

國立交通大學

電子工程學系 電子研究所
碩士論文

應用於處理器驗證之
腳本導引的限制隨機樣本產生器



**Script-Controlled Constrained-Random Pattern
Generator for Processor Verification**

研究生：許瀚蔚

指導教授：黃俊達 博士

中華民國九十八年六月

應用於處理器驗證之
腳本導引的限制隨機樣本產生器
**Script-Controlled Constrained-Random Pattern
Generator for Processor Verification**

研究生：許瀚蔚
指導教授：黃俊達 博士

Student: Han-Wei Hsu
Advisor: Dr. Juinn-Dar Huang



A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical & Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Electronics Engineering & Institute of Electronics

June 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年六月

應用於處理器驗證之腳本導引的限制隨機樣本產生器

研究生：許瀚蔚

指導教授：黃俊達 博士

國立交通大學

電子工程學系 電子研究所碩士班

摘 要

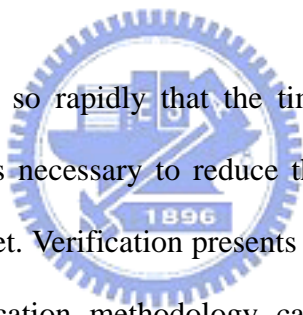
積體電路設計複雜度的快速成長使得整個設計流程所需的時間也跟著拉長，但是在受到上市時間的限制下，縮短開發時程是必須的。在設計過程中驗證這步驟大約佔了全部時程的 60%至 70%，發展新的驗證策略來縮短產品推出的時程是相當重要的。標準參考模型和直接測試在現今的驗證環境中變為基本的方法。驗證策略必然朝向更新的方式，例如以有限制的隨機生產方式來縮短測試樣本的發展時間，進而加快完成積體電路的完整驗證。針對某些被微處理器設計者忽略的不常發生的情境與狀況，以有限制的隨機生產方式產生相對應的測試樣本達到在驗證階段的初期就能及早發現設計的錯誤。在本論文中，詳細的介紹了有限制的隨機生產器以及如何利用我們提供的命令文件(script file)便能簡單地針對指定的情況製造出大量的測試樣本。

Script-Controlled Constrained-Random Pattern Generator for Processor Verification

Student: Han-Wei Hsu Advisor: Dr. Juinn-Dar Huang

Department of Electronics Engineering & Institute of Electronics
National Chiao Tung University

Abstract

The logo of National Chiao Tung University is a circular emblem. It features a gear-like outer border. Inside the circle, there is a stylized representation of a building or a structure, with the year '1896' prominently displayed at the bottom. The entire logo is rendered in a blue color.

IC complexity is increasing so rapidly that the time spent on whole design flow increases in this situation. It is necessary to reduce the development time due to the pressure from the time to market. Verification presents about 60-70% of the total design effort and advances in verification methodology can improve the time to market considerably. Directed tests and golden reference models are becoming the primitive tools in the modern design verification environment. Verification strategies are consequently developed towards advance methodologies like constrained-random approach to reduce verification pattern development time, and speed up the time it takes to achieve complete verification. Constrained-random pattern generation tools create tests for corner cases that the microprocessor designers may not expect and hence find bugs early in the verification stage. This thesis describes the details of the constrained-random generator and the script file that helps easily produce a huge amount of constrained-random patterns for designated corner cases.

誌 謝

首先，我要感謝我的指導教授-黃俊達博士，在碩士三年當中給我的支持和鼓勵，讓我能有良好的研究環境，在自由的學習風氣之下，培養出獨立研究的能力，又能隨時給予寶貴的意見及指導，對老師的感激之情，並非以簡短的文句可以表達。

當然也要感謝養育我長大的父母對我的栽培，沒有他們，就沒有今日的我。接著要感謝的是所有來參與口試的所有教授們，劉建男教授和王俊堯教授，百忙當中抽空前來指導我，讓我受益匪淺，也讓我得到了寶貴的經驗，謝謝你們。

最後也要感謝所有實驗室的同仁們，建德、威號、詠翔學長以及學弟們，跟大家一起修課、做實驗、和討論及分析研究成果更是我人生旅途中一段最值得珍惜的回憶，希望未來在讀書或工作還有機會一起努力。

希望這篇論文能對人類社會有小小的貢獻，如此一來在辛苦也就值得了，再次謝謝大家的幫忙。

Contents

Chinese Abstract.....	i
English Abstract	ii
Acknowledgments.	iii
Contents.....	iv
List of Tables.....	vi
List of Figures.....	vii
Chapter 1 Introduction.....	1
1.1 Motivation	1
1.2 Thesis Organization.....	3
Chapter 2 Related Works.....	4
2.1 Processor Verification	4
2.1.1 Simulation-based Verification.....	4
2.2 Proposed Strategy	6
Chapter 3 Constrained-Random Verification.....	8
3.1 Overview of Verification Flow	8
3.2 User Input Script File	9
3.2.1 Constant, Variable, and Parameter	9
3.2.2 Instruction Syntax in Base Pattern.....	12
3.2.3 Program Flow Mechanism in Base Pattern	16
3.3 Constrained-Random Code Generator	18
Chapter 4 Introduction to Study Case: ACARM9 Processor Core Design	22

4.1	Organization of ACARM9	22
4.2	Multi-cycle Multiplication.....	25
4.3	Verification Strategy	27
4.3.1	Functional Verification.....	27
4.3.2	Coding Style Checking by Linting Free	28
4.3.3	Deterministic Verification	28
Chapter 5	Experimental Results	29
5.1	Experimental Environment.....	29
5.2	Applied to Real Case	30
5.2.1	Simulation Environment.....	30
5.2.2	Script Files and Bugs Found.....	31
5.3	Compare with Pure Random Verification.....	39
Chapter 6	Conclusions and Future Works.....	40
References	41

List of Tables

Table 3.1 Percentage of elements chosen in parameter “reg_1”	11
Table 3.2 Formats with corresponding actions in generator	13
Table 5.1 Experimental environment	29
Table 5.2 Logic shift right with zero shift amounts	32
Table 5.3 Results of shift operations in standard and the model.....	34
Table 5.4 Value of r15 fetched as operand in different source of shift amount	35
Table 5.5 Output of multiplier in standard and RTL.....	38
Table 5.6 Simulation time of two generators	39



List of Figures

Figure 1.1 Reasons of re-spins	2
Figure 2.1 Random verification.....	5
Figure 2.2 Constrained-Random verification with constraint file.....	7
Figure 3.1 Simple architecture of constrained-random code generator.....	8
Figure 3.2 Organization of user input script file	9
Figure 3.3 Format of constant & variable.....	10
Figure 3.4 Example of base pattern	14
Figure 3.5 Possible result of token “[d_2]”	14
Figure 3.6 Possible results of tokens “[reg1[r_1]” & “[reg1[]”	15
Figure 3.7 One possible outcome	15
Figure 3.8 Syntax of loop function and random selection.....	16
Figure 3.9 Flow chat of random selection.....	17
Figure 3.10 Architecture of the generator	18
Figure 3.11 Flow chat of generator	21
Figure 4.1 Block diagram of ACARM9	23
Figure 4.2 Source selection of the address registers.....	24
Figure 4.3 Operand selection.....	24
Figure 4.4 Read/write operation	25
Figure 4.5 Multiplication FSM of ACARM9	26
Figure 4.6 Functional verification flow	27
Figure 5.1 Simulation environment	30
Figure 5.2 Script file of shift operand testing (1)	31

Figure 5.2 Different used bit-range of shift amount register33

Figure 5.4 Script file of shift operand testing (2)35

Figure 5.5 Script file of multiplication instructions36

Figure 5.6 Block diagram of multiplier in RTL38



Chapter 1 Introduction

1.1 Motivation

Today's IC and System-on-Chip (SOC) design trends have placed an increasingly heavy load on the shoulders of verification engineers. Processor functionality, custom logic, software content, and system performance are all getting more complicated at the same time that all the schedules are being pressed. The percentage of the time that is spent on verification in the design flow grows up with the complexity of design. It consumes about 70% of design effort today [1].

Based on the report of the functional verification for IC designs in Collett International 2000, the bugs found in IC designs mostly come from the errors in function and 50% of chips require one or more re-spins. Moreover, Figure 1.1 shows the analysis for the chips which require re-spins and 74% of re-spins are due to functional defect. The fact indicates that superior functional verification methodologies are needed. Advances in verification methodology can improve the time to market when IC designs are more complex than before.

Verification is a process used to demonstrate the functional correctness of a design [2]. The main purpose of functional verification is to ensure that a design implements intended functionality. Functional verification for a design in million-gate-count always needs more than billions of clock cycles in simulation to verify fully.

Simulation of automatically-generated test programs is the main means for verifying complex hardware designs. Random verification may be one selection to speedup the process. Language features such as Verilog random sequence generator can create a great amount of input signals randomly based on a structured set of rules. Such

random-sequence generation schemes are not difficult to produce but can not cover full cases efficiently for larger designs, such as processors.

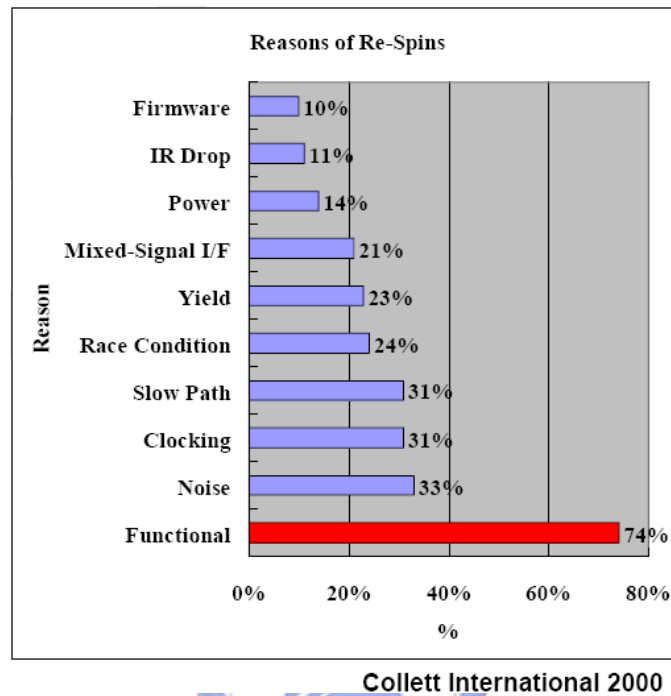


Figure 1.1 Reasons of re-spins [3]

Constrained-random simulation is the main workhorse in today's hardware verification flows. It requires the random generation of input stimuli that obey a set of declaratively specified input constraints, which are then applied to validate given design properties by simulation. Constrained-random verification can offer a highly effective way to deal with the challenges of microprocessor verification [8]. These verification challenges include: complex instruction sets, multiple pipeline stages, in-order or out-of-order execution strategies, instruction parallelism, and other features for some specific applications. The time that traditional direct tests require becomes uncontrollable, and simple random sequences are no longer sufficient to verify a processor fully.

1.2 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 discusses the simulation-based methodology in processor verification. Chapter 3 gives a detailed description on the proposed verification strategy. Chapter 4 makes an introduction to the study case. Chapter 5 describes the experiment setup and simulation result, and Chapter 6 concludes the thesis.



Chapter 2 Related Works

In this chapter, we give a brief description of processor verification and the main idea about our proposed strategy.

2.1 Processor Verification

Verification on processors can be basically classified into two categories: formal verification and simulation-based verification. Formal verification mathematically analyzes the design and verifies if it functions correctly. Simulation-based verification verifies the design by comparing the result of the design with the golden model through simulation. Although the formal verification can check the consistency with the functional specification and verify the equivalence across several design levels, it is difficult to handle a large design due to high computing complexity and memory explosion. So, generating test patterns for the simulation-based verification should play an important role in processor verification.

2.2.1 Simulation-based Verification

In simulation-based approach, it generates test programs automatically to verify the processor through simulation, so called instruction-based verification. To prove functional correctness of a new processor design, deterministic test patterns for simulation would be a common choice. Patterns for basic features verification are like data processing, memory access, branch and interrupt, etc. Except the basic cases, there are patterns for special cases that designed for special applications: over 32-bit shift amount, scalar/vector operations, and block data transfer, etc. These test patterns usually are designed for single instruction, and the completeness is assured by code coverage tool.

As the set of test patterns becomes larger and is hard to handle manually, additional verification strategies are the next necessary selections. Real applications or programs like Dhrystone, Whetstone, DSPstone, and JPEG2000 encoder program are usually used as test bench for processor designs. Most bugs come from unexpected corner cases like different combinations of multiple instructions when using this strategy.

To find out more unexpected corner cases, random verification which can easily generate a great deal of patterns would be used to deal with this problem. Random code generator produces massive random pattern as input to RTL and golden model. Designers compare the output result come from RTL and model to check the consistency or dump the difference information, as show in Figure 2.1. The test pattern produced by pure random generator usually hits a corner case after a long-time simulation and may still lose some cases.

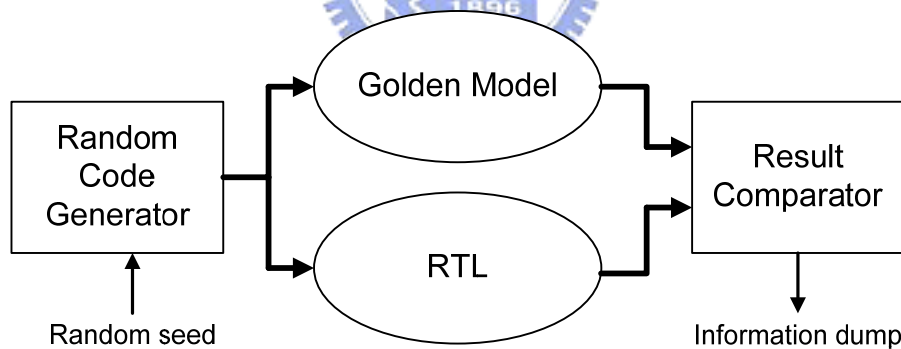


Figure 2.1 Random verification

Many techniques have been proposed for generation of directed test programs. Aharon et al. [14] have proposed a test program generation methodology for functional verification of PowerPC processors in IBM. Miyake et al. [15] have presented a combined scheme of random test generation and specific sequence generation. A

coverage driven test generation technique is presented by Fine et al. [13]. Shen et al. [16] have used the processor to generate tests at run-time by self-modifying code, and performs signature comparison with the one obtained from emulation. Ur and Yadin [19] present a method for generation of assembler test programs that systematically probe the micro- architecture of a PowerPC processor. Iwashita et al. [18] use an FSM based processor modeling to automatically generate test programs. Mishra et al. [17] have proposed a graph-based functional test program generation technique for pipelined processors using model checking. These techniques present the methodologies of generating test patterns and checking with model for processor verification.

2.2 Proposed Verification Strategy

In order to make the generating patterns become efficient for covering all cases, the first step is to add constraints on generating patterns. Figure 2.2 shows that adding a constraint file that contains some rules in random verification. Therefore, the generator can be controlled by writing code generating rules in constrained file and produce the patterns focusing on specific instruction groups and makes a special case happen frequently. To design a suitable rule and syntax for constraint file such that a generator can produce appropriate pattern become the main issue.

The proposed strategy focuses on generating variant test patterns which have specific combinations of different instruction groups. Every segment of single instruction and every instruction in combinations are generated individually. The freedom in generating test patterns can help engineers easily test special cases by writing rule in script file and, moreover, modify a little content of script file for different cases. In contrast with a

verification tool: Cadence Specman Elite [20] which uses a high language: e-language to write the script file, we propose an assembly-like syntax to control more details in the test patterns.

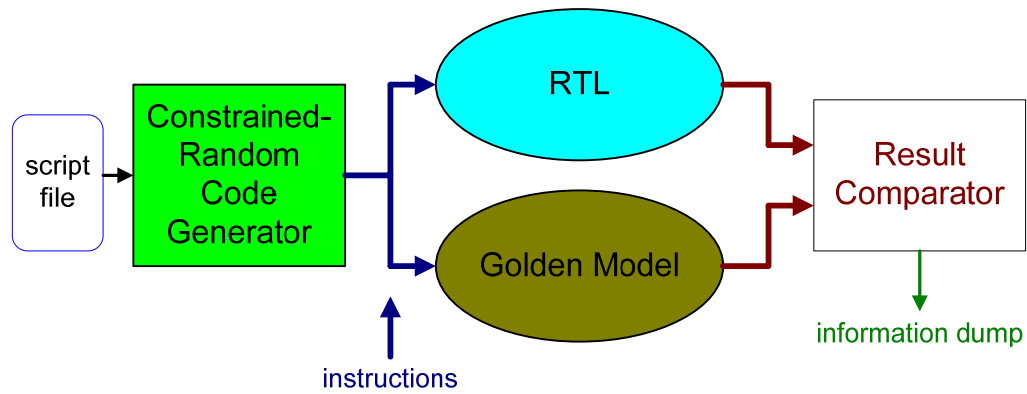


Figure 2.2 Constrained-Random verification with constraint file



Chapter 3 Constrained-Random Verification

In this chapter, the flow of proposed constrained-random verification is introduced in the first section. The remaining sections give the detailed description of the proposed strategy.

3.1 Overview of Verification Flow

As described in Chapter 2, we add some constraint rules on random code generator, and the process of whole verification flow becomes a little more complex than a pure-random one. It can be divided into 3 steps: writing a script file, compiling the script file, and generating then outputting codes.

First, according to the case which would like to be tested, writing the rules of constraint in the script file. It depends on the architecture of the target design. The random code generator compiles the script file and generates assembly codes based on rules then transforms codes into machine codes and outputs the codes at last. Figure 3.1 shows the architecture of constrained-random code generator.

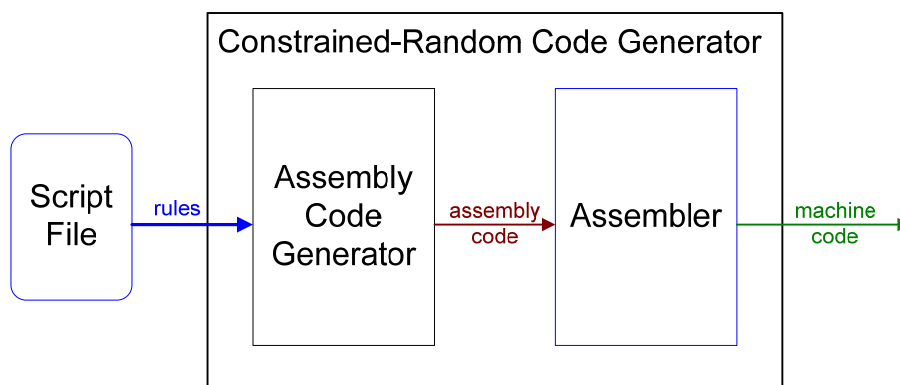


Figure 3.1 Architecture of constrained-random code generator

3.2 User Input Script File

The content of a script file can be separated into two major parts: “*Constraint Setting*” and “*Base Pattern*”. Base pattern is an assembly-like program and controls the program flow of test patterns produced by generator. Every line of codes is a combination of several tokens, and these tokens would be transformed into segments of assembly code by generator. Segments such as operation code or operand in every line of codes are ruled by constraint setting. There are three basic parts: *constant*, *variable*, and *parameter* in constraint setting. Organization of script file is shown in Figure 3.2. Following sections give details about syntax of these two.

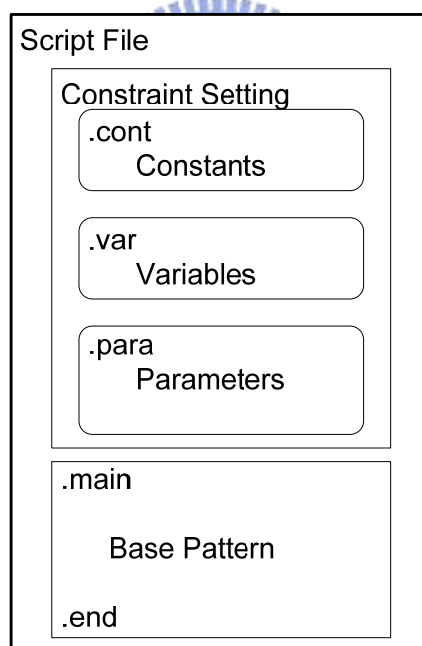


Figure 3.2 Organization of user input script file

3.2.1 Constant, Variable, and Parameter

Constant part is used to assist in controlling the program flow in base pattern. There are two program flow mechanisms: loop function and random selection (introduced in

Section 3.2.2) in the base pattern syntax. The counts of loops and random selections are declared in this part. It is convenient for user to change the counts of loop in this part rather than in base pattern.

Objects declared in variable part are able to store the segments in pattern that is generated this time and can be fetched later in place of generating new one. For example, it can fix two or three registers and make them identical in generating assembly code to fulfill certain cases, or make two registers in different codes to test data hazard. Detailed usage of variable is illustrated in Section 3.2.2. Simple format of constant part and variable part is shown in Figure 3.3.

```
.const
  const_name1 const_value1
  const_name2 const_value2
      ⋮
  const_namen const_valuen
.var
  var_name1
  var_name2
      ⋮
  var_namem
```

Figure 3.3 Format of constant & variable

The range of the segments randomly produced in assembly codes is defined in parameter part. A parameter is a group with elements. When a generator wants to generate one segment, it would randomly select one element in some parameter which is mentioned in the base pattern. User can build a new parameter that contains elements for generator to choose. There are two kinds of format in parameter. One of them has independent elements and can be decided the relative percentage of every element selected by adjusting the corresponding weights. If the weight of some element is

greater, it means that this element has higher probability chosen by generator. On the other hand, an element with smaller weight would be picked in lower probability. The format (i) and one example are shown below:

$$parameter = \{element_1(weight_1), element_2(weight_2), \dots, element_n(weight_n)\} \quad (i)$$

$$reg_1 = \{r1(8), r3(6), r4(3), r11(), sp(2)\} \quad (1)$$

In Example (1), a parameter named “*reg_1*” has five elements: *r1*, *r3*, *r4*, *r11*, and *sp* in the braces. The number in the parentheses behind every element means the corresponding weight, and none for the weight of element *r11* means the default value: 1. So the weights of the five elements: *r1*, *r3*, *r4*, *r11*, and *sp* are 8, 6, 3, 1, and 2. When generator wants to generate a pattern from parameter *reg_1*, the percentage of every element which is chosen in this group are presented in Table 3.1.

Table 3.1 Percentage of elements chosen in parameter “*reg_1*”.

Element	Weight	Percentage
<i>r1</i>	8	$\frac{8}{8+6+3+1+2} \times 100\% = 40\%$
<i>r3</i>	6	$\frac{6}{20} \times 100\% = 30\%$
<i>r4</i>	3	$\frac{3}{20} \times 100\% = 15\%$
<i>r11</i>	1	$\frac{1}{20} \times 100\% = 5\%$
<i>sp</i>	2	$\frac{2}{20} \times 100\% = 10\%$

The other kind of format in parameter is format (ii) with pound sign “#” in front of parameter name. It is used to deal with immediate integer value in assembly code. Contrast to format (i), there are no multiple elements in one parameter. This kind of parameter defines the range of immediate integer value by the minimum and maximum value in the braces, and a factor limits the output value to be multiple of it. The syntax and two examples are shown below.

$$\# \textit{parameter} = \{\textit{min}, \textit{max}, \textit{factor}\} \quad (\text{ii})$$

$$\# \textit{imm1} = \{4, 1000, 4\} \quad (2)$$

$$\# \textit{imm2} = \{-100, 400\} \quad (3)$$

In Example (2), the minimum and maximum values of parameter “#imm1” are 4 and 1000, and the factor is 4. When generator fetches the information of this parameter, it would output immediate integer value which is multiple of 4 and limit the value in the range of 4 to 1000. None for the factor in the Example (3) means the default value: 1. So parameter “#imm2” tells generator to produce immediate integer value in range of -100 to 400.

3.2.2 Instruction Syntax in Base Pattern

We provide an assembly-like syntax to edit the base pattern. Codes in base pattern would be converted into assembly code by the generator. For one line of code, it has several tokens which can be converted into segments of assembly code such as operation code, conditional code, registers, or operands depending on the assembly code syntax of target design. What segment a token will be converted into is decided by variable and parameter. The produced segments can be randomly generated from token individually or specified by user. Token has three kinds of format, and syntax is shown below.

$\$ \text{var}_1[\text{para}_1]$ (iii)

$\$[\text{para}_2]$ (iv)

$\$ \text{var}_3[]$ (v)

In the formats, “*var*” and “*para*” in each token mean variable and parameter. The dollar sign “\$” indicates a token that is recognized by the generator and makes generator start the generating process. For the generator, parameter means to generate new segment, and variable means to store or load segment. Format (iii) is the basic format of token. It has variable and parameter in the brackets. This format tells generator to produce segment from elements in the group “*para*₁” which is declared in parameter part and store the produced segment in variable “*var*₁”. Format (iv) is a token with only parameter in the brackets and tells generator only to produce segment from elements in the group “*para*₂”. In contrast to format (iii), generator would not save the segment when processing format (iv). Format (v) is a token with only variable in front the blank brackets. It tells generator to load segment from variable “*var*₃” that was stored before. Formats of token with corresponding actions in generator are listed in Table 3.2.

Table 3.2 Formats with corresponding actions in generator

	Format	Variable	Parameter	Action in generator
(iii)	$\$ \text{var}_1[\text{para}_1]$	Y	Y	Generate and Store
(iv)	$\$[\text{para}_2]$	N	Y	Generate
(v)	$\$ \text{var}_3[]$	Y	N	Load

A simple case is written for ARM ISA Version 5 [9] and shown in Figure 3.4. We take three tokens in this case which belong to three kinds of formats individually as examples. For the first line of base pattern, the first token “\$[d_2]” which only has parameter belongs to the format (iv). When the generator fetches this token, it would randomly choose one element in parameter “d_2” to replace this token and make it become one segment of assembly code. Figure 3.5 shows this process with one possible result.

```
.var
    reg1
.para
    shift = {lsl(5), lsr(2), asr(), ror()}
    d_2 = {tst(), teq(), cmp(), cmn()}
    r_1 = {r0(), r1(), r2(), r3(), r4(), r5(), r6(), r7(), r8()}
    cond = {eq(), ne(), cc()}
.main
    $[d_2]${cond} $reg1[r_1], r6, ${shift} $[r_1]
    $[d_2] $[r_1], $reg1[]
.end
```

Figure 3.4 Example of base pattern

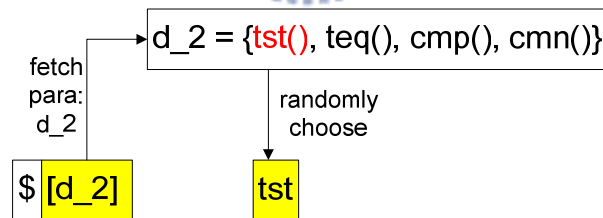


Figure 3.5 Possible result of token “\$[d_2]”

The third token “\$reg1[r_1]” that has both parameter and variable belongs to the format (iii). Generator would store segment in variable “reg1” after randomly choosing one element in parameter “r_1”. The last token “\$reg1[]” that has variable only in the second line belongs to the format (v) and tells the generator to load segment that was produced before. In this case, the loaded segment is the result of the third token

converted in the first line. The processes and possible results of converting these two tokens are shown in Figure 3.6. Characters, signs, and spaces which do not belong to token will be retained in the process. Generator repeats converting these tokens until fetch the line end of code. Figure 3.7 shows one possible outcome of every token in two lines after conversion and final assembly code.

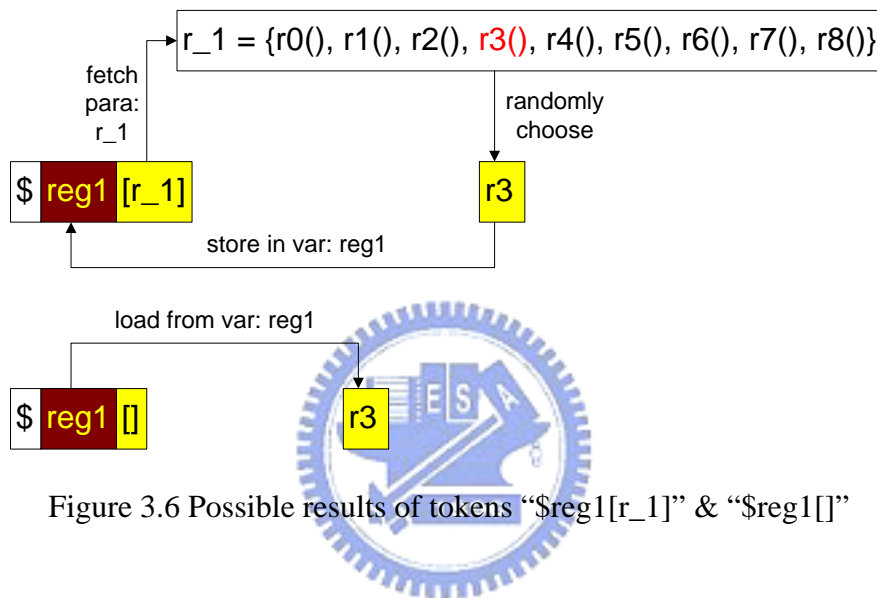


Figure 3.6 Possible results of tokens “\$reg1[r_1]” & “\$reg1[]”

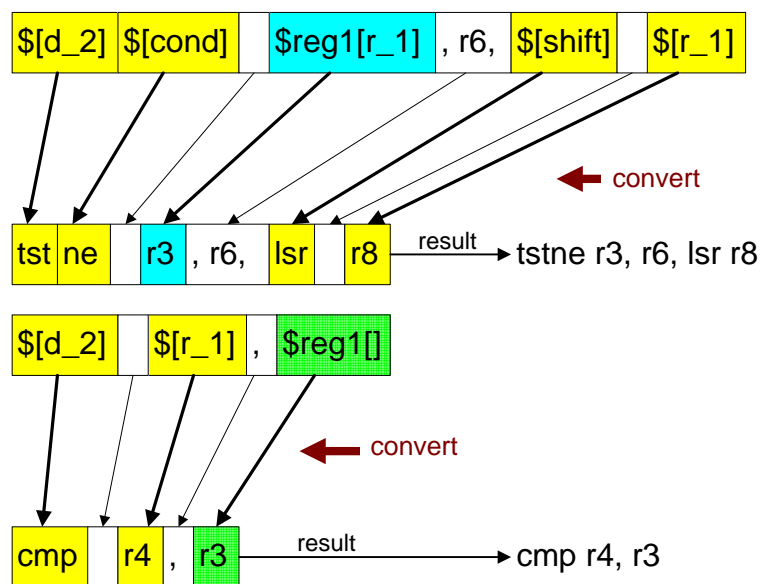


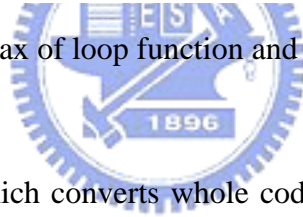
Figure 3.7 One possible outcome

3.2.3 Program Flow Mechanism in Base Pattern

Base pattern has two program flow mechanisms: loop function and random selection. Figure 3.8 shows the syntax of two. They would repeat codes in the braces basically. Compared with the loop function, the syntax of the random selection has additional weights in the parentheses behind dollar sign “\$” and “w”. Weights can be adjusted by user and decide the relative percentage of codes.

<pre> loop (loop count) { code₁ code₂ ⋮ code_n } </pre>	<pre> select (select count) { \$w(weight₁): code₁ \$w(weight₂): code₂ ⋮ \$w(weight_m): code_m } </pre>
---	--

Figure 3.8 Syntax of loop function and random selection



Unlike the loop function which converts whole codes sequentially for one loop, the random selection only selects one line of code in the braces to convert for one loop. For example, a base pattern has five lines of codes in the loop function with loop count n , five lines of codes in the random selection with loop count m . The codes in the loop function will be converted in the sequence which is shown in (4), and the codes in the random selection may be converted in the order shown in (5).

$$\begin{aligned}
 & (code_1 \rightarrow code_2 \rightarrow code_3 \rightarrow code_4 \rightarrow code_5)_{loop1} \rightarrow \\
 & (code_1 \rightarrow \dots \rightarrow code_5)_{loop2} \rightarrow \\
 & \quad \vdots \\
 & (code_1 \rightarrow \dots \rightarrow code_5)_{loopn}
 \end{aligned} \tag{4}$$

$$(code_3)_{loop1} \rightarrow (code_5)_{loop2} \rightarrow (code_1)_{loop3} \rightarrow (code_1)_{loop4} \rightarrow \dots \rightarrow (code_2)_{loopm} \tag{5}$$

When entering the random selection, generator calculates percentage of all codes in the braces first. For one loop, one code is randomly selected to be converted. It would repeat this flow until the count of random selection reaches zero. Random selection can produce random combination of multiple codes that would help user to test some corner cases. Figure 3.9 shows the flow chats of the loop function and the random selection.

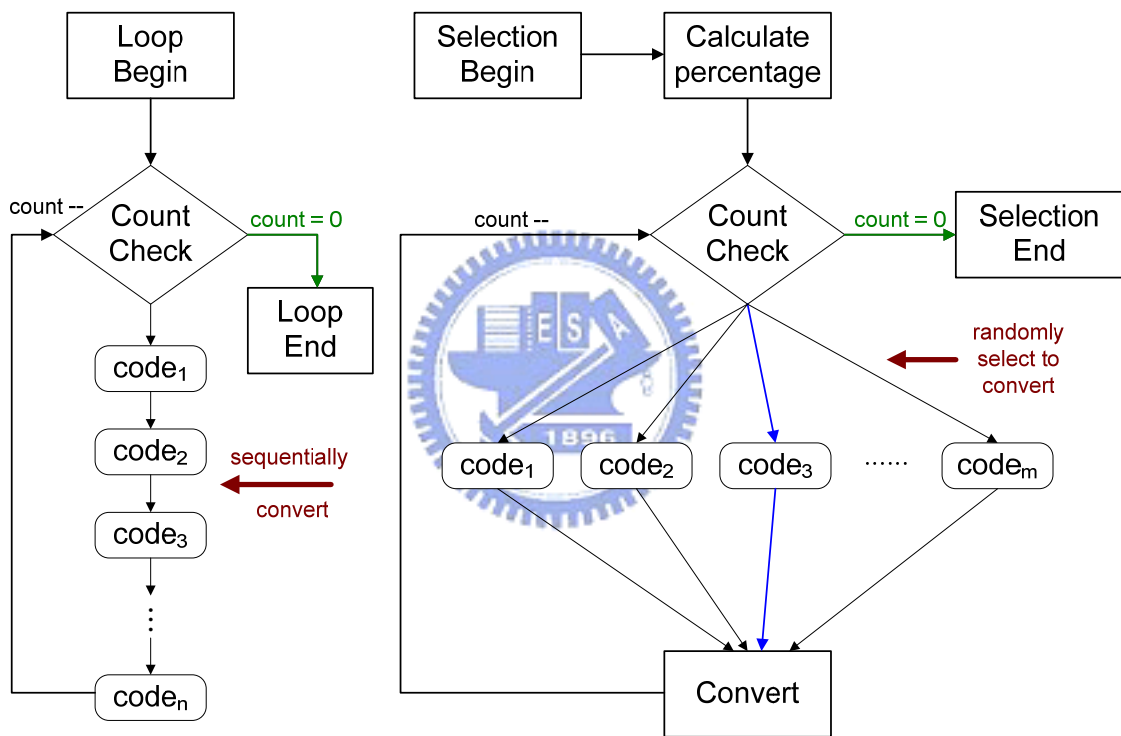


Figure 3.9 Flow chats of loop function and random selection

3.3 Constrained-Random Code Generator

Main function of the constrained-random code generator is to convert the content of script file into test pattern in machine code. The generator is written in SystemC and generates patterns for processor RTL and golden model on the fly. The architecture of the constrained-random code generator is shown in Figure 3.10 and can be divided into several parts. Connections between parts are also shown in Figure 3.10.

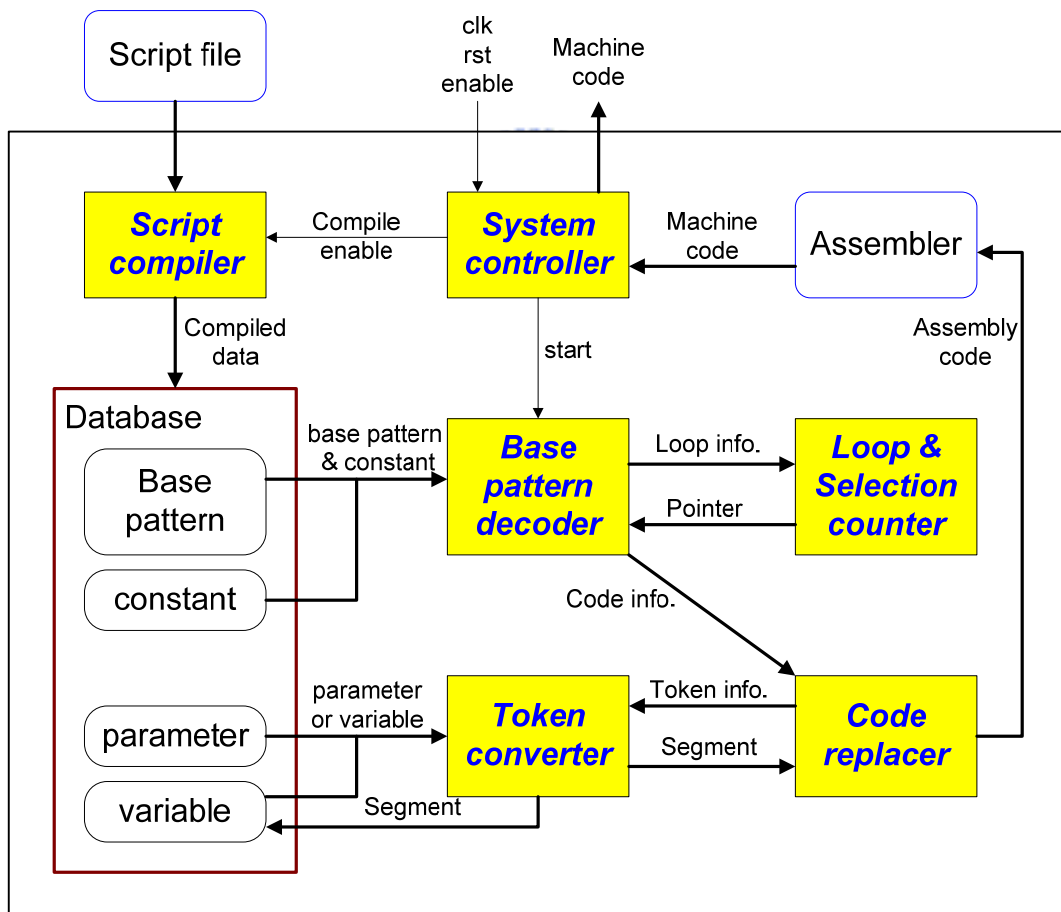


Figure 3.10 Architecture of the generator

- System controller: System controller handles the interface signals to the RTL and model and controls the actions of the Script compiler and the Base pattern decoder.
- Script compiler: Script compiler reads the script file and transforms the content into self defined data type then storing them in the Database.
- Base pattern decoder: Base pattern decoder fetches the content of the base pattern in the Database and decodes then proceeds one of the two actions:
 - ◆ Fetching the loop function or random selection command: The decoder sends the counts of loop, the beginning, and the end pointer to the Loop and Selection counter and receives the pointer of the next code.
 - ◆ Fetching the other code: The decoder sends the code information to Code replacer.
- Loop and Selection counter: The counter calculates the remaining loop count and also calculates the pointer of the next code in the base pattern then sending this pointer to the Base pattern decoder.
- Code replacer: The function sends the information of a token in the received code to the Token converter sequentially and replaces the token with the segment of assembly code returned from the Token converter. After all the tokens are converted, the function will send the complete assembly code to assembler.
- Token converter: Toke converter has three actions for converting a token into a segment of assembly code depending on the format of token: generating from parameter, loading from variable, and storing to variable. The generated segment is returned to the Code replacer.
- Assembler: Assembler function receives the assembly code from the Code replace function and transforms it into machine code for System controller to output.

When simulation starting, generator would set up internal parameters and compile user input script file after receiving the “reset” signal, then entering standby state. Action of generator in standby state for microprocessor RTL and golden model is like an instruction memory. It would output machine code of instruction when receiving “enable” signal. Figure 3.11 shows flow chat of constrained-random code generator.

Generator would fetch one line of code in base pattern and convert it into assembly code. In the process of converting code, there are four flow paths to deal with three kinds of token and characters which do not belong to token. Actions for three kinds of token are noted in Table 3.2 and show in Figure 3.11. For format (iii) and (iv), segment generating from parameter is the first step. Compared to format (iv), additional step for format (iii) is to store segment in variable. There is no generating from parameter in the flow path of converting format (v). Loading segment from variable is instead. Generator would save the produced segment in temporary then deals with the next token or other characters. Unlike tokens, characters which are not part of token are directly saved without generating or loading process.

When the end of the fetched code is read, generator transforms assembly code in temporary into machine code and outputs. Then generator would jump into standby state. Next time receiving enable signal, the next line of code in base pattern would be fetched and transformed into machine code. Generator repeats the process until fetches the end of base pattern.

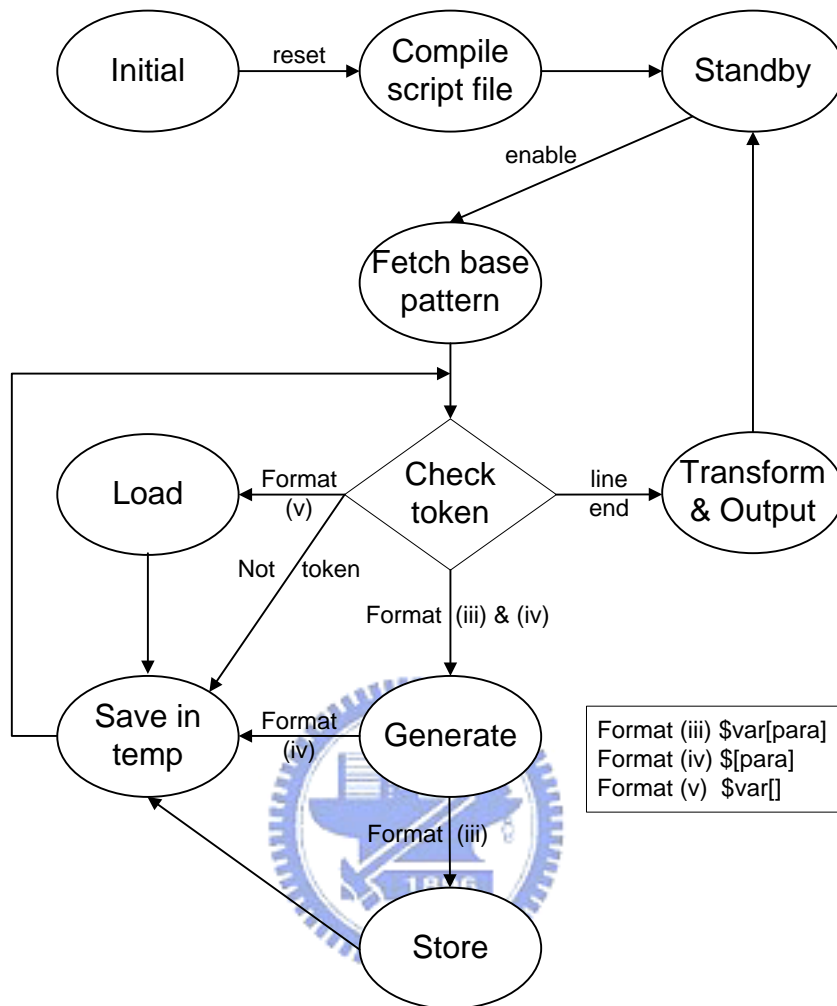


Figure 3.11 Flow chat of generator

Chapter 4 Introduction to Study Case: ACARM9

Processor Core Design

This chapter describes the in-house ACARM9 processor core design. An ARM9E-like [10], 32-bit RISC embedded processor core is implemented in the register-transfer level (RTL) with verilog language. Section 4.1 depicts the organization of ACARM9. Section 4.2 describes the implementation of multi-cycle multiplication, and Section 4.3 describes the verification strategy.

4.1 Organization of ACARM9

The verilog RTL of ACARM9 consists of 13 major functional blocks including the decoder, register file (RF), barrel shifter (BS), arithmetic/logical unit (ALU), 32x32 multiplier (MUL), count leading zero (CLZ), forwarding unit, program counter (PC) selector, operand fetcher (OF), load/store data address generator (DAG), read/write data operation and control unit. The block diagram is shown in Figure 4.1.

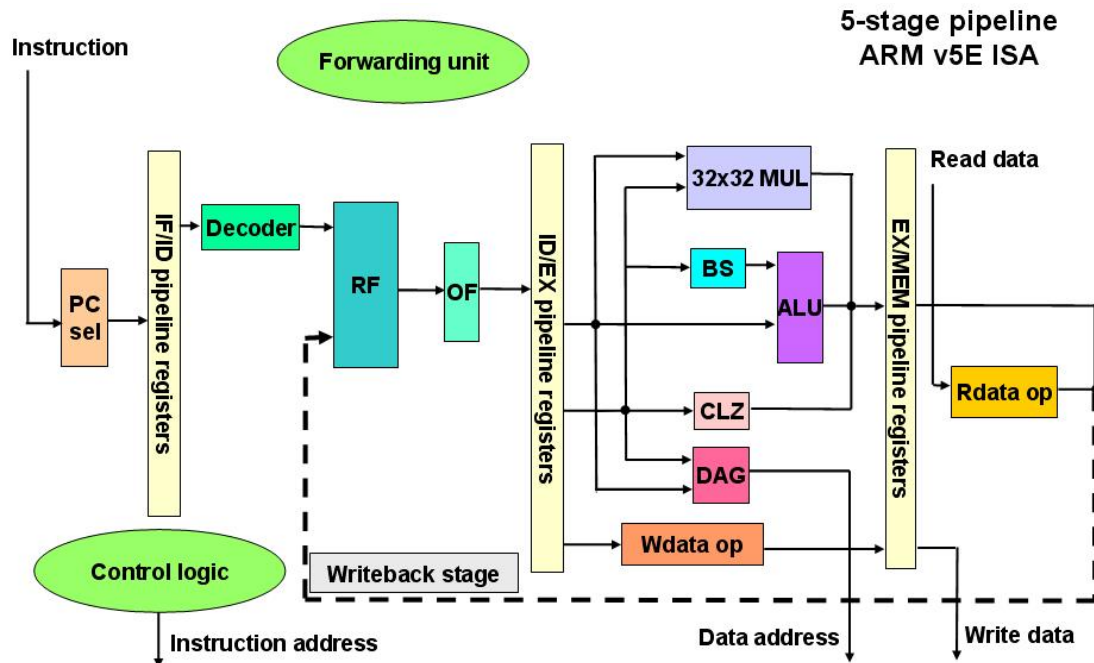


Figure 4.1 Block diagram of ACARM9

- The decoder unit obtains the instruction from IF phase and decodes it to generate all the information needed for the other functional unit.
- The register file is composed of total 37 registers – 31 general-purpose 32-bit registers and 6 status registers.
- The execution stage consists of a BS, an ALU, a CLZ, and a 32x32 multiplier. The arithmetic and logical operations are implemented with the ALU module whose second operand is received the BS to perform shift operations if needed. The 32x32 multiplier produces a 32-bit product. More details of multi-cycle multiply operation will be described in Section 4.2.
- The forwarding unit forwards the output data of EX stage to make sure that the next instruction can get these data as soon as possible.

- The program counter selector chooses on valid address from five sources including the PC incrementer, the ALU output, LDM (load multiple)/STM (store multiple) output, read data and the interrupt as shown in Figure 4.2.

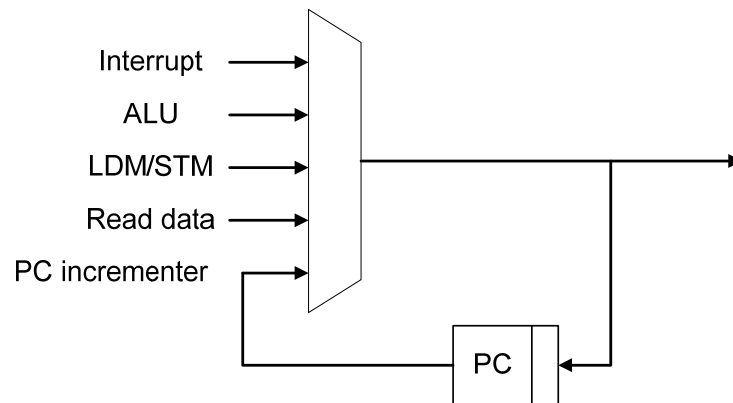


Figure 4.2 Source selection of the address registers

- The operand fetcher select data from register file, read data, ALU output, data address generator, immediate value or forwarding value as shown in Figure 4.3

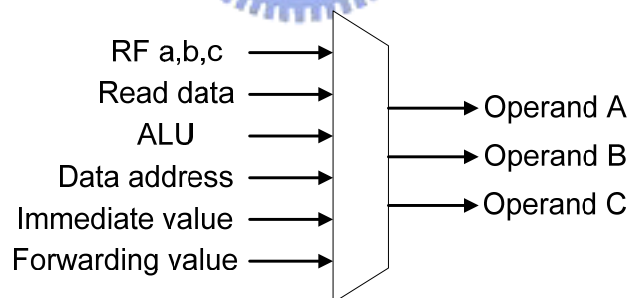


Figure 4.3 Operand selection

- The data address generator calculates the read/write data address on data bus includes the multiple load/store, and branch address on instruction bus.
- The control logic controls all the data flows of combinational logic and all the state transitions of sequential logic.

- The read/write data operation performs data alignment. As shown in Figure 4.4, the read operation shifts byte data and half-word data to the bottom of a 32-bit register with zero-extended or sign-extended when the processor reads byte data or half-word data from memory. For writing half-word data to memory, the write operation copies the low half-word part to the high half-word part to fill the 32-bit data width. For writing byte data to memory, the write operation copies least significant byte to the other three more significant bytes.

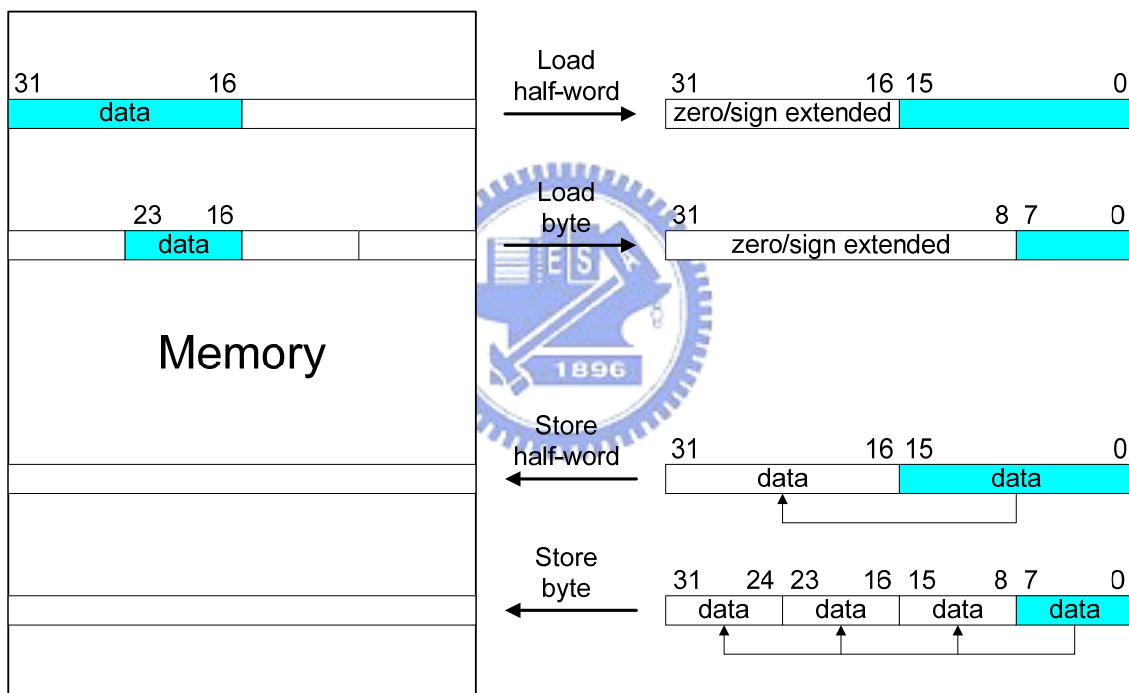


Figure 4.4 Read/write operation

4.2 Multi-cycle Multiplication

For a multiply instruction, source operand A and source operand B are fed to the multiplier directly to perform multiply operation. Figure 4.5 shows the multi-stage multiplication FSM.

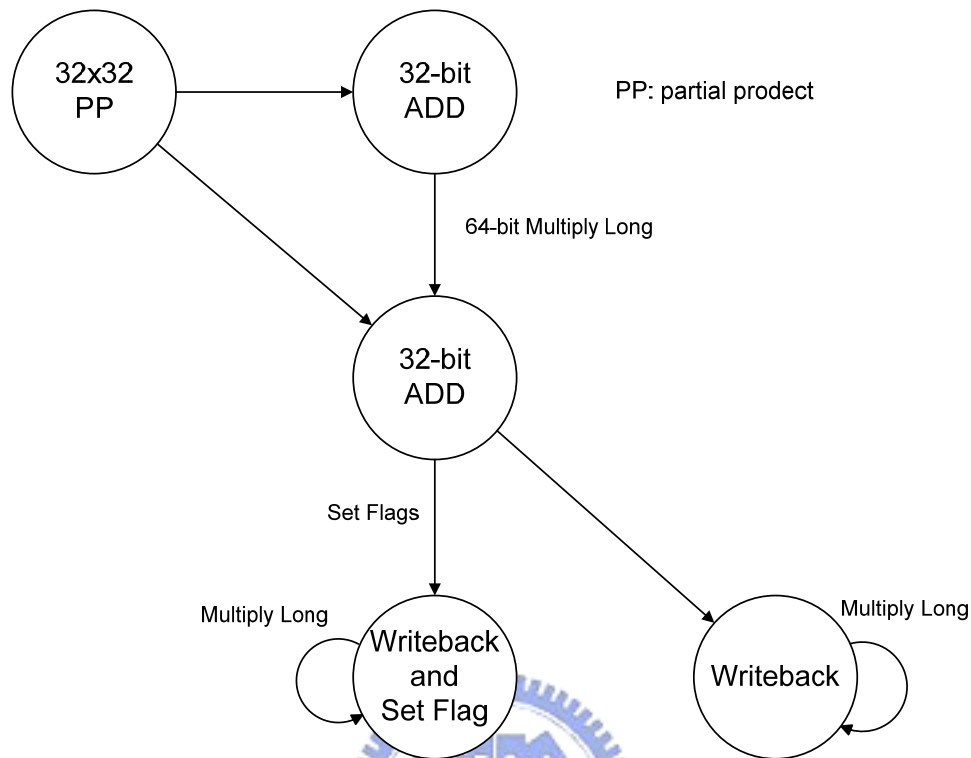


Figure 4.5 Multiplication FSM of ACARM9

A normal multiply instruction with/without accumulation needs two cycles, since two 64-bit partial products are obtained in the first on cycle. The tow low half 32-bit partial products and the value that is accumulated to the product are added by a carry save adder to generate two values, and then add the two values by a 32-bit adder to get final 32-bit result. While all 32-bit of product is valid, two cycles are needed. A long multiply instruction with/without accumulation needs three cycles, after the first cycle to obtain two 64-bit partial products, the 64-bit values need to take two cycles to perform 32-bit addition twice.

While the multi-cycle multiply operation is finished, a finish signal is sent from multiplication FSM to main FSM to tell that the multi-cycle operation is finished and the next instruction may get executed to continue the program flow.

4.3 Verification Strategy

A functional verification flow is proposed in the section. Section 4.3.1 proposes the overall functional verification flow. Section 4.3.2 proposes coding style checking by Linting; Section 4.3.3 proposes a deterministic verification approach.

4.3.1 Functional Verification

The functional verification flow diagram is listed in Figure 4.6 and in each of steps the RTL code is verified with a SystemC behavior model which is designed for matching all the cycle behaviors of ADS. All mismatches during the comparison will cause the flow back to RTL revision step for modification.

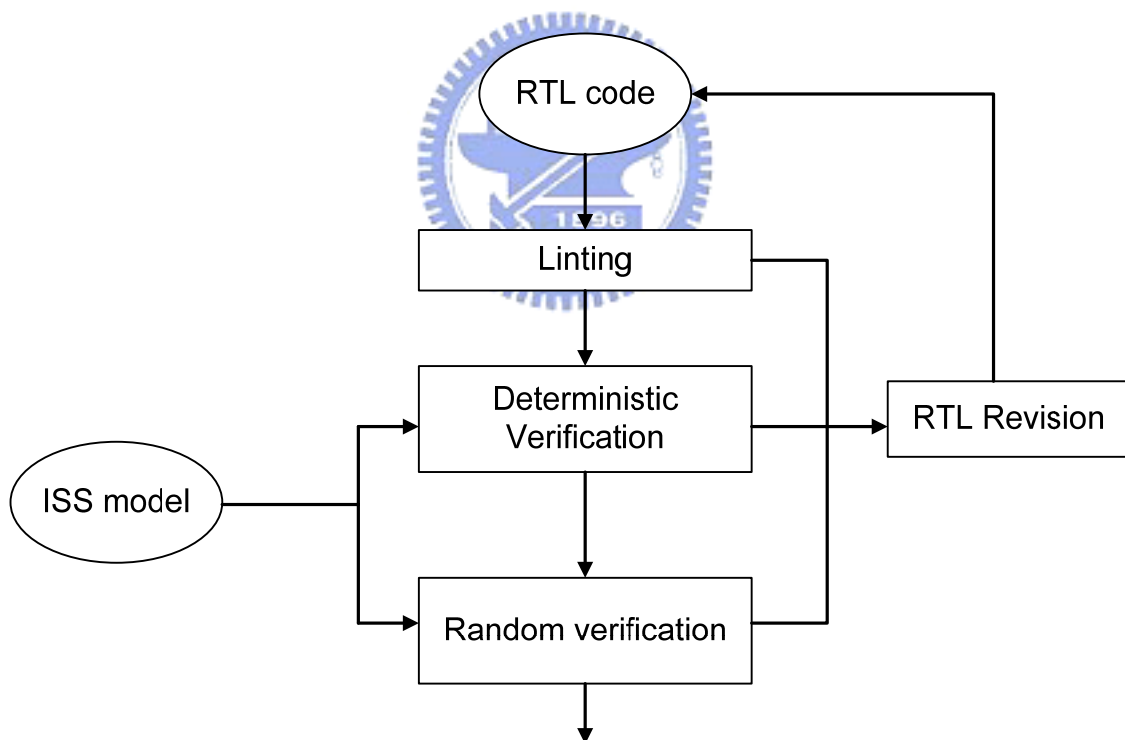


Figure 4.6 Functional verification flow

4.3.2 Coding Style Checking by Linting Free

This section describes a static coding style check which improves the quality of the design in reuse and verification perspectives of RTL code. Checking the coding style of the design by Novas nLint tool with 328 lint rules adopted from Freescale Semiconductor Reuse Standard (SRS) can avoid all kinds of warnings and errors including naming, synthesis simulation common syntax, undeclared objects, unexpected latches, DFT issue, and so on.

4.3.3 Deterministic Verification

This section describes deterministic verification which is made to check all regular cases and special corner case should be confronted with in the simulation phase. Deterministic verification is composed of three parts including specialized handcrafted pattern, real application pattern.

The handcrafted pattern is written in all cases of instructions implemented in the ACARM9 design. It checks all results of instructions to be right in the first step in deterministic verification.

The real application patterns are implemented by some benchmarks like Dhystone, Whetstone, and DSPstone. Moreover a JPEG encoder and an MP3 decoder program are also provided to verify the correctness of the design. This kind of pattern checks real cases met in applications of real world and is essential in deterministic verification phase.

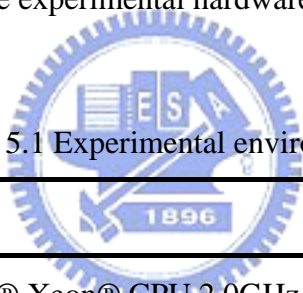
Chapter 5 Experimental Results

This chapter provides the experimental results of the proposed verification strategy. The experimental environment is described in Section 5.1. Section 5.2 presents that proposed strategy is used on ACARM9 processor core introduced in Chapter 4. Results of performance compared with pure-random are present in Section 5.3.

5.1 Experimental Environment

We conduct all the experiments on a HP wx8400 workstation. The commercial simulators are the Cadence NC-Verilog simulator and the Cadence NC-SC simulator. Table 5.1 shows the detail of the experimental hardware and software platform.

Table 5.1 Experimental environment



Hardware
CPU: Intel® Xeon® CPU 2.0GHz
RAM: DDR2-667 ECC FB-DIMM 14GB
Software
OS: CentOS 5 x86_64 (with Linux 2.6 kernel)
Cadence NC-Verilog version 6.1
Cadence NC-SC version 6.1

5.2 Cases Study

We use the proposed verification strategy on in-house ACARM9 core with ARM ISA version 5E [9] fully compatible. To check if the output of RTL correct, efficient two-layered cycle-accurate model [11] is used as a golden model.

5.2.1 Simulation Environment

We run co-simulation for the ACARM9 RTL code and the golden model and use the constrained-random code generator as instruction memory. Generator would send instructions in machine code to both RTL and model when receiving instruction enable signal which comes from RTL. We also use a comparator to collect all outputs of both RTL and model such as external signals and registers and compare them every cycle when running simulation. Comparator would dump detail information if the outputs of RTL and model have mismatches. The information includes signal name or register number, values, and cycle number. Figure 5.1 shows the environment.

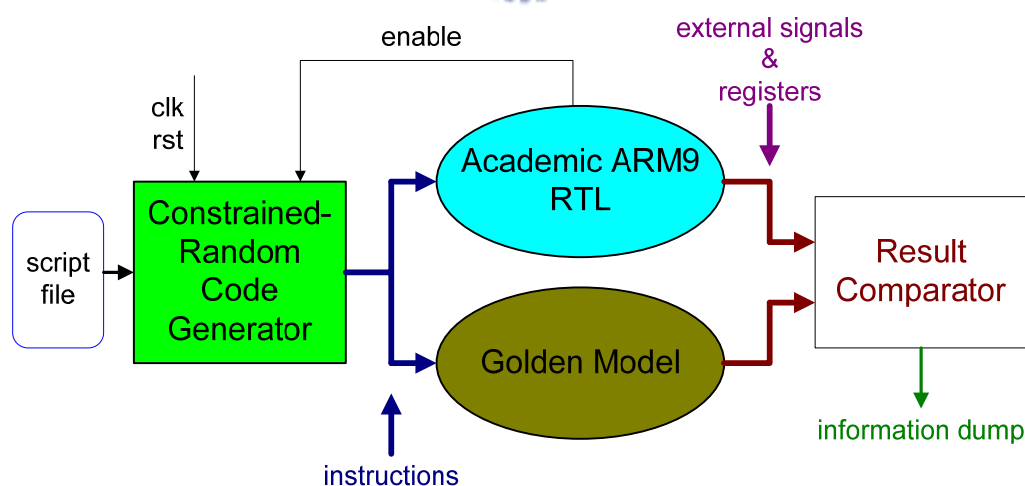


Figure 5.1 Simulation environment

5.2.2 Script Files and Bugs Found

To test different cases on the ACARM9 core, we write different base pattern with corresponding constraint setting in script files. The ACARM9 core has passed the deterministic verification and other real applications such as Dhrystone, Whetstone, and JPEG2000 encoder. Finally, it is used to execute an MP3 program on the FPGA successfully. However we still find two more bugs in the ACARM9 RTL code using the proposed method; moreover, two bugs in the model. The details of the four bugs with contents of script files used are listed below.

(1) Bug of Shift Operation in RTL

In ARM ISA, the operand shift operations can be parallelly executed with other operations. The script file of shift operation testing (1) which focuses on this point is shown in Figure 5.2. The first instruction in the base pattern would give random value to registers by loading from memory. We find one bug in the RTL code and one bug in the model individually by using this script file.

```
.const
  loop_count 100000
.para
  shift = {lsl(), lsr(), asr(), ror()}
  d_1 = {add(), adc(), sub(), sbc(), rsb(), rsc(), and(), eor(), oor(), bic()}
  d_2 = {tst(), teq(), cmp(), cmn(), mov(), mvn()}
  r_1 = {r0(), r1(), r2(), r3(), r4(), r5(), r6(), r7(), r8(), r9(), r10(), r11(),
        r12(), r13(), r14()}
  #imm1 = {0, 32}
.main
  ldmed r0, {r0-r14}
  loop (loop_count)
  {
    ${d_1} ${r_1}, ${r_1}, ${r_1}, ${shift} #${#imm1}
    ${d_1} ${r_1}, ${r_1}, ${r_1}, ${shift} ${r_1}
    ${d_2} ${r_1}, ${r_1}, ${shift} #${#imm1}
    ${d_2} ${r_1}, ${r_1}, ${shift} ${r_1}
  }
.end
```

Figure 5.2 Script file of shift operation testing (1)

The RTL code has calculation error in the logic shift right (*lsr*) operation with register specified shift amount when the shift amount is zero. In ARM ISA, the shift amount of shift operations has two sources: instruction specified or register specified. Different sources will cause different special operation for the particular value of shift amount. The correct outputs of logic shift right with zero shift amounts for different shift amount sources are listed in Table 5.2. For this case, the correct output should be the input operand without shift operation, but the RTL code recognizes it as special case in logic shift right operation with instruction specified shift amount: “*lsr #0*” which is used to encode “*lsr #32*”. It has a zero result with bit 31 of operand as the carry output.

Table 5.2 Logic shift right with zero shift amounts

	Shift amount source	Special case?	Encode	Output
Standard	Instruction specified	Y	<i>lsr#32</i>	Zero out, C = bit 31
	Register specified	N	<i>lsr#0</i>	No effect
RTL	Instruction specified	Y	<i>lsr#32</i>	Zero out, C = bit 31
	Register specified	<i>Y</i>	<i>lsr#32</i>	<i>Zero out, C = bit 31</i>
Note: “C” means the carry-out flag				

(2) Bug of Shift Operation in Model

The model has one bug which is found by the script file of shift operation testing (1), too. The bug happens when the RTL executes the three kinds of shift operation except rotation right with register specified shift amount as the shift amount equal to or greater than 32.

In ARM ISA, only the least significant byte (bit 0 to bit 7) of the contents of the shift amount register is used to determine the shift amount. But the model takes only least 5 bits (bit 0 to bit 4) of the shift amount register as the shift amount. Figure 4.3 shows the different bit-range of the shift amount register used as shift amount in standard and the model. The correct results of this case are result zero for both logic shift left and right, and result filled with bit 31 of the operand for arithmetic shift right. The outputs of the model are shift operation with shift amount which is the remainder of the contents of the register divided by 32. Table 4.3 lists the different results of the three kinds of shift operations in standard and the model.

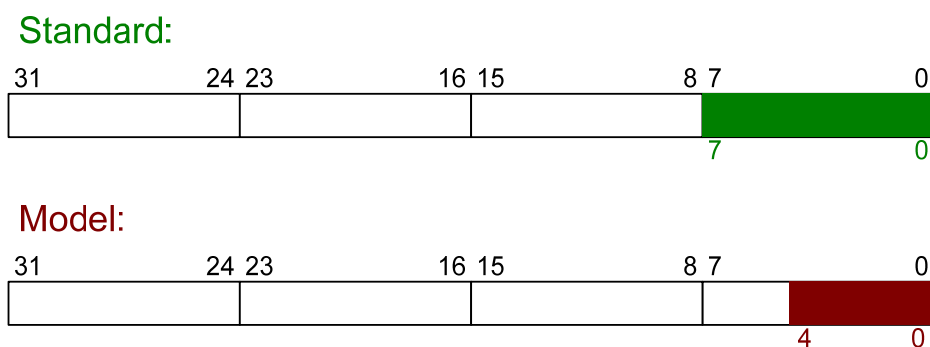


Figure 5.3 Different used bit-range of shift amount register.

Table 5.3 Results of shift operations in standard and the model

	Shift type	Shift amount = 32	Shift amount > 32
Standard	Logic shift right	Zero out, C = bit 31	Zero out, C = 0
	Logic shift left	Zero out, C = bit 0	Zero out, C = 0
	Arithmetic shift right	Filled with bit 31, C = bit 31	
Model	Logic shift right	<i>No effect</i>	<i>Operand shifted with shift amount mod 32, C = corresponding bit</i>
	Logic shift left	<i>No effect</i>	
	Arithmetic shift right	<i>No effect</i>	
<p>Note: shift amount means the least significant byte of the shift amount register</p> <p>“C” means the carry-out flag</p>			

(3) Bug in Usage of the Program Counter (PC)

For the ARM core architecture, register 15 (r15) holds the Program Counter (PC) and has limits when used as a destination register or an operand. We change the script file of shift operand testing (1) by adding a new parameter which has higher probability to produce r15 and modifying tokens in the base pattern. Figure 5.4 shows the script file of shift operand testing (2). One more bug in the model is found by using this script file.

In this script file, r15 is used as an operand. The value of r15 will be different depending on the source of shift amount. It will be the address of the instruction, plus 8 bytes for instruction specified shift amount or 12 bytes for register specified shift amount. The model fetches the value of r15 which is the address of the instruction plus 8 bytes for both cases. Table 5.4 lists the value of r15 for standard and the model with different shift amount source.

```

.const
    loop_count 100000
.para
    shift = {lsl(), lsr(), asr(), ror()}
    d_1 = {add(), adc(), sub(), sbc(), rsb(), rsc(), and(), eor(), oor(), bic()}
    d_2 = {tst(), teq(), cmp(), cmn(), mov(), mvn()}
    r_1 = {r0(), r1(), r2(), r3(), r4(), r5(), r6(), r7(), r8(), r9(), r10(), r11(),
           r12(), r13(), r14()}
    r_2 = {r0(), r1(), r2(), r3(), r4(), r5(), r6(), r7(), r8(), r9(), r10(), r11(),
           r12(), r13(), r14(), r15(30)}
    #imm1 = {0, 32}
.main
    ldmed r0, {r0-r14}
    loop (loop_count)
    {
        ${d_1} ${r_1}, ${r_2}, ${r_2}, ${shift} #${#imm1}
        ${d_1} ${r_1}, ${r_2}, ${r_2}, ${shift} ${r_1}
        ${d_2} ${r_1}, ${r_2}, ${shift} #${#imm1}
        ${d_2} ${r_1}, ${r_2}, ${shift} ${r_1}
    }
.end

```

Figure 5.4 Script file of shift operand testing (2)

Table 5.4 Value of r15 fetched as operand in different source of shift amount

	Instruction specified	Register specified
Standard	PC + 8	PC + 12
Model	PC + 8	<i>PC + 8</i>

(4) Bug in Combination of Two Multiplication Instructions

We find one more bug in the RTL code by the script file shown in Figure 5.5. This script file focuses on all instructions about multiplication in ARM ISA version 5E; moreover, random selection is used to make random combinations of these multiplication instructions.

```
.const
    select_count 500000
.para
    m_1 = {mla(), smlal(), smull(), umlal(), umull()}
    m_2 = {smla(), smlal()}
    xy = {b(), t()}
    r_1 = {r0(), r1(), r2(), r3(), r4(), r5(), r6(), r7(), r8(), r9(), r10(), r11(),
          r12(), r13(), r14()}
.main
    ldmed r0, {r0-r14}
    select (select_count)
    {
        $w(5):$[m_1] $[r_1], $[r_1], $[r_1], $[r_1]
        $w():mul $[r_1], $[r_1], $[r_1]
        $w(8):$[m_2]${xy}${xy} $[r_1], $[r_1], $[r_1], $[r_1]
        $w(4):smul${xy}${xy} $[r_1], $[r_1], $[r_1]
        $w(2):smulw${xy} $[r_1], $[r_1], $[r_1]
        $w(2):smlaw${xy} $[r_1], $[r_1], $[r_1], $[r_1]
    }
.end
```

Figure 5.5 Script file of the multiplication instructions

The bug happens when the RTL sequentially executes a signed or unsigned multiply long (*smull* or *umull*) instruction or a signed or unsigned multiply accumulate long (*smlal* or *umlal*) instruction after an Enhanced DSP instruction: 16-bit signed integer multiply (*smul*<*x*><*y*>) instruction. The multiply long instruction will output total 64-bit result and divide it into higher 32-bit part and lower 32-bit part then output sequentially in two cycles. But the lower 32-bit part of the result of the signed multiply long instruction would become the output of the previous instruction (*smul*<*x*><*y*>).

As the description in Section 4.2, we use a 32x32 multiplier in the RTL code for multiplication and modify it such that the multiplication becomes a multi-cycle operation to fit the timing constraint. A simple block diagram of multiplier in RTL is shown in Figure 5.6, and the multiplication FSM is shown in Figure 4.5. The multiplier deals with multiplication instructions which belong to multiply long instructions and have 64-bit output in three-cycle operation and other normal multiplication instructions have only 32-bit output in tow-cycle operation. When the multiplier handles a multiplication instruction, the controller will save some signals from the decoder in the status registers and control whole the process. The status registers can be separated into several parts which save corresponding control signals for detail functions like signed/unsigned, accumulation, DSP operation, and multiply long, etc. After finishing calculating, the result will be saved in the output register and the controller will clear the status registers and jump into standby state. The multiplier will output the lower 32-bit part of result in the second cycle and higher part in the third cycle for multiply long instructions and output result only in the second cycle for other multiplication instructions.

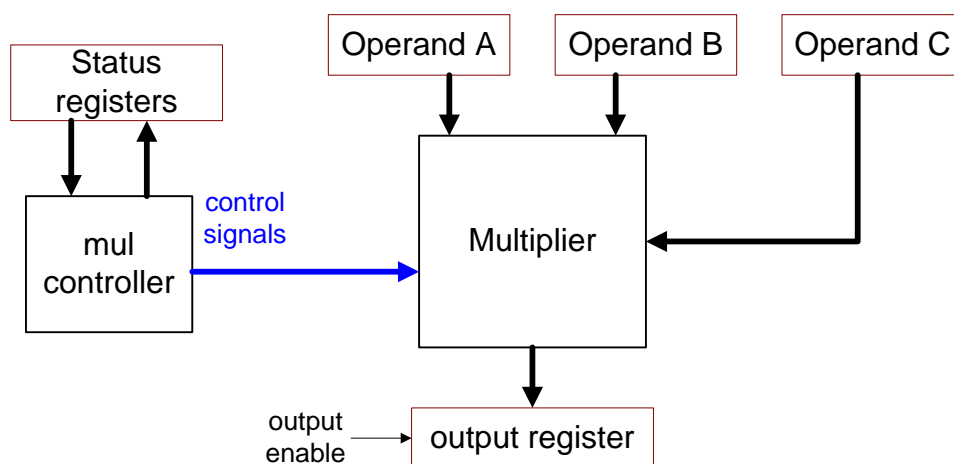


Figure 5.6 Block diagram of multiplier in RTL

But the controller does not clear the status registers after finishing $smul\langle x \rangle \langle y \rangle$ instruction. If other multiplication instruction except the multiply long instructions is executed after $smul\langle x \rangle \langle y \rangle$ instruction, some of the status registers which store the control signals about $smul\langle x \rangle \langle y \rangle$ instruction would be updated and the multiplier would work as normal. If a multiply long instruction is executed after $smul\langle x \rangle \langle y \rangle$ instruction, these status registers would not be all updated. The output register will be locked (output enable signal is low) in the second cycle of the process of multiply long instruction such that the lower 32-bit part of result becomes the result of previous instruction ($smul\langle x \rangle \langle y \rangle$). In the third cycle, the controller sets the output enable signal as “high” and the higher 32-bit part of result is outputted correctly. Table 5.5 lists the all results of the instruction $smul\langle x \rangle \langle y \rangle$ and multiply long instruction in RTL and standard when multiply long instruction is executed after $smul\langle x \rangle \langle y \rangle$ sequentially.

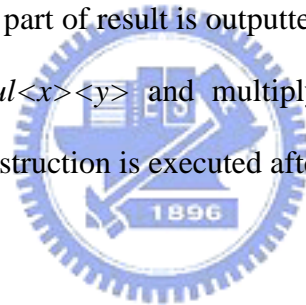


Table 5.5 Output of multiplier in standard and RTL

	Multiply Instruction	Output				
		Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
Standard	$smul\langle x \rangle \langle y \rangle$	None	Result S			
	Multiply long			None	Result lo	Result hi
RTL	$smul\langle x \rangle \langle y \rangle$	None	Result S			
	Multiply long			None	Result S	Result hi

Note: suppose that the output of $smul\langle x \rangle \langle y \rangle$ is “Result S” and the correct outputs of multiply long are “Result lo” and “Result hi”

5.3 Compare with Pure Random Verification

We use a pure random code generator to compare the performance with the proposed constrained-random code generator. The pure random one is also written in SystemC and will output legal machine codes randomly without any constraint. The constrained-random code generator and the pure random code generator are used as an instruction memory in the co-simulation environment individually to measure performance. We run 100,000,000-cycle simulation for both generators and compare their simulation time. The simulation time for each generator are shown in Table 5.6. The proposed generator has about 35% overhead in simulation time when compared to pure random one.

Table 5.6 Simulation time of two generators

Generator	Simulation time (seconds)
Constrained-random	8,314
Pure random	6,156

Chapter 6 Conclusions and Future Works

A constrained-random code generator has been presented in this thesis. The proposed verification strategy provides an all-purposed syntax in the user input script file that helps generate test patterns efficiently. It can define the generated range of every segment in assembly code such as operation code, conditional code, operands, and immediate value. By changing a small part of constraint setting in the script file, generator can easily output different patterns to cover different corner cases. It can also define the relation between segments to test some corner cases such as data hazards. Moreover, the program flow of the output pattern can be also controlled by the base pattern in script file. The generator can output the codes in sequence as the order in the script file or randomly schedule the codes. Finally, the generator is applied on in-house ARM9 core and finds out more bugs even if this core has passed many verification strategies before. However, the proposed strategy cannot cover the verification of external interrupt behaviors at this moment. Consequently, an advance method [12] may be applied to our strategy to test the external interrupt behaviors of a processor.

References

- [1] Marcin Kazmierczak, “White-box verification techniques in Networking ASIC Design”, Thesis Dissertation, Department of Information Technology, Lund Institute of Technology, Sep 2001.
- [2] Janick Bergeron, Writing testbenches: functional verification of HDL models, Kluwer Academic Publishers, Norwell, MA, 2000.
- [3] Source: Collett International 2000.
- [4] Nathan Kitchen, Andreas Kuehlmann, “Stimulus generation for constrained random simulation,” *IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 258-265.
- [5] Prabhat Mishra, Nikil Dutt, “Functional Coverage Driven Test Generation for Validation of Pipelined Processors,” *Design, Automation, and Test in Europe*, pp. 678-683, 2005.
- [6] Ilya Wagner, Valeria Bertacco, Todd Austin, “StressTest: an automatic approach to test generation via activity monitors,” *Annual ACM IEEE Design Automation Conference*, 2005, pp. 783-788.
- [7] Prabhat Mishra, Nikil Dutt, “Specification-driven directed test generation for validation of pipelined processors,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, Issue 3, 2008.
- [8] Jason C. Chen, Synopsys Inc, Applying Constrained-Random Verification to Microprocessors, Dec 2007 at <http://www.edadesignline.com/howto/204800266>
- [9] ARM, ARM Architecture Reference Manual.
- [10] ARM, ARM9E-S Core Revision: r2p1 Technical Reference Manual.

- [11] Chien-De Chiang, Juinn-Dar Huang, “Efficient Two-Layered Cycle-Accurate Modeling Technique for Processor Family with Same Instruction Set Architecture.” In *Proceedings of International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2009, pp. 235-238.
- [12] Fu-Ching Yang, Wen-Kai Huang, Ing-Jer Huang, “Automatic Verification of External Interrupt Behaviors for Microprocessor Design,” *DAC*, 2007, June.
- [13] S. Fine and A. Ziv. “Coverage directed test generation for functional verification using bayesian networks.” In *Proceedings of Design Automation Conference (DAC)*, 2003, pp. 286–291.
- [14] A. Aharon and D. Goodman and M. Levinger and Y. Lichtenstein and Y. Malka and C. Metzger and M. Molcho and G. Shurek. “Test program generation for functional verification of PowerPC processors in IBM.” In *Proceedings of Design Automation Conference (DAC)*, 1995, pp. 279–285.
- [15] J. Miyake and G. Brown and M. Ueda and T. Nishiyama. “Automatic test generation for functional verification of microprocessors.” In *Proceedings of Asian Test Symposium (ATS)*, 1994, pp. 292–297.
- [16] J. Shen and J. Abraham and D. Baker and T. Hurson and M. Kinkade and G. Gervasio and C. Chu and G. Hu. “Functional verification of the equator MAP1000 microprocessor.” In *Proceedings of Design Automation Conference (DAC)*, 1999, pp. 169–174.
- [17] P. Mishra and N. Dutt. “Graph-based functional test program generation for pipelined processors.” In *Proceedings of Design Automation and Test in Europe (DATE)*, pp. 182–187, 2004.

- [18] H. Iwashita and S. Kowatari and T. Nakata and F. Hirose. “Automatic test pattern generation for pipelined processors.” In *Proceedings of International Conference on Computer-Aided Design (ICCAD)* , 1994, pp. 580–583.
- [19] S. Ur and Y. Yadin. “Micro architecture coverage directed generation of test programs.” In *Proceedings of Design Automation Conference (DAC, 1999)*, pp. 175–180.
- [20] Cadence Specman Elite V5.0 For Linux, at http://www.cadence.com/products/functional_ver/specman_elite/index.aspx

