國立台灣大學電機工程研究所碩士論文

指導教授：郭斯彥教授

# 暫存器移轉階層之編譯程式模擬技術

# A Compiled-Code Simulation Technique for RTL Designs

研究生：張凱揮　撰

中華民國九十年六月

國立台灣大學電機工程學研究所

碩士學位論文

# 暫存器移轉階層之編譯程式模擬技術

# A Compiled-Code Simulation Technique for RTL Designs

研 究 生：張凱揮

指導教授：郭斯彥 博士

經考試合格特此證明

碩士學位論文考試委員：

郭斯彥　　王國禎

陳良基

指導教授　：郭斯彥

研究所所長：王維新

中華民國九十年六月

# 誌謝

感謝郭斯彥教授兩年來之悉心指導，讓我能順利完成碩士學位。

感謝亞睿系統設計的黃薈萊博士，提供他的經驗和環境讓我得以對論文的題目做深入的研究。感謝葉宜忠學長提供協助以及在我有疑問時熱心的解答，使我節省了許多摸索的時間。

最後要感謝我的家人和朋友，在他們的支持和鼓勵下讓我能順利的成長，得到目前的成就。

# 摘要

隨著超大型積體電路技術的進步，邏輯模擬日趨重要，許多人也發展了不同的加速模擬的方法，編譯程式模擬技術就是其中之一。前人提出的方法多半是注重於邏輯閘階層。而在現今的設計中，暫存器移轉階層的模擬越來越重要，所以在本論文中作者提出了一個方法以加速暫存器移轉階層及邏輯閘階層的模擬。

目前有一些商業化的編譯程式模擬器，如 Cadence 公司的 NC-Verilog 及 Synopsys 公司的 VCS。他們的方法是重寫一個全新的模擬器，並直接產生機械碼。這種方法的好處是效能可以被最佳化，但缺點是重新開發新的模擬器要很長的時間，而且維護兩組不同的模擬器也需要比較多的人力。

在編譯程式技巧的使用上我們選擇了不同的道路。從以前的分析中，我們發現在模擬時，大部份的時間不是花在排程上，而是花在值的運算上，所以我們決定把直譯式的排程器留下，而將每個設計特有的運算部份寫出為 C 程式碼。如此可以不需要花太多的工夫重寫一個新的模擬器，但仍能得到速度的提升。雖然目前速度的提升仍未能和商業軟體相比，但已較原來的版本有極大的改進。未來如果直接輸出機械碼，效率應可接近於商業化的軟體。

從實驗結果可以看到模擬時間隨著所模擬設計的不同縮短了 25%到 79%。對運算較複雜的設計而言，速度的提升也更多。

# 第一章 導論

隨著積體電路產業的發展，積體電路模擬的問題越來越重要，許多加速模擬的方法也一一被提出。根據模擬演算法的不同，可以分為事件觸發和全模擬兩種。其中事件觸發只模擬輸入改變的邏輯單元，而全模擬是每一次模擬整個電路。根據模擬器設計的不同，又可以分為直譯式和編譯式。在直譯式中有一個中央排程器對資料結構做讀取並運算，在編譯式中則是將運算和資料結構直接轉換為程式碼。

在本論文中將介紹一編譯式模擬器。它是從一個 Verilog 的直譯式模擬器 VCK(Verilog-C Kernel)更改而來。在更改的過程中，我們保留了直譯式的排程器，而將運算的部份轉換為 C 程式碼。

# 第二章 編譯程式概念

在前人的研究中，Lewis 將電路分階層，從輸入起每經過一個邏輯單元和一個輸出節點為一階層。在模擬時依邏輯相、節點相的順序交替模擬，但此種方法只能用在單延遲的模擬中。在多延遲的模擬中，需要一個時間輪來紀錄事件將發生的時間。在一個邏輯單元模擬後，事件將依其延遲插入時間輪相對應的地方。排程器則根據時間輪觸發事件。

在現今的電腦架構中，快取記憶體的使用非常普遍，所以一些傳統的編譯程式演算法會導致效能不如預期。因此 Maurer 提出了「影子演算法」，將資料結構以「影子」表示，以增加快取的使用率。

其他還有一些適用於邏輯匣層次的模擬加速方法。如逆轉演算法、跳躍程式等等。其中一些觀念將在未來的章節中被使用。

# 第三章 編譯程式前置處理

在 Verilog 的語法中，等式有平行處理和序列處理兩種。在編譯程式模擬器中，將全部轉換為序列處理，這個過程叫做「序列碼轉換」。轉換的方法可以利用加入臨時變數或是變更等式的順序來達成。

VCK 的資料結構是以「變數」為基礎，操作變數的運算元和變數合而成為「方程式」。由方程式和變數進一步合成「等式」。等式包含「簡單等式」和「條件等式」。條件等式會跟據條件的值選擇所要執行的等式，而簡單等式是不含條件，一定會執行的等式。許多會被相同條件觸發的等式合起來為一個集合，稱為「狀態」。一個狀態會被寫為一個 C 函式，稱為「狀態函式」

在直譯部份和編譯部份之間的資料結構連結上，狀態函式的指標被存在一個陣列中，直譯式排程器可以根據狀態的編號呼叫相對應的 C 函式。而其他的資料結構會在編譯時寫出模擬時讀回。其中靜態資料如模擬值、變數位元長度直接寫出讀入，而指標則是給每一個資料結構一個獨特的編號。在寫出時寫出編號，在讀回時先建立編號和指標的轉換表，執行時再根據編號查出指標。

# 第四章 編譯程式技術

編譯程式執行速度快的原因是因為在編譯時已知模擬線路的特性，所以可以用特殊且執行速度快的計算函式取代一般而執行速度慢的。所以除了對每一個要模擬的線路要產生出正確的編譯程式外，也要預先製作各種特殊的計算函式。

依目前的計算機架構而言，三十二位元以內的運算都同等複雜，所以計算函式以三十二位元為分界：三十二位元之下、剛好三十二位元、三十二位元之上以及剛好六十四位元都有其最佳化的函式。而在函式的最佳化過程中，編譯器的最佳化功能也可以用來協助最佳化函數的產生。

編譯程式的產生主要是要將資料結構的存取和使用轉換成相對應的 C 程式碼，所以要分別對方程式、簡單等式、條件等式做轉換的動作。資料結構都能轉換之後，編譯程式的產生就成功了。

在前人的研究中有以直接跳躍代替函式呼叫以加速模擬的方法。在標準的 C 中是不能達成的，但可以利用 GCC 的延伸功能達成，叫做「計算跳躍」。利用全區域陣列紀錄每個函數進入位置的指標，再利用全區域變數傳遞參數，就可以答成用直接跳躍取代函式呼叫的目的。

# 第五章 實驗結果

從實驗結果中，可以看出編譯程式的執行速度比直譯式快了百分之二十五至百分之七十九。而在和商業軟體的比較中，在運算很複雜的情形時，目前的速度只比商業軟體慢兩倍左右。在考慮商業軟體最佳化所花的時間和人力下，本論文提出的方法的確是一個易於實行而又能有效能提升模擬速度的方法。

至於計算跳躍的部份，效果並不令人滿意。在複雜運算指令集下利用跳躍取代函數呼叫可以增進執行效率，但在簡單運算指令集中，反而會使執行效率更差，因此目前的模擬器並未實作此一部份。

# 第六章 結論和未來展望

在本論文中提出了一個適用於暫存器移轉階層的編譯程式模擬技術。從實驗結果可以看出因為被模擬電路的不同，模擬時間節省了百分之二十五至百分之七十九不等，平均而言可以節省百分之四十八的模擬時間。一般而言，本論文中提出的方法對於運算複雜的電路最有效益。

編譯時間也是評論一個編譯程式模擬器好壞的重要指標。比較大的電路通常需要比較長的時間來編譯，但更多的 C 程式碼被產生也意味著更多的最佳化處理。如果執行時間夠長，模擬節省下來的時間通常會超過編譯所需的時間。

在未來，如果程式直接輸出機械碼而不是 C 程式碼，就可以省略掉編譯的時間，也可以達到更多的最佳化。二狀態的模擬也是未來可以發展的重點之一。

# Abstract

As the technology of VLSI improves, logic simulation is getting more and more important, and various techniques that increase the speed of simulation are developed. One of them is compiled-code technique. Previous works by Lewis or Maurer focused on gate-level designs. Because RTL simulation is getting important now, we proposed an approach that can accelerate the simulation of RTL designs as well as gate-level designs.

There are some compiled-code simulators available commercially like NC-Verilog from Cadence and VCS from Synopsys. Their approach is to write a new simulator which generates machine code directly. The advantage is that the performance can be optimized to a maximal. The disadvantage is that it takes a very long time to develop a totally new simulator, and takes more effort to maintain two different simulators.

We took a different approach on the use of compiled-code techniques. From some analysis, it is found that the most time consuming part in simulation is not on the scheduler. It is on the evaluations of expressions. Thus we decided to keep the interpretive scheduler and write out design-specific evaluations to C code. It takes much less effort than writing a totally new simulator, while exploiting the advantage of compiled-code techniques. The improvement in performance is significant, though not comparable with commercial products yet. In the future, if machine code is generated directly, the performance will get close to commercial products.

The experimental results show that the simulation time can be shortened between 25% to 79%, depending on the design. For circuits with more complex evaluations, the acceleration could be more significant.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

As the technology of VLSI improves, logic simulation is getting more and more important, and various techniques that increase the speed of simulation are developed. Those techniques can be classified into two categories [1]. In interpretive simulation, a central scheduler traverses the data structures and calls the evaluation routines iteratively. In compiled-code simulation, the fact that the design will not change during simulation is used and can produce customized program that runs efficiently. Previous works on compiled-code simulation include compiled-code event-driven logic simulator [2], Shadow algorithm [3], Inversion algorithm [4], [5], and Branching Programs [6]. All of them focused on gate-level simulation. In the current design strategy, register transfer level (RTL) descriptions are used to verify the correctness of the algorithm or to be synthesized into gate-level circuits. So the RTL simulation is also very important. There are some commercial compiled-code simulators like NC-Verilog from Cadence [7] and VCS from Synopsys. Their approach is to write a new simulator which generates machine code directly. The advantage is that the performance can be optimized to a maximal. The disadvantage is that it takes a very long time to develop a totally new simulator, and takes more effort to maintain two different simulators. In this thesis, we proposed an RTL compiled-code technique, which can accelerate the simulation speed of RTL designs as well as gate-level designs.

The compiled-code simulator in this thesis is modified from an interpretive Verilog simulator named VCK (Verilog-C Kernel). In the interpretive version, the original Verilog design is converted to sets of assignments. Assignments in the same set, which is called a “state” in VCK, have the same triggering conditions. In compiled-code version, those assignments are converted to C code, and each state is converted to a C routine.

The central scheduler of the interpretive simulator is kept in the compiled-code version. Only the evaluations are converted to C code, which make the conversion easy while taking the advantage of compiled-code technique. The idea and the block

diagram of interpretive and compiled-code VCK is shown in Figure 1-1.

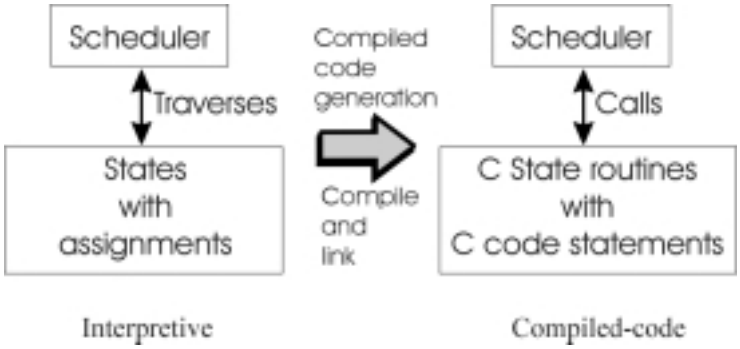


**Figure 1-1. VCK block diagram**

The concept, algorithm and techniques used in the compiled-code simulator will be described in this thesis in different chapters. In the second chapter, previous works on compiled-code techniques and other simulation acceleration methods are introduced. In the third chapter, the preprocessing and conversion of the data structures are described. In the fourth chapter, the compiled code techniques and algorithms are illustrated. In the fifth chapter, some experiments are conducted, and the results are compared. In the sixth chapter, conclusions are given and future work is discussed.

# Chapter 2

# Compiled-code Concepts

## 2.1 Logic Simulation Algorithms

Simulation programs can be characterized by the algorithm used, which is either oblivious or event-driven simulation [1]. In oblivious simulation, all gates are simulated at each time point, and produce new states for the next time point. In event-driven simulation, changes in network state are recorded, and only those gates that might cause a change in network state during a given time point are simulated. Event-driven simulation is potentially more efficient than oblivious simulation because it only evaluates gates that might change, but the overhead of keeping track of those gates is also a concern.

There are two implementation techniques that are in common use. The first one is interpretive simulation, and the second one is compiled simulation. In interpretive simulation, a data structure that represents the network is constructed. A central scheduler iterates over simulated time, calling procedures that evaluate the network elements. In compiled simulation, it uses the fact that network structure does not change during simulation, and produces a customized program that efficiently simulates the network. In the extreme form of optimization, all of the loops in the simulator are unrolled, and pointers to data structures are followed. This produces a program with almost no branch and with all data addressed directly by the code, and leads to much more efficient simulation.

The difference of performance between event-driven and oblivious simulation depends on the network activity. Let us assume that an event-driven simulator executes $n_e$ instructions when evaluating a gate, and that an oblivious simulator executes $n_o$ instructions per gate evaluation, and that the probability of a gate causing a new event during a simulation time point is $p$. The event-driven simulator executes $n_e \times p$ instructions for the evaluation of a gate, while the oblivious simulator executes $n_o$. Clearly, if ($n_e \times p < n_o$), or $p < (n_o / n_e)$, then event-driven simulator will be more efficient than oblivious one. Usually it takes hundreds of instructions for an

interpretive event-driven simulator to evaluate a gate, and it takes only a few instructions for a compiled oblivious simulator to evaluate a gate. So interpretive event-driven simulator can only be more efficient if *p* is very low, which is roughly 1%, suggesting that event-driven simulator is rarely the more efficient approach. When event-driven algorithm is implemented with compiled-code, it usually takes a few tens of instructions to evaluate a gate, making the threshold value of *p* from 4% to 10%, which is usually more efficient than oblivious one.

# 2.2 Compiled-code Simulation

Although the compilation/interpretation and event-driven/oblivious issues are independent, this has rarely been recognized in early simulators, until the time when Lewis proposed a hierarchical compiled-code event-driven logic simulator [2]. In his simulator, he combined compiled-code and event-driven simulation and produced a very efficient logic simulator. Lewis' algorithm is very important and worth a detailed description.

## 2.2.1 Algorithm

Lewis' simulator is named Tortle, it uses two-phase event-driven unit delay simulation algorithm to perform one time point of simulation. The first phase is node phase, which is also called event phase, and the second phase is gate phase, which is also called evaluation phase. The algorithm is shown below.

```
fanout:
for each node in the active_nodes list
     update node_val from next_node_val
     add fanouts of the node to active_gates list
active_nodes list <- {}
simulate:
for each gate in the active_gates list
     simulate the gate
     for each output node of the gate
          if the node value has changed
               save the new value in next_node_val
               add the node to the active_nodes list
active_gates list <- {}
```

The node phase corresponds to the code labeled "fanout:", and the gate phase

corresponds to the code labeled “simulate:”. There are two lists which represent the state of the network. One of them is for the active nodes, and the other one is for the active gates. At each time point, nodes in active nodes list are scanned and their fanouts are added to the active gates list. The logic value of each node is also updated from next_node_val which is calculated for the next time point. The active nodes list is then cleared. The active gates list is then scanned. Each active gate is simulated. If the logic value of any node has been changed, its value will be stored in next_node_val, and the node will be added to the active nodes list. Then the list of active gates is cleared.

In order to implement the algorithm efficiently, a simple list implementation is required for the performance of the simulation, and it is shown below. The list is represented by two arrays *flags* and *items*, and a counter *n* is used to point to the last item in the stack. To add an item *x* to the list, the following code is used:

```
if (!flags[x])
{
     flags[x] = TRUE;
     items[n++] = x;
}
```

This code checks if the item is already in the list. If not, append it to the list, and set the flag.

The first step to convert the algorithm into a compiled one is to generate a customized procedure for the fanout operation associated with each node, and for the simulate operation associated with each gate. The two lists contain the addresses of the procedures that need to be called by the central scheduler. The fanout procedures append addresses of simulation procedures that need to be called to the active nodes list, and the simulate procedures append the addresses of fanout procedures that need to be called to the active gates list. The central scheduler then calls the two lists in turn to simulate the logic net. The customized code is more efficient because the compiler calculates many addresses of the data structures statically which otherwise will need to be traversed at simulation time. Figure 2-1 gives a sample circuit and the code that follows shows the code generated for node 3 and gate 1 with the scheduler calling these procedures.

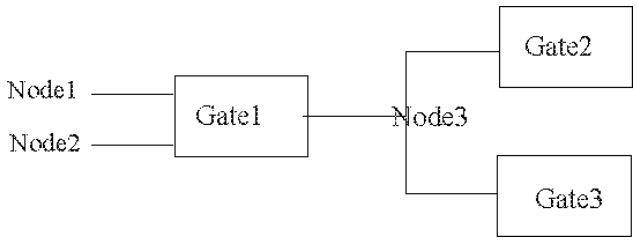


**Figure 2-1. Example circuit**

```
fanout_3()
{
     node_val[3] = next_node_val[3];
     node_active[3] = FALSE;
     if (!gate_active[2])
     {
          gate_active[2] = TRUE;
          *gate_active_ptr++ = &simulate_2;
     }
     if (!gate_active[3])
     {
          gate_active[3] = TRUE;
          *gate_active_ptr++ = &simulate_3;
     }
}

simulate_1()
{
     gate_active[1] = FALSE;
     result = evaluate(node_val[1], node_val[2]);
     if (result != next_node_val[3])
     {
          next_node_val[3] =  result;
          if (! node_active[3])
          {
               node_active[3] = TRUE;
               *node_active_ptr++ = &fanout_3;
          }
     }
}

simulate()
{
```

```
    node_active_ptr = node_active_list;
    while (node_active_ptr != node_active_list_end)
        (*node_active_ptr++)();
    gate_active_ptr = gate_active_list;
    while (gate_active_ptr != gate_active_list_end)
        (*gate_active_ptr++)();
}
```

In this version of compiled-code simulator, the overhead due to procedure calls is considerable. This overhead can be eliminated with a technique called “threaded code” [5]. Threaded code is an implementation with two optimizations: tail recursion optimization and loop unrolling. In the current version of simulator, procedures return to the scheduler and then the scheduler calls another procedure. In threaded code version, the procedure jumps directly to the next procedure, so that procedure calls are eliminated. The threaded code version of the same net is described below.

```
fanout_3:
{
    node_val[3] = next_node_val[3];
    node_active[3] = FALSE;
    if (!gate_active[2])
    {
        gate_active[2] = TRUE;
        *gate_active_ptr++ = &simulate_2;
    }
    if (!gate_active[3])
    {
        gate_active[3] = TRUE;
        *gate_active_ptr++ = &simulate_3;
    }
    node_active_ptr++;
    goto *node_active_ptr;
}

simulate_1:
{
    gate_active[1] = FALSE;
    result = evaluate(node_val[1], node_val[2]);
    if (result != next_node_val[3])
    {
        next_node_val[3] =  result;
```

```
            if (! node_active[3])
            {
                node_active[3] = TRUE;
                *node_active_ptr++ = &fanout_3;
            }
        }
        gate_active_ptr++;
        goto *gate_active_ptr;
    }

    simulate()
    {
            *node_active_ptr++ = &done_fanout;
            gate_active_ptr = gate_active_list;
            goto *node_active_list;
        done_fanout:
            *gate_active_ptr++ = &done_simulate;
            node_active_ptr = node_active_list;
            goto *gate_active_list;
        done_simulate:
    }
```

ANSI C does not support jumping to an address saved in a pointer, but most modern C compilers have the option to optimize procedure calls and produce threaded-code machine codes. A C extension provided by GCC called “computed-goto” can also be used to implement threaded-code programs.

## 2.2.2 Example Circuit

Now let us see an example step by step to further illustrate the algorithm. The example circuit is given in Figure 2-2.

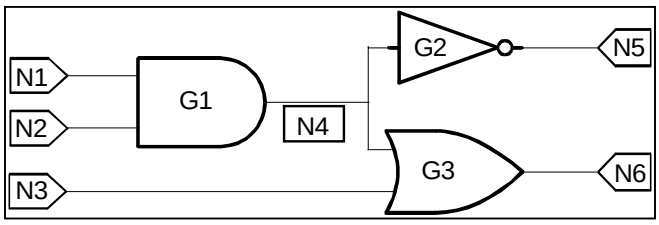


**Figure 2-2. Example circuit**

Assume that the procedures for Nodes N1, N2, … , N6 are fanout_1, fanout_2, …, fanout_6, respectively. The procedures for G1, G2 and G3 are simulate_1, simulate_2, and simulate_3, respectively. The value of node_active or

gate_active is represented by shading the gate or node. Shaded gate or node means the value of its gate_active or node_active is true. The transition "A→B" above the wire means the node_val and next_node_val of the node. It is represented as "next_node_val → node_val". The value above the input wire is its current input. See Figure 2-3 to 2-16 and Table 2-1 for detailed steps.

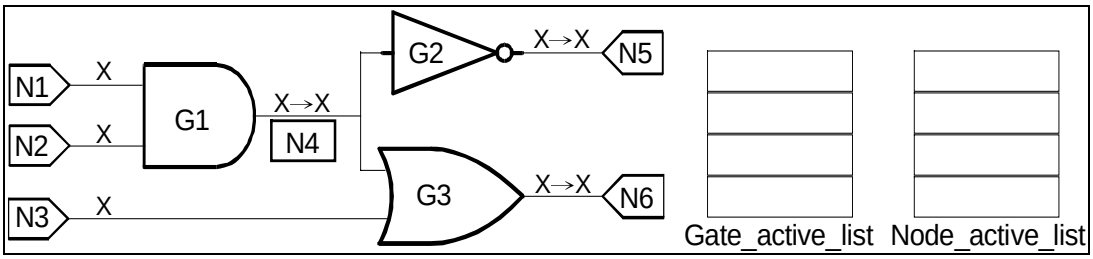


**Figure 2-3. Initial condition**

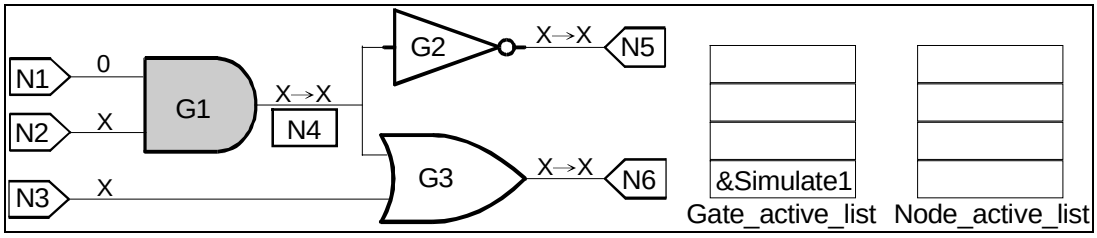


**Figure 2-4. First step**

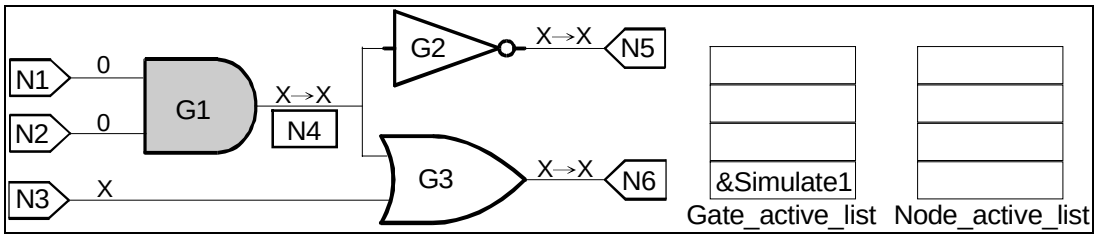


**Figure 2-5. Second step**

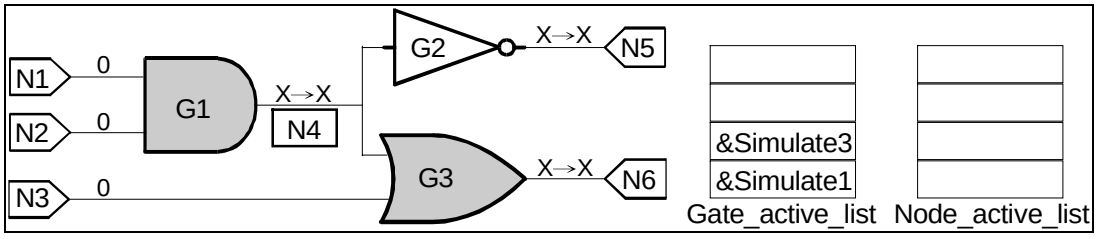


**Figure 2-6. Third step**

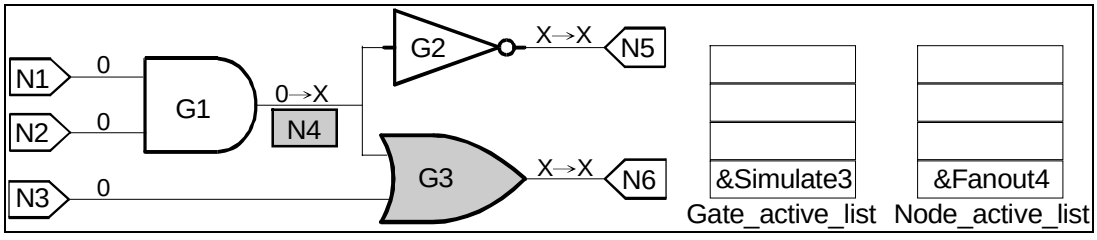


**Figure 2-7. Forth step**

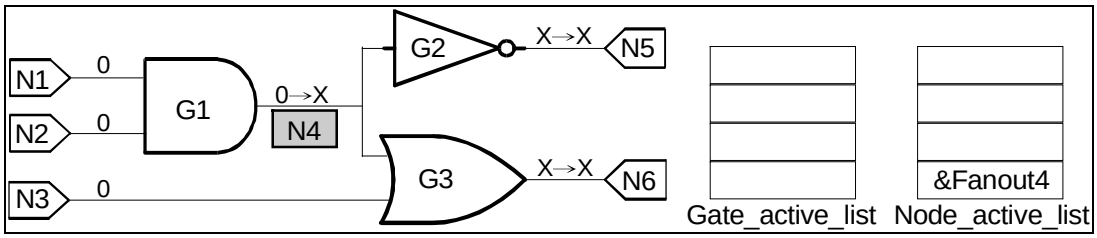


**Figure 2-8. Fifth step**

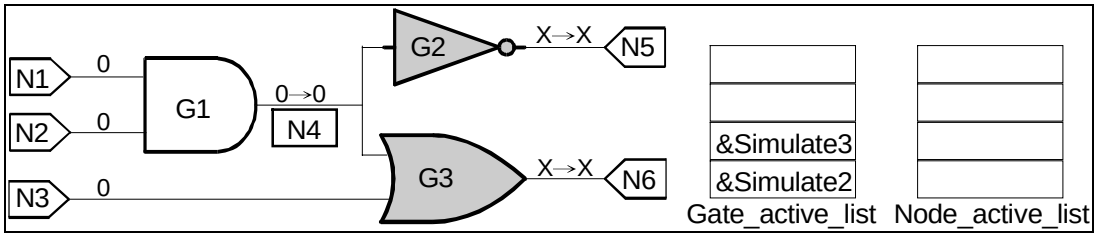


**Figure 2-9. Sixth step**

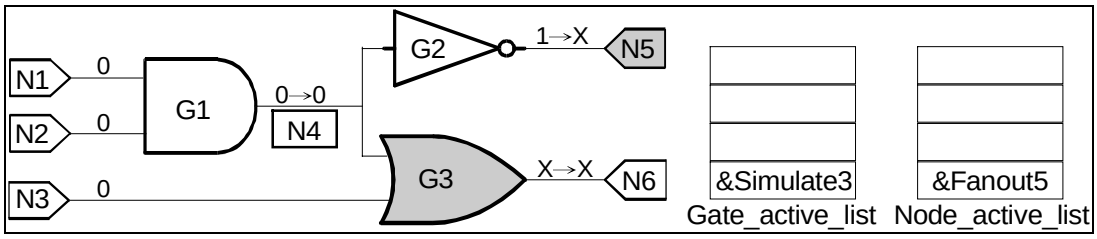


**Figure 2-10. Seventh step**

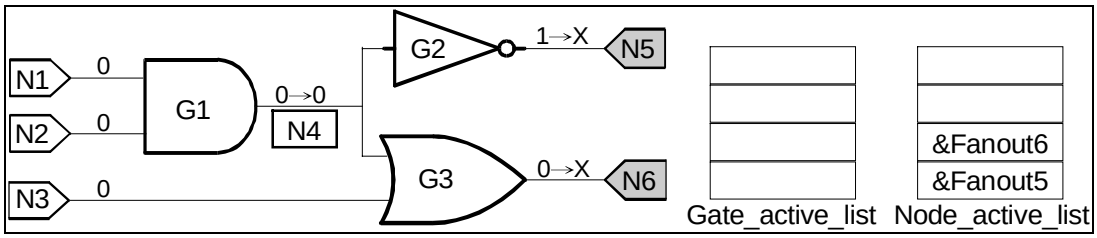


**Figure 2-11. Eighth step**

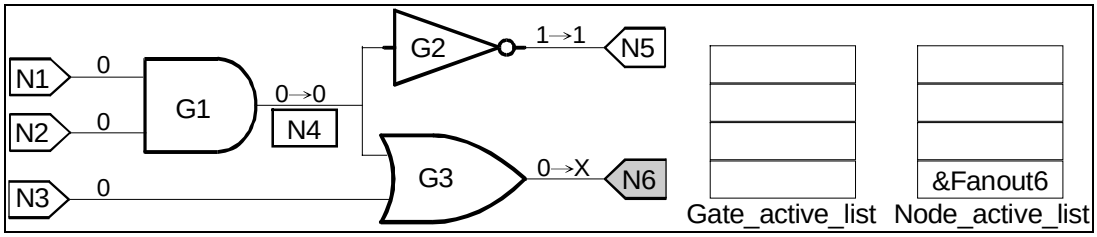


**Figure 2-12. Ninth step**

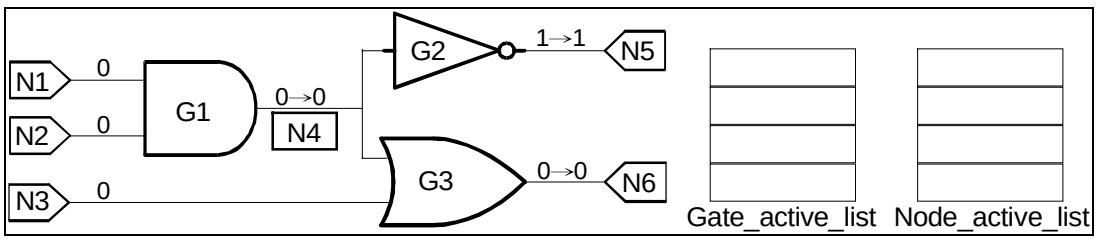


**Figure 2-13. Tenth step**

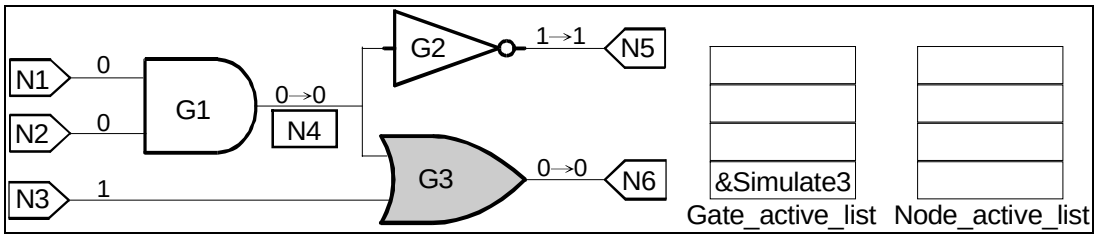


**Figure 2-14. Eleventh step**

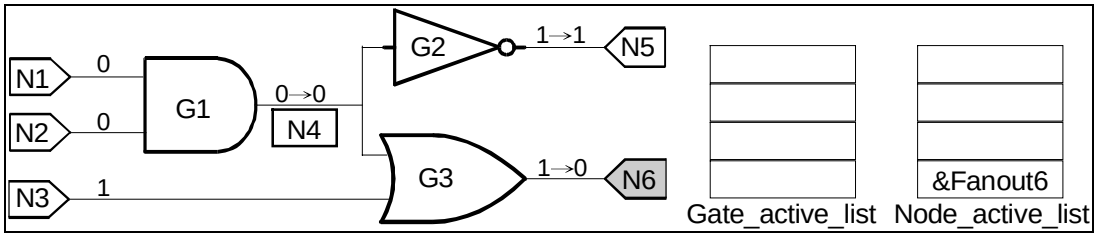


**Figure 2-15. Twelfth step**

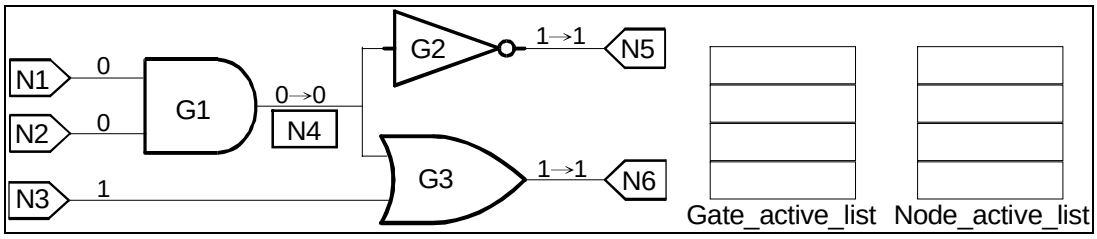


**Figure 2-16. Thirteenth step**

The actions performed in each step can be summarized in the following table:

**Table 2-1. Steps of simulation**

| Step | Actions performed |
|---|---|
| 1 | The value of node 1 is changed to 0, gate 1 is marked active, simulation routine "&simulate1" for gate 1 is added to the gate_active_list queue |
| 2 | The value of node 2 is changed to 0, gate 1 is already active so no further action is taken |
| 3 | The value of node 3 is changed to 0, gate 3 is marked active, simulation routine "&simulate3" for gate 3 is added to the gate_active_list queue |
| 4 | Simulation routine for gate 1 is removed from the gate_active_list queue and executed, next_node_val of node 4 is changed to 0, gate 1 is marked inactive, node 4 is marked active and the simulation routine of node 4 "&fanout4" is added to the node_active_list queue |
| 5 | Simulation routine for gate 3 is removed from the gate_active_list queue and executed, gate 3 is marked inactive, next_node_val of the gate is still X so no further action is taken |
| 6 | Simulation routine for node 4 is removed from the node_active_list queue and executed, the value of the node is changed to 0, gate 2 and gate 3 are marked active, the corresponding simulation routines are added to the gate_active_list queue |

| | |
|---|---|
| 7 | Simulation routine for gate 2 is removed from the gate_active_list queue and executed, gate 2 is marked inactive, next_node_val of node 5 is set to 1, node 5 is marked active and the simulation routine for node 5 is added to the node_active_list |
| 8 | Simulation routine for gate 3 is removed from the gate_active_list queue and executed, gate 3 is marked inactive, next_node_val of node 6 is set to 0, node 6 is marked active and the simulation routine for node 6 is added to the node_active_list |
| 9 | Simulation routine for node 5 is removed from the node_active_list and executed, node 5 is marked inactive, and its value is set to 1 |
| 10 | Simulation routine for node 6 is removed from the node_active_list and executed, node 6 is marked inactive, and its value is set to 0 |
| 11 | Node 3 is set to 1, gate 3 is marked active, simulation routine for gate 3 is added to the gate_active_list queue |
| 12 | Simulation routine for gate 3 is removed from the gate_active_list and executed, its next_node_val is set to 1, node 6 is marked active, and the simulation routine for node 6 is added to the node_active_list queue |
| 13 | Simulation routine for node 6 is removed from the node_active_list queue, node 6 is marked inactive, and its value is set to 1 |

# 2.3 Multi-delay Algorithms

Every gate introduces a delay to the signals propagating through it. In modeling the behavior of a gate [10], the function and the delay is separated as indicated in Figure 2-17. After the logic function of the gate is evaluated, the calculation of the delay is performed.

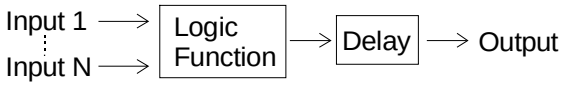


**Figure 2-17. Separation between function and delay in modeling a gate**

If the results of the simulation of the gates go transparently to the output without any delay, it is called zero-delay model. If all the gates have the same delay, the delay can be scaled to one and this model is called unit-delay. If the gates in a circuit have different delays, then it is called multi-delay model.

The compiled-code simulator described earlier uses a unit-delay timing model. In unit-delay model, the value of any node can only change at the next time point. It cannot change at two or more time units later. Clearly, this model is only useful for simple functional logic verification. Real circuits need more complex timing model to describe the delay of logic gates, which is multi-delay model.

In event-driven unit-delay model, there are only two queues: the gate queue and the event queue. In the gate phase, all the gates in the gate queue are simulated and

produce a queue of events; while in the event phase, all the events in the queue are processed and produce a queue of gates. Because there are only two fixed queues, the generation and storage of events and gates are simple. Furthermore, it is possible to suppress the generation of an event if the new value from the simulated gate is identical to its old value.

Event-driven multi-delay model is also a two-phase process, but because gates may have different delays, events may stay in the queue for many cycles. Therefore the simulator must keep track of the current time, and the event phase should only process the events that correspond to that time. Furthermore, the suppression of the generation of events is no longer simple.

Since only the events that correspond to the current time will be processed, it is necessary to sort the events according to time. The simplest way of sorting the events is the use of time-wheel [9]. The time-wheel is an array of event queues, with each queue corresponding to a single time. Usually the maximum delay is longer than the length of the time wheel, so the time wheel is usually implemented by a circular list. The position of the event queue is indexed by the time modulo the length of the time-wheel.

Because the number of events generated in each phase is not a constant, the queues in the time-wheel are usually implemented by linked lists. The structure of the event queue is illustrated in Figure 2-18.

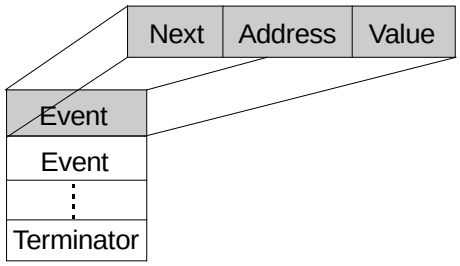


**Figure 2-18. The event queue**

There are three fields in an event item. The “next” field points to the next event, the “address” field points to the address of the event routine, and the “value” field keeps the value of the node.

In the event-phase, it first selects the correct queue that corresponds to the current time, then executes all the event routines found in the event queue. It first pops an event from the queue, executes the routine associated with the event, and then

pops and jumps to the next event in the queue, until the terminator is reached. After processing each event, the value of the result is compared to the value stored in the event. If these two values are different, the value stored in the event is updated, and the gates in the fan-out set of the event is added to the gate queue. In order to avoid adding the same gate to the gate queue twice, a valid flag is usually used. An example event handler is given below.

```
new_value = TIME_WHEEL[current_time]^.value;
POP TIME_WHEEL[current_time];
if new_value ≠ net_value then
     net_value := new_value;
     PUSH gate_1;
     PUSH gate_2;
     ... ...
     go to TIME_WHEEL[current_time]^.routine;
endif
```

The final entry in the event queue is the terminator which consists of a simple jump to the first gate simulation routine and is never deleted.

When a gate simulation routine is executed, it first pops itself off the gate queue. It then starts simulation using the current value of the inputs and stores the value of its outputs in temporary variables. It then creates events for each output and inserts them into correct event queue. To do that, it first calculates the delay of the gate, modulo it to the size of the time wheel, and then the number will be the index to the correct queue. After inserting the events, the routine branches to the next gate simulation routine in the gate queue. An example gate simulation routine is given below.

```
POP GATE_QUEUE;
temp_value := EVALUATE(inputs);
new_queue := (current_time + gate_delay) MOD time_wheel_size;
PUSH output_event_handlers and temp_value onto TIME_WHEEL[new_queue];
go to GATE_QUEUE^.routine;
```

The final entry in the gate queue is also a termination routine, but this routine is more complex than the event termination routine. The gate termination routine needs to check if there is any event in the event queue. If yes, it increments the current time, finds the correct event queue in the time wheel, and then branches to the first event simulation routine in the event queue. If not, it either stops the simulation or reads the

next input vector and starts from time zero again.

To begin the simulation, the simulator reads a vector of inputs and sets the current time to zero. It then compares the new input values with current values. If any value is different, the event associated with the input is queued for the net at time zero. Finally, the scheduler selects the first routine in the time-zero event queue and branches to the routine.

# 2.4 Cache Effects

As the speed of CPU improves, cache is getting more and more important. Some traditional acceleration used in compiled-code simulators is now no longer beneficial. For example, creating a routine for every gate makes the routine specific to the gate so that optimization can be done. But it also means that the routine will only be executed once in each time unit, which makes the cache useless. For another example, threaded code technique eliminates procedure calls, but direct jump to an address far away makes the pre-paging of cache fail and causes high cache miss rates. Since caches can be several times faster than main memory, some new techniques are developed to use caches efficiently.

## 2.4.1 General Techiniques

To deal with the problems compile-code techniques brought to cache, there are some solutions. The first one is not to create specific routines for each gate, but to create some generic routines instead. For example, all AND gates with two inputs can share the same routine. In this way, spatial locality is improved and there will be much less cache misses.

The second technique is to re-roll part of the routings. For example, some "for-loops" which has been unrolled is now re-rolled. Unrolled loops are usually much larger and are hard to keep in the cache, while re-rolled loops can usually be run entirely in the cache. It depends on the size and the count of a loop to determine whether the loop should be unrolled or not. No matter how, it is not always good to unroll loops in a computer with cache.

## 2.4.2 The Shadow Algorithm

In the shadow algorithm [3], each gate and each net in the circuit being simulated

is represented by a data structure called shadow. Shadow contains information specific to the gate or net, such as fan-in and fan-out counts. It also contains the addresses of simulation routines for the gate or net.

The shadow algorithm can be used in either compiled or interpreted simulators. In compiled simulators, simulation routine for each gate is unique; while in interpreted simulators, some generic routines are used in less common occurring objects such as gates with large number of inputs or nets with large number of outputs.

The simulation of an input vector is represented by a linked list of shadows which is created dynamically at simulation time. The shadow algorithm uses a method similar to Lewis' compiled-code simulator with threaded-code technique. That is, the shadows are managed as a linked-list queue. When simulation begins, it tests the primary inputs for changes, and then alternate between two phases to finish the simulation. The two phases are net simulation phase and gate simulation phase. In net simulation phase, it determines whether any change has taken place in the net, and adds gates to the linked list if necessary. In gate simulation phase, it adds new net structures to the net queue after each simulation. The final routine in either gate or net linked-list is the trailer routine, which switches from the current phase to the other phase and performs some housekeeping operations. Each simulation routine for the gate or net removes the head element from the queue, loads the next element, and branches to the processing routine for the new shadow.

The reason why the shadow algorithm is advantageous to a computer with cache is that it provides spatial and temporal locality. The shadow keeps all the information of the net or gate it represents, so all data needed in the simulation routine will be fetched into the cache together. Some routines are reused in the shadow algorithm, so there is a larger chance that the simulation routine being called is already in the cache.

## 2.5 Other Simulation Algorithms

Besides the algorithms mentioned above, there are some other algorithms worth introducing. Most of them focused on accelerating the simulation of logic gates using the characteristics of logic simulation. Some of the ideas are used later this thesis.

## 2.5.1 The Inversion Algorithm

The inversion algorithm [4] is an event-driven algorithm for digital simulation. It focused on eliminating unnecessary gate simulations if there is no change in the output of a gate. The basic principal of the algorithm is that no gate should be simulated unless its output is guaranteed to change. Therefore, when an event occurs on the input of a gate, it is necessary to determine if the change will propagate to the output of the gate, and suppress the gate simulation if no propagation will occur.

When an event occurs on the input of a gate, the inversion algorithm performs some tests to determine whether the event will propagate through the gate. The test differs according to the gate type. In the most basic form, the inversion algorithm supports eight gate types: AND, NAND, OR, NOR, XOR, XNOR, NOT and BUFFER. These types are enough for almost all gate-level circuits. However, it is not necessary to provide specific tests for each of the gate types. Only three sets of tests are enough. The first set is for NOT and BUFFER, the second set is for XOR and XNOR, and the third set is for AND, NAND, OR and NOR gates.

The tests for the XOR, XNOR, NOT and BUFFER are trivial because a change in any input will always change the output. But because the number of inputs is more than one in XOR and XNOR gates, two consecutive changes in the inputs will cancel each other and eliminate the need of an event. Therefore different tests are used for XOR and XNOR gates. In XOR and XNOR gates, it first tests if the output event is scheduled. If not, schedule it. If it is already scheduled, deschedule it. In NOT and BUFFER gates, the output event is always scheduled. The event handler for XOR and XNOR are given below.

```
if OutputEvent not scheduled then
     schedule OutputEvent;
else
     deschedule OutputEvent;
endif
```

The tests for AND, OR, NAND and NOR are more complex, and are based on the counting algorithm [11] originally described by Schuler. The original algorithm is shown below. It is assumed that there has been a change in an input X to the gate G. The dominant value is 1 for OR and NOR gates, and is 0 for AND and NAND gates.

```
if (Value.of.X == Dominant.Value.of.G) then
     Count.of.G = Count.of.G + 1;
     if (Count.of.G == 1) then
          Output.of.G = Dominant.Value.of.G;
else
     Count.of.G = Count.of.G – 1;
     if (Count.of.G == 0) then
          Output.of.G = not Dominant.Value.of.G;
```

The counting algorithm given above assigns a value to the output of G if and only if the output changes value. However, its usage in the inversion algorithm is different from the original one. It is used to predict changes rather than computing output values. The goal is to schedule and deschedule events. The underlying scheduling technique is based on the shadow algorithm, and each event is represented by the structure shown in Figure 2-19.

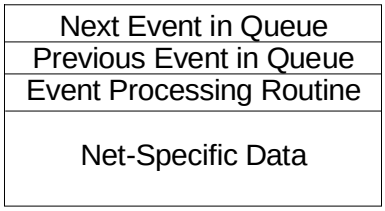


**Figure 2-19. Shadow structure of the inversion algorithm**

The inversion algorithm generates an event structure that is specific for each input net. Each of the event structure contains an indirect pointer to the event-processing routine for the net. The use of indirect pointer allows event handlers to be changed at run time. Since successive events on an input net will cause the gate to alternate between dominant and nondominant value, the test for dominant values can be eliminated. The inversion algorithm uses two event handlers, one for dominant values and the other for nondominant values. When the dominant-value event-handler executes, it replaces the address of event-processing-routine in the event structure with the address of the nondominant event handler. The nondominant-value event handler also does the same thing. The two event handlers thus execute alternatively and no test for dominant value is required. The event handlers for dominant and nondominant values are illustrated below.

Dominant:

```
decrement DominantCount;
if (DominantCount == 0) then
```

```
    if (OutputEvent not scheduled) then
        schedule OutputEvent;
    else
        deschedule OutputEvent;
EventProcessingRoutine = AddressOf NonDominant;
```

NonDominant:

```
increment DominantCount;
if (DominantCount == 1) then
    if (OutputEvent not scheduled) then
        schedule OutputEvent;
    else
        deschedule OutputEvent;
EventProcessingRoutine = AddressOf Dominant;
```

As the code shows, the simulation of a gate is reduced to simple inversion operations. Since the operation of the inversion algorithm does not need net values, most gate simulations can be eliminated entirely. Only the schedule and deschedule of events are necessary for the simulation to work. For a net which is visible to users, the net value must be kept, and another event handler to keep the correct net value is added to the simulation of the gate.

There are two passes in the inversion algorithm. A first pass to determine the dominant or nondominant of a net is required before the simulation starts. So a traditional simulation is necessary for the initialization. After the initialization, a second pass to simulate the circuit using the inversion algorithm can be started.

The inversion algorithm is competitive with levelized compiled code simulation, even at high activity rates. If the activity rate is reduced, the performance increases correspondingly, while the performance of levelized compiled code remains the same. The inversion algorithm can also be used interpretively without too much loss in performance. A modified version is proposed for three-value simulations [5], which makes this algorithm more capable for current designs. However, this algorithm focused on gate-level, and cannot be used in RTL simulation.

## 2.5.2 Branching Programs

To accelerate the evaluation of logic gates, there is another method called “branching programs” [6]. It focused on delay-independent cycle-based functional

verification. It is based on decision diagram, which converts logic computation to branch decisions according to the input. Branching program uses compiled-code technique and can run faster than levelized compiled-code simulation.

The basic of the branching program is the decision diagram [12]. Cerny [13] provided a method to derive branching programs from decision diagrams. Although his work was originally done for switch-level circuits, it can be applied to logic level simulation easily. A decision diagram evaluates a Boolean function by a sequence of decisions, which are usually two-way branches. Each decision is based on the value of one of the inputs. An example diagram for the function x1*x2+x3 is shown in Figure 2-20.

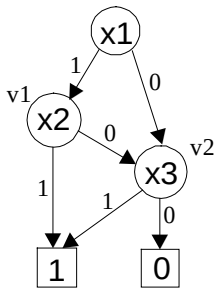


**Figure 2-20. Example decision diagram for x1*x2+x3**

The straightforward isomorphism between a decision diagram and a branching program has been pointed out by many researchers. A branching program evaluates its function using a sequence of multi-way branches, which depends on the value of one of the inputs. The branching program for Figure 2-20 is given below:

```
        if (x1) then
            goto v1;
        else
            goto v2;
v1:     if (x2) then
            return 1;
        else
            goto v2;
v2:     if (x3) then
            return 1;
        else return 0;
```

Since specialized program is created for each circuit, the branching program method is a form of compiled-code simulation. The number of branches depends on

the number of inputs instead of the number of gates, which is an advantage over gate simulation techniques like levelized compiled-code simulator. But there are still some problems to concern. The first one is how to extend the single output function to multiple outputs. The second one is that the size of the decision diagram grows rapidly, resulting in a program that can be hard to compile, hard to store and which causes memory thrashing. Therefore some approaches were proposed.

The approach proposed for the first problem is to use the characteristic functions for multi-output functions [14]. For the second problem, a set of decision diagrams is used instead of one. Between them, the characteristic functions in this set compute all the outputs. Thus the whole circuit is partitioned to many small circuits. Simulation between the partitioned circuits is done with the levelized compiled-code technique, where each partition is treated as a large gate. There is a trade off between the size of the program and the speed of the simulation. Larger partitions cause a larger program in size but runs faster, while smaller partitions cause a smaller program in size but runs slower. In the extreme case, if there is only one partition, then the whole circuit uses the branching program. In another extreme case, if there is a partition for each gate, then it is reduced to levelized compiled-code simulation.

There is an up to 10X speedup over levelized compiled code simulation using branching programs. The idea of branching programs is used in the expansion of conditional assignments in this thesis, which will be discussed in later chapters.

# Chapter 3

# Compile-code Preprocessing

## 3.1 Basic Concepts

The simulator described in this paper is called VCK (Verilog-C Kernel) [15] and uses Verilog hardware description language as its source language. Verilog is a complete and complex language which can be used to design and simulate the hardware. There are two levels of modeling in Verilog: the behavioral level modeling and the gate level modeling.

In behavioral modeling, the user can describe the behavior of the circuit using high-level statements like if-then or while-loops. The main purpose of this model is to verify the correctness of the algorithm or to use some other tools to synthesize the circuit into gate level.

In gate level modeling, the circuit is described in logic gates like and-gates, or-gates or some larger macros like full-adders. In cell-based designs, there is almost a direct mapping between the gate-level model and the layout of the chip. It is the most detailed description of a circuit before layout.

In order to simulate the hardware, there are some constructs that are not available in ordinary programming languages and these constructs must be added to the behavioral model. For example, all blocks in a hardware circuit are processing concurrently, so Verilog is basically a parallel processing language and provides parallel processing statements like non-blocking assignment, fork-join, etc. Hardware circuits usually depend on some signals to stimulate part of its logic, so Verilog provides a variety of timing controls like event control and delayed control.

No matter how complex the hardware description language is designed, the idea is very simple: The state of the circuit is stored in some variables. There are assignments between these variables. There are also some events waiting to trigger these assignments, no matter the triggers are from delayed or event control. What the simulator needs to do is to trigger the assignments at the right time and update the variables according to the evaluation results of the assignments.

In VCK, the source file will be converted into many blocks of statements called “states”. There are assignments, including simple assignments and conditional assignments, in the states. The assignments in a state share the same triggering conditions. For example, they are from the same delay control or they are triggered by the same event. The conditional assignment such as “if-then” or “mux” has a condition and can determine which assignments to be evaluated. There is a scheduler determining the state to be executed at a time point. The implementation of the scheduler is beyond the scope of this thesis and will not be further discussed. The author will focus on the assignments in a state and discuss the compiled-code techniques that applied to it.

There are two kinds of assignments in Verilog: The blocking assignments and the non-blocking assignments. In blocking assignments, the assignments are executed serially like ordinary programming languages. For example, in the following code:

```
a = b;
b = a;
```

Assume “a” is originally 1 and “b” is originally 2, then both a and b will become 2 after these assignments. To be more detailed, the value of “a” is first changed to the value of “b”, and then “b” is assigned the value of “a”, which is the original value of “b”. So both values in these two variables are the original value of “b”.

In non-blocking assignments, the assignments are executed concurrently and have no time dependency between them. For example,

```
a <= b;
b <= a;
```

“a” will get the value of “b”, and “b” will get the value of “a”. The two values are swapped. For example, in the beginning, if “a” is 1, and “b” is 2, then “a” will become 2 and “b” will become 1 after these assignments.

The most straightforward way to execute non-blocking assignments is to evaluate the RHS (right-hand-side) of the assignments, store the results in temporary variables, and then update all the variables in the LHS (left-hand-side) at the end of the time unit.

There are two ways to arrange the assignments in a state. The first one is to convert all the assignments to blocking ones, which can be executed serially. The

second one is to convert all the assignments to non-blocking ones, which need to store the results of the RHS of assignments and update the LHS at the end of the time unit. There is much more overhead in the second method, so the author decided to implement the first one. That is, convert the assignments to non-blocking ones so that each variable in the LHS can be updated when the assignment is executed. This process is called serial-code-conversion.

Blocking assignments in Verilog are essentially serial-code so no further processing is necessary. The problem that remains is to convert the non-blocking assignments into serial ones.

Executing non-blocking assignments sequentially certainly will produce incorrect results. In the value-swapping problem discussed above, executing the non-blocking assignments sequentially will cause the two variables to share the same value rather than swapping them. The solution is to add a temporary variable between the two assignments. Actually, adding a temporary variable to every assignment will solve the problem, but it is obviously too redundant. In the value-swapping problem, we need only one temporary variable instead of two. So an algorithm is designed to solve this problem. The algorithm will be described in section 3.3. Now the data structures used in VCK are first described as the base to understand the algorithms.

# 3.2 Data Structures

## 3.2.1 Variables

Variable is the basic data storage in VCK. It stores the simulated value and some other information of a variable in the Verilog design. A variable does not use other data structures and is used by other data structures like equations. The simulated value is stored in a data type called “value_t”, which stores the Verilog 4-value data.

## 3.2.2 Equations

There are two basic constructs in the RHS of assignments and in conditions: The variables (or operands) and the operations. In VCK, all assignments and conditions are combinations of two data structures: “equation” and “variable”. Variable has been discussed in the previous subsection. Equation is a data structure that connects the variables with the operations that manipulate them. Every equation has a value_t

called “sim_v” which keeps the current simulation value of the equation.

There are three types of equations: unary, binary and associative. Unary equations are operations that only have one operand, like “negation” and “reduction unary and”. Binary equations are operations that have two operands, like “division” and “bit-wise binary and”. Associative equations are operations that are associative, like “addition” and “multiplication”. Each equation has a field called “operation_type” that represents the operation of the equation. Equation also has fields that point to variables or other equations. Figure 3-1 shows some examples and illustrates the usage of them. The “operation_type” of the equation is enclosed in the parentheses. The variables are written directly using their original variable names. There are two pointers used in an equation to connect to other equations or variables, which are named “side” and “down”, and is represented in the graph by drawing “side to” or “down to” the equation.

| Type | Example | Representation |
|---|---|---|
| Unary | !a | equation(!)<br>\|<br>a |
| Binary | a / b | equation(/)<br>\|<br>a – b |
| Associative | a + b + c | equation(+)<br>\|<br>a – b – c |

**Figure 3-1. Equation examples**

In unary operations, the operand is attached under the equation and has its side empty. In binary operations, the operands are attached under the equation and form a chain. The second operand is chained on the side of the first operand. Some operations like addition and multiplication are associative so that all the operands are chained side by side to their previous operand. They are called associative equations.

## 3.2.3 Simple Assignments

Simple assignment is an assignment without any condition. The data structure is simple. There is a pointer to the variable in the LHS of the assignment, and a pointer to an equation which represents the RHS of the assignment. There is also some other information about the assignment, like bit-select or part-select, which selects only a portion of the bits to be assigned.

## 3.2.4 Conditional Assignments

Conditional assignment is an assignment with a condition. To execute a conditional assignment, the condition is first evaluated, and then the result is used to select a block of assignments to execute according to the tag of the block.

For example, in the following code:

```
if (a == 1)
     b = 2;
else
     b = 3;
```

In the conditional assignment, “a == 1” is the condition, “true” is the tag of the assignment “b = 2”, and “default” is the tag of the assignment “b = 3”. To execute the conditional assignment, “a == 1” is first evaluated, and then its value is compared with the tags of the assignment. If it is “true”, then assignment “b = 2” will be selected and evaluated. If not, the assignment “b = 3” will be executed.

A graph illustrating the data structure is given in Figure 3-2. CAssignment is an abbreviation of “conditional assignment”. “Assignments” is a linked list of assignments to be executed if the tag matches the condition.

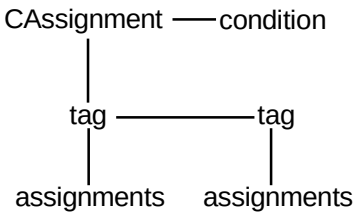


**Figure 3-2. Data structure of conditional assignment**

## 3.2.5 States

A state is a linked list of assignments, including simple and conditional assignments. The assignments are triggered under the same condition. The data structure is shown in Figure 3-3.


**Figure 3-3. Data structure of state**

# 3.3 Serial Code Conversion

To convert parallel assignments into serial ones, we must consider the sequential dependency between the assignments. If a variable in the LHS of an assignment is later used in the RHS of another assignment, the later assignment will get an incorrect value of the variable. To solve this problem, later assignments must have a way to access the original value of the variable in their RHS. Because there are two kinds of assignments in a state, “simple assignments” and “conditional assignments”, they will be discussed separately.

## 3.3.1 Simple Assignments

Simple assignment is an assignment without a condition and tags. To convert simple assignments, first scan all the variables in the LHS of assignments. If any of them is used in the RHS of the assignments after it, create a temporary variable, assign the new value to the temporary variable, and update the original variable after all the assignments are finished. For example, if we have the following non-blocking assignments:

```
a <= b;
b <= a;
c <= a;
```

First we scan the LHS of the assignments and see if the variable in the LHS appears in the RHS of the assignments after it. In the first assignment, “a” is used in the second and the third assignments, so a temporary variable must be created and an assignment to update “a” must be added to the end of the assignments. Assume the temporary variable for “a” is named “a_t”, then the assignment will become:

```
a_t = b;
b = a;
c = a;
a = a_t;
```

Now we continue to process the second assignment “b <= a”. Since b is not used in the RHS of any assignment after it, it is left unchanged. So is the third assignment “c <= a”. This completes the serial code conversion.

### 3.3.2 Conditional Assignments

Besides simple assignments, there are also conditional assignments in a state. They are treated almost the same as simple assignments. The conditions are treated the same as the RHS of simple assignments, and the branches are all assumed to be taken. For example, in the following code:

```
if (a == b)
     c = d;
else
     c = e;
```

Variables “a”, “b”, “d”, and “e” are considered to be in the RHS of assignments, and “c” is considered to be in the LHS of assignments. If “a” has appeared in the LHS of an assignment before this code segment, and “c” will appear in the RHS of some assignments after this code segment, the code will be converted to:

```
if (a == b)
     c_t = d;
else
     c_t = e;
……
a = a_t;
c = c_t;
```

### 3.3.3 Serial-Code Conversion Algorithm

The algorithm can be summarized in the pseudo-code as shown in Figure 3-4.

```
assi1 = foreach(all_assignments)
{
     vl = LHS(assi1);
     if (!marked(v1))
     {
          assi2 = foreach(assignments_after_assi1)
          {
               v2 = foreach(RHS_variable(assi2))
               {
                    if (v1 == v2)
                    {
                         add_to_temp_list(v1);
                         mark(v1);
```

```
                        goto finish;
                    }
                }
            }
        }
finish:
    if marked(v1)
        print_temp_assign(assi1);
    else
        print_original_assign(assi1);
}
v3 = foreach(variable_in_temp_list)
{
    print_assign_val_temp(v3);
}
```

**Figure 3-4. Serial code conversion algorithm**

Where “foreach(RHS_variable(assignment))” returns the variables in the RHS of the assignment in turn, and “LHS(assignment)” returns the variable in the LHS of the assignment. “print_temp_assign(assi1)” prints the assignment “assi1” with its LHS changed to a temporary variable. If no temporary variable is used, then the routine “print_original_assign(assi1)” prints the original assignment “assi1”. The routine “print_assign_val_temp(variable)” prints a line in the source code to assign the temporary variable to the original variable like “a = a_t;”. After the dependency of a variable between two assignments is found, the variable is marked so that it will not be processed twice. To determine whether a variable is marked or not, use the function “marked(variable)”.

## 3.3.4 Order Swapping

It is possible to eliminate the use of temporary variables if we swap the order of the assignments. For example, in the following non-blocking assignments:

```
a <= b;
c <= a;
```

In the original algorithm, since the RHS of the second assignment uses the LHS of the first one, a temporary variable will be introduced as the following:

```
a_t = b;
c = a;
a = a_t;
```

But if we swap the two assignments, the temporary variable will no longer be necessary. The serialized code will be:

```
c = a;
a = b;
```

To formulate this problem into graph theory, the following transformation is proposed:

(1) Transform each assignment to a single node, and then arrange them in a line according to the sequential order of the assignments.

(2) If the variable in the LHS of an assignment is used in the RHS of another assignment, draw an arrow between the nodes that correspond to the assignments, starting from the node with the variable being used.

An example of the graph is given below. The assignments are in Figure 3-5, and the graph is in Figure 3-6.

```
a <= b + c;
c <= a + b;
d <= a + e;
e <= a + d;
```

**Figure 3-5. Order swapping example assignments**

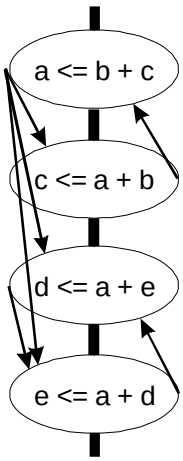


**Figure 3-6. Order swapping example graph**

An arrow pointing down represents a dependency between the assignments, and a temporary variable need to be used. So the goal of the problem is to swap the nodes

so that the number of nodes with arrows pointing down is minimized.

The solution space of this problem is obviously O(n!). Although some heuristics can be used to solve this problem, an optimal solution is difficult to obtain. So it is currently not implemented now.

After serial code conversion, we have assignments that can be executed serially.

# 3.4 Interpretive to Compiled-Code Conversion

The original version of VCK is interpretive. After the original design is converted to assignments in states, a scheduler evaluates the assignments in the state. In compiled version, the scheduler is only slightly modified. All the assignments in a state are converted to a C routine, so the scheduler only needs to call the correct C routine according to the state it should evaluate. So there are two problems to be solved. The first one is how to connect the interpretive and compiled part so that the scheduler can call the correct C state routines. The second one is how to convert the assignments in a state to a C routine, including the pointers and the data structures like variables and equations.

## 3.4.1 State Calling

In the interpretive version of VCK, after an event is triggered, the corresponding state must be executed. In compiled version, all states are precompiled into C routines, but the routine to be called is determined at run time. So there must be a conversion.

To call the state routines, an array of pointers to the state routines is built at the compilation time. This way any state routine can be called using an index, which is its state number. A compiled code example is shown below, where the global array state_ctask_g[] keeps the pointers to the routines:

```
int (*state_ctask_g[])() =
{
   state_1_ctask,
   state_2_ctask,
   state_3_ctask
};
int state_1_ctask(){…}
```

```
int state_2_ctask(){…}
int state_3_ctask(){…}
```

In the code above, each routine “state_n_ctask()” represents a state. In the scheduler, it calls the correct routine using the following code with state number n.

```
int (*addr_ctask)();
addr_ctask = state_ctask_g[n];
result = (*addr_ctask)();
```

Where “addr_ctask” is a variable which stores the address to the correct routine obtained from state_ctask_g, and “result” is an integer for the return value of the state routine.

## 3.4.2 Data Structure Conversions

There are two phases in compiled-code simulation; one is the compilation phase, and the other one is the simulation phase. In compilation phase, the design is converted to C code, and then is compiled into an executable file. In simulation phase, the design is loaded and simulated. Since compiled-VCK uses a mixed scheme with interpretive scheduler and compiled evaluation routines, the compiled-code part must be able to access the same data structures as the interpretive part. So at the simulation time, there must be a way to recover back the same data structures as that of the compilation time.

One solution to rebuild the same data structures is to parse the original design again, but it is very inefficient. The best way is to write out the data structures already parsed to a file and then read it back at simulation time. If the values simulated so far are also written to a file with the data structures, the simulation can be restarted from the time the last simulation is stopped. It is much more flexible and is adopted in the current implementation. So our solution is to write out the data structures at the compilation time and read them back at the simulation time.

The data structures used in the compiled-code part are mostly variables and equations. There are two kinds of information saved in the data structures: static and dynamic. Static information includes the bit-length of the variable and the current simulation value, etc. This information will not be changed between the compilation time and the simulation time. Dynamic information includes pointers to other data structures, which could be different between the compilation time and the simulation

time. To write out the static information, just save them into a file. But writing out dynamic information is more complex and worth further discussion.

Our solution is to give each variable and equation a unique identifier, which is an integer. The identifier of the variable is called variable_id, and the identifier of the equation is called equation_id. When writing out pointers to a file, write the identifier numbers instead of the values of the pointers. When the design is reloaded at simulation time, the memory of the data structures are reallocated, and a table which maps the identifiers to the addresses of the data structures is built. Later in simulation, the compiled-code part can get the addresses of the pointers by this mapping table. The conversion process is shown in Figure 3-7.

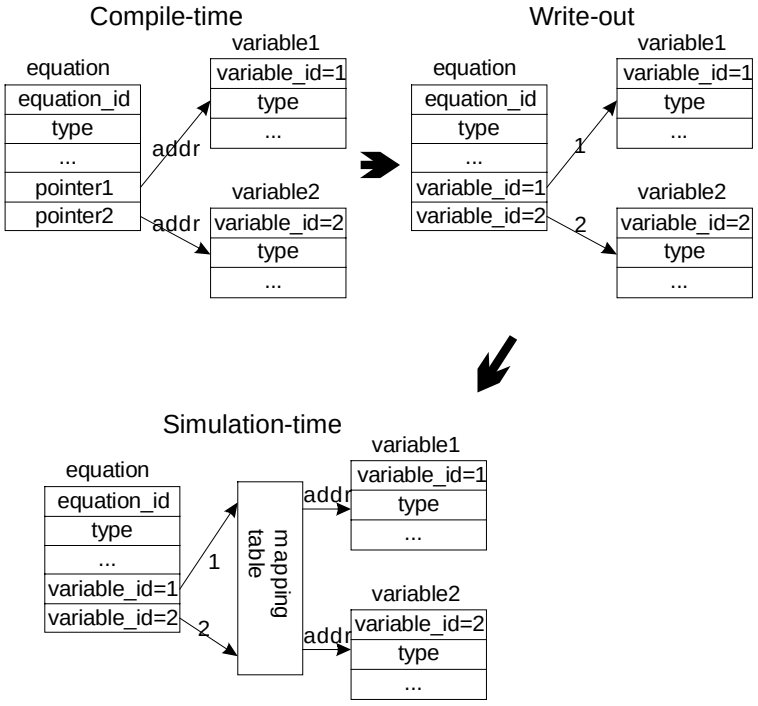


**Figure 3-7 Pointer conversion**

## 3.4.3 Compiled-Code Data Structures

When compiling the design, the equations are written out and flattened as a series of C evaluation routine calls. All evaluation routines have prefix “cc_” in compiled-code VCK. For example, the equation “a + b” will become something like:

```
cc_value_add(a, b, result);
```

And a * b +c would become

```
cc_value_multiply(a, b,value1);
cc_value_add(value1, c, value2);
```

Where value*N* is the sim_v of the equation; “a”, “b”, and “c” are the pointers to the value field of the corresponding variables respectively. Since all the values are pointed to by pointers, and all the variables are recreated when the compiled executable file starts simulation, it is obvious that we cannot know the addresses of these pointers at compilation time. So there must be a mechanism to get the correct addresses to the values when simulation starts.

As stated in the previous subsection, the pointers are written out as identifiers. So before the evaluation routine is executed, a conversion routine which converts the identifier to a pointer must be called first. The conversion is based on the mapping table created when the memories of the data structures are reallocated, as explained in the previous subsection. Assume the name of the routine is index_get_value, then the equation “a + b” is now converted to:

```
a = index_get_value(1);
b = index_get_value(2);
result = index_get_value(3);
cc_value_add(a, b, result);
```

Where the identifier of “a” is 1, the identifier of “b” is 2, and the identifier of “result” is 3.

After these conversions, the data structures are again available to the compiled-code routines.

# Chapter 4

# Compiled-code Techniques

## 4.1 Basic Concepts

There are two reasons why the compiled-code techniques proposed by Lewis, Maurer, etc. can accelerate the simulation. The first one is the elimination of a central scheduler, and the second one is specific routines for gate and net evaluations, which removes many redundancies. In VCK, the central scheduler is kept, so the compiled-code focuses on the evaluations. As described in the previous chapter, it focuses on accelerating the evaluations of the equations.

The benefit of compiling a design into C code is from the fact that the design will not be changed during simulation. Therefore specific statements can be created to evaluate the equations. For example, if we have a statement:

```
a = b + c;
```

In the interpretive version of VCK, it first reads the assignment, gets its RHS equation, evaluates the operator, and then finally stores the result to the variable “a” on the LHS. In compiled-code, since we already know the type and bit-length of the variables, we can generate efficient code like:

```
a = b + c;
a = a & bit_mask;
```

Where bit_mask masks off bits not needed. After the C code is compiled, very efficient assembly code like the following can be generated:

```
load regb
load regc
add  regb, regc
mask  bitm // remove the un-needed bits
store rega
```

Compared to hundreds of instructions needed in interpretive version, the improvement is obvious. But because the number system in Verilog is 4-valued, including “0”, “1”, “X” and “Z”, the evaluation routines cannot be this simple. But it

can still be greatly simplified.

Since the target language of the code generator is C, a C compiler is needed to compile the generated code. There are many optimization options in C compilers. These optimizations will be used and will affect the code and evaluation routines we generate.

There are two different parts of code needed to be generated. One is a pre-compiled library which consists of evaluation routines that will be called by the simulator, and the other is the C state routines for each specific Verilog design. There are different techniques in generating them and will be discussed separately.

# 4.2 Evaluation Routines

Evaluation routines are routines used to evaluate equations. For example, there are routines to add, multiply, divide, compare, etc. The evaluation routines in interpretive VCK are generic routines, which can handle all bit-length and types of operands. For example, in the addition routine, it must handle bit-length from one to a very large number. It must also handle operands with type signed integer, unsigned integer and floating-point. Since the design is fixed at the compilation time, the type and bit-length of the operands are also fixed. Specific evaluation routines in type and bit-length can thus be called to save simulation efforts.

The evaluation routines are simplified from original ones. When simplifying the routines, computer architecture and the compiler being used must be taken into account.

## 4.2.1 Computer Architecture

The computers being used now are mostly 32-bit. The “int” type in C is also 32-bit in current implementations; so 32-bit is a boundary to our simplification. All operations less than 32 bits are all the same in current computers. For example, 1-bit bit-or is the same complex as 2-bit bit-or. 32-bit itself is also a special case. For example, no mask is necessary if the operands are all 32 bits. 64 bits is also a special case and is treated separately. All bit-length above 32 bits and is not 64 bits are evaluated using the generic routines.

For example, in the integer addition routine, the original routine looks like:

```
if (op1_type == SIGNED)
     sign_extention(op1, nints);
if (op2_type == SIGNED)
     sign_extention(op2, nints);
for (i = 0; i< nints; i++)
{
     op1_aval = get_op_aval(op1, i);
     op2_aval = get_op_aval(op2, i);
     result_aval = op1_aval + op2_aval;
     if (i == nints –1)    //Last group, may need mask
          result_aval = result_aval & bitmask;
     put_op_aval(result_aval, i);
}
```

The “int” type in C is 32 bits, so the operands are added in 32 bits each time. Variable “nints” is the bit-length of the target variable divided by 32. The bit-length of the operands may not be multiples of 32 bits, so a bit-mask is needed in the last 32 bits.

Routine “get_op_aval” gets the value of the operand on the i-th position, while routine “put_op_aval” puts it back. The value system in Verilog is 4-valued, so two bits are needed to represent a value, which is called a-bit and b-bit. The meaning of the combinations of these two bits is shown in Figure 4-1. For simplification, here only a-bits are considered, and the carry between each 32 bits is ignored. So only a-bits of the operands are used, which is called “aval” in the text.

| A-bit | B-bit | Meaning |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | Z |
| 1 | 1 | X |

**Figure 4-1. Verilog 4-value representations**

The routine first inspects if any of the operand is signed. If it is, sign-extend it to the bit-length of the target bit-length. It then iterates each 32-bit and uses the addition in C to add these values. The result is stored to the target operand in the last part of the code.

The routines can be simplified if the type and the bit-length are already known. For example, if both the operands are known to be unsigned, the two “if” statements can be eliminated, which saves some comparison and branches, as the following code shows:

```
for (i = 0; i< nints; i++)
{
     op1_aval = get_op_aval(op1, i);
     op2_aval = get_op_aval(op2, i);
     result_aval = op1_aval + op2_aval;
     if (i == nints –1)    //Last group, may need mask
          result_aval = result_aval & bitmask;
     put_op_aval(result_aval, i);
}
```

If the operands are known to be unsigned, and the bit-length is less than 32 bits, the for-loop can be eliminated, yielding the following code:

```
op1_aval = get_op_aval(op1, 0);
op2_aval = get_op_aval(op2, 0);
result_aval = op1_aval + op2_aval;
result_aval = result_aval & bitmask;
put_op_aval(result_aval, 0);
```

If the bit-length is exactly 32 bits, the bit-mask can also be eliminated, which eliminates line 4 in the previous code.

If the bit-length is exactly 64 bits, the bit-mask can be eliminated, and the for-loop can be unrolled. Yielding the following code:

```
op1_aval = get_op_aval(op1, 0);
op2_aval = get_op_aval(op2, 0);
result_aval = op1_aval + op2_aval;
put_op_aval(result_aval, 0);
op1_aval = get_op_aval(op1, 1);
op2_aval = get_op_aval(op2, 1);
result_aval = op1_aval + op2_aval;
put_op_aval(result_aval, 1);
```

In other cases, the original routine is used as the generic routine.

Besides the bit-length of the computer, the instruction set can also affect the performance of the compiled-code simulator. One example is RISC and CISC architecture. In an optimization technique called computed-goto, there is some improvement in speed on Intel CISC CPUs, but there is no improvement at all on SUN UltraSparc RISC CPUs. The detail of computed-goto will be discussed in later section.

## 4.2.2 Compiler Optimizations

The compiler used in the current implementation is GCC. There are some optimizations that can be used in compiled-code generation. They are discussed below.

#### 4.2.2.1 Threaded-Code Optimization:

It is possible for GCC to produce threaded-code object/executables using the optimization flag "-finline-functions". For example, if we have procedure “b” which calls procedure “a” twice:

```
void a()
{
       statement a;
}
void b()
{
       a();
       a();
}
```

After optimization, procedure b() will become

```
void b()
{
       statement a;
       statement a;
}
```

That is, the procedure calls are eliminated. But there are some limitations. First, if procedure a() is precompiled as an object file, it is not possible for GCC to optimize it. The optimization also cannot be done if procedure a() is in another .c file. The only way for GCC to produce threaded-code is to include all procedures in the same .c file. Since it will take a lot of time to compile all the routines in a large C file, it is not used in the current implementation, and is replaced with a technique called computed-goto.

#### 4.2.2.2 Loop-Unrolling:

GCC can do loop-unrolling with optimization flag "-funroll-loops". It will unroll loops whose number of iterations can be determined at the compilation time or run time. For example,

```
for (i = 0;i < 2; i++)
{
	statment a;
}
```

After optimization, it will become

```
	statment a;
	statment a;
```

Which will save some jumps and comparisons.

There is something worth noticing. Loop unrolling is not always done by GCC. Unrolling a block of code which is very large usually results in a huge program in size and also increases the probability of cache misses. It will only make the program run even slower. GCC has its own way to decide whether to unroll a loop or not, and our implementation just follows it.

If there is a variable that never changes its value once it is assigned, and it is used in the loop condition, then it will be treated as a constant and will be unrolled. For example,

```
int v = 2;
for (i = 0; i < v; i++)
	statement a;
```

After optimization, it will become

```
	statement a;
	statement a;
```

#### 4.2.2.3 Other Optimizations

GCC can eliminate an "if" block if its condition is constant and is always evaluated false. For example,

```
if (1 == 2)
	statement a;
```

The whole block inside "if" will be eliminated and will not generate any code. But a variable with its value never changed after its first assignment may not have the same effect. For example,

```
int a = 1;
if (a == 2)
	statement b;
```

GCC will either produce code to evaluate the condition or eliminate the “if” block. It depends on the distance between the assignment of the variable and the place it is used.

#### 4.2.2.4 Summary

To create specific routines from a generic routine, the following methods can be used:

1. When there is a loop, find the variable which controls its number of iteration, and assign it a constant.
2. When there is an “if” block, change its variables used in the condition to constants.

For example, in the addition routine described in the previous section, the 64-bit version can be created with loop-unrolling by GCC instead of by hand. Then the code will be:

```
for (i = 0; i< 2; i++)
{
     op1_aval = get_op_aval(op1, i);
     op2_aval = get_op_aval(op2, i);
     result_aval = op1_aval + op2_aval;
     put_op_aval(result_aval, i);
}
```

GCC will unroll the loop automatically.

# 4.3 Compiled-code Generation

There are two parts in the code generation. The first part is for initialization, and the second part is the C state routines. In initialization, there are also two parts. The first one is variable declaration, and the second one is memory allocation and pointer conversion.

## 4.3.1 Initialization

There are two phases in the initialization. The first phase is variable declaration, and the second phase is memory allocation and pointer conversion.

In the variable declaration phase, all the variables that will be used later are

declared. Since all the operands are of the type value_p, all declarations are in the form:

```
value_p valueID;
```

Where ID is the identifier number of the variable. For example, value1 is the value of variable with identifier 1. To declare them, the assignments are traversed at the compilation time and all variables, including temporary variables, are written out in the form just described. The valueID is only a pointer, it must be assigned to a valid address at the simulation time.

Variables in the Verilog design are saved to a file and are recreated before the simulation starts. A mapping table of the variable_id to the address of the variable is already created. So a call to the mapping routine “index_get_value” is enough.

## 4.3.2 State Routine Generation

As mentioned in the previous chapters, there are assignments in a state, and the assignments are composed of variables and equations. The data structures can be illustrated in the following figures:


**Figure 4-2. State**

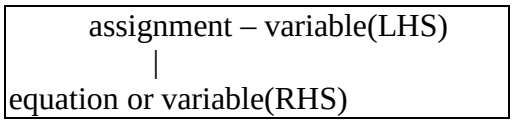


**Figure 4-3. Simple Assignment**

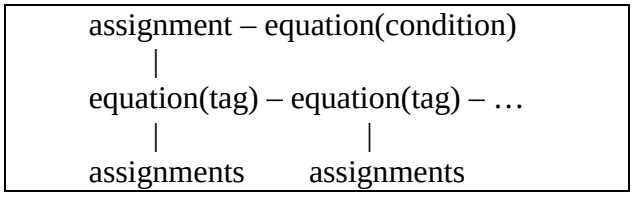


**Figure 4-4. Conditional Assignment**

The goal is to write the evaluation of the assignments to C code. Since the assignments are composed of equations, the basic step is to write the equations to blocks of C code. This procedure is called “equation expansion.”

### 4.3.2.1 Equation Expansion

The algorithm to write out an equation is recursive in nature. There are

termination cases and recursive cases to consider. Let us consider the termination cases first.

The termination case is an equation which has no equation as its operand. That is, all of its operands are variables. There are three different cases of equations: Unary, binary and associative.

Unary equations are equations with only one operand, as Figure 4-5 shows:

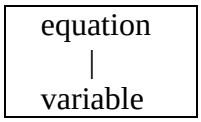


**Figure 4-5. Unary equation**

Assume the ID of the equation is 0, the ID of the variable is 1, and the operation type of the equation is “negate”, then the following code is generated:

```
cc_value_negate(value1, value0);
```

That is, value0 will be the negation of value1.

Binary equations are equations with two operands, as Figure 4-6 shows:

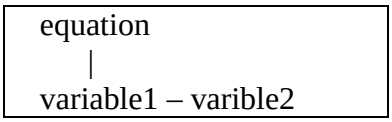


**Figure 4-6. Binary equation**

Assume the ID of the equation is 0, the ID of variable1 is 1, the ID of variable2 is 2, and the operation type of the equation is “divide”, then the following code is generated:

```
cc_value_divide(value1, value2, value0);
```

An associative equation is an equation with operation that is associative and which the operands can be chained to a list, as Figure 4-7 shows:

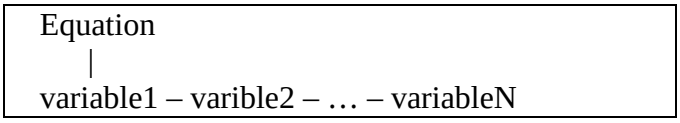


**Figure 4-7. Associative equation**

Assume the ID of the equation is 0, and the ID of variable*M* is *M* respectively. Also assume the operation type of the equation is “add”, then the following code is generated:

```
cc_value_add(value1, value2, value0);
cc_value_add(value0, value3, value0);
……
cc_value_add(value0, valueN, value0);
```

If any of the operands in an equation is an equation instead of a variable, the routine to expand the equation is called recursively.

#### 4.3.2.2 Equation Expansion Algorithm

The following code is the algorithm to expand the equations. The name of the main routine is “expand_eq”. The routine to expand unary equations is “expand_uni_eq”, and the routine to expand binary equations is “expand_bi_eq”. The routine to expand associative equations is “expand_asso_eq”. It is similar to previous ones and is omitted for simplicity.

```
procedure expand_eq(equation eq)
     gen_code(“{“};
     switch (eq.type)
          case UNARY:
               expand_uni_eq(eq);
               break;
          case BINARY:
               expand_bi_eq(eq);
               break;
          case ASSOCIATIVE:
               expand_asso_eq(eq);
               break;
     gen_code(“}“);
end

procedure expand_uni_eq(equation eq)
     op1_id = fetch_op1_id(eq);
     if (fetch_op1_type(eq) == EQUATION) then
          expand_eq(fetch_op1(eq));
     gen_code(eq.op, op1_id, eq.id);
end

procedure expand_bi_eq(equation eq)
     op1_id = fetch_op1_id(eq);
     op2_id = fetch_op2_id(eq);
     if (fetch_op1_type(eq) == EQUATION) then
          expand_eq(fetch_op1(eq));
```

```
    if (fetch_op2_type(eq) == EQUATION) then
        expand_eq(fetch_op2(eq));
    gen_code(eq.op, op1_id,  op2_id, eq.id);
end
```

The routine “gen_code” generates the C code to be written out as compiled-code. The evaluation routine is chosen according to the operation type of the equation (in field “eq.op”) and the type and the bit-length of the operands. For example, if eq.op is negate, op1_id is 1, op2_id is 2, then the following routine call will be generated:

```
cc_value_negate(value1, value2);
```

Real program contains many cases and is far more complex than the code shown here.

#### 4.3.2.3 Simple Assignment Expansion

Since equations can be expanded now, assignments can be converted into C code. There are two kinds of assignments: Simple assignments and conditional assignments. They will be discussed separately.

Simple assignment contains only one assignment without any conditions. The structure is shown in Figure 4-8.

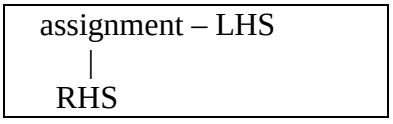


**Figure 4-8. Simple assignment**

Where the LHS is a variable, and the RHS is an equation or a variable. Assume the ID of LHS is 1, and the ID of RHS is 2, then the following code is generated for simple assignment.

```
cc_value_copy(value2, value1);
```

If there are part-select or bit-select in the assignment, another routine is used:

```
cc_value_part_to_part(value2, msb2, lsb2, value1, msb1, lsb1);
```

If the RHS is an equation, a call to expand_eq(RHS) is first executed before the code to copy the value is generated.

#### 4.3.2.4 Conditional Assignment Expansion

Conditional assignment contains a condition, more than one tags, and blocks of assignments. Condition and tags are equations. The block of assignments are linked to

the “down” of tag. The structure is shown in Figure 4-9.

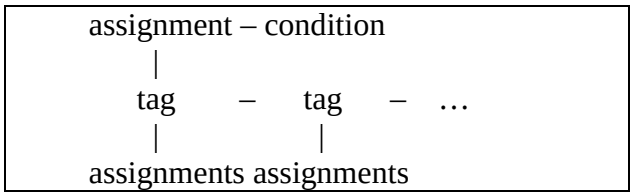


**Figure 4-9. Conditional assignment**

There are two kinds of conditional assignment. The first one is simple conditional assignment, and the second one is complex conditional assignment. The data structure is the same, but the condition and the tags are different.

In simple conditional assignment, the condition is simple and can be represented by a jump-table. That is, all cases that the condition can generate can be enumerated. Efficient C code using “switch” can thus be generated. For example, in the following Verilog case structure:

```
case (a)
       1'b0: b = 0;
       1'b1: b = 1;
       default b = 'bx;
endcase
```

Assume the variable_id of “a” is 0, the variable_id of “b” is 1, the variable_id for “0” is 2, the variable_id for “1” is 3, and the variable_id for “x” is 100. (Constant number is also treated as a variable in VCK.) Then the following C code will be generated:

```
{
       int index, xz;

       index = cc_value_to_int(value0, &is_xz);
       if (is_xz)
          index = 2;
       switch (index)
       {
       case 0:
          cc_value_simple_copy(value2, value1);
          break;
       case 1:
          cc_value_simple_copy(value3, value1);
          break;
```

```
        case 2:
            cc_value_simple_copy(value100, value1);
            break;
        default:
            break;
        }
}
```

Where cc_value_to_int converts the Verilog 4-value data in value0 to an integer. If the data in value0 is “X” or “Z”, then “is_xz” will be true and index will be 2. The case conditions in C correspond to the tags in the data structure of conditional assignment. The statements under “case” correspond to the assignments that are attached to the tag, which use “cc_value_simple_copy” to copy the first operand to the second.

In complex conditional assignments, all the cases that the condition can generate cannot be enumerated, so more complex if-then structure must be used to generate the C code. Each tag becomes an “if” statement, and the assignments under it become the “then” part. The end of the “then” part is a statement which jumps out of the conditional assignment.

For example, a statement like:

```
if (a < b)
     c = d;
next statement……
```

Will be converted to something like:

```
if (cc_value_compare_case_lt(value0, value1))
{
     cc_value_simple_copy(value2, value3);
     goto label0;
}
label0:
next statement……
```

Where the variable_id of “a” is 0, the variable_id of “b” is 1, the variable_id of “c” is 3, and the variable_id of “d” is 2. Routine “cc_value_compare_case_lt” compares two values and returns true if the first operand is less than the second.

#### 4.3.2.5 Assignment Expansion Algorithm

There are two kinds of assignments, one is simple assignment, and the other is conditional assignment. Procedure “assignment_generate” is the main routine that generates the code of an assignment. The procedure “generage_sa” is for simple assignments, while procedure “generate_ca” expands conditional assignments. The pseudo code is shown in Figure 4-10.

```
procedure assignment_generate()
    foreach (assignment A in the state) do
        if (A is a simple assignment) then
            generate_sa(A);
        else
            generate_ca(A);
end

procedure generate_sa(assignment A)
    R = equation_expand(RHS of A);
    generate_code(assign R to LHS);
end

procedure generate_ca(assignment A)
    C = equation_expand(condition of A);
    foreach(tag T of A)
        generate_code(compare C with T);
        assignment_generate(assignments under T);
        generate_code(goto end_label);
    generate_code(end_label);
end
```

**Figure 4-10. Assignment expansion algorithm**

There are some differences between complex conditional assignment and simple conditional assignment. In simple conditional assignment, the line “compare C with T” generates a “case” statement; while in complex conditional assignment, an “if-then” statement is generated.

# 4.4 Computed Goto

The evaluation routines used in compiled-code VCK are invoked by procedure calls, which is different from Lewis’ threaded-code technique. It is not very efficient to make so many routine calls, and there are several ways to solve this problem. One is to generate threaded C code directly. In this method, all routine calls are flattened and chained to a line, which will largely increase the code size. The second method is

to use the GCC optimization flag “-finline-functions”. But it requires that all the evaluation routines be compiled with the generated code, which takes a long time to compile. The third method is to “jump” to a procedure rather than calling it, and then jump back to the scheduler. The problem is that ANSI-C does not support jumping to a label in another routine.

There is a C-extension in GCC which solves the problem, which is called “computed-goto” [16]. In computed-goto, it can use the labels as values, so the address of a label can be obtained and stored to a variable. The concept and the usage of computed-goto is introduced first.

### 4.4.1 Concept and Usage

The address of a label defined in the current function can be obtained with the unary operator ‘&&’. The value has type ‘void *’. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *pointer;
 ...
pointer = &&label_name;
 ...
label_name:
```

To use the value stored in the variable, it must be able to be jumped to. This is done with the computed-goto statement ‘goto *EXP;’ For example,

```
goto *pointer;
```

Any expression of type ‘void *’ is allowed.

To make the idea more clear, an example is given below. Assume we have the following code:

```
void *label_pointer;
     label_pointer = &&goto_label;
     statement_1;
     statement_2;
     goto *label_pointer;
     statement_3;
     statement_4;
goto_label:
     statement_5;
```

When the program is run, the following statement sequence will be executed:

```
statement_1;
statement_2;
statement_5;
```

In the statement “goto *label_pointer”, the program directly jumps to the code after “goto_label”, which has its address saved in “label_pointer”.

## 4.4.2 Implementation

The reason to use computed-goto is to replace routine calls. Since any label in any routine can be jumped to now, routine calls can be replaced by goto. In this way, the routine structures can be kept, so that only needed routines will be linked into the executable file. And the improvement in performance with direct jumping is still achieved.

There are still two problems to be dealt with. The first one is the pass of arguments, and the second one is to get the addresses of the labels defined in a routine.

#### 4.4.2.1 Argument Passing

The arguments used in the evaluation routines are similar. In unary-operation evaluation routines, there are usually two arguments. One is the “from_value”, and the other is the “to_value”. They are all passed by pointers. In binary-operation evaluation routines, there are usually three arguments. They are the first operand, the second operand and the value to be saved to. They are also passed by pointers. In some other cases, some integers like “msb” or “lsb” will also need to be passed to the evaluation routines.

In computed-goto, no argument passing is possible, so the arguments must be passed by global variables. The evaluation routines can be changed by replacing the arguments with global variables. For example, if the first argument of the original evaluation routine is “opand1”, and the global variable to pass this argument is now “value_arg1_g”, then just rename all the “opand1” in the routine to “value_arg1_g”. Return values can also be passed by global variables. Since there is no recursive call in evaluation routines, this strategy works well.

#### 4.4.2.2 Address Calculation

The address of a label to be jumped to must be calculated and saved before the label can be used. The address of a label can only be obtained only if the code to get the address is in the same routine with the label. So the address of the label must be saved to a global variable for other routines to use it.

To do so, an array of the pointers to the labels is created. Assume the name of the array is “function_array_g[]”, then the address of the label is stored in the array in the first time the routine is called, which is also the only time it will be called. A routine has a unique index to the array so that correct label can be saved to correct position. For example, if the label to the addition routine has index “1”, and the label to the subtraction routine has index “2”, then we can jump to the addition routine using the label function_array_g[1], and jump to the subtraction routine using the label function_array_g[2].

To illustrate the idea, let us see the following example. The following code is a routine that adds two values:

```
void cc_value_add(args…)
{
     original code…
     return;
}
```

To save the address of the label to the global array, the code is changed to:

```
void cc_value_add(args…)
{
     function_array_g[CC_VALUE_ADD] = &&cc_value_add;
     return;
cc_value_add:
     original code…
}
```

Where cc_value_add is the label which will be jumped to in later simulations, and CC_VALUE_ADD is an integer indexing the label.

#### 4.2.2.3 Summary

Now the addresses of the labels can be obtained, and the arguments can be passed to and back from the routines, computed-goto can be used to accelerate the simulation. The evaluation routines and the code generated must be modified in order

to use it.

In the compiled-code generated, if the original procedure call to the evaluation routine “addition” is:

```
cc_value_add(opand1, opand2, result_val);
```

Then it will be changed to:

```
value_arg1_g = opand1;
value_arg2_g = opand2;
value_arg3_g = result_val;
return_point_g = &&return_point_1;
goto *function_array_g[CC_VALUE_ADD];
return_point_1;
```

Where return_point_1 is a label for the routine to jump back. And the evaluation routine which originally looks like:

```
void cc_value_add(args…)
{
     original code…
     return;
}
```

Will be changed to:

```
void cc_value_add()
{
     function_array_g[CC_VALUE_ADD] = &&cc_value_add;
     return;
cc_value_add:
     original code with arguments changed to global variables…
     goto *return_point_g;
}
```

After these modifications, the flow of the execution will be:

```
value_arg1_g = opand1;
value_arg2_g = opand2;
value_arg3_g = result_val;
return_point_g = &&return_point_1;
goto *function_array_g[CC_VALUE_ADD];
(original code in routine cc_value_add with arguments changed to global
variables…)
```

```
goto *return_point_g;
(statements after return_point_1……)
```

Before the simulation starts, all the routines which will be used must be called once to fill the label pointer array “function_array_g”. Since it is only called once, it will not affect the performance much. After the initialization, there will not be any more routine calls to these routines. All routine calls are replaced by direct jumps.

It seems that computed-goto will save some time compared to function calls. Surprisingly, although it is true on PC, it is not the case on SUN workstations; therefore it is not currently implemented. The reason will be discussed with the experimental results.

# Chapter 5

# Experimental Results

## 5.1 Compiled-code technique

There are four test cases used in the experiments. Mul16 is a Wallace Tree 16*16 multiplier. MC6805 is a CPU core. AAL1 and AAL2 are ATM chip designs. Mul16 is gate-level. MC6805, AAL1 and AAL2 are RTL. Mul16 is very evaluation intensive.

We conducted all the experiments on two platforms. PC is Intel Pentium III 733MHz with 256M memory. SUN is Sun UltraSparc 440MHz with 256M memory. Preprocessing time includes compiled-code generation and compilation. The results are shown in Table 5-1. The executables were not optimized by the compiler.

**Table 5-1. Experimental results of compiled-code**

| Test \ Method | Interpretive (sec) | Compiled-code(sec) | | Time saved (%) |
|---|---|---|---|---|
| | | Preprocessing | Simulation | |
| Mul16(PC) | 444.8 | 7.2 | 94.0 | 79 |
| Mul16(SUN) | 1212.4 | 7.9 | 274.5 | 77 |
| MC6805(PC) | 35 | 15.9 | 26.3 | 25 |
| MC6805(SUN) | 89.4 | 32.9 | 58.8 | 34 |
| AAL1(PC) | 90.3 | 4.1 | 55.8 | 38 |
| AAL1(SUN) | 326.5 | 8.7 | 194.8 | 40 |
| AAL2(PC) | 1251.8 | 9.0 | 665.3 | 47 |
| AAL2(SUN) | 3968.1 | 14.5 | 2108.3 | 47 |

We also conducted experiments on the same test cases with commercial simulators including Verilog XL (version 2.5.16) and NC-Verilog (version 1.1). The experiments were conducted on SUN UltraSparc 440MHz with 256M memory. The executables were optimized by GCC using flag “-O2”. The results are shown in Table 5-2. IVCK means interpretive VCK, and CVCK means compiled-code VCK. The numbers are the simulation time, and the unit is second.

**Table 5-3. Experimental results on IVCK, CVCK, XL and NC-Verilog**

| Test \ Method | IVCK | CVCK | Verilog-XL | NC-Verilog |
|---|---|---|---|---|
| Mul16 | 456.6 | 144.8 | 182.7 | 60 |
| MC6805 | 50.4 | 35.2 | 31.2 | 5.51 |
| AAL1 | 144.9 | 108.1 | 42.1 | 8.56 |
| AAL2 | 1056 | 671.5 | 313.9 | 47 |

# 5.2 Computed Goto

The test is to compare the performance difference between computed-goto and function call. The test iterates $10^9$ times on PC, and iterates $10^8$ times on SUN. The performance difference is due to the different methods used. The results are shown in Table 5-3.

**Table 5-3. Experimental results of computed goto**

| Platform/method | Computed goto(sec) | Function call(sec) | Ratio |
|---|---|---|---|
| PC | 19.17 | 24.12 | 0.79 |
| SUN | 11.88 | 6.15 | 1.93 |

# 5.3 Discussions

## 5.3.1 Performance Comparison

From the experimental results, it is found that the circuits that take the most advantage of the techniques proposed in this thesis are those with a lot of evaluations. Evaluations are greatly simplified in compiled-code; therefore the most acceleration is achieved.

Because system tasks and function calls in the original Verilog design are kept interpreted without compiling to C code, test-benches which use them a lot will not be accelerated much. For example, test cases which use $random a lot do not show much improvement in speed. However, random inputs are not that important in test-benches. It is because that random inputs usually generate random outputs, which is not very useful in functional verifications. So it is not a very serious problem currently.

It takes time to compile a design, so compilation time must be taken into account when evaluating the performance of a compiled-code simulator. It takes more time to compile a large C file, but more C code generated also means more acceleration. Actually, when the simulation runs long enough, the time saved can almost always compensate the time spent on compiling the design.

From the comparison with commercial simulators, it is found that for evaluation intensive test cases like Mul16, it is only about 2 times slower than NC-Verilog. Consider the efforts to write a new simulator and optimize the generated machine code, the techniques proposed in this thesis is a much easier way to obtain significant improvement in simulation speed. For test cases that are less evaluation intensive, the

performance is not comparable with commercial products yet. However, compiled-code VCK still runs much faster than its interpretive counterpart.

## 5.3.2 Computed Goto

The experimental results on computed-goto are somewhat surprising. It takes even more time if computed-goto is used on SUN workstations. The explanation can be found in the difference of the computer architectures. Intel CPU is CISC, and SUN UltraSparc is RISC. It takes more time to call a function than to jump in PC, but it takes about the same time in SUN. The difference can be observed from disassembled code of the executable file. In PC, a function call like:

```
func1(a1, b1, c1);
```

will be compiled to:

```
mov    0xfffffff4(%ebp),%eax
push   %eax
mov    0xfffffff8(%ebp),%eax
push   %eax
mov    0xfffffffc(%ebp),%eax
push   %eax
call   80483a0 <func1>
add    $0xc,%esp
```

If computed goto is used, an assignment of argument like:

```
arg1_g = &arg1;
```

will become:

```
lea    0xfffffff8(%ebp),%edx
mov    %edx,0x8049588
```

And

```
goto *array[0];
```

will become:

```
jmp    *0x80495a0
```

While in UltraSparc, the function call will be compiled to:

```
ld  [ %fp + -20 ], %o0
ld  [ %fp + -24 ], %o1
```

```
ld  [ %fp + -28 ], %o2
call  10534 <func1>
```

The argument assignment will become:

```
sethi  %hi(0x20800), %o1
or  %o1, 0x28, %o0     ! 20828 <arg3_g>
add  %fp, -32, %o1
st  %o1, [ %o0 ]
```

And the goto will become:

```
sethi  %hi(0x20800), %o1
or  %o1, 0x30, %o0     ! 20830 <array>
ld  [ %o0 ], %o1
jmp  %o1
```

In PC, the passing of assignments in function calls is about the same complex as assigning the arguments to global variables. Since calling a subroutine saves some registers to the stack automatically, it takes longer CPU cycles than a jump. Therefore some cycles are saved if jump is used instead of function call.

In UltraSparc, every instruction takes about the same amount of cycles. Calling a function with three arguments takes only 4 lines of assembly code, while computed-goto with three arguments is compiled to 16 lines. Therefore it is obvious that using computed-goto is not beneficial in RISC architecture.

# Chapter 6

# Conclusions and Future Work

A register transfer level compiled-code simulator is proposed in this thesis. An approach and the algorithm to convert data structures from interpretive simulation to compiled-code are proposed. Many optimization issues are also discussed. From the experimental results in section 5.1, it can be seen that the acceleration saves simulation time between 25% to 79%, depending on the design. An average acceleration of about 48% saving in simulation time is achieved from the technique proposed in this thesis. In general, designs with more complicated evaluations are more beneficial to use the compiled-code technique.

Compile time is also an important factor to judge the performance of compiled-code simulator. Larger circuits usually require longer time to compile. However, larger C code also means greater optimization. If the running time is long enough, the time used to compile the design can usually be compensated by the time saved in simulation.

In the future, machine code can be generated instead of C code. It saves the time to compile the design, and more efficient code can be generated. Two-state simulation is another acceleration technique that can be used in future work.

# Reference


[1] Z. Barzilai, J. L. Carter, and J. D. Rutledge, “HSS – A high-speed simulator.,” IEEE Transactions on Computer-Aided Design, vol. 6, July 1987, pp. 601-617.

[2] D. M. Lewis, “A hierarchical compiled-code event-driven logic simulator,” IEEE Transactions on Computer-Aided Design, June 1991, pp. 726-737.

[3] Peter M. Maurer, “The Shadow algorithm: A scheduling technique for both compiled and interpreted simulation,” IEEE Transactions on Computer-Aided Design, vol. 12, September 1993, pp. 1411-1413.

[4] Peter M. Maurer, “The Inversion algorithm for digital simulation,” IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 16, NO. 7, July 1997, pp. 762-769.

[5] P. Maurer and W. Schilp, “Three-valued simulation with the inversion algorithm,” Department of Computer Science Engineering., University of South Florida, Tampa, Tech. Rep. DA-27, 1995.

[6] Pranav Ashar and Sharad Malik, “Fast Functional Simulation Using Branching Programs,” IEEE 1995, pp. 408-412

[7] NC-Verilog Help, Version 1.1, Cadence Design Systems, 1997.

[8] J. R. Bell, “Threaded code,” Communication of ACM, vol. 16, no. 6, June 1973, pp. 370-372.

[9] Y. S. Lee and P. M. Maurer, “Two new techniques for multi-delay compiled logic simulation,” in Proc. 29th Design Automation Conference, June 1992, pp. 420-423.

[10] Micron Abramovici, Melvin A. Breuer and Arthur D. Friedman, “Digital Systems Testing and Testable Design”, Computer Science Press, 1990, pp 52-53.

[11] D. Schuler, “Simulation of NAND logic,” in Proc. COMPCON’72, Sep 1972, pp. 243-245.

[12] S.B. Akers, “Binary decision diagrams,” in IEEE Transactions on Computers, vol. C-27, June 1978, pp. 509-516

[13] A. Cerny and J. Gecsei, “Simulation of MOS circuits by decision diagrams,” in

IEEE Transactions on CAD, vol. C-4, October 1985, pp. 685-693

[14] A. Cerny, “An approach to unified methodology of combinational switching circuits,” in IEEE Transactions on Computers, vol. 27, August 1977

[15] VCK User’s Manual, Version 1.1, Avery Design Systems, 2001

[16] GCC online document, Version 2.91