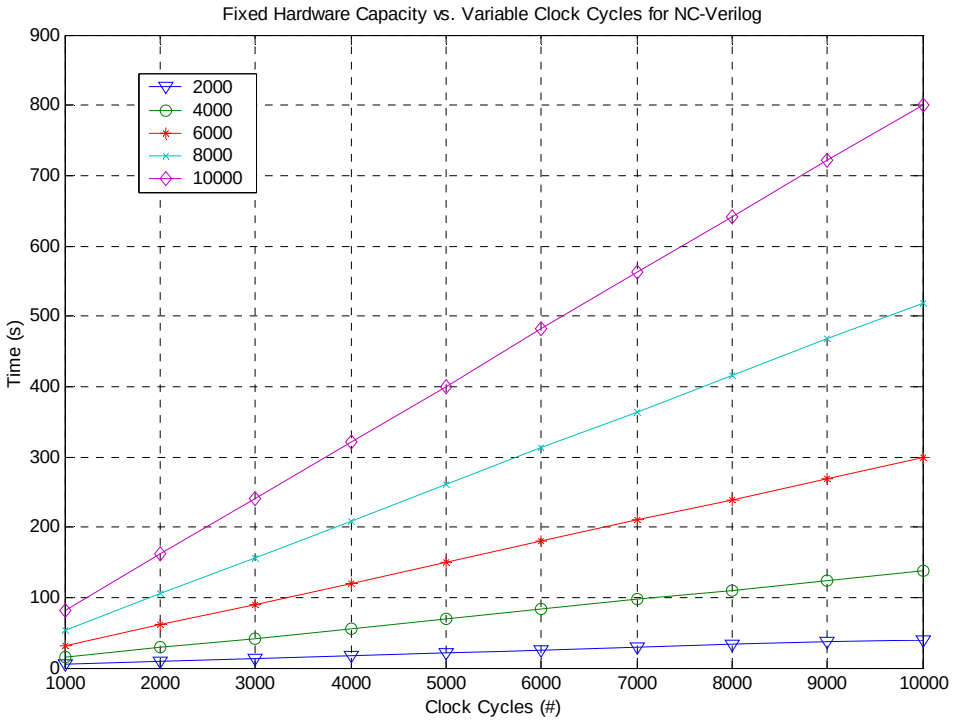
Shift register on fixed hardware capacity 8,000 bits for NC-Verilog					
# of clock cycle	1000	2000	3000	4000	5000
behavioral simulation time (secs)	53.1	105.5	156.5	208.7	260
# of clock cycle	6000	7000	8000	9000	10000
behavioral simulation time (secs)	312.3	364.2	416.3	468.1	519.1

[Table 9: Shift register on fixed hardware capacity 8,000 bits for NC-Verilog]

Shift register on fixed hardware capacity 10,000 bits for NC-Verilog					
# of clock cycle	1000	2000	3000	4000	5000
behavioral simulation time (secs)	81.2	161.3	241.6	321.8	400.6
# of clock cycle	6000	7000	8000	9000	10000
behavioral simulation time (secs)	481.9	562.1	641.9	721.8	801.3

[Table 10: Shift register on fixed hardware capacity 10,000 bits for NC-Verilog]

The values from the [Table 6] to the [Table 10] are plotted on the [Figure 5] and the fitting function for each line will be found.



[Figure 5: Fixed hardware capacity vs. variable clock cycles for NC-Verilog]

It is very obvious, for fixed hardware capacity to variable clock cycles, the time of simulation associates with increasing clock cycles linearly. The fitting equations for each line in the [Figure 5] are:

Hardware Volume 2000: Time = $3.89*10^{-3}*$ Clock_cycles + 1.21 (2)

Hardware Volume 4000: Time = $1.37*10^{-2}*$ Clock_cycles + 0.9 (3)

Hardware Volume 6000: Time = $2.89*10^{-2}*$ Clock_cycles + 1.0 (4)

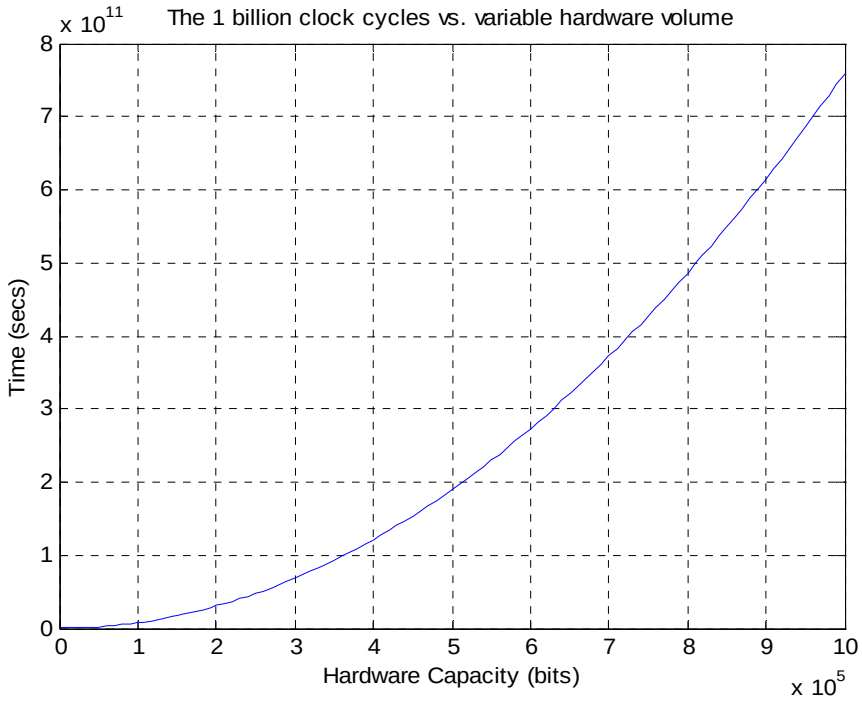
Hardware Volume 8000: Time = $5.18*10^{-2}*$ Clock_cycles + 1.3 (5)

Hardware Volume 10000: Time = $8.00*10^{-2}*$ Clock_cycles + 1.2 (6)

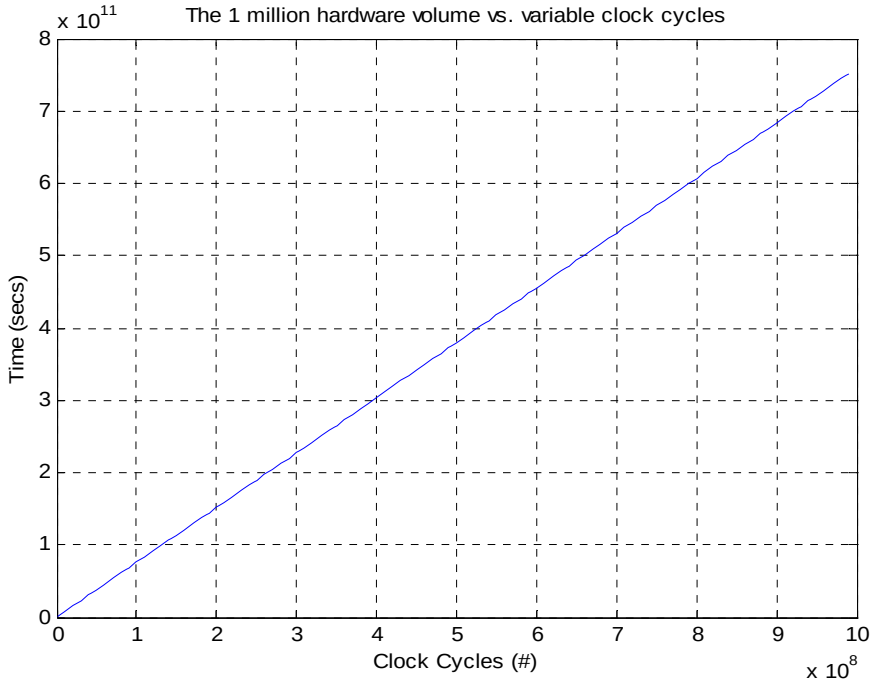
The curve on the [Figure 4] is known that it could be fitted by a quadratic equation (1). In here, the procedure is repeated and equation (2), (4), and (6) are selected to plug into "Time = a * Hardware_volume^2 + b * Hardware_volume + c" respectively. The parameter a, b, and c can be refined out and an equation combined Hardware_Volume and Clock_cycles can be used to estimate the simulation time for similar design. The equation is:

$$\text{Time} = (7.59*10^{-10}*\text{Clock\_cycles} + 1.22*10^{-8})* (\text{Hareware\_volume}^2) + (4.06*10^{-7}*\text{Clock\_cycles} - 1.45*10^{-4})* (\text{Hardware\_volume}) + (4*10^{-5}*\text{Clock\_cycles} +1.43) \quad (7)$$

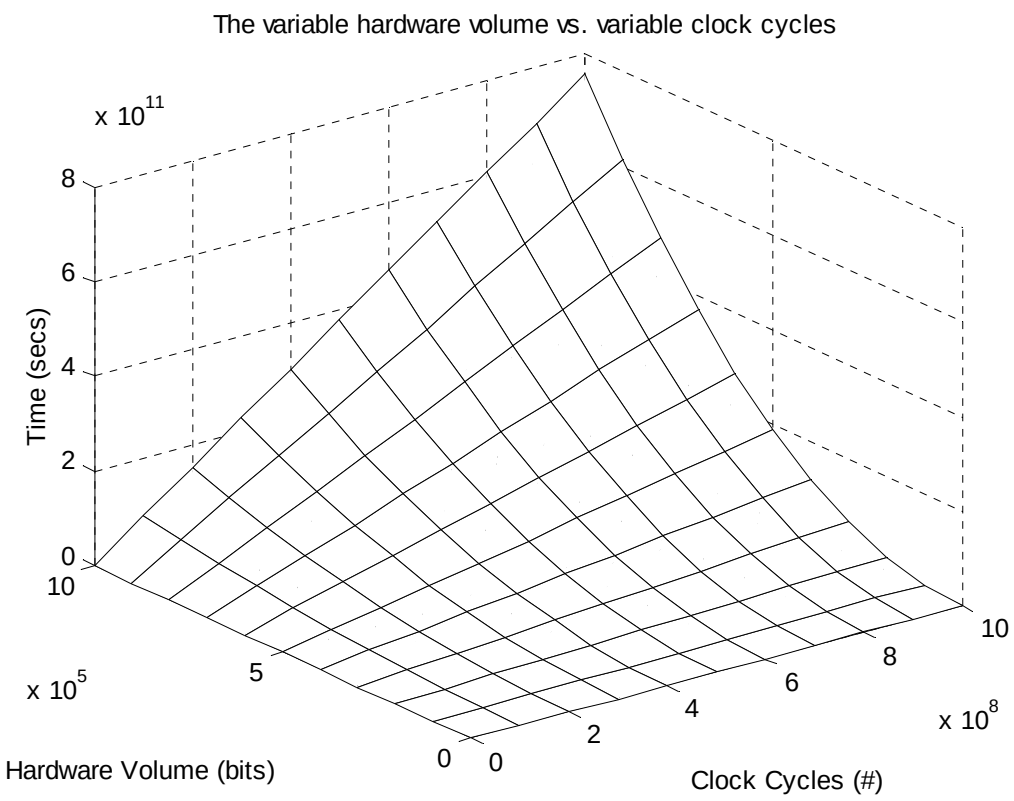
Let us have a little discussion for equation (7). First, the figure for clock cycles at 1,000,000,000 (1 billion) to variable hardware volume from 1 to 1,000,000 (1 million) will be drawn to the [Figure 6: The 1 billion clock cycles vs. variable hardware volume]. Then the hardware volume is set to 1,000,000 (1 million) and the clock cycle is changed from 1 to 1,000,000,000 (1 billion) in the [Figure 7: The 1 million hardware volume vs. variable clock cycles]. Finally, a graphic with two variables for the variable clock cycle and the variable hardware volume is put into the [Figure 8: The variable hardware volume vs. variable clock cycles]. Two arbitrary big values of 1 billion clock cycles and 1 million hardware volume are selected; the test time for this case at this system approximates to be $7.594*10^{11}$ seconds. It is about 24080.4 years. This result is not accepted in the real world since no one can wait for such a long time. From the view of this point, the testing for a large circuit is pessimistic. The speeding up skills will have a great helpful here.



[Figure 6: The 1 billion clock cycles vs. variable hardware volume]



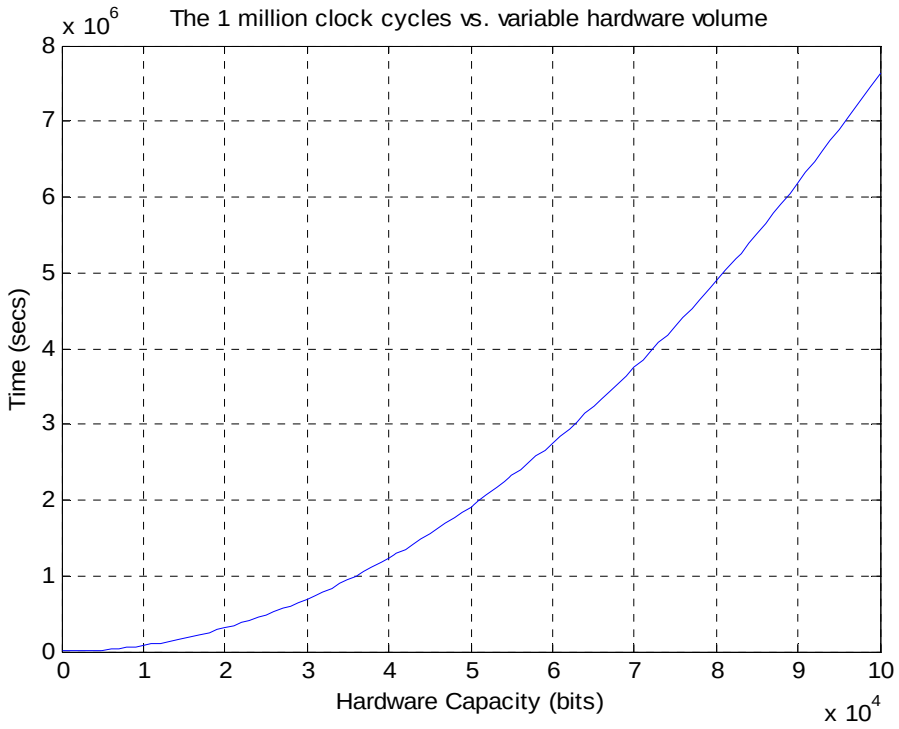
[Figure 7: The 1 million hardware volume vs. variable clock cycles]



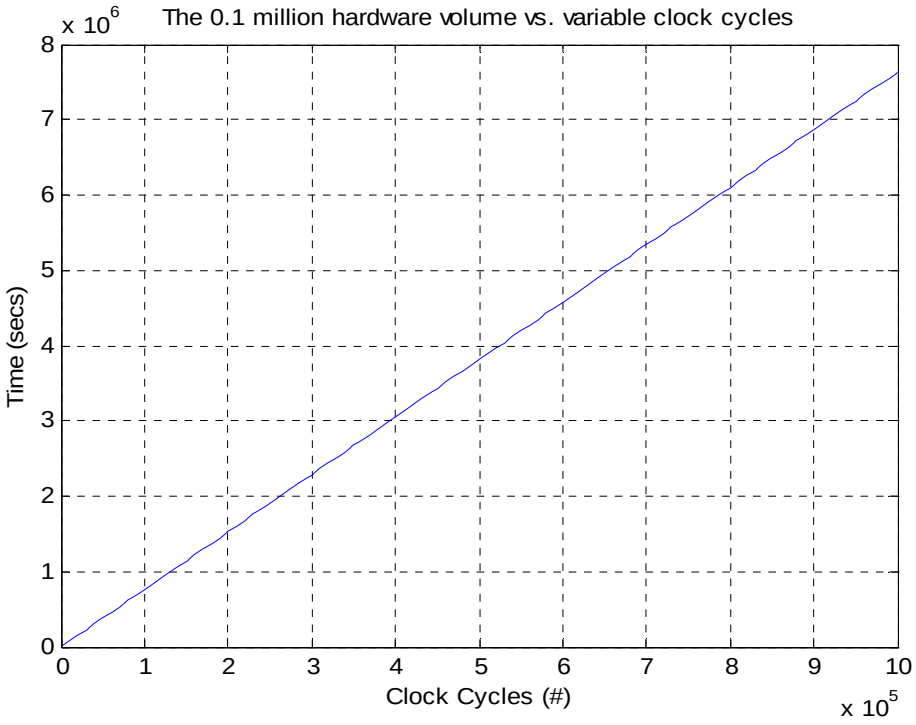
[Figure 8: The variable hardware volume vs. variable clock cycles]

In order to have a more realistic investigation, another arbitrary set of parameters is picked up: 1 million clock cycles and 0.1 million hardware volume. The results are below: [Figure 9: The 1 million clock cycles vs. variable hardware volume], [Figure10: The 0.1 million hardware volume vs. variable clock cycles], and [Figure 11: The variable hardware volume vs. variable clock cycles for smaller value].

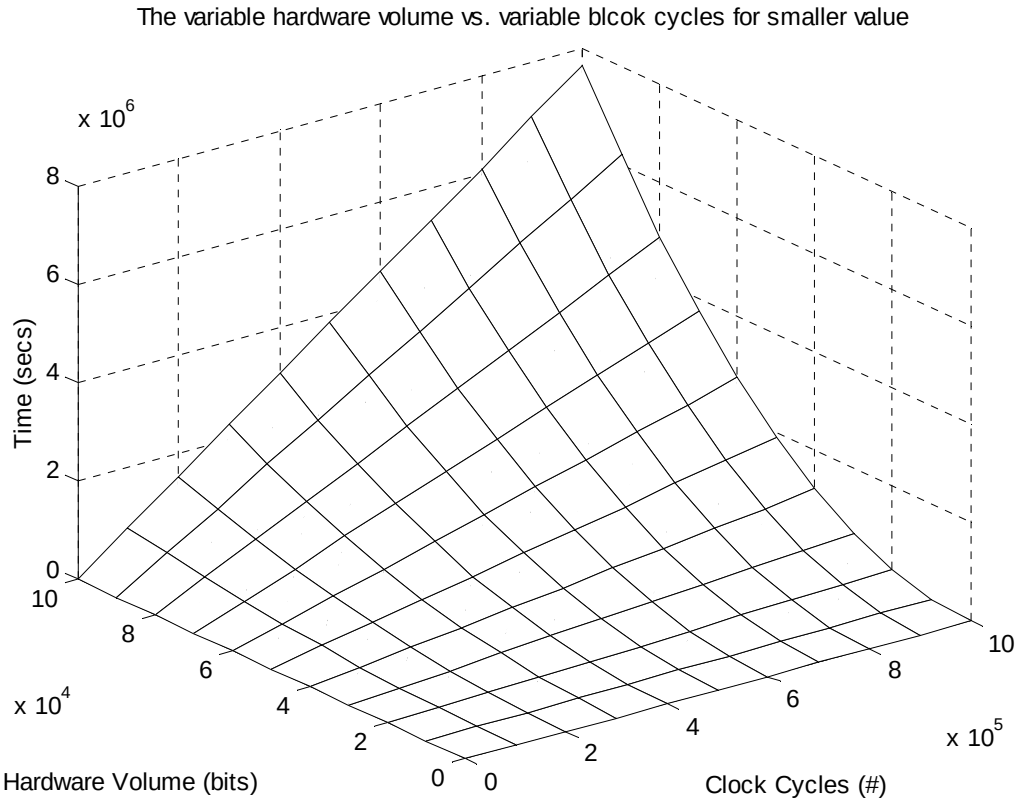
The maximum time for this set of parameters is 7630777.93 seconds (88.31 days).



[Figure 9: The 1 million clock cycles vs. variable hardware volume]



[Figure10: The 0.1 million hardware volume vs. variable clock cycles]



[Figure 11: The variable hardware volume vs. variable clock cycles for smaller value]

The equation (7) is for predicting the simulation time of the behavioral NC-Verilog, and it can be expanded by multiplying the Tool-Factors from the [Table 4] for the structural XL, behavioral XL, and structural NC codes. It turns out equation (8).

$$\text{Time} = (7.59*10^{-10}*\text{Clock\_cycles} + 1.22*10^{-8})* (\text{Hareware\_volume}^2) +$$
$$(4.06*10^{-7}*\text{Clock\_cycles} - 1.45*10^{-4})* (\text{Hardware\_volume}) +$$
$$(4*10^{-5}*\text{Clock\_cycles} +1.43) \quad * \{\text{Tool-Factor}\} \qquad (8)$$

All the tests above use the default system configuration and the library is in the same directory of tested codes in order to save communicating time. It is hardware related for structural codes; in other words the synthesizer will pick up the actual modules from the library for the simulator to run the circuit, for instance the df001 in the [Code & Instruction 3]. Therefore the equation (8) is just for predicting simulation time of the structural codes where the same type of registers (df001) used. But for the behavioral codes, if the program format is similar and no matter how its function is, equation (8) can be applied to estimate the simulation time. Some examples will be raised to verify this result in the next section.

Some samples and expending the application

Example one: The [Code & Instruction 1] is selected for behavioral Verilog-XL test. The Hardware_volume is 10,000 and the Clock_cycles is 5,000.

$$\text{Time} = \{(7.59*10^{-10}*10000 + 1.22*10^{-8})*(5000^2) + (4.06*10^{-7}*10000 - 1.45*10^{-4})*$$
$$(5000) + (4*10^{-5}*10000 + 1.43)\} * \{6.896\}$$

The result is 1458.22 seconds and the actual simulating result is 1844.8 seconds. The error is about 20% of actual testing time.

Example two: The behavioral NC-Verilog code in the [Code & Instruction 1] will be tested with 20,000 clock cycles and 11,000 bits of hardware capacity.

$$\text{Time} = \{(7.59*10^{-10}*20000 + 1.22*10^{-8})*(11000^2) +$$
$$(4.06*10^{-7}*20000 - 1.45*10^{-4})*(11000) + (4*10^{-5}*20000 + 1.43)\} * \{1\}$$

The result is 1928.21 seconds and the actual simulating result is 1932.2 seconds. Because the behavioral NC-Verilog is selected as base in this study; therefore the error is less than 0.2% of actual testing time.

Example three: The [Code & Instruction 3] is reused for the structural Verilog-XL test. The Hardware_volume is 500 and the Clock_cycles is 30,000.

$$\text{Time} = \{(7.59*10^{-10}*30000 + 1.22*10^{-8})*(500^2) + (4.06*10^{-7}*30000 - 1.45*10^{-4})*$$
$$(500) + (4*10^{-5}*30000 + 1.43)\} * \{22.584\}$$

The result is 323.80 seconds and the actual simulating result is 463.8 seconds. The error is about 30% of actual testing time.

Example four: The [Code & Instruction 3] is used again for the structural NC-Verilog test. The Hardware_volume is 500 and the Clock_cycles is 30,000 the same as above.

$$\text{Time} = \{(7.59*10^{-10}*30000 + 1.22*10^{-8})*(500^2) + (4.06*10^{-7}*30000 - 1.45*10^{-4})*$$
$$(500) + (4*10^{-5}*30000 + 1.43)\} * \{4.72\}$$

The result is 67.68 seconds and the actual simulating result is 96.1 seconds. The error is about 30% of actual testing time.

Example five: The [Code & Instruction 3] is applied again for the structural NC-Verilog test. The Hardware_volume is 1,000 and the Clock_cycles is 300,000.

Time = $\{(7.59*10^{-10}*300000 + 1.22*10^{-8})*(1000^2) +$

$(4.06*10^{-7}*300000 - 1.45*10^{-4})*(1000) + (4*10^{-5}*300000 +1.43)\} * \{4.72\}$

The result is 1712.42 seconds and the actual simulating result is 1879.4 seconds. The error is about 9 % to the actual testing time.

Example six: For the behavioral codes, if a program has similar format with the [Code & Instruction 1], equation (8) can be applied. In the previous cases, just one bit register is simulated. The dimension is expanded here. An 8x10 shift register will be tested in the behavioral NC-Verilog code. The code is in the [Code & Instruction 6: The behavioral code of 8 bit wide shift register]. This code is synthesized into structural code by synthesizer in order to make sure this is the right thing, and it is in the [Code & Instruction 7: The structural code of 8 bit wide shift register].

Simulate the [Code & Instruction 7] with the same test bench [Code & Instruction 4] at 100,000 clock cycles, and the result comes out of 28.2 seconds where it is 9.15 seconds by the equation (8) with parameters filled into the following calculation.

Time = $\{(7.59*10^{-10}*100000 + 1.22*10^{-8})*(80^2) + (4.06*10^{-7}*100000 - 1.45*10^{-4})*$

$(80) + (4*10^{-5}*100000 +1.43)\} * \{1\}$

This result does not surprise us since the format of the code has been modified. The code in the [Code & Instruction 6] and the code in the [Code & Instruction 1], which is used to derivative the equation (8), share the same format. Using the same way of predicting the simulation time for the other codes (structural XL, behavioral XL, and

structural NC), a new Tool-Factor is derived for this case. In other words, the relationship of the code in the [Code & Instruction 6] to our equation (8) will be found. It is easy enough just by using 28.2 / 9.15 and the result is 3.08. Now the clock cycles are increased to 200,000, redo the test, and plug the values into the equation (8) again.

$$\text{Time} = \{(7.59*10^{-10}*200000 + 1.22*10^{-8})*(80^2) + (4.06*10^{-7}*200000 - 1.45*10^{-4})*$$
$$(80) + (4*10^{-5}*200000 + 1.43)\} * \{3.08\}$$

The data from above calculation is 52.02 seconds and from the real test is 54.7 seconds. By applying this method, we can expand our shift register to any dimensions. The error here is about 5%.

Because the purpose here is to show the idea of making a prediction for simulation time, the derivation of the Tool-Factor is very rough and the Tool-Factor (actually the relationship in other words) could be an equation. This will cause a great error since we ignore the detail of the curve trends. The behavioral NC-Verilog code used to estimate the behavioral NC-Verilog code itself has a very satisfied result because it does not have any derivation problem for the Tool-Factor.

CHAPTER III CONCLUSION

There still is no standard for the performance test of hardware simulation. In this thesis, a method is developed which you can use to predict the simulation time of your system. If the simulation time of a design could be known before the simulator is launched, it is beneficial for arranging the schedule of your project. From the investigation for a one bit shift register in our system, if the clock cycle is 1 billion and the hardware capacity is 1 million, the simulation time was 24080.4 years for the behavioral NC-Verilog code. This result implied that it is impossible to have such a big design on this system. When the clock cycle is reduced to 1 million and the hardware capacity is changed to 0.1 million for a one bit shift register of the behavioral NC-Verilog code, the simulation can be finished within 3 months (about 88.31 days). This result is a lower bound for the simulation time of hardware systems with greater complexity than a shift register. In general by selecting one similar behavioral code as a base, the prediction can be extended from the base case.

REFERENCES

[1] Zhicheng Wang and Peter M. Maurer, “LECSIM: A Levelized Event Driven Compiled Logic Simulator”, Proceedings of the 27th ACM/IEEE Design Automation conference, pp. 491-496, June 1990.

[2] Peter M. Maurere, “Event Driven Simulation Without Loops or Conditionals”, Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design, pp. 23-36, 2000.

[3] Robert S. French and Monica S. Lam and Jeremy R. Levitt and Kunle Olukotun, “A General Method for Compiling Event-Driven Simulations”, Design Automation Conference, pp. 151-156, 1995.

[4] Gary York, Robert Mueller-Thuns, Jagat Patel and Derek Beatty, “An Integrated Environment for HDL Verification”, Proceedings of IEEE/Verilog HDL conference, pp. 9-18, 1995.

[5] Charlie Burns, “An Architecture for a Verilog Hardware Accelerator”, Proceedings of IEEE/Verilog HDL conference, pp. 2-11, 1996.

[6] Matthias Bauer, Wolfgang Ecker, Renate Henftling and Andreas Zinn, “A Method for Accelerating Test Environments”, 25th Euromicro Conference (EUROMICRO '99), pp. 1-4, 1999.

[7] Young-Il Kim and Chong-Min Kyung, “Automatic Translation of Behavioral Testbench for Fully Accelerated Simulation”, Proceedings of the 2004 IEEE/ACM international conference on Computer-aided design, pp.218-221, 2004.

[8] John Willoughby, “Synthesis Support for Design Partitioning”, Proceedings of IEEE/Verilog HDL conference, pp.32-37, 1997.

[9] Monte Becker, “Faster Verilog Simulations Using A Cycle Based Programming Methodology”, Proceedings of IEEE/Verilog HDL conference, pp.24-31, 1996.

[10] Kunle Olukotun, Mark Heinrich and David ofelt, “Digital System Simulation: Methodologies and Examples”, Proceedings of the 35th ACM/IEEE Design Automation conference, pp.658-663, 1998.

[11] Roland E. Wunderlich, Thomas F. Wenishch, Babak Falsafi and James C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", Proceedings of the 30th annual international symposium on computer architecture, pp.84-95, 2003.

[12] Kathleen Meade, "NC-Verilog Tips for Maximizing Simulation Performance", **http://apsfv-news.cadenceasia.com/apsfv-news/newsletters/2002-01-EN/NCVerilogTips.html**

[13] Cadence®, "Difference between the Cadence NC-Verilog and Verilog-XL Simulators", **http://apsfv-news.cadenceasia.com/apsfv-news/newsletters/2002-01-EN/Differences_between_NCV_VXL.html**

[14] Weyuker, E.J. and Vokolos, F.I., "Experience with performance testing of software systems: issues, an approach, and case study", IEEE Transactions on Software Engineering, Volume: 26, Issue: 12, pp.1147-1156, 2000.

[15] David A. Patterson and John L. hennessy, "Computer Organization & Design the Hardware / Software Interface", Morgan Kaufmann, 1998.

[16] Zainalabedin Navabi, "Verilog Digital System Design", McGraw-Hill, 1999.

[17] Neil H. E. Weste and Kamran Eshraghian, "Principles of CMOS VLSI Design A System Perspective second edition", Addison Wesley, 1994

APPENDIXES

[Code & Instruction 1: The behavioral code of shift register]

```
module shiftregister(d, clk, q);
 input   d, clk ;
 output [Hardware_Volume : 0] q ;  //change Hardware_Volume to test
 reg  [Hardware_Volume : 0] q;    // change Hardware_Volume to test
 reg insignal ;
 integer k;

 always @ (posedge clk)
 begin

   for(k= Hardware_Volume; k>=1; k=k-1)  // change Hardware_Volume to test
    begin
      q[k] = q[k-1];
    end
    q[0] =  d;

 end
endmodule
```

[Code & Instruction 2: The translation of behavioral code to structural code]

```
    do_remove_design –all      //delete existed compiled files
    read_ver shiftregister.v      //read the behavior file
    do_build_generic          //optimize logic
    set_wire_load_mode enclosed
    read_alf ami350hxsc3.alf     //read the reference library
    do_optimize    //map logic into cells from technology library and minimize area
    write_ver -hier shiftregister_syn.v   //write structural code into file
```

[Code & Instruction 3: The structural code of shift register]

```
// Generated by ac_shell v5.10-s071 on Mon Mar 14 01:15:51 CST 2005.
// Restrictions concerning the use of Ambit BuildGates are covered in the
// license agreement.  Distribution to third party EDA vendors is
// strictly prohibited.

module shiftregister(d, clk, q);

        input d;
        input clk;
        output [999:0] q;

        df001 q_reg_0(.Q(q[0]), .C(clk), .D(d));
        df001 q_reg_1(.Q(q[1]), .C(clk), .D(q[0]));
        df001 q_reg_2(.Q(q[2]), .C(clk), .D(q[1]));
        df001 q_reg_3(.Q(q[3]), .C(clk), .D(q[2]));
        df001 q_reg_4(.Q(q[4]), .C(clk), .D(q[3]));
                        :
                        :  (Omission)
                        :
        df001 q_reg_996(.Q(q[996]), .C(clk), .D(q[995]));
        df001 q_reg_997(.Q(q[997]), .C(clk), .D(q[996]));
        df001 q_reg_998(.Q(q[998]), .C(clk), .D(q[997]));
        df001 q_reg_999(.Q(q[999]), .C(clk), .D(q[998]));
endmodule
```

[Code & Instruction 4: The test bench of shift register]

```
`timescale 1ns/10ps
module shiftregister_testbench();
reg d, clk;
wire [Hardware_Volume:0]  q;   // Hardware_Volume the same as above
integer  k, i;
shiftregister test(.d(d) , .clk(clk) , .q(q));


//start to testing the shift registers
initial begin
  clk = 1'b0;
  #1;
  d = 1'b1;
  #1;
  i = 0;

   for (k=0; k<= Clock_time_2; k=k+1) begin   // # of Clock_time_2 for simulation

      clk = ~clk ;
      #1;
      i = i + 1;
      if(i == 2) begin
        i = 0;
        d = ~d;
        #1;
      end

     $display ("Time=%t  q=%b  k=%d  clk=%b ", $realtime, q[Hardware_Volume], k,
                        clk);   //Hardware_Volume as above

    end
end
endmodule
```

[Code & Instruction 5: The simulation instructions]

```
ncverilog <test bench> <target file> -v <library ami350hxsc.d.lib> +ncstatus
verilog <test bench> <target file> -v <library ami350hxsc.d.lib>
```

[Code & Instruction 6: The behavioral code of 8 bit wide shift register]

```
module shiftregister(d, clk, q);
 input   d, clk ;
 output [79 : 0] q ;  //change Hardware_Volume to test
 reg  [79 : 0] q;    // change Hardware_Volume to test
 reg insignal ;
 integer k;

 always @ (posedge clk)
 begin


    for(k= 9; k>=1; k=k-1)  // change Hardware_Volume to test
     begin
       q[k] = q[k-1];
       q[k+10] = q[k-1+10];
       q[k+20] = q[k-1+20];
       q[k+30] = q[k-1+30];
       q[k+40] = q[k-1+40];
       q[k+50] = q[k-1+50];
       q[k+60] = q[k-1+60];
       q[k+70] = q[k-1+70];
     end

     q[0] =  d;
     q[10] =  d;
     q[20] =  d;
     q[30] =  d;
     q[40] =  d;
     q[50] =  d;
     q[60] =  d;
     q[70] =  d;

 end
endmodule
```

[Code & Instruction 7: The structural code of 8 bit wide shift register]

```
// Generated by ac_shell v5.10-s071 on Thu Mar 17 21:22:30 CST 2005.
// Restrictions concerning the use of Ambit BuildGates are covered in the
// license agreement.  Distribution to third party EDA vendors is
// strictly prohibited.

module shiftregister(d, clk, q);

        input d;
        input clk;
        output [79:0] q;


        df001 q_reg_0(.Q(q[0]), .C(clk), .D(d));
        df001 q_reg_1(.Q(q[1]), .C(clk), .D(q[0]));
        df001 q_reg_2(.Q(q[2]), .C(clk), .D(q[1]));
        df001 q_reg_3(.Q(q[3]), .C(clk), .D(q[2]));
        df001 q_reg_4(.Q(q[4]), .C(clk), .D(q[3]));
        df001 q_reg_5(.Q(q[5]), .C(clk), .D(q[4]));
        df001 q_reg_6(.Q(q[6]), .C(clk), .D(q[5]));
        df001 q_reg_7(.Q(q[7]), .C(clk), .D(q[6]));
        df001 q_reg_8(.Q(q[8]), .C(clk), .D(q[7]));
        df001 q_reg_9(.Q(q[9]), .C(clk), .D(q[8]));
        df001 q_reg_10(.Q(q[10]), .C(clk), .D(d));
        df001 q_reg_11(.Q(q[11]), .C(clk), .D(q[10]));
        df001 q_reg_12(.Q(q[12]), .C(clk), .D(q[11]));
        df001 q_reg_13(.Q(q[13]), .C(clk), .D(q[12]));
        df001 q_reg_14(.Q(q[14]), .C(clk), .D(q[13]));
        df001 q_reg_15(.Q(q[15]), .C(clk), .D(q[14]));
        df001 q_reg_16(.Q(q[16]), .C(clk), .D(q[15]));
        df001 q_reg_17(.Q(q[17]), .C(clk), .D(q[16]));
        df001 q_reg_18(.Q(q[18]), .C(clk), .D(q[17]));
        df001 q_reg_19(.Q(q[19]), .C(clk), .D(q[18]));
        df001 q_reg_20(.Q(q[20]), .C(clk), .D(d));
        df001 q_reg_21(.Q(q[21]), .C(clk), .D(q[20]));
        df001 q_reg_22(.Q(q[22]), .C(clk), .D(q[21]));
        df001 q_reg_23(.Q(q[23]), .C(clk), .D(q[22]));
        df001 q_reg_24(.Q(q[24]), .C(clk), .D(q[23]));
        df001 q_reg_25(.Q(q[25]), .C(clk), .D(q[24]));
        df001 q_reg_26(.Q(q[26]), .C(clk), .D(q[25]));
        df001 q_reg_27(.Q(q[27]), .C(clk), .D(q[26]));
        df001 q_reg_28(.Q(q[28]), .C(clk), .D(q[27]));
        df001 q_reg_29(.Q(q[29]), .C(clk), .D(q[28]));
        df001 q_reg_30(.Q(q[30]), .C(clk), .D(d));
        df001 q_reg_31(.Q(q[31]), .C(clk), .D(q[30]));
```

```
df001 q_reg_32(.Q(q[32]), .C(clk), .D(q[31]));
df001 q_reg_33(.Q(q[33]), .C(clk), .D(q[32]));
df001 q_reg_34(.Q(q[34]), .C(clk), .D(q[33]));
df001 q_reg_35(.Q(q[35]), .C(clk), .D(q[34]));
df001 q_reg_36(.Q(q[36]), .C(clk), .D(q[35]));
df001 q_reg_37(.Q(q[37]), .C(clk), .D(q[36]));
df001 q_reg_38(.Q(q[38]), .C(clk), .D(q[37]));
df001 q_reg_39(.Q(q[39]), .C(clk), .D(q[38]));
df001 q_reg_40(.Q(q[40]), .C(clk), .D(d));
df001 q_reg_41(.Q(q[41]), .C(clk), .D(q[40]));
df001 q_reg_42(.Q(q[42]), .C(clk), .D(q[41]));
df001 q_reg_43(.Q(q[43]), .C(clk), .D(q[42]));
df001 q_reg_44(.Q(q[44]), .C(clk), .D(q[43]));
df001 q_reg_45(.Q(q[45]), .C(clk), .D(q[44]));
df001 q_reg_46(.Q(q[46]), .C(clk), .D(q[45]));
df001 q_reg_47(.Q(q[47]), .C(clk), .D(q[46]));
df001 q_reg_48(.Q(q[48]), .C(clk), .D(q[47]));
df001 q_reg_49(.Q(q[49]), .C(clk), .D(q[48]));
df001 q_reg_50(.Q(q[50]), .C(clk), .D(d));
df001 q_reg_51(.Q(q[51]), .C(clk), .D(q[50]));
df001 q_reg_52(.Q(q[52]), .C(clk), .D(q[51]));
df001 q_reg_53(.Q(q[53]), .C(clk), .D(q[52]));
df001 q_reg_54(.Q(q[54]), .C(clk), .D(q[53]));
df001 q_reg_55(.Q(q[55]), .C(clk), .D(q[54]));
df001 q_reg_56(.Q(q[56]), .C(clk), .D(q[55]));
df001 q_reg_57(.Q(q[57]), .C(clk), .D(q[56]));
df001 q_reg_58(.Q(q[58]), .C(clk), .D(q[57]));
df001 q_reg_59(.Q(q[59]), .C(clk), .D(q[58]));
df001 q_reg_60(.Q(q[60]), .C(clk), .D(d));
df001 q_reg_61(.Q(q[61]), .C(clk), .D(q[60]));
df001 q_reg_62(.Q(q[62]), .C(clk), .D(q[61]));
df001 q_reg_63(.Q(q[63]), .C(clk), .D(q[62]));
df001 q_reg_64(.Q(q[64]), .C(clk), .D(q[63]));
df001 q_reg_65(.Q(q[65]), .C(clk), .D(q[64]));
df001 q_reg_66(.Q(q[66]), .C(clk), .D(q[65]));
df001 q_reg_67(.Q(q[67]), .C(clk), .D(q[66]));
df001 q_reg_68(.Q(q[68]), .C(clk), .D(q[67]));
df001 q_reg_69(.Q(q[69]), .C(clk), .D(q[68]));
df001 q_reg_70(.Q(q[70]), .C(clk), .D(d));
df001 q_reg_71(.Q(q[71]), .C(clk), .D(q[70]));
df001 q_reg_72(.Q(q[72]), .C(clk), .D(q[71]));
df001 q_reg_73(.Q(q[73]), .C(clk), .D(q[72]));
df001 q_reg_74(.Q(q[74]), .C(clk), .D(q[73]));
df001 q_reg_75(.Q(q[75]), .C(clk), .D(q[74]));
df001 q_reg_76(.Q(q[76]), .C(clk), .D(q[75]));
df001 q_reg_77(.Q(q[77]), .C(clk), .D(q[76]));
```

```
        df001 q_reg_78(.Q(q[78]), .C(clk), .D(q[77]));
        df001 q_reg_79(.Q(q[79]), .C(clk), .D(q[78]));
endmodule
```

VITA

Ting-Chang (Tom) Chen

Candidate for the Degree of

Master of Science

Thesis: AN INVESTIGATION OF THE SIMULATION PERFORMANCE OF VERILOG FOR LARGE CIRCUITS

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Born in Tainan County, Taiwan, Dec. 26, 1970; the son of Chen, Wen-Liang and Shieh, Shu-Ju.

Education: Graduated from National Kaohsiung Institute of Technology, Kaohsiung City, Taiwan, in June 1991; received Bachelor of Science degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma, in August 2003; completed the requirements for the Master of Science degree with a major in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma, in May 2005.

Experience:

Technical Support Engineer: Siemens Limited, Taiwan (Feb. 1998 – July 2000): Trainer of PLC, Call-desk service, Lecturer of workshop, Field trouble shooting, Technical coordinator of distributors, PLC program design

Project Engineer: Formosa Chemical & Fiber Co., Taiwan (April 1995 - Jan, 1998): Controlling system design, PLC program design, Director of construction and commissioning

PC Sales & Maintainer: Heh-Hwang Computer Company, Taiwan (May - Nov. 1994): Sales of personal computer, Customer support and maintainer

Engineer: Ching-Jyi Machine Company, Taiwan (Sep. 1993 – May 1994): Designer & assembler for PLC controlling system, PLC program design and commissioning