

# 國立交通大學

資訊工程系

碩士論文

魚骨：無時脈高能源效率堆疊



Fish-Bone: A Clock-less Power-efficient Stack

研究生：沈銘峰

指導教授：陳昌居 博士

中華民國九十四年六月

魚骨：無時脈高能源效率堆疊

# Fish-Bone: A Clock-less Power-efficient Stack

研究生：沈銘峰

Student : Ming-Fung Shen

指導教授：陳昌居

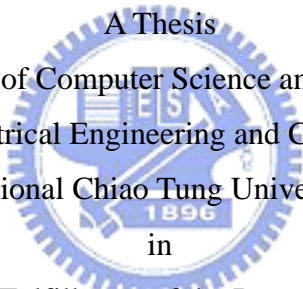
Advisor : Chang-Jiu Chen

國立交通大學

資訊工程學系

碩士論文

A Thesis  
Submitted to Department of Computer Science and Information Engineering  
College of Electrical Engineering and Computer Science  
National Chiao Tung University  
in  
partial Fulfillment of the Requirements  
for the Degree of Master  
in  
Computer Science and Information Engineering  
June 2005

The logo of National Chiao Tung University is a circular seal. It features a blue outer ring with the university's name in Chinese characters. Inside the ring, there is a central emblem with a book and a torch, and the year '1896' is inscribed at the bottom of the inner circle.

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

# 魚骨:無時脈高能源效率堆疊

研究生：沈銘峰

指導教授：陳昌居

國立交通大學 資訊工程學系

## 摘要

低耗電的設計在現在以及未來都是設計的主要方針，尤其在不失速度效能的前提下，低耗電的設計更獨具其優勢。過去有些追求速度效能的研究設計，但是其能源消耗都不盡理想。在此論文中，我們提出一個無時脈高能源效率的堆疊設計以及實作。基於 GasP 的設計模組概念作延伸，利用主從暫存概念，以及增加暫存，盡可能減少資料在堆疊內部裡的搬運次數來減低能源的消耗。和[1]的設計方法比較，在 42 個一位元儲存空間的堆疊實驗中，於 HSPICE 在速度 2 GHz 下以 UMC 0.18 製程模擬結果指出，根據不同的輸入命令序列其平均耗電量可以省電達該設計約百分之十七。然而與線性堆疊比較下，實驗之平均耗電量魚骨堆疊更只有其設計之百分之八點三左右。魚骨堆疊的每個堆疊動作的執行時間和儲存空間、資料的長度及命名序列長度沒有關係，其時間是固定的，而其能源消耗則和堆疊命令序列有絕對的相關性。

# **Fish-Bone: A Clock-less Power-efficient Stack**

**Student : Ming-Fung Shen Advisor : Dr. Chang-Jiu Chen**

**Department of Computer Science and Information Engineering**

National Chiao-Tung University

## **Abstract**

Low-power circuit design is the fashion of the future design guideline, especially not losing the speed performance. So far some research on stack is performed. However, their power performances are not good enough. The purpose of this paper is to present a power-efficient stack with clock-less design technique and its implementation. Based on GasP asynchronous control family from Sun Microsystems laboratories, we reduced the data movements in stack with the concept of master-slave temporal storages and n-place linear storages to low down the power consumption. For the ease of comparing with other targets in experiment, we implemented our stack with the same number of storages as the target's one. Results from HSPICE simulations with UMC 0.18 model file show that our stack saved averagely 17% in power consumption with 100 random command sequences that are sized 100 compared to the re-implementation of original design in [1]. More than that, we gained averagely 91.39% in power consumption compared to linear stack. The cycle time is independent of the number of data items in the stack and the data width. It has constant time property. The energy consumption per stack operation depends on the sequence of stack operations.

# Acknowledgment

完成這篇論文要感謝很多夥伴，首先感謝指導老師陳昌居老師在這兩年內給予的環境和指導。特別感謝黃年時學長，在完成這篇論文的期間一起熬夜討論以及引導，以及陳仕偉同學的討論協助及幫忙。也感激同實驗室的夥伴們曾經提供的協助以及生活上的點點滴滴。最重要的是感謝我的家人給我的支持，讓我能夠穩定的求學以及研究。



# CONTENTS

摘要 .....	i
Abstract.....	ii
Acknowledgment.....	iii
CONTENTS .....	iv
LIST OF FIGURES .....	vi
LIST OF TABLES.....	viii
CHAPTER 1 INTRODUCTION.....	1
1.1 BENEFITS WITH ASYNCHRONOUS DESIGN .....	1
1.2 THE CORE OF STACK MACHINES .....	2
1.3 OBVIOUS IMPLEMENTATIONS AND PRIOR WORKS .....	3
1.4 MORE POWER-EFFICIENT STACKS.....	4
CHAPTER 2 RELATED WORKS.....	6
2.1 STACK WITH RAM IMPLEMENTATION.....	6
2.2 STACK WITH LINEAR ARRAY OF CELLS .....	7
2.3 A HYBRID STACK CONSISTS OF TREE STACKS AND THREE-LEVEL THREE-PLACE LINEAR STACKS .....	9
2.3.1 <i>A Design Based on GasP Modules</i> .....	10
2.3.2 <i>Tree Stacks</i> .....	12
2.3.3 <i>Three-level Three-place Linear Stacks</i> .....	13
2.3.4 <i>The Conversion from FSM to GasP Modules</i> .....	15
2.3.5 <i>A Hybrid Stack</i> .....	18
2.4 THE FIRST BREATH OF FISH-BONE STACKS .....	20
CHAPTER 3 FISH-BONE STACK .....	23
3.1 MOTIVATIONS AND IDEAS .....	23
3.1.1 <i>Benefits form Reducing Data Item Moves</i> .....	23
3.1.2 <i>A Solution with Reducing Data Item Moves</i> .....	23
3.2 ARCHITECTURE OF FISH-BONE STACKS .....	27
3.2.1 <i>The Internal Actions in Fish-Bone Stacks</i> .....	27
3.2.2 <i>An Example of Executing A Small Command Sequence</i> .....	29
3.3 THE IMPLEMENTATION OF CONTROL PATH.....	34
3.4 REAL IMPLEMENTATION OF A FISH-BONE STACK.....	41
3.4.1 <i>Top View of A Fish-Bone Stack</i> .....	41

3.4.2 Implementation of GasP Modules .....	42
3.4.3 Implementation of State Keeper.....	44
3.4.4 Implementation of Data Storage.....	45
3.4.5 Implementation of Condition Maintainer .....	46
<b>CHAPTER 4 IMPLEMENTATION IMPROVEMENTS.....</b>	<b>47</b>
<b>CHAPTER 5 THE EXTENSION OF STACK PLACE .....</b>	<b>50</b>
5.1 STRAIGHTFORWARD EXTENSION .....	50
5.2 EXTENSION IN TREE STRUCTURE .....	50
5.3 EXTENSION IN RECURSIVE STRUCTURE .....	51
5.4 TREE STRUCTURE USED IN POWER CONSUMPTION COMPARISON.....	51
<b>CHAPTER 6 POWER CONSUMPTION EVALUATION .....</b>	<b>52</b>
6.1 POWER CONSUMPTION TESTING ON A THREE-LEVEL THREE-PLACE LINEAR STACK AND A FISH-BONE STACK .....	52
6.2 POWER CONSUMPTION TESTING ON BOTH STACKS WITH DIFFERENT LEAF NODE DESIGNS .....	53
6.3 DATA MOVEMENTS COUNTING .....	57
6.4 USED TRANSISTORS COUNTING .....	59
6.5 ANALYSIS OF RESULT FROM SIMULATIONS.....	60
<b>CHAPTER 7 CONCLUSIONS AND FUTURE WORKS .....</b>	<b>62</b>
<b>REFERENCE.....</b>	<b>63</b>

# LIST OF FIGURES

Figure 2-1: Stack with RAM implementation. ....	6
Figure 2-2 : A simple linear stack .....	8
Figure 2-3 : GasP with self-resetting NAND.....	11
Figure 2-4 : A two-place tree stack .....	12
Figure 2-5 : A two-place tree stack: Finite state machine specification. ....	13
Figure 2-6 : A three-level three-place linear stack .....	15
Figure 2-7 : A three-place linear stack cell.....	15
Figure 2-8 : the converting from FSM to GasP module. ....	17
Figure 2-9 : a hybrid stack with 42 storage places consists of a tree stack and 3-level 3-place linear stacks.....	19
Figure 2-10 : the structures of two type of stack.....	20
Figure 2-11 : The difference of internal data moves between 3-level 3-place linear stack and Fish-Bone stack.....	22
Figure 3-1 : The data-path of a three-level three-place linear stack .....	25
Figure 3-2 : the data-path of a Fish-Bone stack .....	26
Figure 3-3 : Continuous put commands on an empty Fish-Bone stack .....	32
Figure 3-4 : Continuous get commands on an empty Fish-Bone stack.....	33
Figure 3-5 : A Fish-Bone stack: Finite state machine specification.....	36
Figure 3-6 : the diagram of control path converted from FSM of Fish-Bone stack .....	40
Figure 3-7 : a top view of a complete Fish-Bone stack with both control-path and data-path.....	42
Figure 3-8 : the modification from the original designs .....	43
Figure 3-9 : the circuit of a keeper .....	45



**Figure 3-10 : the implementation of data storage .....45**  
**Figure 3-11 : the implementation of condition maintainer .....46**  
**Figure 4-1 : The difference of two kind handling with condition signals .....48**  
**Figure 4-2 : the view of the final circuit of a Fish-Bone stack .....49**  
**Figure 5-1 : Extension in a recursive structure .....51**  
**Figure 6-1 : the trend of the relation between the difference of moves and the  
P(HS)/P(FB).....61**



# LIST OF TABLES

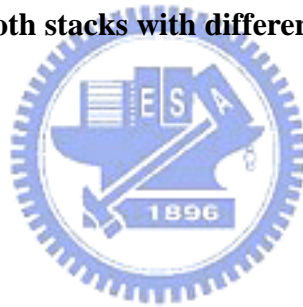
**Table 3-1 : The events stimulated by relative GasP modules in round-robin fashion and the data flow controlled by the six states in Fish-Bone stacks ..35**

**Table 6-1 : The difference of average power consumption between one 3-level 3-place linear stack and one Fish-Bone stack.....53**

**Table 6-2 : Results of average power consumption between the Hybrid stack with 3-level 3-place linear stacks (HS), the Hybrid stack with Fish-Bone stack (FB), and the linear stack (LS) .....57**

**Table 6-3 : The relation between the power consumption and the number of their data movements.....59**

**Table 6-4 : the counting of both stacks with different leaf node designs .....60**



# Chapter 1 Introduction

The goal of this thesis is to implement an asynchronous fast and more power-efficient stack. In this chapter, first, a briefly introduction to asynchronous design is depicted. After that, we introduce the need of the fast and power efficient stack hardware design. Third, obvious implementations and some other related researches done in this field are exhibited. Then, we introduce the idea of our design, Fish-Bone. Finally, the organization of a Fish-Bone stack is roughly described.

## 1.1 Benefits with Asynchronous Design

Asynchronous circuits keep the one of two major assumptions, signals are binary, but remove the other assumption that time is discrete. The asynchronous design style has several possible benefits:

*No clock skew* - Clock skew is the difference in arrival time of the clock signal at different parts of the circuit. Because of the definition of asynchronous circuits, no globally distributed clock, we don't need to worry about the clock skew. However, system with synchronous design often slows down their circuits to accommodate the skew.

*Average-case instead of worst-case performance* - Many asynchronous systems sense only a computation has completed, allowing them to exhibit average-case performance. Synchronous circuits must wait until all possible computations have completed, yielding worst-case performance.

*No global timing issues* - In synchronous system, the system clock, and thus the system performance, is decided by the critical path. So, most portions of a circuit must be optimized to achieve the highest clock rate. Since no globally

timing issues in asynchronous circuits and the speed is dictated by the circuit path currently in operation, the optimization for speed performance of rarely used portions of the circuits can be ignored without adversely affecting whole system performance.

*Lower power consumption* - Most synchronous circuits need to toggle clock lines, and pre-charge and discharge signals, in portions of a circuit unused in the current computation. However, in asynchronous circuits, the transitions only in used areas involved the current computation consume the power.

Some other potential advantages detailed discussed in Scott Hauck article [2], like *better technology migration, robustness*, etc. High-speed asynchronous design is increasingly becoming an attractive alternative to full-custom synchronous design because of its freedom from clock distribution and clock skew problems, and some other advantages. However, we focus on the low power property in our design, a power-efficient stack. We call that *Fish-Bone* because of the likeness between the data movement diagram and the rest of a fish stake we left on our dinner table.

Because of lots of advantages, a lot of asynchronous designs have been researched. For examples, basic gates like Muller-C [3],etc., Small components like asynchronous adders [4] and multipliers [5],etc., architecture designs like counterflow architecture [6],etc., whole system designs like Philips 8051 [7], Amulet processor family [8], TITAC [9, 10], etc..

## 1.2 The Core of Stack Machines

Although virtually every processor today uses a load-store register architecture, stack architectures attract attention again due to the success of Java. The intermediate language of Java, the Java bytecodes, is stack based and therefore a , is also stack based. Faster stack hardware can archive high performance during

executing stack operations. More than that, the power consumption efficiency is needed to be put more attention. The core of a whole stack machine is a hardware stack. And high percentage of power consumption caused from the core [11]. How about a fast and power-efficient stack instead of the core?

### 1.3 Obvious Implementations and Prior Works

An obvious implementation is a RAM with top-of-stack pointer. The trivial solution causes the long cycle time and high energy consumption because of the large fan-in loads that must be driven for each put and get action, although the high density of RAMs consumes little area per data item. Furthermore, the cycle time and energy consumption grows with the size of the RAM.

The second trivial solution is implemented with a linear array of cells, where each cell can store data items. The linear of cells may offer a shorter cycle time because cells communicate only with their neighbors. More, the first cell of the array always contains the data item on top of the stack. The contact windows between this stack and the environment is only the top cell, so stack commands have only small loads to drive. However, a potential disadvantage of such an implementation is that the average power consumption of stack command sequence, put actions and get actions, can still be quite high, because the involved cells is the whole chain of the linear array. To complete a put or get command causes all items in the array to move.

Both of the above two implementations can be designed easily with synchronous design style. The second one can be implemented with synchronous design for parallel data movements in a linear array, or be implemented with asynchronous design but more latch cost or more complex control circuits are needed. Using master-slave latch architecture for cells of the linear array is a solution to remove the need of the distributed clock signals, but the disadvantage of power consumption is

still quite high [12].

Jo Ebergen has proposed another high speed and energy performance stack [1]. It is an asynchronous stack design based on GasP asynchronous control circuit family. The low power and high speed properties hold a special attraction to us. The energy consumption per stack operation only depends on the sequence of stack operations, and the cycle time is independent of the number of data items in the stack and the data width. The high performance stack consists of a tree stack and some three-place linear stacks. A GasP network compatible design was developed, and the protection of underflow and overflow was also implemented.

Other related works on stacks includes Alain Martin's lazy stack [13] and stack design by Mark Josephs et al. [14] However, these designs have a longer cycle time and the cycle time depends on the size of the stack.

## 1.4 More Power-efficient Stacks

Let us concern the trend from the linear array type stacks to n-place linear stacks. The main idea is to reduce the data movements in stacks. The architecture of linear array of cells stack naturally causes all cells operate, even those cells are not needed to be involved. That is, if a stack has n storages, it means that a command, put or get, causes n internal operations. In Jo Ebergen's article, "a fast and energy-efficient stack", the number of internal data moves is improved to 1 to 5 in a 42-place stack example. The average number of internal data moves is about 3.67.

We would focus the attention on the power consumption. How to save more power? In this thesis, a more power-efficient stack with underflow and overflow protections was implemented. Results from HSPICE simulations show that it gained averagely 17.06% in power consumption compared to Jo Ebergen's one. Moreover, its consumed average power is just averagely one over thirteen to asynchronous linear

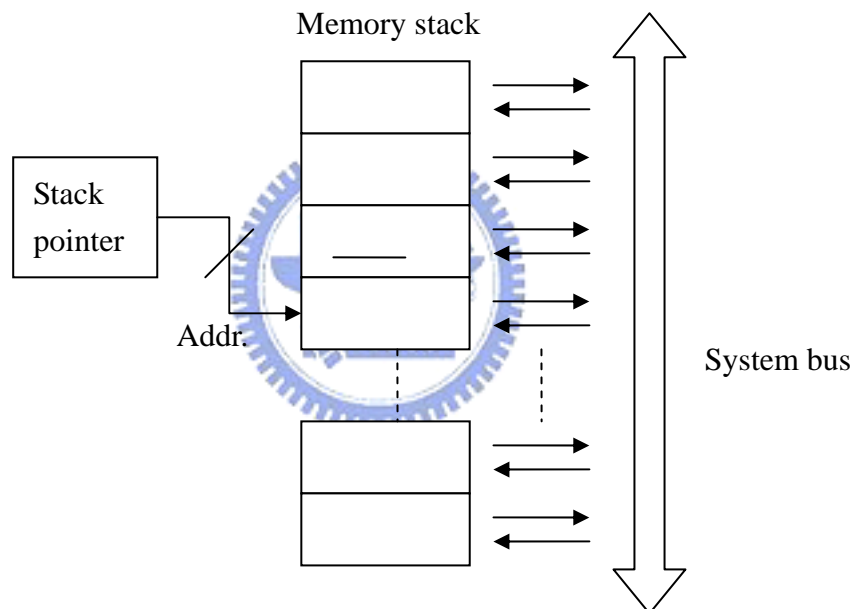
stacks. We will begin with considering how many data movements can be reduced, how to reduce the data movements, and how to save more power.



# Chapter 2 Related Works

In this chapter, we will give some backgrounds and go into the detail about some related works. In the first two sections, intuitional implementations mentioned in the end of the previous chapter are described. Next, a hybrid stack is discussed and finally we give the life to the Fish-Bone stack.

## 2.1 Stack with RAM Implementation



**Figure 2-1: a stack with RAM implementation. The address bus has quite loads in memory stack to drive, and the data communication between memory stack and system bus needs longer time because data outside needs to be passed to all memory stack cells.**

Figure 2-1 shows a portion of a memory organized as a stack, and the details are discussed in [15]. The diagram shows the simple RAM implementation of stack. Put and get commands affect stack pointer, and the result of a decoder fan out to each place of the memory stack corresponding to make one of the places in memory stack

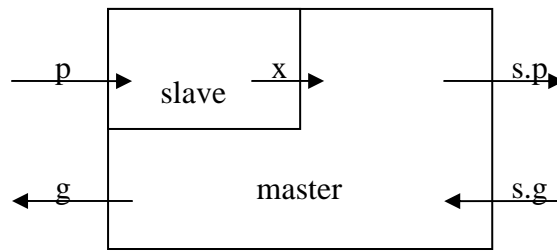


work. Because of the large loads that must be driven for each put and get action, the cycle time and energy consumption can be high. Data movement between memory stack and system bus costs also much time and energy because the same reason, large fan out. Furthermore, the cycle time and energy consumption grows with the size of the memory stack in RAMs. However, the implementation has an advantage on area cost because of the high densities of RAMs, such an implementation consumes very little area per data item. In next section, a solution to the problem, long cycle time and high energy consumption caused by large fan-out will be described.

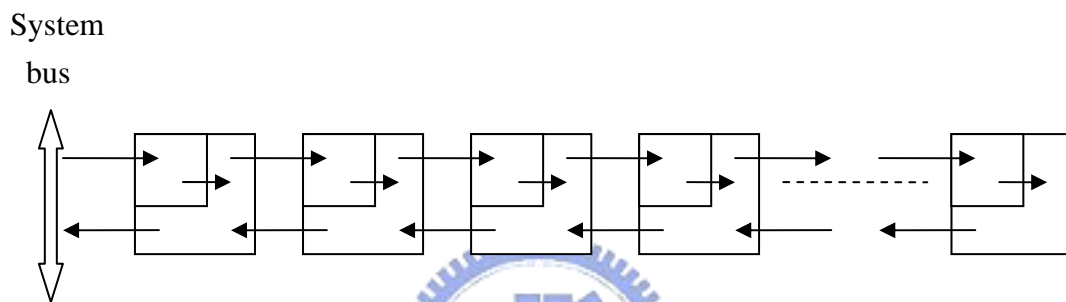
## 2.2 Stack with Linear Array of Cells

A problem caused by large fan-out and large fan-in can be solved with this implementation, stack with linear array of cells. There is only one window that contacts to the environment. This is a design that Jo Ebergen has proposed [12]. Figure 2-2(a) illustrates all data movements of successful puts and gets of a cell. Figure 2-2(b) shows a simple stack which consists of a linear array of cells. In Figure 2-2(a), the master-slave storage architecture was shown. Each cell has one slave storage location and one master storage location. Each put action  $p$  moves the incoming data item from the predecessor (the predecessor of the first cell is the environment) into the slave location. Then, the internal action causes action  $s.p$  puts the data item residing in the master location into the next cell, or called sub-stack. Finally, action  $x$  moves the data item form the slave location into the master location. The cell's behavior then repeats until the end of the linear array of cells. When get command acts, the slave location doesn't need to work. Every get action  $g$  moves the data item residing in the master location to the predecessor, and subsequently action  $s.g$  get a data item from the sub-stack. Like the procedure of put action, the cell's

behavior then repeats until the end of the linear array of cells.



(a) Cell with data movements



(b) Asynchronous linear array of stack

**Figure 2-2 : A simple linear stack: (a) A cell and its related data movements (b) A simple stack with linear array of cells**

Such an implementation may have a short cycle time, because communications are local and involve only small loads. However, there are several shortcomings. First, each put cycle contains three internal actions,  $p$ ,  $s.p$ , and  $x$ , which takes longer time and costs more energy than two internal actions,  $g$  and  $s.g$ , in a get cycle. Second, each put and get command on the cell propagates to the sub-stack. Consequently, every put action results in pushing every data item in each cell deeper into the stack, and every get action results in pulling every data item in each cell further out of the stack. That is the power consumption per put action or get action is proportional to the number of items in the stack. Furthermore, each cell contains two storage locations

but store only one data item in this implementation. The shortcoming brings in the increasing area demands. Finally, the simple implementation doesn't have underflow and overflow protections of the stack: a put action on a full stack loses the data item at the bottom of the stack, and a get on an empty stack will yield an unknown value.

The design idea we get here is the constant time response and local communication for low power, although there are still shortcomings needed to be solved.

### **2.3 A Hybrid Stack Consists of Tree Stacks and Three-level Three-place Linear Stacks**

Hybrid stack is an asynchronous stack was proposed by Jo Ebergen [1]. The implementation consists of some linear arrays of cells, but each one has  $n$  storage locations and each storage location can hold a data item. More precisely, the stack consists of two kind of  $n$ -place linear stack, one is 2-place linear stack, or called *tree stack* and the other is *three-level three-place linear stack*. Some ideas are useful to improve the performance and solve the problems that were resulted in by the simple implementation of linear array of cells, or call it 1- place linear stack. First, put and get commands on the stack and sub-stack rotate through the storage locations of the cell in a round-robin fashion. Furthermore, the cell performs actions on the sub-stack only when necessary. That is, only when the cell become full, the cell performs a put action on the sub-stack, and only when the cell becomes empty, the cell performs a get action on the sub-stack. And through this idea, lots of unnecessary data item movements are removed. Second, this is a stack with overflow and underflow protections.

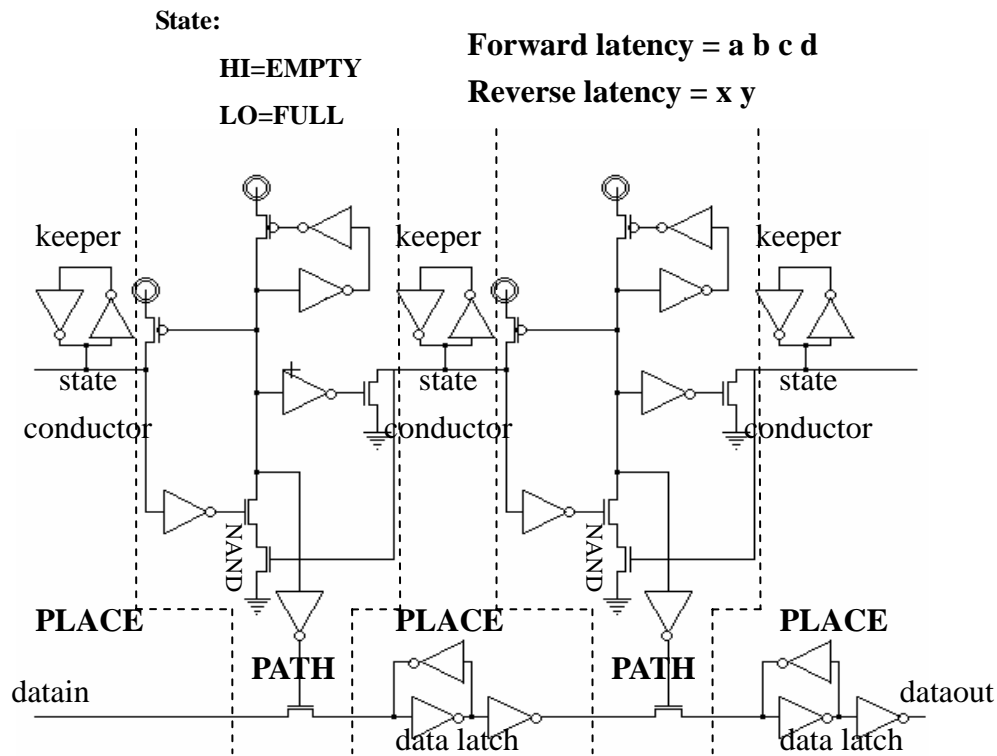
### 2.3.1 A Design Based on GasP Modules

GasP is an asynchronous control circuit family. Some years ago Molnar [16] articulated the basic control requirement for an asynchronous pipeline. Molnar's "asP\*" control system used a flip-flop in each PLACE to record its state and a NAND gate in each PATH to detect the conditions prerequisite to action. The words "PLACE" and "PATH" is used to distinguish two kinds of circuits: a PLACE holds data whereas a PATH controls the flow of data between PLACES. Molnar's asP\* circuit was symmetric in form, and so its forward latency and reverse latency were the same. The last three letters in the name GasP acknowledges its asP\* ancestry. Figure 2-3 describes a simple example of GasP network that forms a FIFO queue and their actions [17]. Each PATH circuits controlling the flow of data between stages must act only when both its predecessor PLACE is valid and its successor PLACE is also valid. In this example, the predecessor PLACE is valid when FULL, and the successor PLACE is valid when EMPTY.

As Molnar pointed out, a PATH must accomplish things when it fires:

- (1) It must make data latches momentarily transparent.
- (2) It must declare its predecessor stage EMPTY.
- (3) It must declare its successor stage FULL.
- (4) To reset the output of the series N-type transistors to the inactive.

GasP circuits store each state on a single wire that is called state conductor. In this GasP pipeline, each PLACE has a state conductor to indicate whether it is FULL or EMPTY. It is simplest to understand GasP circuits using the state encoding HI = EMPTY, LO = FULL for all state conductors.



**Figure 2-3 : GasP with self-resetting NAND**

In Figure 2-3, a complete action performs like below:

- (1) when the keepers of the successor and the predecessor of a PATH is correspondingly stated EMPTY and FULL.
- (2) The latch drive signal from inverter [gc] is a short positive pulse suitable for making the N-type transistor pass the gates at the bottom of the figure momentarily transparent to copy data forward.
- (3) Inverter [c] and N-type transistor [d] drive the drive the successor state conductor LO, meaning FULL.
- (4) P-type transistor [y] drives the predecessor state conductor HI, meaning EMPTY.
- (5) Delaying inverter at the top of the figure and a P-type transistor reset the NAND function after a short controllable delay.

We used transistor sizing for the GasP modules for the correct functionality of the

design. the theory behind sizing transistors is based on Logical Effort and explained in [18]. This theory permits us to calculate quickly the transistor sized of each gate for given gate delays more easily. However, the theory is researched for synchronous circuits. So the theory is used for giving the initial guess value more precisely. Then we must use simulator to check the correctness of the circuits.

### 2.3.2 Tree Stacks

The hybrid stack in [1] consists of a tree stack and some three-level three-place stacks. And a tree stack consists of several  $n$ -place cells, like the nodes in a linear stack. The number,  $n$ , of places in a cell is at least 2. Every cell has  $n$  sub-stack, one for each place. To keep the tree stack simple and to obtain a short cycle time, the original design was chosen  $n = 2$ . A node performing a tree stack appears in Figure 2-4 along with two sub-stacks.

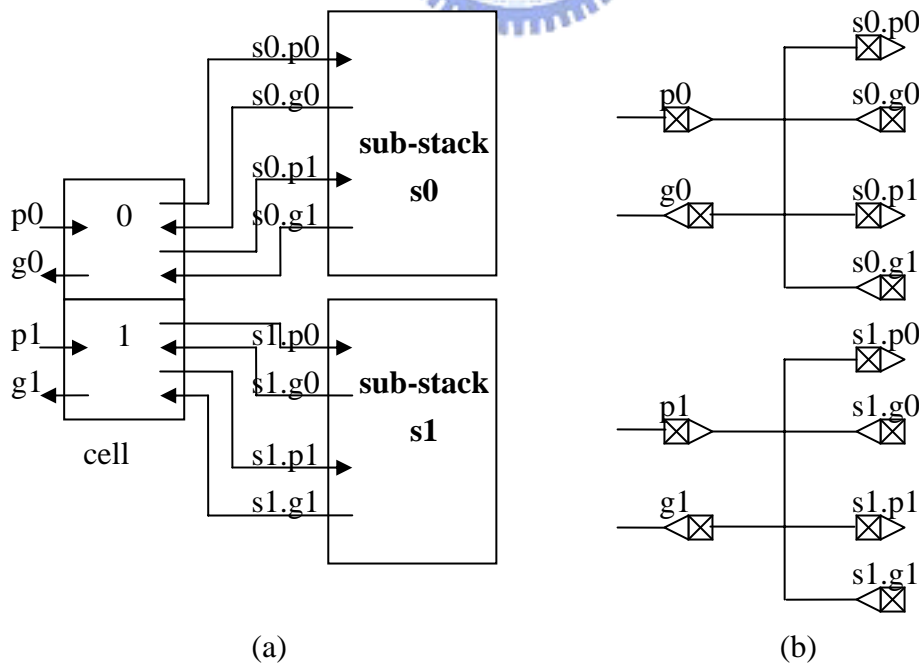
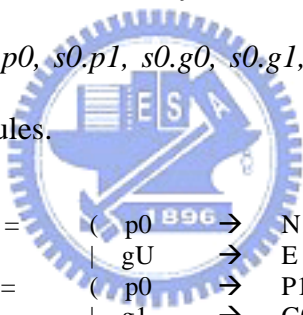


Figure 2-4 : A two-place tree stack: (a) Cell and sub-stack with data movements (b) Data path

The behavior of each tree cell with respect to put and get command is similar to that of the linear stack cell. When using a two-place cell, like the example in Figure 2-4, the puts and gets rotate through the place 0 and 1. After the cell gets a put command and puts a data item in place  $i$ , the cell puts the data item residing in place  $i+1$ , if any, into sub-stack  $i+1$ . Similarly, after the cell gets a get command and gets a data item from place  $i$  and place  $i-1$  is empty, the cell gets a data item for place  $i-1$  from the sub-stack  $i-1$ , where additions are modulo 2 because of the rotating use of the cell.

The tree stack is easily implemented with GasP modules converted from FSM. Figure 2-5 shows the FSM specification and all possible sequences of puts and gets. In the case,  $\{E, F, N0, N1, P0, P1, G0, G1\}$  are states kept in keepers formed by a latch, and  $\{p0, p1, g0, g1, s0.p0, s0.p1, s0.g0, s0.g1, s1.p0, s1.p1, s1.g0, s1.g1\}$  are event controlled by GasP modules.



```

E = ( p0 → N1
    | gU → E )
N0 = ( p0 → P1
    | g1 → G0 )
N1 = ( p1 → P0
    | g0 → G1 )
F = ( pU → F
    | g1 → N1 )
P0 = ( s0.p0 → N0
    | s0.p1 → N0
    | s0.pU → F )
P1 = ( s1.p0 → N1
    | s1.p1 → N1 )
G0 = ( s0.g0 → N1
    | s0.g1 → N1 )
G1 = ( s1.g0 → N0
    | s1.g1 → N0
    | s1.gU → E )

```

Figure 2-5 : A two-place tree stack: Finite state machine specification.

### 2.3.3 Three-level Three-place Linear Stacks

Different type of stacks that compose a hybrid stack is three-level three-place

linear stacks. The three-level three-place stacks play the roles of the main storage of a hybrid stack, so the three-level three-place stacks are placed at the leaf of the tree. There are two directions to extend the storage, hierarchically extension (place) and vertically extension (level). However, Jo Ebergen chose three-level and three-place for his design for ease and because of some analysis. There is some analysis in his article. Figure 2-6 shows a three-level three-place stack. Three one-level three-place linear stacks serially compose a three-level one. The behavior is like tree stacks. Puts and gets on the stack and sub-stack rotate through the storage locations of the cell in a round-robin fashion. And each cell performs actions only when necessary. Initially, the state empty,  $E$ , is specially set as high for waiting the coming put or get command. Like the first state description in figure 2-7, two possible commands may come. Put command comes to put a data item in storage location 0 and set state  $N1$ , or get command comes as the  $gU$  action in FSM specification and keep the state  $E$  high. Then, in state  $Ni$  when the environment puts an item in location  $i$  of the cell, the cell checks that the neighbor location  $i+1$  is empty for a potential next put action, where the additions is modulo 3. On the one hand if storage location  $i+1$  is full, then the cell puts the item residing in the storage location  $i+1$  into its corresponding storage location of sub-stack  $i+1$  and enter state  $N(i+1)$ , and on the other if the storage location  $i+1$  is empty, then the cell directly enters state  $N(i+1)$ . The get commands follow the similar rules. In state  $F$ , the stack cell can execute a get action  $g2$  on storage location 2 or an unsuccessful put action  $pU$ . Then, in state  $Ni$ , when the environment gets a data item form storage location  $i-1$ , the cell checks that the neighbor location  $i-2$  is full for a potential next get action form the environment, where the minus is modulo 3. On the one hand if storage location  $i-2$  is empty, then the cell gets an item from the sub-stack for storage location  $i-2$  and enters state  $N(i-1)$ ,



and on the other if storage location  $i-2$  is full, then the cell enters state  $N(i-1)$ .

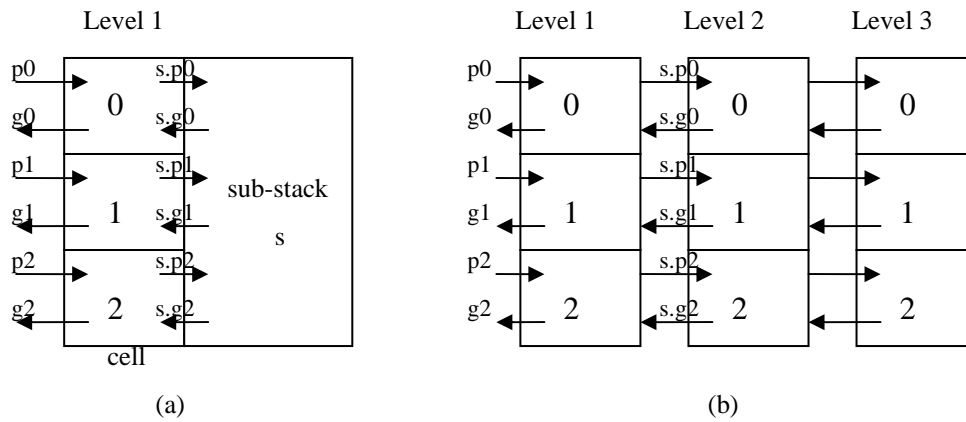


Figure 2-6 : A three-level three-place linear stack: (a) cell and sub-stack with data movements (b)

### 3-level 3-place stack and data movements

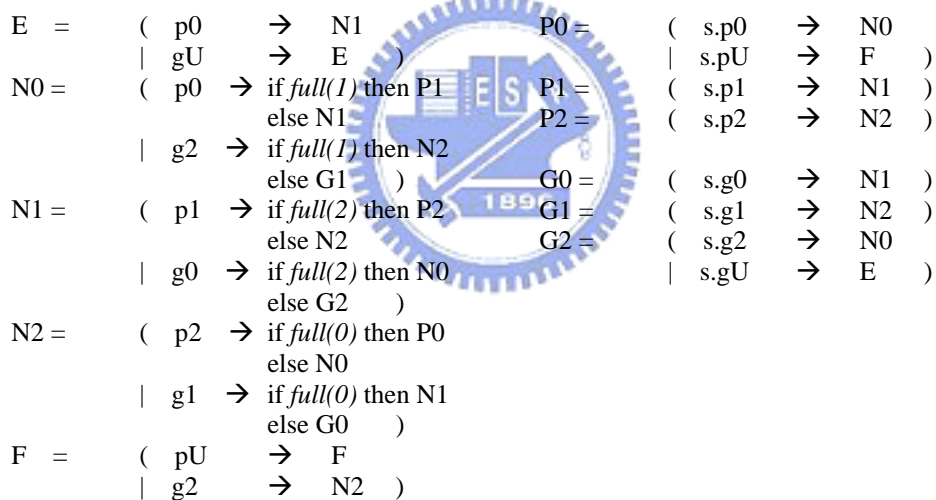


Figure 2-7 : A three-place linear stack cell: Finite state machine specification.

## 2.3.4 The Conversion from FSM to GasP Modules

Figure 2-8 gives simple examples of the conversion from FSM to GasP modules. A GasP module can be identically designed for some unique functions. It is based on a basic structure, a NAND gate like with resetting output ability. Besides, every optional addition, like (1)(2)(3)(4) in the figure, can be added to basic GasP modules

when they are needed to commit some functions. Initially, the GasP event is set to high, the left input is set to low, the right input is set to high for waiting the command coming from the left input. The controlled components by GasP modules are low active in normal. Besides, the condition maintainers are not included in the GasP design. Precisely, there is an extra combinational circuit that maintains the condition signals according to some related GasP events. There are sometimes not only one condition needed in a GasP module, we can use basic logic gate, AND gates or OR gates, to merge those signals.

Figure 2-8(b) shows the added wires and components needed for non-resetting input port and resetting input port. Figure 2-8(c) shows the added components needed for non-resetting input port, resetting input port and next state port. Figure 2-8(d) shows the added components needed for two side resetting input ports and two next state ports with a condition. As mentioned before, the w/l ratio of transistors in GasP modules are needed to be concerned for the correctness of functionalities. Too many added components will lead to slow response of the output of the GasP module. The loads of a transistor MOS is about 4 or less times to its fan-in in suggestion. There are a lot of different GasP modules according to different added components. Sometimes we need to reverse the active type, high to low or low to high, to commit our design requirements, so some components of N-type transistors will be replaced with P-type transistors and some components of P-type transistors will be replaced with N-type transistors.

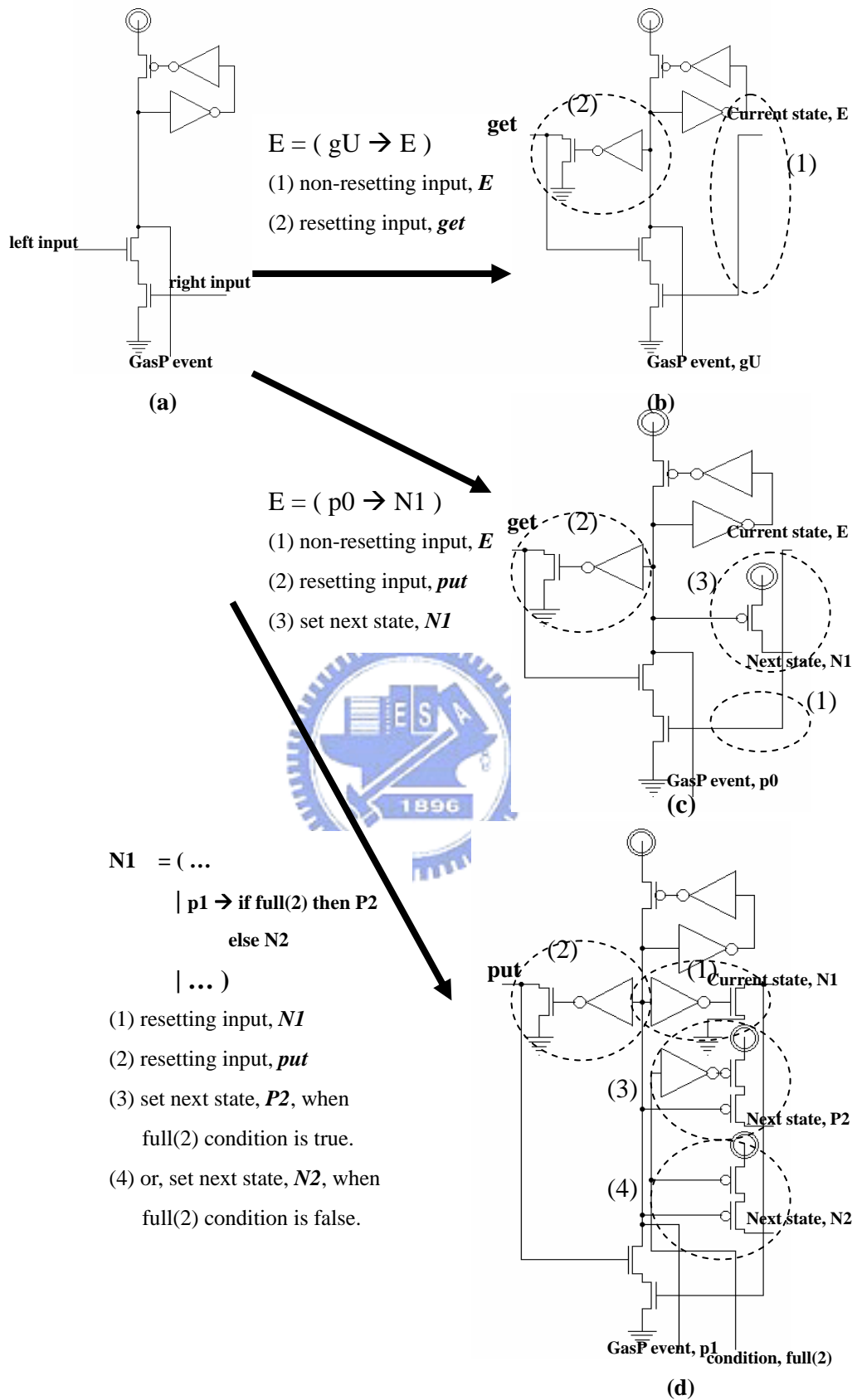


Figure 2-8 : the converting from FSM to GasP module: (a) The basic structure of a GasP module

- (b) Use N-type transistor to reset input, and do nothing for non-resetting input (c) More than case (b), use a P-type transistor to set next state. (d) A GasP module with a condition signal.

### 2.3.5 A Hybrid Stack

For the benefit from the advantages offered by two types of stack, three-level three-place linear stacks and tree stacks, a hybrid stack had been constructed as shown in Figure 2-9.

Some advantages are offered from tree stacks:

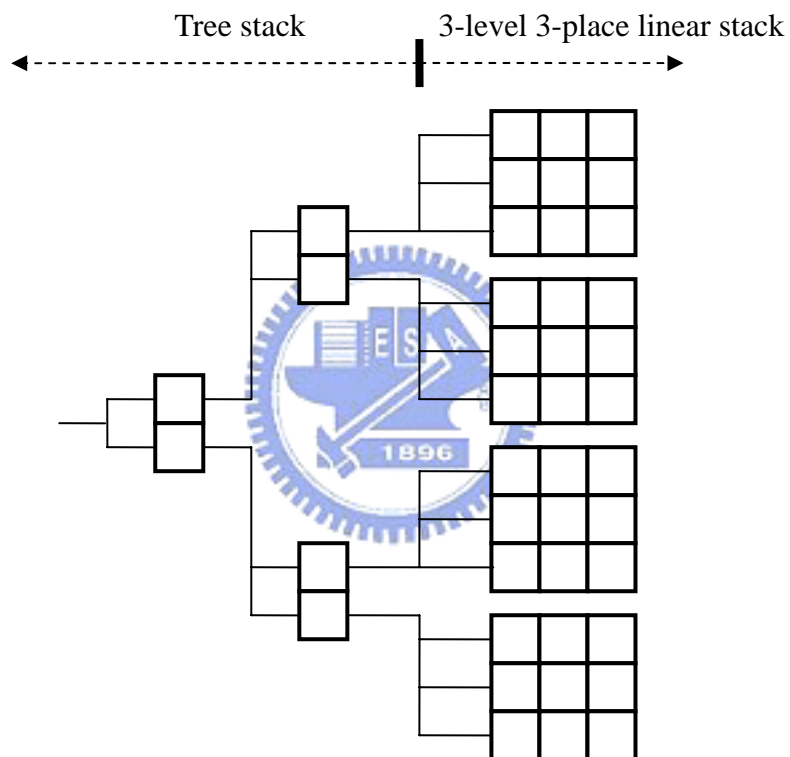
- (1) ***Less data movements per cycle:*** Each put or get action cycle of one cell contains at most two data moves. Here, the cell means a node of a tree stack or a level of a 3-level 3-place linear stack.
- (2) ***Annihilation of propagating puts and gets:*** Each put or get action can only involve one path of the tree to be pushed down into or pulled up from the stack by a cell. Consequently, the total number of data moves in the tree stack is logarithmic in the number of items in the stack.
- (3) ***No waste in storage locations:*** Each cell of tree stacks has two places and each place can hold a data item.
- (4) ***With underflow and overflow protections***

Some advantages are offered from 3-level 3-place stacks:

- (1) ***Less data movements per cycle:*** it is the same as tree stack node described as the mention above.
- (2) ***Annihilation of propagating puts and gets:*** A put action propagates down sub-stack only when the cell has 2 items. Similarly, a get action propagates down the sub-stack only when the cell has 1 item.
- (3) ***No waste in storage locations:*** it is also the same as tree stack node described as the mention above.

(4) *With underflow and overflow protections*

In the example of Figure 2-9, the hybrid stack is formed as a tree, and there are 42 storage places. The range of the number of internal data moves is from 1 to 5. for example, the number, 1, occurs when executing a put command with empty stack and when executing a get command with full stack. In the worst case, there are five internal data moves because of three level structure of the stack and two more data moves may be needed in the leaf stack.

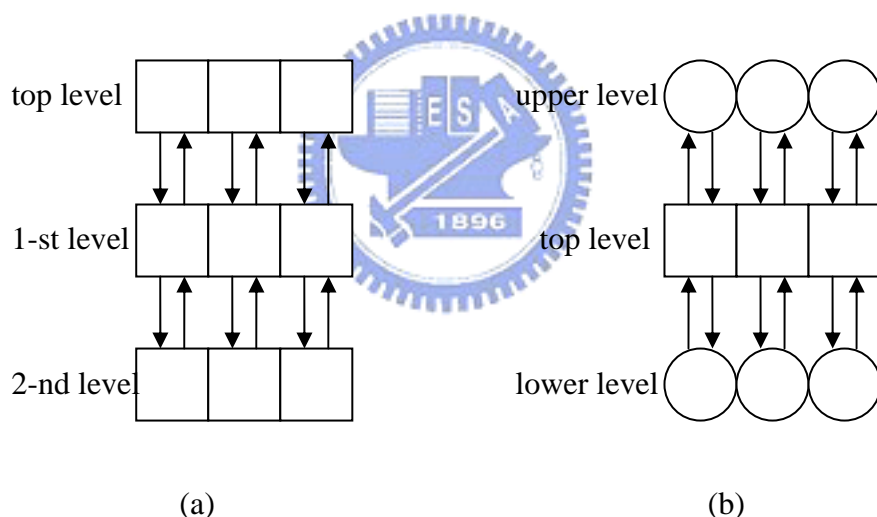


**Figure 2-9 : a hybrid stack with 42 storage places consists of a tree stack and 3-level 3-place linear stacks.**

However, we can not use one of two type stack only to compose a suitable size stack. If we choose tree stacks as the components, the too deep depth will lead to more internal data moves in executing a put or a get command than that in hybrid stack type. On the other, if we choose only 3-level 3-place linear stack as the components, the extension on storage locations will be a problem.

## 2.4 The First Breath of Fish-Bone Stacks

Almost every thing is complete in hybrid stack design, which is a design with basic underflow and overflow protections, efficiently using each storage locations, and reduced data moves of a cell in a command cycle. However, the energy consumption seem not so perfect. We focus on the power consumption of the 3-level 3-place linear stack, and ignore the tree stack which is designed for enlarging storage locations. We find that the number of data moves is the key point. Less data item moves leads to less power consumption. The maximal number of data item moves in the design of 3-level 3-place linear stack node is three because there are three levels to propagate. How about less data moves in a new design?



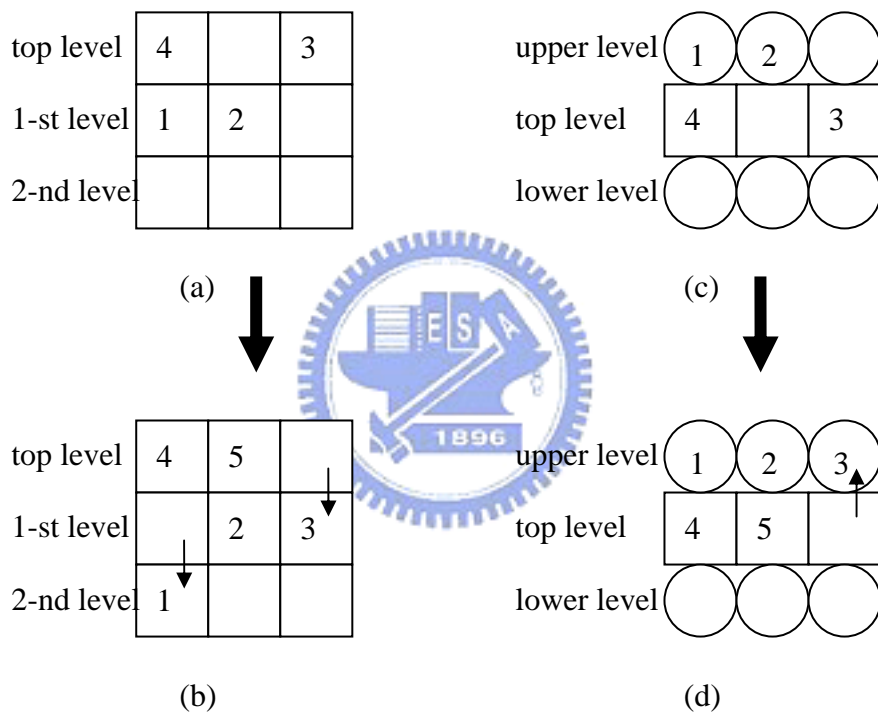
**Figure 2-10 : the structures of two type of stack: (a) Three-level three-place linear stack (b) Fish-Bone stack**

A solution was proposed in this thesis. The key difference between the three-level three-place linear stack and Fish-Bone stack is the design of the remainders besides their top level as shown in Figure 2-10. Each storage location of the top level has more than one storage location to push into or pull from in Fish-Bone stack, upper level and lower level, but there is only one storage location for

three-level three-place linear stack.

An advantage offered from the new design is the less internal data item moves during executing a stack command. And, Figure 2-11 shows an easy case that a command needs less data moves in a leaf node of a hybrid stack. For easily explaining the difference, we only take the leaf node, three-level three-place linear stack, to compare with our design, Fish-Bone stack. A case with continuous five put commands applied to an empty stack and the process of applying the fifth put command. Figure 2-11(a) shows a three-level three-place linear stack, and the state after finishing the fourth put command of continuous five put commands. The number in the grids means the order of data item that was put into, and the empty grids means no occupancy of those storage locations. Figure 2-11(b) shows the state after the fifth put command. The item 5 was put at the second storage location of the top level, storage location 1 (the naming of the storage locations is form 0 to 2), and resulting the top level to full state. Then, the item 3 was pushed to next level for the next potential put command. For the same reason, the item 1 in the second level was pushed to the third level for the next potential put command on the second level three-place linear stack. There are totally three data item moves to finish the fifth put command in three-level three-place linear stack. There are only two data item moves to finish the same commands in our design, Fish-Bone stack. Figure 2-11(c) shows our Fish-Bone stack, and the state after finishing the fourth put command of continuous five put commands. Figure 2-11(b) shows the state after the fifth put command. The item 5 was put at the second storage location of the top level, storage location 1, and resulting the top level to full state. Then, the item 3 was pushed to the upper level for the next potential put command. For the same reason, the item 1 in the upper level was pushed to the outside Fish-Bone stack for the next potential put command on the upper level by nature.

However, we use Fish-Bone stack as the leaf node for comparing with three-level three-place stack, so we set some parameters of Fish-Bone stack to high for disable the propagation to the outside of the Fish-Bone stack. Consequently, the item 1 in the upper level of Fish-Bone stack is not pushed outside and does nothing. We save the data item move of the item 1, so there are totally two data item moves to finish the fifth put command in Fish-Bone stack.



**Figure 2-11 : The difference of internal data moves between 3-level 3-place linear stack and Fish-Bone stack: (a)(b) The process of the fifth put command on 3-level 3-place linear stack (c)(d) The process of the fifth put command on Fish-Bone stack**



# Chapter 3 Fish-Bone Stack

## 3.1 Motivations and Ideas

As mentioned in previous sectiono, there is a novel stack that can consume lower power and can offer almost the same speed performance. We call it Fish-Bone stack because of the likeness of the architecture of Fish-Bone stack between real Fish-Bones.

### 3.1.1 Benefits form Reducing Data Item Moves

A key point of this thesis is to propose a novel architecture that can use less data item moves to complete every put and get command and consequently provide more power-efficient performance. The less data item moves results in less power consumption because of less charging and discharging process in transistors. We had implemented a one-bit stack and compare it to other designs. The effect of the power saving by reducing the internal data item moves will be enlarged in normal use case, like an 8-bits data width stack, a 16-bit data width stack, even a 32-bit data width stack.

### 3.1.2 A Solution with Reducing Data Item Moves

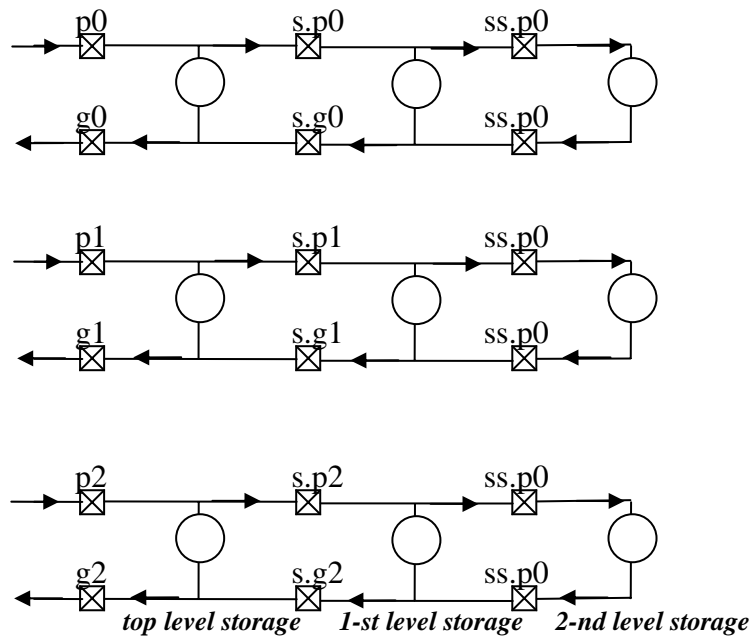
As mentioned above, we understand that the relation between data item moves and the power consumption of a stack. In this section, the architecture of Fish-Bone was shown to explain how to reduce the data item moves against almost perfect design, three-level three-place linear stacks.

Figure 3-1 and Figure 3-2 shows two architectures for a simple comparison.

Figure 3-1 shows the data-path of three-level three-place linear stacks. It is architecture with three levels of three-place linear stacks. The arrows mean the flow of data, the circles mean data latches used for really store data items, and the squares with a cross inside mean gates that control the pass of the data. The naming rules of transmission gates which are controlled by corresponding GasP modules are trivial, and are described below:

- (1)  $\{p0, p1, p2\}$  are transmission gates that control the pass of data form the environment to the top level storage, and the number, 0, 1, and 2, is an index for the order of storage location.
- (2)  $\{g0, g1, g2\}$  are transmission gates that control the pass of data form the top level storage to the environment, and the number, 0, 1, and 2, is an index for the order of storage location.
- (3) The words “s.” and “ss.” are used just to distinguish the different transmission gates those are in different level. Concatenating “s.” to  $\{p0, p1, p2, g0, g1, g2\}$  means the transmission gates that control the pass of data between the top level three-place linear stack and 1-st level three-place linear stack, and concatenating “ss.” means those that control the pass of data between the 1-st level three-place linear stack an the 2-nd level three-place linear stack.

When an empty stack gets a put command, the GasP,  $p0$ , will be active, then the corresponding transmission gate also marked  $p0$  in Figure 3-1 will be open to make the data outside flow into the stack. And the data item is stored at the first storage location, 0, in top-level three-place linear stack. The next put command will be put at the second storage location in accordance with the specification of the FSM of the three-place linear stack. Other more actions are mentioned in section 2.3.3.



**Figure 3-1 : The data-path of a three-level three-place linear stack**

Figure 3-2 shows the data-path of a Fish-Bone stack. The key difference between a three-level three-place linear stack shown in Figure 3-1 is the depth from the top-level. As same as the descriptions in Figure 3-1, the naming rules in Fish-Bone stacks are almost same. There are some paralleled transmission-gates near the spinal canal marked with “c” in the end are control the pass of data between the environment and the top-level of a Fish-Bone stack, like  $\{up0c, dp0c\}$ ,  $\{up1c, dp1c\}$ ,  $\{up2c, dp2c\}$  for put actions on the top level storage locations, and  $\{ug0c, dg0c\}$ ,  $\{ug1c, dg1c\}$ ,  $\{ug2c, dg2c\}$  for get actions on the top level storage locations. The paralleled transmission gates not work at one time. There is only one transmission gate of them active and the other is not. And, others are marked “u” or “d” in the end are used for distinguish the transmission gates that control the pass of data between the top level and the upper level or the lower level. Besides, the head character, “u” or “d”, of the paralleled transmission gates on the spinal canal are used to distinguish the

transmission gates that are pass the data between the top level and the upper level or the lower level in two conditions.

The depth is three maximally from the top level in a three-level three-place linear stack, but the depth is only two maximally from the top level in a Fish-Bone stack. Although there is just one improvement, this is a large improvement in percentage view, 33%.

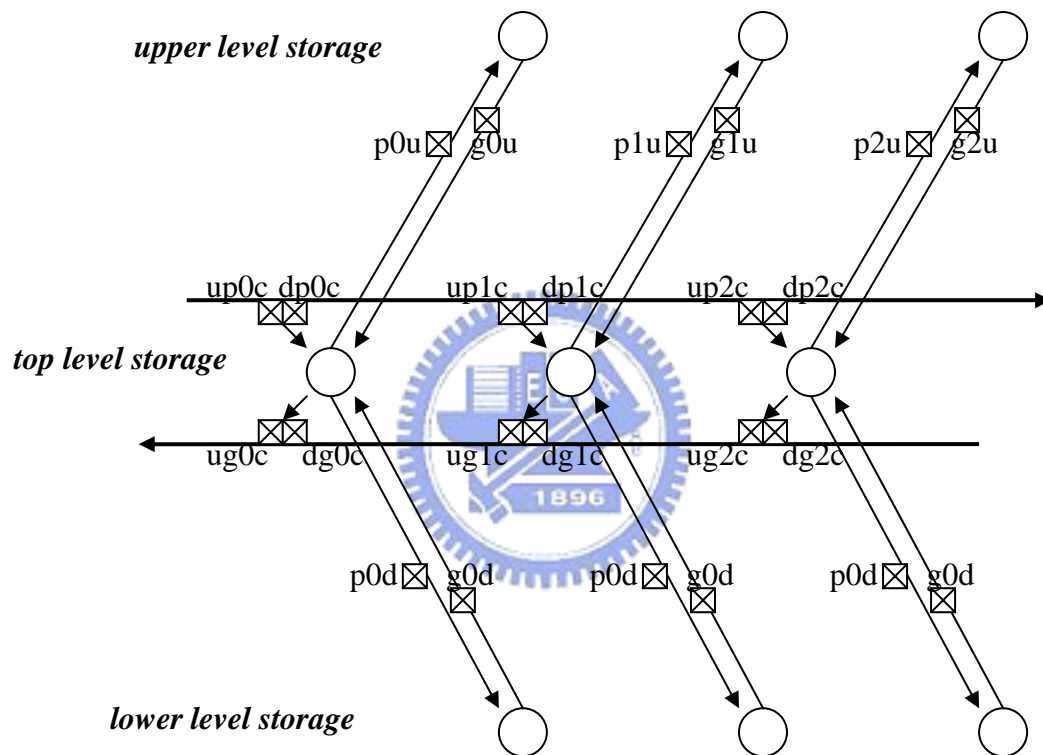


Figure 3-2 : the data-path of a Fish-Bone stack

Here is simple inference. Because of the same behaviors of the top levels in both designs, we assume the total internal data movement is  $T$  for a random command sequence. The internal data movement is the summary of  $T$  and data moves in the other levels. Consider the three-level three-place linear stack. For a random command sequence, the size can be infinite but countable. And, the commands passed from the top level of stack are also infinite and countable. We assume the number of data

moves caused by these passed command as  $\{s_1, s_2, s_3, \dots, s_n\}$  respectively. And we can conclude that each in  $\{s_1, s_2, s_3, \dots, s_n\}$  is less than or equal to 2 by the specification in previous section. Because of the same behavior of the top level in both designs, the passed command from the top level is also same. And for the same reason, we assume the number of data moves caused by those passed command in a Fish-Bone stack as  $\{f_1, f_2, f_3, \dots, f_n\}$  respectively. We can also conclude that each in  $\{f_1, f_2, f_3, \dots, f_n\}$  is less than 2 by the specification in the previous section. So, by summarizing the data movements of both designs, we can easily conclude that the data moves in a Fish-Bone stack is clearly less than in a three-level three-place stack.

## 3.2 Architecture of Fish-Bone Stacks

The architecture of a Fish-Bone stack is shown in Figure 3-2 previously. The spinal canal contains three data storage locations at the joints that connect six fishbone totally, and each fishbone contains a data storage location in the end. The spinal canal is the top level of a Fish-Bone stack; the top half of the fishbone is the upper level of a Fish-Bone stack; and the bottom half of the fishbone is the lower level of a Fish-Bone stack. The data communication is only between the different levels. And communications between different indexes of data storage locations are not allowed.

### 3.2.1 The Internal Actions in Fish-Bone Stacks

The principles of the Fish-Bone stack design is to prepare a space for the next potential put command and to prepare a data item for the next potential get command. The design rules result in shorter time to complete a request when a Fish-Bone stack gets one. Simple steps of internal actions are described below:

Put commands:

- (1) A data item was put at a storage location indexed “ $i$ ” according to the specification of the Fish-Bone stack.
- (2) One of two GasP modules,  $p(i+1)u$  (between the upper level of the stack and the top level of the stack) and  $p(i+1)d$  (between the lower level of the stack and the top level of the stack), will be active to empty out the storage location for the next potential put command on the top level. And, the data residing in the original storage location will be passed to the second level if it is valid, where the additions are modulo 3.
- (3) One of two GasP modules,  $s.p(i+2)u$  and  $s.p(i+2)d$ , will be active to empty out the storage location for the next put command on the second level, upper level or lower level. And, the data residing in the original storage location of the second level will be passed outside of the stack if it is valid, where the additions are modulo 3.

Get commands: the actions are just opposite to the rules of put commands.

- (1) A data item was get from a storage location indexed “ $i$ ” according to the specification of the Fish-Bone stack.
- (2) One of two GasP modules,  $g(i+1)u$  (between the upper level of the stack and the top level of the stack) and  $g(i+1)d$  (between the lower level of the stack and the top level of the stack), will be active to pull out a data item from the storage location of the second level for the next potential internal get command on the top level if it is valid, where the additions are modulo 3.
- (3) One of two GasP modules,  $s.p(i+2)u$  and  $s.p(i+2)d$ , will be active to pull out a data item form the storage location of the second level for the next

potential internal get command on the second level, upper level or lower level, if it is valid, where the additions are modulo 3.

### 3.2.2 An Example of Executing A Small Command Sequence

In this section, a small command sequence applied to a Fish-Bone stack was shown in Figure 3-3 and Figure 3-4. The command sequence consists of ten continuous put commands with concatenating ten continuous get commands was applied to a Fish-Bone stack. The process of put commands is shown in Figure 3-3, and the process of get commands is shown in Figure 3-4.

In Figure 3-3(a), data item indexed 1 was put in the storage location indexed 0 of the top level. There are no need to propagate the data residing in the storage location indexed 1 of the top level to one of the second level and no need to propagate the data residing in the one of the second level to outside of the stack because some mechanisms are applied to control the need of the propagation according to some condition flags. The number of data moves needed is only one. Data item indexed 2 was directly put in the storage location indexed 1 of the top level like the process of the put of data item indexed 1 in Figure 3-3(b). The number of this put command is also one. In Figure 3-3(c), data item indexed 3 was put in the storage location indexed 2 of the top level. Different from Figure3-3(a)(b), the data item residing in storage location indexed 0 of the top level must be moved to the second level for the next potential put command. It is not necessary to move the data residing the storage locations of the second level to outside of the stack because the stack is used as the leaf node here. Some parameters are set to make the stack consider that the outside stacks are full and there is no capacity for more data. The behavior of the forth and the fifth put command in the stack is almost the same as the third put command shown in

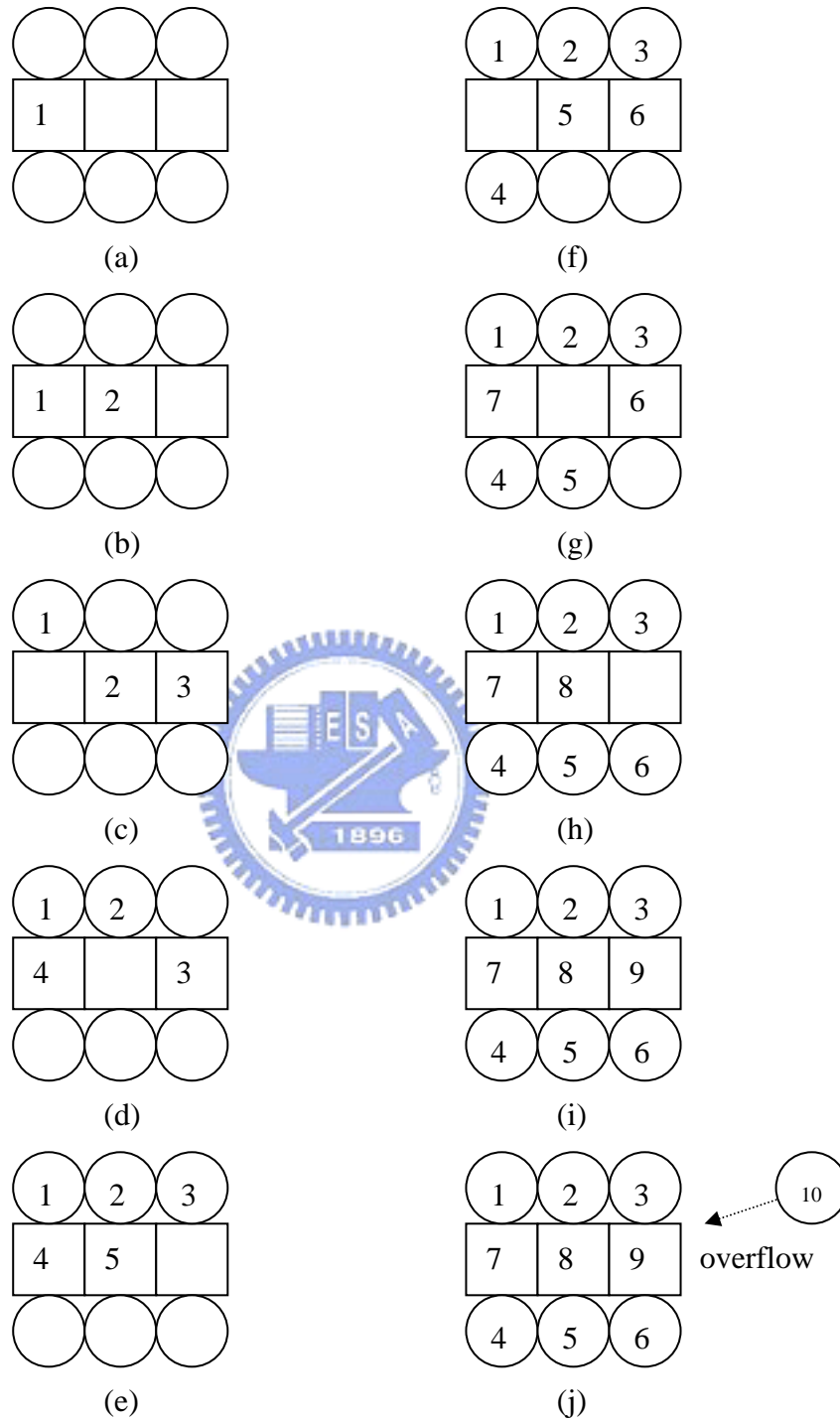
Figure 3-3(d)(e). The numbers of data movements involved by these two put commands are both two. Figure 3-3(f)(g)(h) shows the sixth, seventh, and eighth put commands. The behavior of them is similar to the previous three put commands. The difference is that the second level means the lower level not the upper level here. However, the numbers of internal data moves of them are still two. Figure 3-3(i) shows the ninth put command applied to the stack. The data item is directly put into the last space and no more internal actions are stimulated because some mechanisms block these GasPs to active. So, the number of data movements involved by this put command is one only. One more put command, the tenth put command, on the full stack causes the overflow signal response shown in Figure 3-3(j).

The processes for get commands are counter to the process of put commands. In Figure 3-4(a), data item indexed 9 was gotten from the storage location indexed 2 of the top level. There are no need to pull the data residing in the storage location indexed 1 of the second level to the top level and no need to pull the data outside into stack because some mechanisms are applied to control the need of the propagation according to condition flags. More details will be discussed in the next sections. The number of data movements involved is only one. Data item indexed 8 was directly gotten from the storage location indexed 1 of the top level like the process of the get of data item indexed 9 in Figure 3-4(b). The internal data move of this get command is also once. In Figure 3-4(c), data item indexed 7 was gotten from the storage location indexed 1 of the top level. Different from Figure3-4(a)(b), the data item residing in storage location indexed 2 in one of the second levels must be moved to the top level for the next potential get command. It is not necessary to move the data outside into the stack because the stack is used as the leaf node here. Some parameters are set to make the stack consider that the outside stacks are empty and there is no

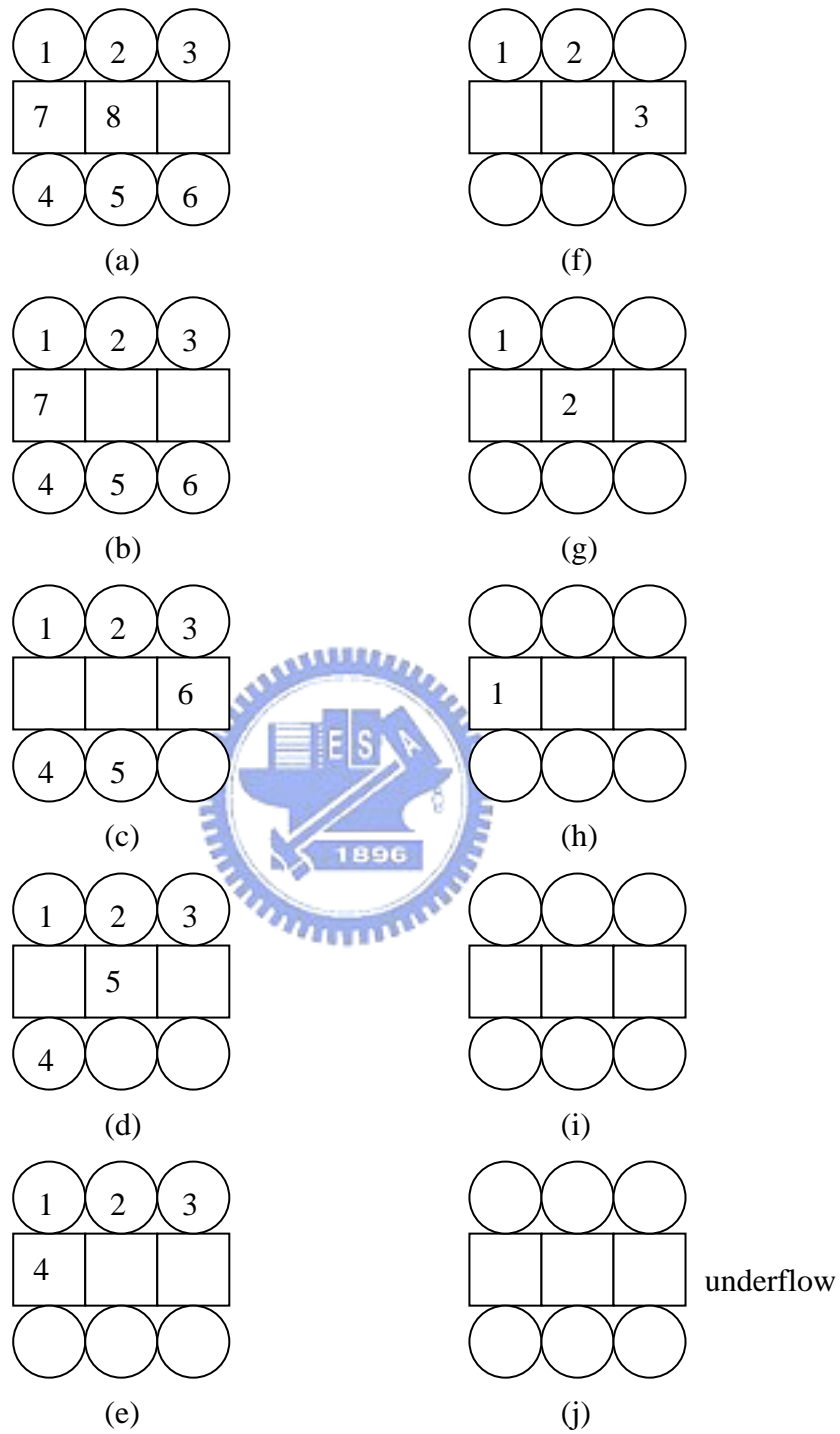


more data item outside. The behavior of the forth and the fifth put commands are almost the similar to the third get command shown in Figure 3-4(d)(e). The numbers of the internal data movement by these two get commands are both two. Figure 3-4(f)(g)(h) shows the sixth, seventh, and eighth get commands of the stack. The behaviors of them are similar to the previous three get commands. The difference is that the second level means the upper level not the lower level here. However, the numbers of internal data moves of them are still two. Figure 3-4(i) shows the ninth get command applied to the stack. The data item is directly gotten out of stack and no more internal actions are stimulated because some mechanisms block these GasPs to active. So, the number of data movements by this get command is one only. One more get command, the tenth put command, on the empty stack causes the underflow signal active response in Figure 3-4(j).





**Figure 3-3 : Continuous put commands on an empty Fish-Bone stack**



**Figure 3-4 : Continuous get commands on an empty Fish-Bone stack**

### 3.3 The Implementation of Control Path

A finite state machine specifying all sequences of moves for Fish-Bone stacks appears in Figure 3-5. All events represent moves between storage locations. A bar '|' in a specification separates the alternative sequences of events, and a comma ',' separates the paralleled sequences of events. The naming of the storage location is a number concatenating a character to distinguish nine storage locations. The number means the index of the level of stack, and the character decides which level, "c" for top level, "d" for lower level, and "u" for upper level. For example, condition  $full(2d)$  means the full signal that states the storage location indexed 2 in lower level of a Fish-Bone stack. Puts and gets on a Fish-Bone stack rotate through the storage locations of the top level and the storage locations of the two second levels in a round-robin fashion, like the stack pointers in a circular pointer implementation. Furthermore, each internal action is performed on the two level of a Fish-Bone stack only when necessary. More precisely, only when the top level of a Fish-Bone stack becomes full, an internal put action is performed on the second level of the Fish-Bone stack, and only when the top level becomes empty, an internal get action is performed on the second level of the Fish-Bone stack. Like the design in three-level three-place linear stacks, unsuccessful put and get actions, denoted by  $pU$  and  $gU$  respectively, in order to protect against overflow and underflow. An unsuccessful put action occurs when the environment wants to put a data item in to the stack, but the stack is full. The data item will be lost and the overflow signal will be set. An unsuccessful get action occurs when the environment wants to get a data item from the stack, but the stack is empty. The get action doesn't cause any move and the underflow signal will be set.

Initially all storage locations are empty, all output of GasP modules are set as

high to block their corresponding transmission gates, and all *states* that control the flow the control path are set as low excluding the state *E*. In state *E*, representing the empty stack with no data item in a Fish-Bone stack with nine storage locations, the stack can execute a put action *ep0c* on *0c* or an unsuccessful get action *gU*. In state *F*, representing the full stack, the stack can execute a get action *fg2c* on *2c* or an unsuccessful put action *pU*.

Put commands						Get commands					
N0	N1	N2	N3	N4	N5	N0	N1	N2	N3	N4	N5
<i>Outside of stack</i>						<i>Outside of stack</i>					
	sp0u	sp1u	sp2u						sg0u	sg1u	sg2u
<i>Upper level</i>						<i>Upper level</i>					
		p0u	p1u	p2u				g0u	g1u	g2u	
<i>Top level</i>						<i>Top level</i>					
up0c	up1c	up2c				ug0c	ug1c	ug2c			
<i>Environment</i>						<i>Environment</i>					
			dp0c	dp1c	dp2c	dg2c				dg0c	dg1c
<i>Top level</i>						<i>Top level</i>					
p1d	p2d				p0d	g1d	g2d				g0d
<i>Lower level</i>						<i>Lower level</i>					
sp2d				sp0d	sp1d	sg0d	sg1d	sg2d			
<i>Outside of stack</i>						<i>Outside of stack</i>					

**Table 3-1 : The events stimulated by relative GasP modules in round-robin fashion and the data flow controlled by the six states in Fish-Bone stacks**

```

E   = (  gU  → E
        /  ep0c → N1 )
N0  = (  sp2d  if full(2d)
        ,  p1d  if full(1c)
        ,  up0c → N1
        /  sg0d  if !full(0d)
        ,  g1d  if !full(1c)
        ,  dg2c → N5 )
N1  = {  sp0u  if full(0u) & !full(s0u),
        ,  p2d  if full(2c),
        ,  up1c → N2
        /  sg1d  if !full(1d) & !empty(s1d)
        ,  g2d  if !full(2c) & full(2d)
        ,  ug0c → if !full(2u) & !full(2d) then E
                  else N0 )
N2  = (  sp1u  if full(1u) & !full(s1u)
        ,  p0u  if full(0c) & !full(0u)
        ,  up2c → if full(0u) & full(0d) then F
                  else N3
        /  sg2d  if !full(2d) & !empty(s2d)
        ,  g0u  if !full(0c)
        ,  ug1c → N1 )
N3  = (  sp2u  if full(2u)
        ,  p1u  if full(1c)
        ,  dp0c → N4
        /  sg0u  if !full(0u)
        ,  g1u  if !full(1c)
        ,  ug2c → N2 )
N4  = (  sp0d  if full(0d)
        ,  p2u  if full(2c)
        ,  dp1c → N5
        /  sg1u  if !full(1u)
        ,  g2u  if !full(2c)
        ,  dg0c → N3 )
N5  = (  sp1d  if full(1d)
        ,  p0d  if full(0c)
        ,  dp2c → N0
        /  sg2u  if !full(2u)
        ,  g0d  if !full(0c)
        ,  dg1c → N4 )
F   = (  gU  → F
        /  fg2c → N5 }

```

Figure 3-5 : A Fish-Bone stack: Finite state machine specification

The specification stipulates that puts and gets rotate through the storage locations, and that the top level moves data items to or from the second levels only when necessary. There are six states used to control six different data passing shown in table 3-1. The data flow is always from the center, the environment, to the two ends. It begins from the environment up to the top level, then upper level and finally to the outside of stack, and in other direction, down to the top level, then lower level and finally to the outside of stack. The repeated “top level” in Table 3-1 means the same one, but the repeated “outside of stack” is for two different outside storages. There are two types of GasP modules for put command and other two types of GasP modules for get command designed for the same storage locations in the top level because there are two different event group need to be fired at different time.

The specification of the FSM is in Figure 3-5 and more easily to be understood in Table 3-1. For example, when in state *N0* a put command is executed, three internal actions, *up0c*, *p1d* and *sp2d*, will be possibly active depending on the full condition signals of relative storage locations. On the other hand, when a get command is executed, three internal actions, *dg2c*, *g1d*, and *sg0d*, will be possibly active depending on some other condition signals. When in state *N1* a put command is executed, three internal actions, *up1c*, *p2d* and *sp0u*, will be possibly active depending on the full condition signals of relative storage locations. On the other hand, when a get command is executed, three internal actions, *ug0c*, *g2d*, and *sg1d*, will be possibly active depending on the some other condition signals. When in state *N2* a put command is executed, three internal actions, *up2c*, *p0u* and *sp1u*, will be possibly active depending on some condition signals. On the other, when a get command is executed, three internal actions, *ug1c*, *g0u*, *sg2d*, will be possibly active depending on some condition signals. When in state *N3* a put command is executed on the stack,

three internal actions,  $dp0c$ ,  $p1u$  and  $sp2u$ , will be possibly active depending on some condition signals. On the other, when a get command is executed, three internal actions,  $ug2c$ ,  $g1u$ ,  $sg0u$ , will be possibly active depending on some condition signals. When in state  $N4$  a put command is executed, three internal actions,  $dp1c$ ,  $p2u$  and  $sp0d$ , will be possibly active depending on some condition signals. On the other, when a get command is executed, three internal actions,  $dg0c$ ,  $g2u$ ,  $sg1u$ , will be possibly active depending on some condition signals. When in state  $N5$  a put command is executed, three internal actions,  $dp2c$ ,  $p0d$  and  $sp1d$ , will be possibly active depending on some condition signals. On the other, when a get command is executed, three internal actions,  $dg1c$ ,  $g0d$ ,  $sg2u$ , will be possibly active depending on some condition signals.

For normal case in the specification of the FSM, the rules of the conditions that restrict the respective GasP modules are intuitional. The stack fires a GasP module,  $pid$  or  $piu$ , after checking whether the storage location  $ic$  is full. The stack fires GasP modules,  $spid$  or  $spiu$ , after checking whether the storage locations  $id$  or  $iu$  is full respectively. If the storage location  $ic$  is empty, there is no need to fire GasP module,  $pid$  or  $piu$ . Because of the same reason, if the storage location,  $id$  or  $iu$ , is empty, there is no need to fire GasP module,  $spid$  or  $spiu$ , respectively. The stack fires a GasP module,  $gid$  or  $giu$ , after checking whether the storage location  $ic$  is empty. The stack fires GasP module,  $sgid$  or  $sgiu$ , after checking whether the storage location  $id$  or  $iu$  is empty respectively. If the storage location  $ic$  is full, there is no space for the gotten data of the firing GasP module,  $gid$  or  $giu$ . Because of the same reason, if the storage location,  $id$  or  $iu$ , is full, there is no space for the gotten data of the firing GasP module,  $sgid$  or  $sgiu$ , respectively. There are some special additional conditions in states  $N1$  and  $N2$ .



During executing put command in state  $N1$ , an additional condition,  $!full(sOu)$ , of an internal action  $spOu$  is needed to be checked because there is a case that the stack has the last two space with all full outside stacks but normally it doesn't. In this case,  $spOu$  can't be fired because no more space can be put in the outside stack of storage location  $Ou$ . And then, the stack enters state  $N2$ . There is also a special case when the stack has one only space with all full outside stacks. The firing of GasP module,  $pOu$ , is restricted by an additional condition,  $!full(Ou)$ , because the previous put action doesn't move the data outside the stack and the two data residing in  $Oc$  and  $Ou$  are valid. And, no more moves in this case. Besides, GasP module,  $spIu$ , can't be fired because of the same reasons. Every space is used for storage in the almost full stack, and the  $spIu$  can't be fired or some data item in the outside stack of storage location  $Iu$  will be overwritten. Furthermore, GasP module,  $up2c$ , is fired after checking the fullness of the both side of the storage location,  $Ou$  and  $Od$ . If they are both true, the stack enters state  $F$ . If not, the stack immediately enters state  $N3$ .

During executing get command in state  $N2$ , also an additional condition,  $!empty(s2d)$ , of an internal get action  $sg2d$  is needed to be checked because the similar reasons in above paragraph. In the case,  $sg2d$  is not needed to be fired because the almost empty stack with last two data items has no data item outside the stack of storage location  $2d$  but normally it doesn't. And then, the stack enters state  $N1$ . A special case when the stack has the last one data item with all empty outside stacks. The firing of GasP module,  $g2d$ , is restricted by an additional condition,  $full(2d)$ , because there is no data item in storage location  $2d$  in almost empty stack with the last one data item inside. Furthermore, GasP module,  $ugOc$ , is fired after checking the emptiness of the both side of the storage location,  $2u$  and  $2d$ . If they are both true, the stack enters state  $E$ . If not, the stack immediately enters state  $N0$ .

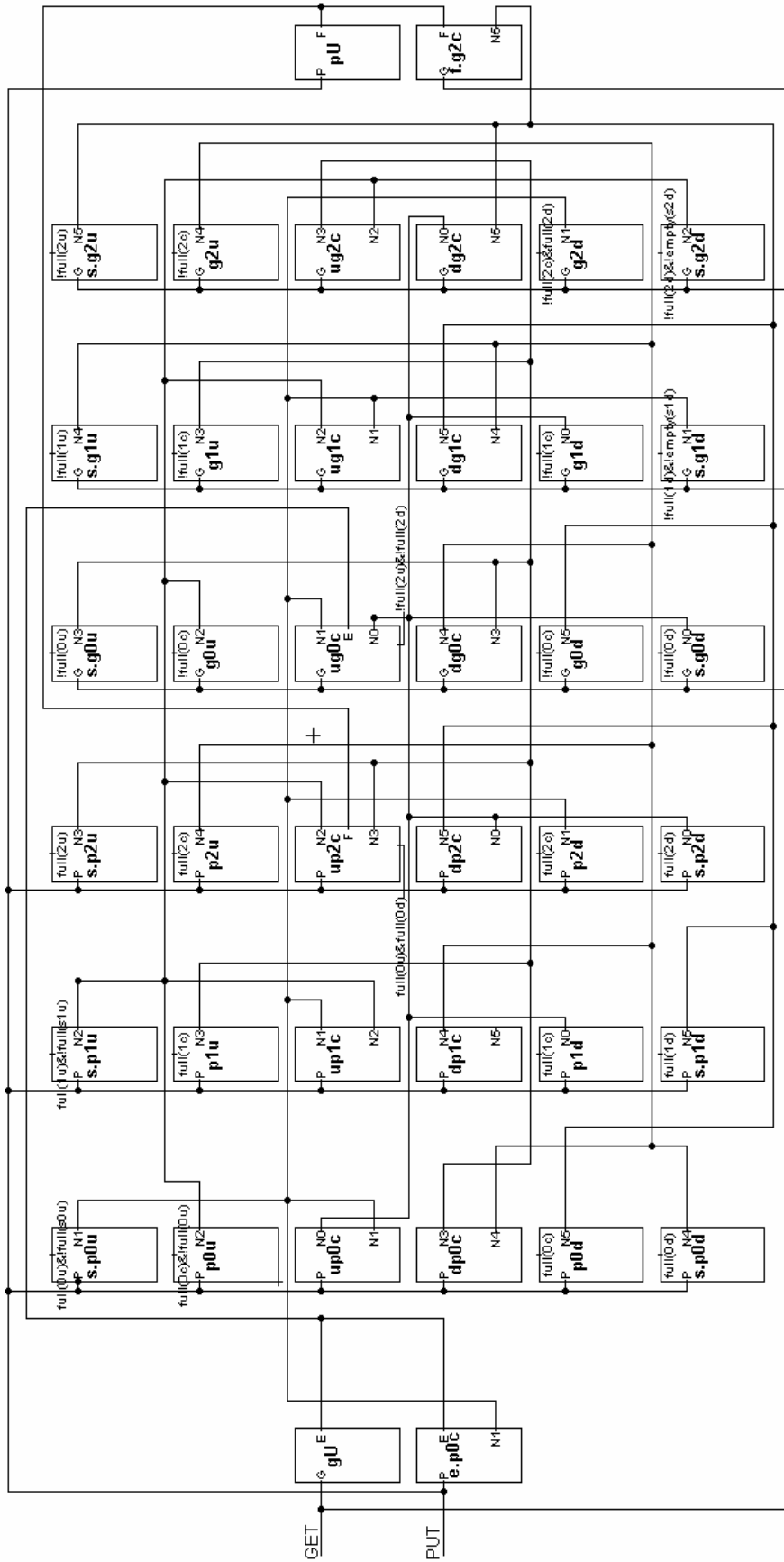


Figure 3-6 : the diagram of control path converted from FSM of Fish-Bone stack

Figure 3-6 shows the converted control path from the finite state machine in Figure 3-5.

### 3.4 Real Implementation of A Fish-Bone Stack

In this section, each component that composes a Fish-Bone stack will be described. First, a top view of a Fish-Bone stack in section 3.4.1. Then, each kind of components will be shown in following sections 3.4.2 to 3.4.6

#### 3.4.1 Top View of A Fish-Bone Stack

In Figure 3-7, a roughly whole design is shown in top view and all of the symbols will be implemented as real circuits. The dashed squares represent a Fish-Bone stack, so there are 7 Fish-Bone stacks in Figure 3-7. The six small ones are the outside stacks, but we disable the function in our later experiments. The symbol drawn with a square and a cross inside represents GasP modules. The symbol drawn with a circle and a character “L” inside represents storage locations. A triangle represents a condition signal that affects the activity of the beside GasP module. And the condition signals are maintained by some condition maintainers. More detailed design can be associated with Figure 3-6 in associatively thinking. The corresponding GasP modules are named as the same words. There is a note for those twelve GsaP modules,  $\{spiu, spid, sgiu, sgid\}$ . Those GasP modules in Figure 3-6 are designed for the control signals as the put or get command signals on the outside stacks, so they are not marked in Figure 3-7.

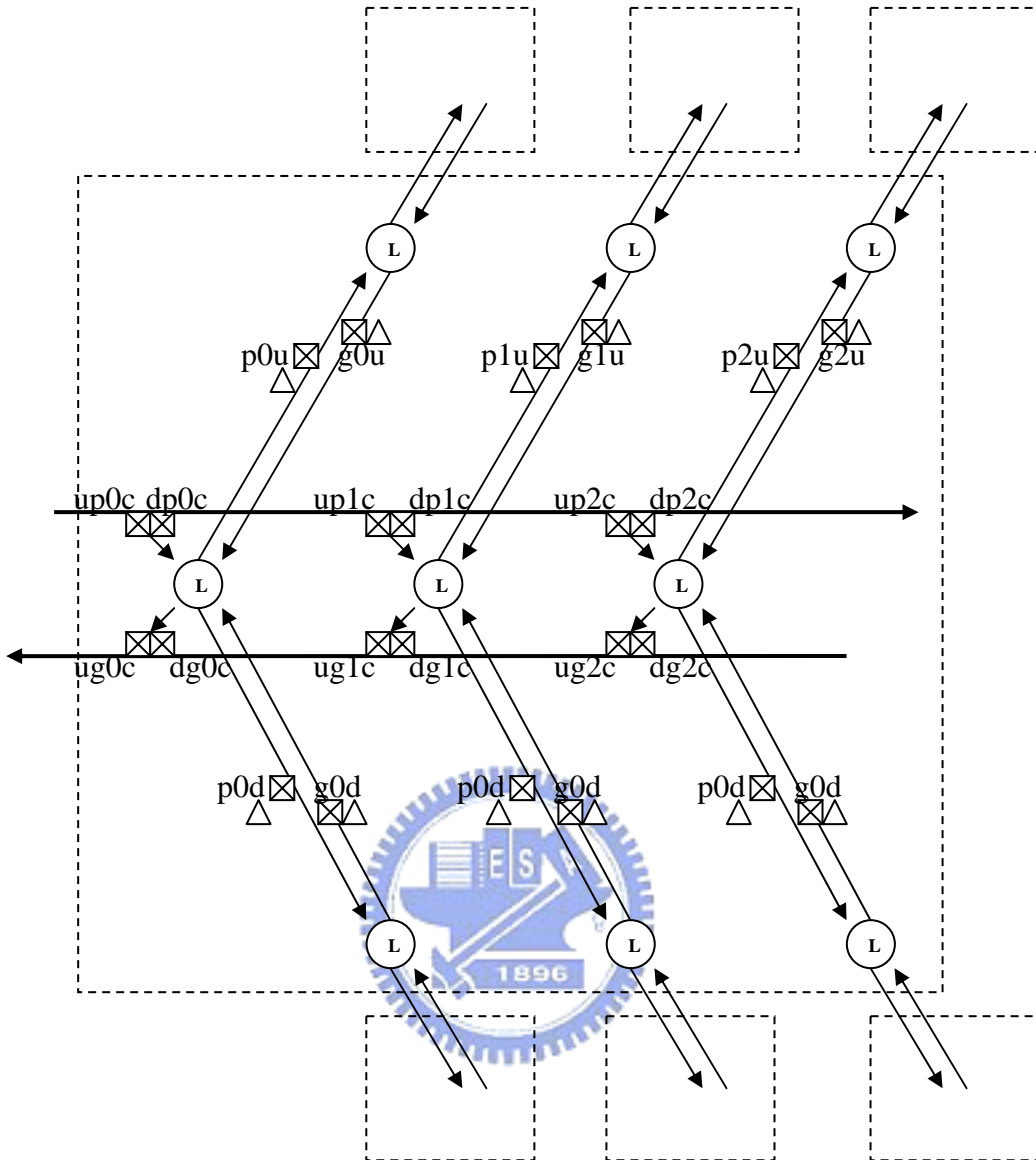
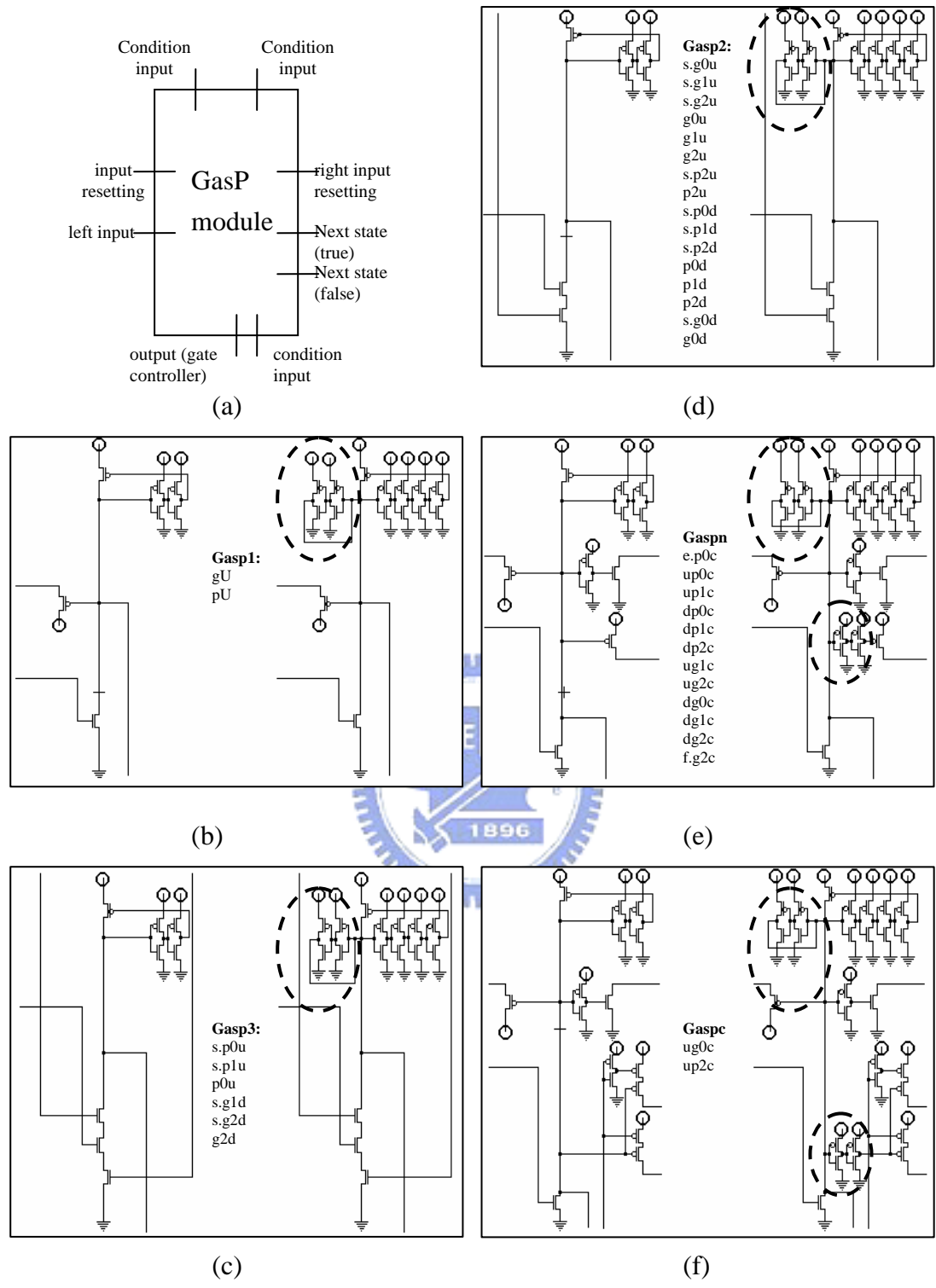


Figure 3-7 : a top view of a complete Fish-Bone stack with both control-path and data-path

### 3.4.2 Implementation of GasP Modules

The idea of designing GasP modules is same as the mention is section 2.3.4. There are several kinds of GsasP modules used in a Fish-Bone stack. The design methodologies are similar. However, there are some difference between our designs and the original ideas for the functionalities. Figure 3-8 shows the modification from the original design mentioned in Jo Ebergen's research to our GasP modules.



**Figure 3-8 : the modification from the original designs**

Because of the requirement of speed performance we extract the current state input as mentioned in section 2.3.4 outside the GasP modules. There are a special

circuits designed for handling them. The reasons are discussed in the next chapter. The methodology of the extraction is using states to control the passing of the input command signal into GasP modules. The left input (command signal) resetting are reverse by using a P-type transistor to set the input high ,and the right input (current state) resetting are unchanged. Besides these significant modifications, other simple modifications of each used GasP modules are shown in the figure. Figure 3-8(a) shows a diagram of a GasP module and it possible in/out ports. In Figure 3-8(b)(c)(d)(e)(f), the left side of each block is the original design and the right side is the design after modifying. The dashed circles show the position of modifications.

There are some simple modifications:

- (a) An additional keeper composed of two serial inverters with connecting the head and the tail is used at the upper dashed circle in each design for keeping the voltage for a long time. The design in original concept in GasP module will cause a function error in a long time because of the power dissipation power in MOS.
- (b) Some buffers are added at needed positions. In GasP network design, there is an important concept- keeping the time delay same for every gate, or keeping the time delay at the right range. The time delay of a gate is not long enough with UMC 0.18 process in the original design, so we add some buffers at needed positions shown at the bottom of each block.

### 3.4.3 Implementation of State Keeper

State keepers are designed to keep the voltage of states. It is a one port circuit with two serial inverters (the tail one is weak inverter) and the head and the tail are

connected together. The detailed circuit is shown in Figure 3-9.

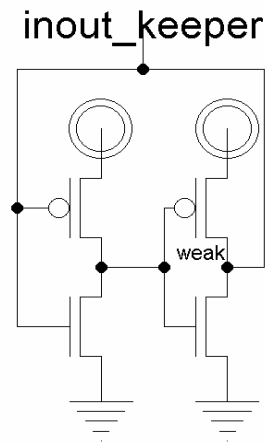


Figure 3-9 : the circuit of a keeper

### 3.4.4 Implementation of Data Storage

The data storages are implemented with two normal inverters, one weak inverter and a serial similar transmission gate device. The two port design is for the perfect output for data. And the similar transmission gate device is used to weaken the feedback electric current provided by the weak inverter.

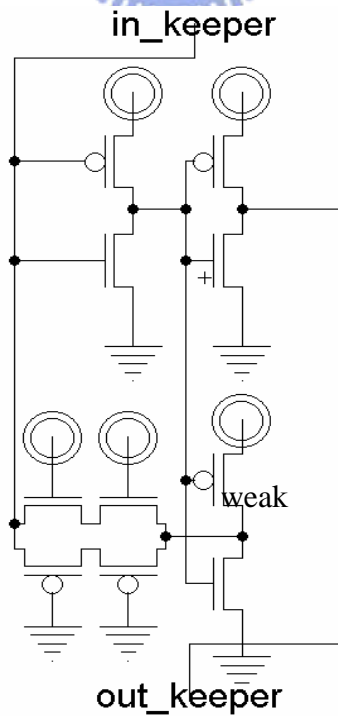
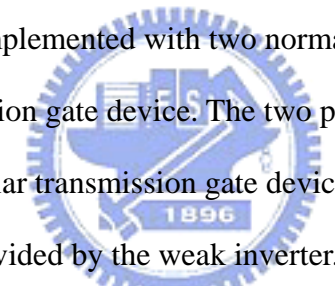


Figure 3-10 : the implementation of data storage

### 3.4.5 Implementation of Condition Maintainer

There are many full condition signals used in Fish-Bone stacks. These signals are maintained by several condition maintainers. There are four kinds of condition maintainer depending on the number of factors that affects the condition signal. For example, if there are  $n$  GasP modules that are active to make a storage location empty, the number of the control input in the left side of the condition maintainer will be  $n$ . Besides, if there are  $m$  GasP modules that are active to make a storage location full, the number of the control input the right side of the condition maintainer will be  $m$ .

The circuit is implemented with a data storage at the center, some N-type transistors at the right side and some P-type transistors at the left side. The data storage is used for keeping the condition status. One of the N-type transistors is active to set the storage high, and one of the P-type transistors is active to set the storage low. The init port is used to initialize the status of the condition maintainer, and  $Q$  and  $Qbar$  are output ports for state and bar of the state respectively.

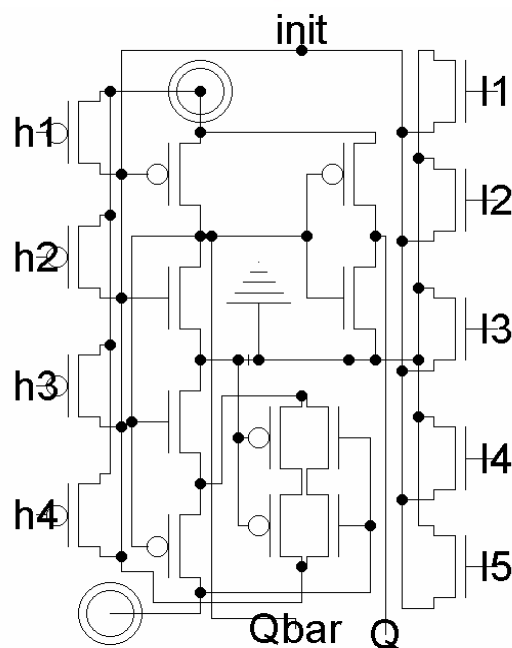


Figure 3-11 : the implementation of condition maintainer



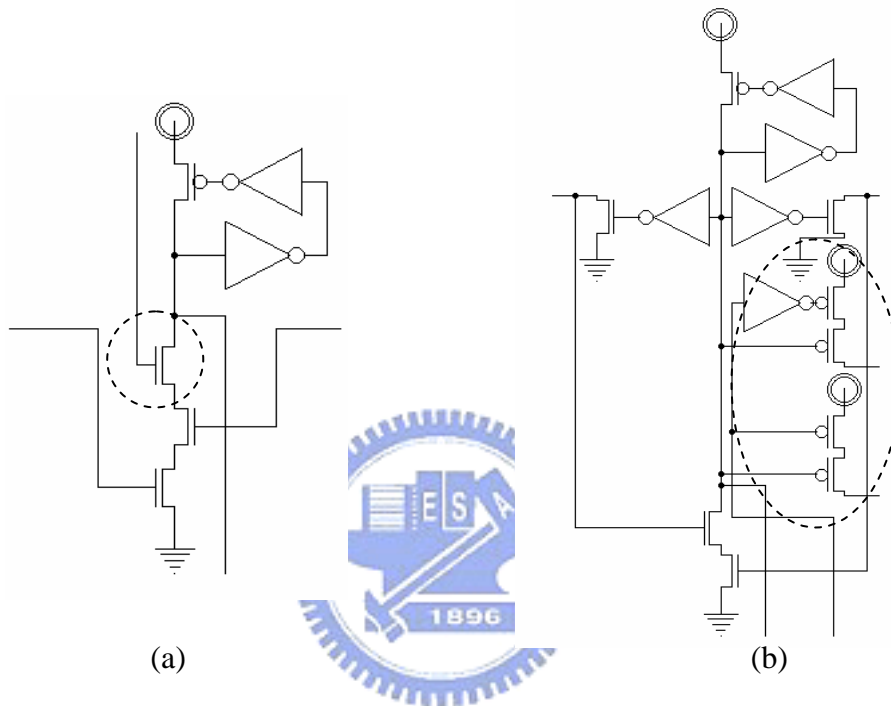
# Chapter 4 Implementation Improvements

We proposed the details of the real implementation of Fish-Bone stacks in chapter 3. Figure 3-6 shows the block diagram of the control path. There are twenty loads of GasP modules needed to be driven by each of put and get signals. The so large loads to be charged and discharged need quite time to complete. And, the version of design can not commit the time requirement in communicating with other stacks at high speed. So there are some improvements needed in real implementation of Fish-Bone stacks to improve the speed performance without unproportionate power consumption.

Basically, there are some small components, like inverters for inverting two possible value  $\{0,1\}$ , N-type transistors for providing perfect strong low signal with high signals, and P-type transistors for providing perfect strong high signal with low signals. Besides, NOR gates are used to merge condition signals. There are two kind of usage on condition signals. If the condition signals are used to affect states, the condition signal is connected to the bottom of the GasP modules. If the condition signals are used to affect the passing of data flow, the condition signal is connected to the upper of the GasP modules. The difference is shown in Figure 4-1. Figure 4-1(a) shows the first case, and Figure 4-1(b) shows the other. When need, a signal is needed to be merged from two condition signals, and we use NOR gates to merge them.

To down low the loads of the put and get signal on GasP modules, we use a trick to solve the problem. Besides solving the problem, less power consumption is another gift brought by that modification. We use states to control the need of other GasP modules and use transmission gates to block the electric currents to pass into the stack. Precisely, we extract the circuits of current state out of GasP modules, and use

transmission gates to isolate the passing of current between put signal offered by the environment and the real need in Fish-Bone stack. So that, we save a lot of power dissipation on the useless charging and discharging in the modified Fish-Bone stack. Because of these modifications, some original GasP module designs with low active property are reversely used. Some components are added for providing better signals.



**Figure 4-1 : The difference of two kind handling with condition signals : (a) condition signals affect the passing of data flow (b) condition signal affect next state**

The final circuit is shown in Figure 4-2. The figure is just for viewing the dimension. And the details have been talked about in previous sections.

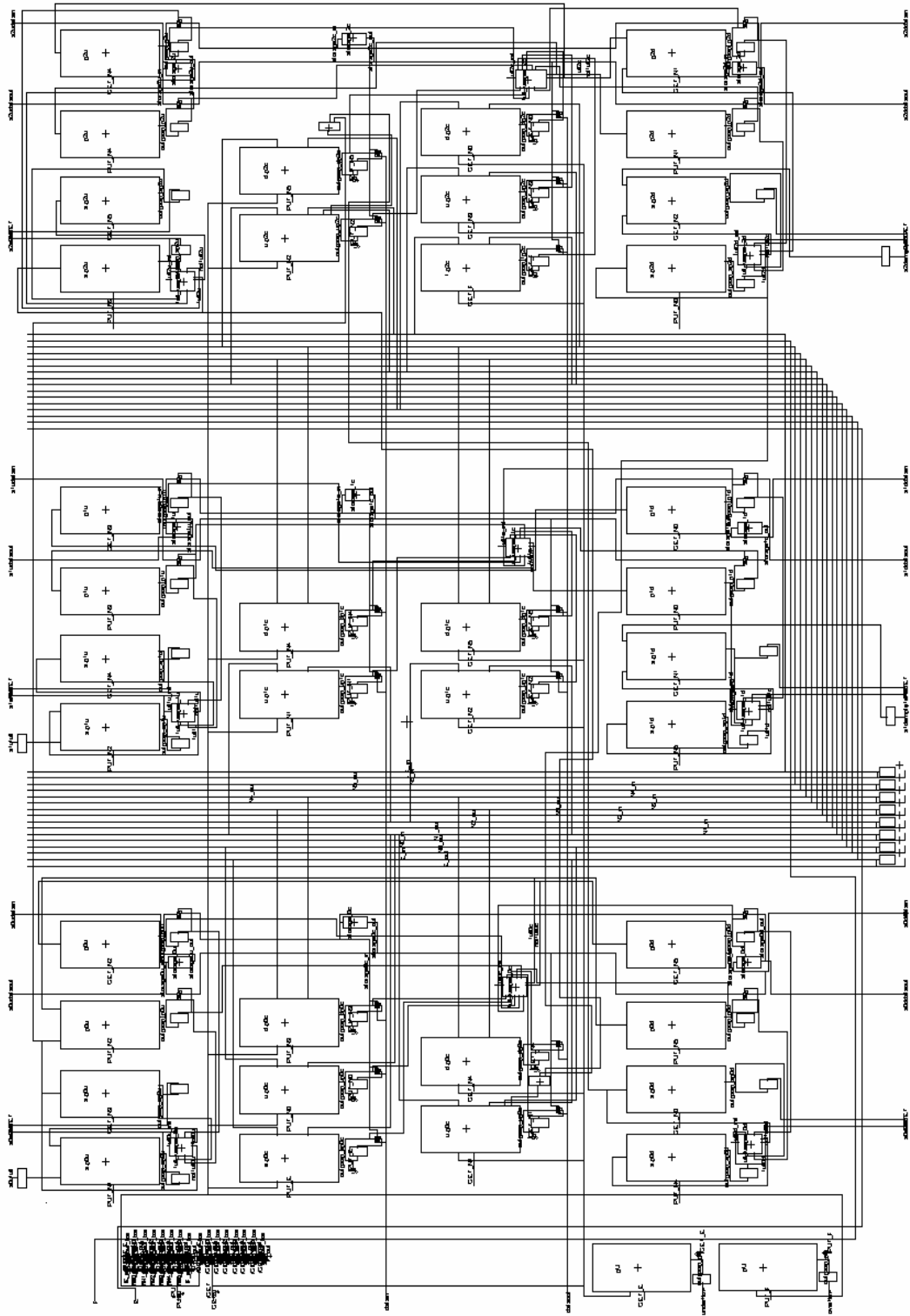


Figure 4-2 : the view of the final circuit of a Fish-Bone stack

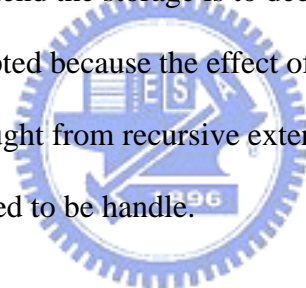
# Chapter 5 The Extension of Stack Place

There are some methods to extend the storage of stack on Fish-Bone stack design. In this chapter, we progress step by step to reveal the better solution to extend.

## 5.1 Straightforward Extension

There is an intuitional solution to extend to the storage of Fish-Bone stacks. The method is just to enlarge the length of the spinal canal of Fish-Bone. However, the enlarged Fish-Bone stacks result in larger loads that put and get signals need to drive. So longer time is needed to complete a put or get command.

The other direction to extend the storage is to deepen the level to a Fish-Bone stack. The solution is not adopted because the effect of power saving brought from that idea is similar to that brought from recursive extension talked about later but more complex controls are need to be handle.



## 5.2 Extension in Tree Structure

The idea is from Jo Ebergen's research. A two-place linear stack is applied to construct the tree stack. As a binary tree structure, a more level brings about two times the storages of the original one. Fish-Bone stacks are used as the leaf nodes. The maximal internal data movements are decided by the number of levels, and there are three levels in a three-level three-place linear stack but two in a Fish-Bone stack. So in both designs with 42 places, the maximal internal data movements in Jo Ebergen's design is 5 but 4 in the stack with Fish-Bone stacks. Besides, there are more than one kinds of components used in constructing the stack. And, more timing problems caused by several different components will happen. However, there is a more simple

design method to extend storage.

### 5.3 Extension in Recursive Structure

A simple solution may have the better power performance is in a recursive structure. A structure is shown in Figure 5-1. Each square with nine grids inside is a Fish-Bone stack. The structure with 63(9\*7) storage locations with maximal four internal data movements is more power-efficient than the extension in the tree structure in ratiocination. However, the rapid increasing in storages is not always suitable for every design.

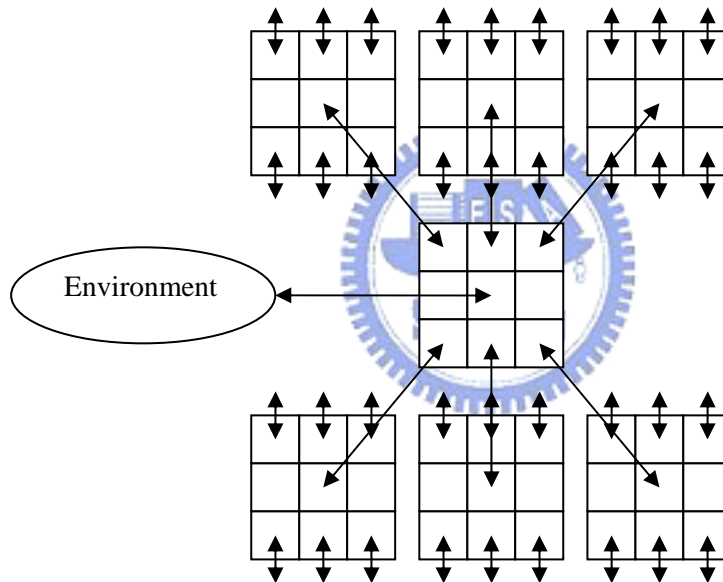


Figure 5-1 : Extension in a recursive structure

### 5.4 Tree Structure Used in Power Consumption Comparison

For the ease to compare with the hybrid stack design in Jo Ebergn's research. We chose the structure described in section 5.2 for power comparison. A 42-place stack consisting of a two-level tree stack structure and four Fish-Bone stacks are used in our experiments for power comparison. And we take the structure for later discussion in the following section.

# Chapter 6 Power Consumption Evaluation

We implemented the Fish-Bone stacks in transistor level and simulated them in the UMC 180nm process (simulation of dynamic dissipation only). Besides, the simulator is HSPICE 2003-09-sp1. In order to be able to compare with the “Hybrid stack”, we re-implemented the design with the same requirements as possible. The goal of our design is for lower power consumption, so we just replaced the three-level three-place linear stacks with our design, Fish-Bone stacks. In power consumption experiment, we set the speed at 2Ghz (almost the limit of the ability, but some tolerances is reserved) to both designs, or 500ps time interval between two commands. There are some results in the following sub-sections. A comparison of one three-level three-place linear stack and one Fish-Bone stack will be shown in section 6.1. In section 6.2, the counts of data moves in both two designs are compared. Furthermore, the results of power consumption are shown and their comparison in section 6.3. Finally, there is an easy analysis represented in section 6.4.

## 6.1 Power Consumption Testing on A Three-level Three-place Linear Stack and A Fish-Bone Stack

For getting the power consumption of the leaf node, we extract out these designs and test the power consumption only. There are four special data sequence and only one stack command sequence chosen for both designs. A rough difference of power consumption is shown in this experiment. The results of HSPICE simulator are shown in Table 6-1. The stack command sequence is composed of ten continuous put commands and concatenating ten continuous get commands. The tenth command and the twentieth command of the command sequence are for the testing of underflow and

overflow protections. In the results shown in Table 6-1, the difference of the average power consumption is quite large. It is about 50% of the average power consumption of a Fish-Bone stack.

input data	3-level 3-place linear stack (mW)	Fish-Bone stack (mW)	Difference (mW)
<b>1110001110</b>	16.760	11.412	5.348
	<i>146.86%</i>	<i>100%</i>	<i>46.86%</i>
<b>1111111111</b>	15.989	10.453	5.536
	<i>152.96%</i>	<i>100%</i>	<i>52.96%</i>
<b>0000000000</b>	15.523	10.146	5.377
	<i>153.00%</i>	<i>100%</i>	<i>53.00%</i>
<b>1010101010</b>	16.822	11.567	5.255
	<i>145.43%</i>	<i>100%</i>	<i>45.43%</i>
<b>average</b>	<b>16.274</b>	<b>10.895</b>	<b>5.379</b>
	<i>149.56%</i>	<i>100%</i>	<i>49.56%</i>

Table 6-1 : The difference of average power consumption between one 3-level 3-place linear stack and one Fish-Bone stack

## 6.2 Power Consumption Testing on Both Stacks with Different Leaf Node Designs

In the section, we concern the power consumption of two stacks with different leaf node stacks, three-level three-place linear stacks and Fish-Bone stacks. For the ease to compare with the “Hybrid stack”, we replace the leaf node stacks with our Fish-Bone stacks. The number of storage locations is still 42. Besides, we set some initial values, so that the Fish-Bone stacks can be used as leaf node. All the conditions gotten from the outside of the Fish-Bone stack are set as high, like *!full(s0u)*, *!empty(s1d)*, *!full(s1u)*, and *!empty(s2d)*. We randomly generate 100 test data with random combination sized 100 of put and get command sequences as the input. Table 6-2 shows the results from simulations. The table exhibits that the average power consumption of a Hybrid stack with three-level three-place linear

stacks (HS) is averagely 120.46% to a Hybrid stack with Fish-Bone stacks (FB) and that of a linear stack (LS) is averagely 1315.47% to FB. The extent of the ratio of averagely gained power here seems lower than that in the previous section. That's because the frequently used components are tree stacks, not the leaf nodes.

Consequently, the ratio of the gained power is lower than that with comparing the leaf nodes only.

No.	Hybrid stack (HS) (mW)	Hybrid stack with Fish-Bone stack(FB) (mW)	Linear stack (LS) (mW)	Difference to FB (mW)	$P_{HS}/P_{FB}$	Difference to FB (mW)	$P_{LS}/P_{FB}$
1	27.320	24.341	325.17	2.979	112.24%	300.829	1335.89%
2	28.394	24.730	353.17	3.664	114.82%	328.440	1428.10%
3	29.740	25.033	322.26	4.707	118.80%	297.227	1287.34%
4	30.850	25.350	342.56	5.500	121.70%	317.210	1351.32%
5	32.900	26.640	344.06	6.260	123.50%	317.420	1291.52%
6	27.260	23.620	359.35	3.640	115.41%	335.730	1521.38%
7	27.650	24.110	339.78	3.540	114.68%	315.670	1409.29%
8	30.980	25.680	360.24	5.300	120.64%	334.560	1402.80%
9	28.590	24.750	335.77	3.840	115.52%	311.020	1356.65%
10	26.560	23.510	311.76	3.050	112.97%	288.250	1326.07%
11	31.520	25.772	321.84	5.748	122.30%	296.068	1248.80%
12	31.780	25.780	342.65	6.000	123.27%	316.870	1329.13%
13	27.630	24.441	333.98	3.189	113.05%	309.539	1366.47%
14	31.160	25.865	342.98	5.295	120.47%	317.115	1326.04%
15	31.170	25.810	324.12	5.360	120.77%	298.310	1255.79%
16	34.250	26.903	350.13	7.347	127.31%	323.227	1301.45%
17	33.104	26.649	350.35	6.455	124.22%	323.701	1314.68%
18	32.310	26.283	334.45	6.027	122.93%	308.167	1272.50%
19	33.060	26.660	339.60	6.400	124.01%	312.940	1273.82%
20	30.910	25.552	328.79	5.358	120.97%	303.238	1286.75%
21	31.670	26.127	335.87	5.543	121.22%	309.743	1285.53%
22	27.840	24.428	341.12	3.412	113.97%	316.692	1396.43%
23	32.300	26.318	336.68	5.982	122.73%	310.362	1279.28%



24	27.750	24.132	355.27	3.618	114.99%	331.138	1472.19%
25	29.360	24.383	316.68	4.977	120.41%	292.297	1298.77%
26	33.040	26.135	342.53	6.905	126.42%	316.395	1310.62%
27	30.670	25.207	364.47	5.463	121.67%	339.263	1445.91%
28	29.660	24.887	344.38	4.773	119.18%	319.493	1383.77%
29	33.110	26.857	339.50	6.253	123.28%	312.643	1264.10%
30	31.320	25.474	316.84	5.846	122.95%	291.366	1243.78%
31	31.980	26.530	333.19	5.450	120.54%	306.660	1255.90%
32	32.570	25.450	346.25	7.120	127.98%	320.800	1360.51%
33	30.880	26.000	344.20	4.880	118.77%	318.200	1323.85%
34	29.220	25.380	337.85	3.840	115.13%	312.470	1331.17%
35	31.320	25.820	325.79	5.500	121.30%	299.970	1261.77%
36	29.490	25.430	356.75	4.060	115.97%	331.320	1402.87%
37	27.910	24.920	336.74	2.990	112.00%	311.820	1351.28%
38	30.400	25.170	337.53	5.230	120.78%	312.358	1340.99%
39	28.900	25.370	337.68	3.530	113.91%	312.313	1331.03%
40	30.510	25.290	358.22	5.220	120.64%	332.930	1416.45%
41	31.690	26.490	322.21	5.200	119.63%	295.720	1216.35%
42	33.280	26.730	330.75	6.550	124.50%	304.020	1237.37%
43	32.110	26.240	343.68	5.870	122.37%	317.440	1309.76%
44	31.980	26.170	335.50	5.810	122.20%	309.330	1282.00%
45	29.130	24.840	358.76	4.290	117.27%	333.920	1444.28%
46	30.650	25.240	320.92	5.410	121.43%	295.680	1271.47%
47	31.710	25.960	347.52	5.750	122.15%	321.560	1338.67%
48	30.980	25.640	325.28	5.340	120.83%	299.640	1268.64%
49	32.370	25.960	331.46	6.410	124.69%	305.500	1276.81%
50	31.650	26.330	319.73	5.320	120.21%	293.400	1214.32%
51	31.000	25.750	329.09	5.250	120.39%	303.340	1278.02%
52	28.020	24.360	340.81	3.660	115.02%	316.450	1399.06%
53	33.180	26.190	345.97	6.990	126.69%	319.780	1321.00%
54	31.260	26.160	352.69	5.100	119.50%	326.530	1348.20%
55	28.540	24.860	338.63	3.680	114.80%	313.770	1362.15%
56	32.340	26.350	357.25	5.990	122.73%	330.900	1355.79%
57	33.110	26.480	333.90	6.630	125.04%	307.420	1260.95%
58	31.320	26.220	346.46	5.100	119.45%	320.240	1321.36%
59	34.400	26.810	356.44	7.590	128.31%	329.630	1329.50%

60	28.550	24.960	351.97	3.590	114.38%	327.010	1410.14%
61	32.350	26.180	325.41	6.170	123.57%	299.230	1242.97%
62	31.490	26.000	355.51	5.490	121.12%	329.510	1367.35%
63	32.820	26.630	318.64	6.190	123.24%	292.010	1196.55%
64	29.400	25.280	321.29	4.120	116.30%	296.010	1270.93%
65	32.050	25.690	355.14	6.360	124.76%	329.450	1382.41%
66	32.580	26.510	321.41	6.070	122.90%	294.900	1212.41%
67	33.070	26.560	346.19	6.510	124.51%	319.630	1303.43%
68	31.830	26.240	360.37	5.590	121.30%	334.130	1373.36%
69	31.420	26.180	361.52	5.240	120.02%	335.340	1380.90%
70	28.060	24.930	325.58	3.130	112.56%	300.650	1305.98%
71	32.730	26.470	334.30	6.260	123.65%	307.830	1262.94%
72	33.210	26.280	328.80	6.930	126.37%	302.520	1251.14%
73	27.210	23.680	363.10	3.530	114.91%	339.420	1533.36%
74	31.060	25.920	317.78	5.140	119.83%	291.860	1226.00%
75	30.760	25.410	344.06	5.350	121.05%	318.650	1354.03%
76	29.190	24.940	347.31	4.250	117.04%	322.370	1392.58%
77	33.590	26.720	326.54	6.870	125.71%	299.820	1222.08%
78	26.550	23.810	323.65	2.740	111.51%	299.840	1359.30%
79	30.850	25.670	330.93	5.180	120.18%	305.260	1289.17%
80	28.890	25.160	346.56	3.730	114.83%	321.400	1377.42%
81	32.510	26.290	318.06	6.220	123.66%	291.770	1209.81%
82	31.260	25.970	329.33	5.290	120.37%	303.360	1268.12%
83	32.800	26.210	331.51	6.590	125.14%	305.300	1264.82%
84	31.050	25.750	341.02	5.300	120.58%	315.270	1324.35%
85	28.870	25.110	347.66	3.760	114.97%	322.550	1384.55%
86	32.300	26.340	322.45	5.960	122.63%	296.110	1224.18%
87	28.330	25.040	325.87	3.290	113.14%	300.830	1301.40%
88	31.150	26.030	339.89	5.120	119.67%	313.860	1305.76%
89	33.580	26.890	331.67	6.690	124.88%	304.780	1233.43%
90	33.760	26.900	344.98	6.860	125.50%	318.080	1282.45%
91	32.020	26.060	316.73	5.960	122.87%	290.670	1215.39%
92	32.060	26.150	323.97	5.910	122.60%	297.820	1238.89%
93	29.460	24.730	343.55	4.730	119.13%	318.820	1389.20%
94	31.500	26.070	329.58	5.430	120.83%	303.510	1264.21%
95	32.450	26.460	331.51	5.990	122.64%	305.050	1252.87%

96	33.070	26.250	326.83	6.820	125.98%	300.580	1245.07%
97	30.820	25.580	325.22	5.240	120.48%	299.640	1271.38%
98	33.350	26.760	354.30	6.590	124.63%	327.540	1323.99%
99	31.190	25.780	332.87	5.410	120.99%	307.090	1291.19%
100	33.510	26.710	337.66	6.800	125.46%	310.950	1264.17%
Avg.	30.985	25.699	337.683	5.286	120.46%	311.98	1315.47%

**Table 6-2 : Results of average power consumption between the Hybrid stack with 3-level 3-place linear stacks (HS), the Hybrid stack with Fish-Bone stack (FB), and the linear stack (LS)**

### 6.3 Data Movements Counting

We implemented a utility to simulate the behavior of HS and FB and to count the internal data moves in HS and FB designs. The generated 100 random data are simulated for the analysis of the relation between the power consumption and the number of their data movements. Furthermore, the saved data movements are also concerned because we consider that is the main factor to affect the extent of power consumption. We reorder the index with the increasing number of data movements for easily observing the relation. Results are shown in Table 6-3.

No.	Power Difference between HS and FB (mW)	$P_{HS}/P_{FB}$	Data moves in HS	Data moves in FB	Difference of data moves
1	2.98	112.24%	332	328	4
70	3.13	112.56%	336	332	4
39	3.53	113.91%	348	343	5
87	3.29	113.14%	340	333	7
78	2.74	111.51%	324	316	8
37	2.99	112.00%	336	327	9
10	3.05	112.97%	315	303	12
22	3.41	113.97%	332	320	12
60	3.59	114.38%	338	326	12
2	3.66	114.82%	337	325	12
80	3.73	114.83%	344	332	12
73	3.53	114.91%	323	311	12
85	3.76	114.97%	341	329	12
24	3.62	114.99%	329	317	12
52	3.66	115.02%	335	323	12
6	3.64	115.41%	320	308	12

36	4.06	115.97%	350	338	12
13	3.19	113.05%	334	320	14
55	3.68	114.80%	340	326	14
64	4.12	116.30%	349	335	14
33	4.88	118.77%	364	350	14
58	5.10	119.45%	372	358	14
7	3.54	114.68%	330	315	15
9	3.84	115.52%	340	325	15
34	3.84	115.13%	354	338	16
69	5.24	120.02%	368	352	16
45	4.29	117.27%	350	332	18
3	4.71	118.80%	357	339	18
76	4.25	117.04%	352	332	20
93	4.73	119.13%	353	333	20
54	5.10	119.50%	368	348	20
41	5.20	119.63%	374	354	20
74	5.14	119.83%	366	346	20
50	5.32	120.21%	376	356	20
88	5.12	119.67%	374	353	21
51	5.25	120.39%	372	351	21
31	5.45	120.54%	378	357	21
8	5.30	120.64%	364	342	22
40	5.22	120.64%	360	338	22
14	5.30	120.47%	372	349	23
28	4.77	119.18%	360	336	24
79	5.18	120.18%	368	344	24
82	5.29	120.37%	372	348	24
25	4.98	120.41%	348	324	24
97	5.24	120.48%	366	342	24
84	5.30	120.58%	369	345	24
15	5.36	120.77%	369	345	24
38	5.23	120.78%	358	334	24
48	5.34	120.83%	364	340	24
94	5.43	120.83%	372	348	24
20	5.36	120.97%	363	339	24
62	5.49	121.12%	376	352	24
21	5.54	121.22%	374	350	24
46	5.41	121.43%	367	343	24
27	5.46	121.67%	359	335	24
11	5.75	122.30%	374	350	24
92	5.91	122.60%	376	352	24
86	5.96	122.63%	382	358	24
95	5.99	122.64%	385	361	24
91	5.96	122.87%	376	352	24
66	6.07	122.90%	378	354	24
30	5.85	122.95%	370	346	24
44	5.81	122.20%	374	349	25
99	5.41	120.99%	376	350	26

75	5.35	121.05%	368	342	26
35	5.50	121.30%	373	347	26
68	5.59	121.30%	372	346	26
4	5.50	121.70%	362	336	26
56	5.99	122.73%	379	353	26
29	6.25	123.28%	386	360	26
47	5.75	122.15%	378	351	27
43	5.87	122.37%	384	357	27
61	6.17	123.57%	384	357	27
23	5.98	122.73%	382	354	28
63	6.19	123.24%	386	358	28
32	7.12	127.98%	386	358	28
100	6.80	125.46%	396	367	29
18	6.03	122.93%	384	354	30
12	6.00	123.27%	377	347	30
71	6.26	123.65%	388	358	30
42	6.55	124.50%	392	362	30
98	6.59	124.63%	392	362	30
65	6.36	124.76%	376	346	30
5	6.26	123.50%	386	355	31
19	6.40	124.01%	392	361	31
67	6.51	124.51%	392	361	31
81	6.22	123.66%	386	354	32
83	6.59	125.14%	394	362	32
49	6.41	124.69%	382	349	33
77	6.87	125.71%	396	363	33
89	6.69	124.88%	394	359	35
90	6.86	125.50%	396	361	35
17	6.46	124.22%	395	359	36
96	6.82	125.98%	397	361	36
72	6.93	126.37%	394	358	36
26	6.91	126.42%	394	358	36
53	6.99	126.69%	392	356	36
16	7.35	127.31%	404	368	36
59	7.59	128.31%	400	364	36
57	6.63	125.04%	398	361	37
Avg.	5.29	120.46%	367.6	344.82	22.78

**Table 6-3 : The relation between the power consumption and the number of their data movements**

## 6.4 Used Transistors Counting

Although the Fish-Bone stack design is more complex than the design of the original hybrid stack, it is surprising that the number of used transistors is not more

than that used in the original hybrid stack. The numbers of the transistors used by two stacks are in Table 6-4. Although the more complex controls are needed to be handles and more GasP modules are needed, the number of transistors used is not increasing. The hybrid stack with three-level three-place linear stacks (HS) used 5198 transistors and the hybrid stack with Fish-Bone stacks (FB) used 4902 transistors only. Furthermore, the stack with Fish-Bone stacks got the better power performance, about 83% of the power consumed by HS.

The tricky situation occurs because there are many GasP modules used in Fish-Bone stacks are just for generating the control signals, not for the next state, and few transistors are needed to complete those GasP modules. Besides, lots of components are extracted out of the GasP modules. However, Table 6-4 doesn't provide precise information on area cost because the transistor used in both design are not uniform. The sizes of transistors are turned for committing the delay requirement in both designs. So, the information in Table 6-4 is just for a reference.

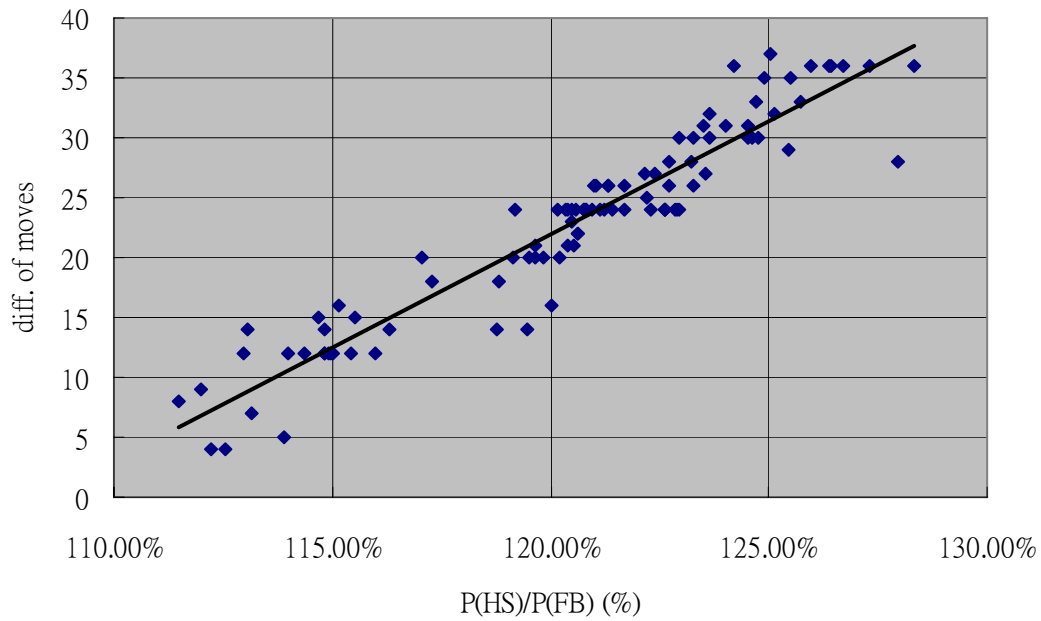
	Hybrid stack with 3-level 3-place linear stacks (HS)	Hybrid stack with Fish-Bone stacks (FB)	Linear stack (LS)
<b>Used transistor count</b>	5198	4902	N/A
	106.58%	100%	N/A
<b>Average power consumption (mW)</b>	30.985	25.699	337.683
	120.46%	100%	1315.47%

**Table 6-4 : the counting of both stacks with different leaf node designs**

## 6.5 Analysis of Result from Simulations

Results from simulations are in Table 6-1 to Table 6-4. The idea that to reduce data movements can low down the power consumption is proved through these experiments. Figure 6-1 shows the trend of the power consumption with the

increasing difference of the data moves between HS and FB. The x axis is the ratio of  $P(HS)$  and  $P(FB)$  ( $P(HS)$  is the power consumed by HS and  $P(FB)$  is the power consumed by FB). The curve is not always increasing, but it goes in an increasing fashion. The reason is that some data items are actually not moved in some of the input commands because the target and the source are same value. And, there is no charge and discharge on these relative transistors.



**Figure 6-1 : the trend of the relation between the difference of moves and the  $P(HS)/P(FB)$**

# Chapter 7 Conclusions and Future Works

In this thesis, we proposed a power-efficient stack, named Fish-Bone stack, with fewer internal data movements during executing stack commands. Fish-Bone stack can execute stack commands within maximally two internal data movements in a dimension three Fish-Bone stack. We consider that the fewer internal data movements result in less power consumption and we have proved that with the results from simulations by HSPICE.

The implementations of Fish-Bone stacks do not include wire delay information. The weakness of that without wire delay information caused the imprecise results gotten from simulations in experiments. However, we reserved a tolerance on speed when designing the Fish-Bone stack. The average power consumption of a hybrid stack with Fish-Bone stacks with generated 100 random combination and sized 100 stack command sequences and random data input is 25.699mW, and the value is about 83% to the average consumed power of a hybrid stack with three-level three-place linear stacks with no more transistors. Besides, in the experiment of power consumption with leaf stacks only, the results shows average 49.56% power consumption gained to a three-level three-place linear stack by a Fish-Bone stack.

Then about the future works, more precise results from simulations can be gotten with the layout of the designs. Also, we consider that using Fish-Bone stacks only will lead to less power consumption as mentioned before. An idea of an extension in aster fashion was shown in section 5.3. And, there is only one kind of component to recursively compose a stack design. The easy and simple extension may cause some new problems in timing. The designs are only clock-less new and other properties of asynchronous circuits need more researches to reveal more.



# Reference

- [1] J. Ebergen, D. Finchelstein, R. Kao, J. Lexau, and D. Hopkins, "A fast and energy-efficient stack," *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pp. 7, 2004.
- [2] S. Hauck, "Asynchronous design methodologies: an overview," *Proceedings of the IEEE*, vol. 83, pp. 69, 1995.
- [3] T. H. Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, "Automatic synthesis of asynchronous circuits from high-level specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, pp. 1185, 1989.
- [4] C. Fu-Chiung, S. H. Unger, and M. Theobald, "Self-timed carry-lookahead adders," *Computers, IEEE Transactions on*, vol. 49, pp. 659, 2000.
- [5] A. Efthymiou, W. Suntiarnontut, J. Garside, and L. E. M. Brackenbury, "An asynchronous, iterative implementation of the original Booth multiplication algorithm," *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pp. 207, 2004.
- [6] T. Werner and V. Akella, "Counterflow pipeline based dynamic instruction scheduling," *Advanced Research in Asynchronous Circuits and Systems, 1996. Proceedings., Second International Symposium on*, pp. 69, 1996.
- [7] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80C51 microcontroller," *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on*, pp. 96, 1998.
- [8] N. Parlangeau and A. Marchal, "AMULET: automatic multisensor speech labelling and event tracking: study of the spatio-temporal correlations in voiceless plosive production," *1996. ICSLP 96. Proceedings., Fourth International Conference on*, vol. 3, pp. 1926, 1996.
- [9] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: design of a quasi-delay-insensitive microprocessor," *Design & Test of Computers, IEEE*, vol. 11, pp. 50, 1994.
- [10] A. Takamura, M. Imai, M. Ozawa, I. Fukasaku, T. Fujii, M. Kuwako, Y. Ueno, and T. Nanya, "TITAC-2: an asynchronous 32-bit microprocessor," presented at Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific, 1998.
- [11] P. Lavoie, D. Haccoun, and Y. Savaria, "A systolic architecture for fast stack

- sequential decoders," *Communications, IEEE Transactions on*, vol. 42, pp. 324, 1994.
- [12] J. C. Ebergen and S. Gingras, "An asynchronous stack with constant response time.," *Technical Report CS-93-11, Computer Science Dept., Univ. of Waterloo, Canada*, 1993.
- [13] A. J. Martin, "A synthesis method for self-timed VLSI circuits," *In Proc. International Conf. Computer Design (ICCD)*, pp. 224-229, 1987.
- [14] M. B. Josephs and J. T. Udding, "Implementing a stack as a delay-insensitive circuit," in *Asynchronous Design Methodologies*, vol. A-28 of IFIP Transactions, S. Furber and M. Edwards, Eds.: Elsevier Science Publishers, 1993, pp. 123-135.
- [15] M. M. Mano and C. R. Kime, "Instructure set architecture," in *Logic and comuter design fundamentals*. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 2000, pp. 482-486.
- [16] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau, S. M. Fairbanks, and I. E. Sutherland, "Two FIFO ring performance experiments," *Proceedings of the IEEE*, vol. 87, pp. 297, 1999.
- [17] I. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pp. 46, 2001.
- [18] J. Ebergen, J. Gainsley, and P. Cunningham, "Transistor sizing: how to control the speed and energy consumption of a circuit," *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pp. 51, 2004.