國立交通大學

資訊工程學系

碩士論文

在超長指令字的數位訊號處理器下的指令排程以降低能量消耗為目的

Instruction Level Scheduling for Low-Power on VLIW DSP

研究生：楊偉帆

指導教授：陳正 教授

中華民國 九十四年六月

在超長指令字的數位訊號處理器下的指令排程以降低能量消耗為目的

Instruction Level Scheduling for Low-Power on VLIW DSP

研究生：楊偉帆　　Student : Wei-Fan Yang

指導教授：陳正 教授　　Advisor : Prof. Cheng Chen

國立交通大學

資訊工程學系

碩士論文

A Thesis

Submitted to

Institute 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

Hsinchu, Taiwan, Republic of China

中華民國 九十四年 六月

在超長指令字架構下的數位訊號處理器作指令排程以降低能量消耗為目的

研究生：楊偉帆　　　指導教授：陳正 教授

國立交通大學資訊工程學系碩士班

摘要

個人攜帶式產品在現代生活中已經越來越普及。例如手機、數位照相機、PDA 等等。他們大都是靠充電電池運作，所以如何降低他們的電力消耗以延長他們的使用時間變成一個很重要的議題。而在處理器消耗能量的比例方面，指令的匯流排因為變化很頻繁，所以它佔了很大的比例。其中 switching activities 是影響指令匯流排能量消耗的一個很重要的因素。我們在本篇論文中針對降低 switching activities 提出一個方法 *Greedy Switching Activities Scheduling* (GSAS)。GSAS 包含了兩個階段，第一個階段是針對指令做排程並以降低 switching activities 為目的。根據實驗結果 GSAS 比 MSAS 更省能量消耗。第二階段是針對第一階段排出的 schedule 做 registers re-assign 的動作，目的也是爲了降低 switching activities，而根據實驗結果可以發現第二階段可以更近一步的結省電力的消耗。

Instruction Level Scheduling for Low-Power on VLIW DSP

Student : Wei-Fan Yang Advisor : Prof. Cheng Chen

Institute of Computer Science and Information Engineering National Chiao Tung University

Abstract

Portable devices become so popular nowadays. How to reduce the power consumption in these portable devices, such as digital camera, cellular phone, PDA, etc., becomes a more and more important issue. Switching activities is one of the most important factors in power dissipation. In this thesis, we focus on reducing the power consumption of application on VLIW architectures by reducing switching activities on the instruction bus. An algorithm, *Greedy Switching Activities Scheduling* (GSAS) is proposed. The algorithm contains two phases. The phase one of GSAS can reduce more switching activities than MSAS and it can save more time than MSAS. The phase two of GSAS can reduce the switching activities further by re-assigning the registers. The experimental results show effectiveness of the method.

Acknowledgements

I would like to express my sincere thanks to my advisor, Prof. Cheng Chen, for his supervision and advice. Without his guidance and encouragement, I could not finish this thesis. I also thank Prof. Jyj-Jiun Shann and Dr. Guan-Joe Lai for their valuable suggestions.

There are many others whom I wish to thank. My thanks to Yi-Hsuan Lee for her kindly advice suggestion. Wen-Pin Liu, Chia-Chun Lee, Che-Yin Liao and Ming-Xian Cai are delightful fellowa, I felt happy and relaxed because of your presence.

Finally, I am grateful to my dearest family. They accompany me all the time.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

A VLIW processor has multiple functional units. It can process several instructions simultaneously and is widely adapted in DSP processors[8-9]. In embedded systems, high performance digital signal processing (DSP) used in image processing, multimedia, wireless security, etc., needs to be processed with high data throughput[10]. Moreover, most embedded systems, such as digital camera, cellular phone, PDA, etc., get the power from batteries and are used widely in the world. Therefore, it becomes an important problem to reduce the power consumption in embedded systems to lengthen the time of using the portable devices[10-14][19].

We focus on reducing the power consumption of application on VLIW architectures by reducing transition activities on the instruction bus. Due to large capacitance and high transition activities, buses consume a significant fraction of total power dissipation in a processor [22]. Recent research for various processors [1-2] shows that the instruction sequence of an application plays an important role in its energy consumption. Thus, new research directions in power optimization have begun to address the issues of instruction-level scheduling for reducing energy consumption [10-16]. MSAS was an instruction-level scheduling algorithm and was designed to minimize switching activities as much as possible[12]. It assigns instructions by using the min-cost maximum bipartite matching. However, it takes too much time to find minimum weight maximum bipartite matching and may not find the minimal switching activities.

Hence, in this thesis, we propose a method named *Greedy Switching Activities Scheduling* (GSAS) which is an efficient instruction-level scheduling algorithm for VLIW DSP with lower power consumption in instruction bus. It contains two phases. The phase one uses a greedy method to choose the minimal switching activities and schedule the instruction.

That is, we only schedule one instruction causing minimal switching activities in one iteration. The phase two re-assigns the registers used by the instructions to reduce the switching activities. It also re-assigns the registers in a greedy way which chooses the register causing the minimal switching activities. Compared with MSAS, GSAS has a better performance in reducing power when switching activities plays an important role in energy consumption. Moreover, GSAS saves more time in comparing with MSAS.

This thesis is organized as follows. In chapter 2, we will introduce the fundamental background, the related work and the motivation of our method. In chapter 3, the proposed machine architecture of our experiments will be introduced and then GSAS will be presented in detail. The experimental results will be presented in the chapter 4. Finally, we will conclude our thesis in chapter 5, and list the future work of our research.

Chapter 2. Fundamental Background and Related Work

In this chapter, we will introduce the fundamental background of the problem and some basic definitions. Then the power cost function will be presented. Third, we will introduce some related work, including MSAS (Minimizing Switching Activities Scheduling), Horizontal and Vertical Scheduling, PRRS (Power Reduction Rotation Scheduling), and SAMLS (Switching-Activity Minimization Loop Scheduling). Finally, we will give a briefing of our motivation.

2.1 Fundamental Background[8-10]

A long instruction is a very-long instruction word executed by a VLIW CPU during each clock cycle. Sub-instructions are several parallel instructions composing long instruction. Fig. 2.1 shows that what is a long instruction and what is a sub-instruction. In a TI TMS320C6000 CPU, it can execute up to eight 32-bit instructions per cycle since the C6000 core CPU consists of eight functional units[8]. In Fig. 2.1, from instruction A to instruction H, each one is a 32-bit instruction that can be executed simultaneously. Hence, there are eight sub-instructions in Fig. 2.1, and eight sub-instructions compose one long instruction.

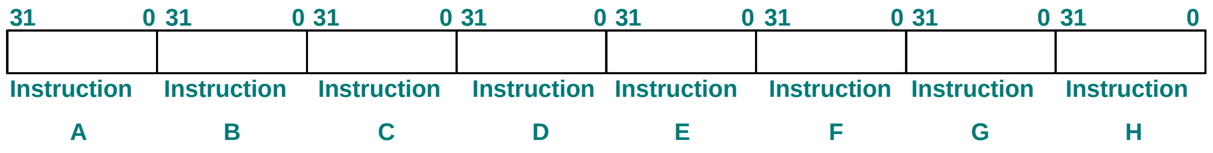
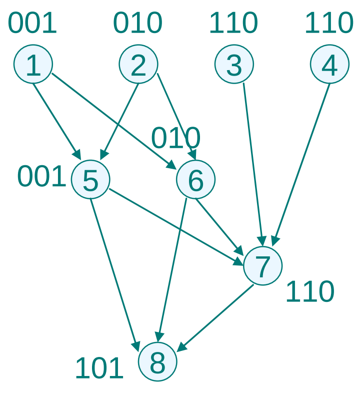


Fig. 2.1 A VLIW (very long instruction word) in TI TMS320C6000 CPU



(a)

Step	FU1 (000)	FU2 (000)
1	1 (001)	2 (010)
2	3 (110)	4 (110)
3	5 (001)	6 (010)
4	7 (110)	
5	8 (101)	

(b)

Fig 2.2(a) A given DAG ; (b) A schedule of left DAG

After knowing what a sub-instruction is, we will use it in the following definition, which is an important input of our algorithm.

Definition 2.1. A Directed Acyclic Graph (DAG) *G*=(*V*, *E*, *Bit_String*(*u*)) is a node-weighted graph, where *V* is the set of nodes and each node represents a sub-instruction, and *E* is the edge set and an edge between two nodes denotes a dependency relation, and *Bit_String*(*u*) is a function to represents the machine code for each node $u \in V$.

A DAG is an input of our algorithm. Fig 2.2(a) is an example of DAG. From node 1 to node 8, each node is a sub-instruction and belongs to V. Each edge between two nodes denotes the dependency relation. For example, node 6 could not be executed before node 1 and node 2 are executed. The binary strings beside the nodes are their machine code. Hence, *Bit_String(*node 5*)* is 001 and *Bit_String(*node 7*)* is 110.

Definition 2.2. The location of a node in a schedule of a DAG is a two dimensions array. Each element of this array is a sub-instruction.

Fig 2.2(b) is a schedule of Fig 2.2(a). We use (*i*, *j*) to denote the location of a sub-instruction in a schedule, where *i* is the row and *j* is the column. The row number represents the clock cycle in which the sub-instruction is executed. The column number represents the functional unit on which the sub-instruction is executed. For example the location of node 6 is (3, 2). On the other hand, we can use (3, 2) to represents node 6.

Next, we will introduce what is switching activities. Basically, it is one of the most important factors of calculating the consumption of power[10]. Switching activities is the Hamming Distance of two consecutive instructions. Hamming Distance is the number of bit differences between two binary strings.

Instruction A	1001 0010 1010 0111 1010 1101 0001 0000
Instruction B	1100 1011 0001 1101 0000 1011 0101 0100
Different Bit	0101 1001 1011 1010 1010 0110 0100 0100

Switching activities = Hamming Distance =15

Fig 2.3 An example of how to calculate switching activities

Fig 2.3 shows that how we calculate the switching activities between two instructions. We suppose that instruction A and instruction B are two contiguous sub-instructions on the same functional unit and the binary strings are their machine code. We can observe that the number of different bit of two binary strings is 15, so the switching activities of two instructions is 15.

2.2 Power Cost Function

In this section, we will introduce the energy consumption equation based on the energy model proposed by [10]. We will use this energy model to evaluate our method.

As we mention before, a VLIW processor executes a long instruction during each clock cycle. A long instruction is composed of several parallel sub-instructions. The power onsumption to execute a long instruction during a clock cycle, P_{cycle}, can be computed by :

$$P_{cycle} = P_{base} + \sum_{Inst_i} \{ P_{Inst_i} + SP(i, j)\} \qquad (1)$$

where P_{base} is the base power needed to support a long instruction execution even when a long instruction contains only NOPs. P_{inst_i} is the power to execute a sub-instruction I_i on a functional unit, and SP(i, j) is the switching power caused by switching activities between $Inst_i$ (current sub-instruction) and $Inst_j$(last sub-instruction) executed on the same functional unit.

The switching power is proportional to the number of transitions. So

$$SP(i, j) = \alpha \cdot WHD(Bit\_String(Inst\_i), Bit\_String(Inst\_j)) \qquad (2)$$

where α is a power coefficient representing the consumed power per transition, and WHD (Weighted Hamming Distance) is a function used to compute the number of transitions between $Inst_i$ and $Inst_j$. Let X = Bit_String($Inst_i$) and Y = Bit_String($Inst_j$), and WHD(X, Y) is :

$$WHD(X, Y) = \sum_{i=1}^{K} w_i \cdot (X[i] \oplus Y[j]) \qquad (3)$$

Where w_i is the weight of a transition. w_i is used to denote the weight for power consumption caused by one transition on different units.

The energy consumption of a program is the summation of all its power consumption during each clock cycle. Let S be a schedule for an application and L the schedule length of S. Then the energy consumption of schedule S, Es, can be computed by

$$Es = \sum_{K=1}^{L} P^{(K)}_{cycle} = L * P_{base} + \sum_{K=1}^{L} \sum_{Inst_i^{(K)}} P_{Insti} + \sum_{K=1}^{L} \sum_{Inst_i^{(K)}} SP^{(K)}(i, j)\} \qquad (4)$$

$\sum\sum P$ is the summation of basic power consumptions for all sub-instructions of an application that does change with different schedules. P_{base} is a constant and $P_{base} * L$ varies with schedule length for specific VLIW processor. $\sum\sum SP(i,j)$ is the switching power and changes with different schedules. Therefore, schedule length and switching activities need to be considered together in instruction-level scheduling techniques in order to minimize energy consumption of a program.

2.3 Related Work[11-13][19]

In this section, we'll show how Minimizing Switching Activities Scheduling (MSAS) works, and then Horizontal and Vertical Scheduling will also be introduced. Finally, we will introduce Power Reduction Rotation Scheduling (PRRS) and Switching-Activity Minimization Loop Scheduling (SAMLS) .

2.3.1 Minimizing Switching Activities Scheduling[12]

The algorithm, Minimizing Switching Activities Scheduling (MSAS), was designed to solve a special case of instruction-level energy-minimization scheduling, i.e., the case when switching activities play the most important role in energy consumption.

When P_{base} is very small compare with α (in equation 2 in section 2.2), the energy of a schedule depends mainly on switching activities. For example, when P_{base} equals 0.1 and α equals 1, then we need to reduce 10 control steps in schedule length to count one bit switch. Thus, the MSAS algorithm was designed to minimize switching activities as much as possible.

On the other side, considering the performance, the algorithm also wants to minimize schedule length. Hence, MSAS algorithm minimizes switching activities in first priority and still considers schedule length. Since most previous work focus on one functional unit, the algorithm takes advantage of multiple functional units under VLIW architectures.

In this algorithm, the input is the DAG we have defined it in previous section and the output is a schedule with switching activities minimization. Due to the existence of the dependency in DAG, we can only schedule a node after all its parent nodes have been scheduled. The scheduling problem with switching activities minimization is how to find a matching between functional units and ready nodes in such a way that it can minimize the total switching activities in every scheduling step. This is equivalent to the min-cost weighted bipartite matching problem. Thus, in the first scheduling step of MSAS algorithm, it creates a weighted bipartite graph G_{BM}, where the vertices of one side are the set of the functional units and the vertices of the other side are the nodes in ready list and the weight of the edge between the node and the functional unit is the switching activities when the node is assigned to the functional unit. Then it assigns nodes based the min-cost maximum bipartite matching. After assigning the nodes, it updates the nodes in the ready list according to the DAG and the machine codes of the functional units. Then it repeatedly creates the weighted bipartite graph and assigns the nodes until all the nodes are scheduled.

The schedule created by MSAS can reduce total switching activities since it considers the min-cost maximum bipartite matching in each step. It is know that finding a min-cost maximum bipartite matching take $O(n^3)$ by the Hungarian Method[20]. Let N be the number of functional unit. In every scheduling step, it needs at most $O((N+|V|)^3)$ to find minimum weight maximum bipartite matching using Hungarian Method and the scheduling step is at most $|V|$. Thus, the complexity of MSAS is $O(|V|*(N+|V|)^3)$. It takes too much time to find minimum weight maximum bipartite matching and it may not find the minimal switching activities in some cases. We will show you in our motivation.

2.3.2 Horizontal Scheduling and Vertical Scheduling[13]

Both high performance and low power are two important objectives of complier optimization. Thus, in [13], the authors propose a two-phase instruction scheduling approach.

In the first, instructions are scheduled by list schedule for performance. Then, in the second phase, horizontal and vertical scheduling methods are employed to re-arrange the codes reducing the power without incurring performance penalty.

We first introduce the horizontal scheduling algorithm which re-schedules the instruction components of a long instruction to minimize switching activities of instruction bus. Suppose we have n VLIW instructions which have been scheduled by list schedule, then the horizontal scheduling won't change the control step of each long instruction and the component of each long instruction, but it will try to re-arrange the position of each sub-instruction of a long instruction to reduce the switching activities. The way of re-schedule the position of sub-instructions of a long instruction is to create the weighted bipartite graph G_{BM} between the long instruction which is re-scheduled and the long-instruction is considered to re-schedule right now. In G_{BM}, the vertices of two sides are the sub-instructions of two long instructions and the weight of the edge is the switching activities between two sub-instructions. Like as MSAS, the horizontal scheduling finds the min-cost maximum bipartite matching and re-arranges the position of each sub-instruction according to the min-cost maximum bipartite matching. For example, in figure 2.4, U1 to U4 are the sub-instructions in the last long instruction already scheduled and L1 to L4 are the sub-instructions of a long instruction to be re-scheduled. Thus, it creates bipartite matching between them and finds the min-cost maximal matching. Then it re-schedules the positions of L1 to L4 according to the matching.

This algorithm repeatedly creates weighted bipartite graph and re-arranges the positions of the sub-instructions of each long instruction from the first long instruction to the last long instruction in the schedule created by list schedule. After re-scheduling all the long instructions, we can get a schedule which has less total switching activities.

Next, we will introduce the vertical scheduling. The vertical scheduling is similar with horizontal scheduling, but it allows sub-instructions to move across long instructions.

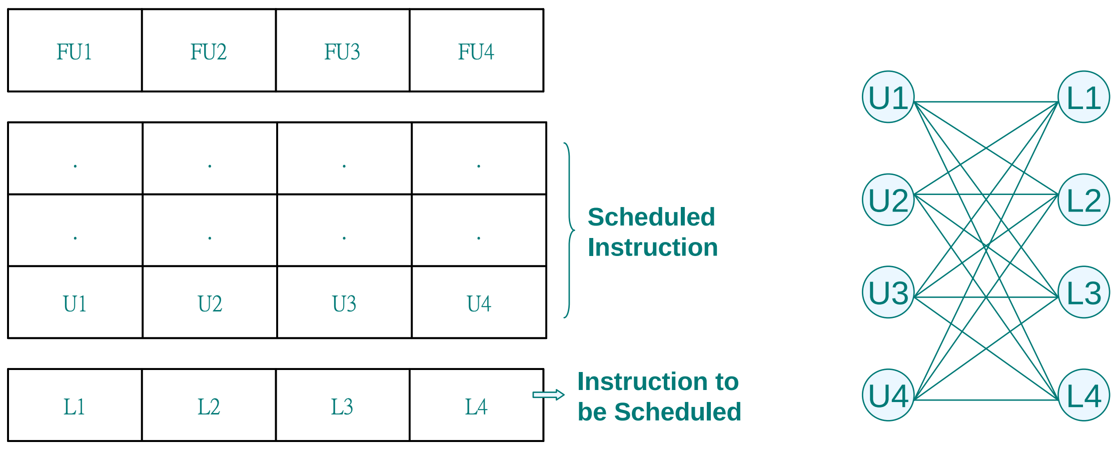


Fig. 2.4 An example of bipartite matching for horizontal scheduling

That is, the horizontal scheduling won’t change each sub-instruction’s control step but the vertical scheduling. How can vertical scheduling do this? Because it uses a window size *w* to decide the weighted bipartite graph between the sub-instructions in the last long instruction already scheduled and the sub-instructions in the next *w* long instructions that satisfy data dependence constraint. We can say that horizontal scheduling is a special case of vertical scheduling when the window size $w = 1$. Like as the horizontal scheduling, the vertical scheduling finds the min-cost maximum bipartite matching and re-arranges the position of each sub-instruction according to the min-cost maximum bipartite matching repeatedly until all sub-instructions are scheduled.

The two algorithms present the essential idea of their low power optimization. It requires the functional units of target VLIW architectures to be identical. Thus, they can only perform sub-instructions swapping with identical functional units on target host without performance penalty. However, the functional units are normally classified into several classes in most of VLIW architecture designs. The swapping can only be done with functional units of the same class. This is the main constraint of their method.

2.3.3 Power Reduction Rotation Scheduling[11]

The algorithm, Power Reduction Rotation Scheduling, was designed to minimize both switching activities and scheduling length for loop applications and is based on rotation scheduling[3].

Rotation Scheduling presented in[3] is a scheduling technique used to optimize a loop schedule with resource constraints. The main goal of rotation scheduling is to reduce the schedule length of a loop application. It transforms a schedule to more compact one iteratively. *Retiming*[21] can be used to break the intra-dependence between instructions in a loop application, so that the rotation can be done to reduce the schedule length of a loop application. In each step of rotation, nodes in the first row of the schedule are rotation down. By doing so, the nodes in the first row are re-scheduled to the earliest possible available locations. From retiming point of view, each node gets retimed once by drawing one delay from each of incoming edges of the node and adding one delay to each of its outgoing edges in the DFG. The new location of the node in the schedule must also obey the precedence relation in the new retimed graph.

The Power Reduction Rotation Scheduling is totally based on rotation scheduling. In addition, each node needed to be rotated must to be scheduled on the location with minimum switching activities. So it can achieve the goal to reduce power consumption. The disadvantage of PRRS is that it was designed for the loop application and it can not be use in none loop application. It needs an initial schedule to be its input and it takes extra time to create the initial schedule.

2.3.4 Switching-Activity Minimization Loop Scheduling[19]

Switching-Activities Minimization Loop Scheduling is an improving algorithm of PRRS which was developed to reduce both schedule length and switching activities of a loop application.

Switching-Activity Minimization Loop Scheduling (SAMLS) was based on rotation scheduling and bipartite matching. In the first phase of SAMLS, it performed the same thing as PRRS did. Then the schedule created by phase one will be the input of the phase two of SAMLS. In the phase two of SAMLS, it performed the same thing as horizontal scheduling did.

As the results of experiments, SAMLS has a little better performance than that of PRRS in reducing switching activities, but SAMLS takes much more time than that of PRRS.

2.4 Motivation

From the relative work, we can observe that reducing the switching activities is an important factor of reducing total power consumption, especially when switching activities play the most important role in energy consumption. Hence, we focus our working on reducing switching activities as much as possible, despite it may take more control steps to complete the application.

In section 2.3.1 and section 2.3.2, we can see that both algorithms create weighted bipartite graph. Then MSAS finds the min-cost maximum bipartite matching and horizontal scheduling finds the maximum bipartite weighted matching. Both of them try to find the minimal switching activities in each allocation. We can observe that it may not find the minimal switching activities. For example, figure 2.5(a) is a simple example of DAG., and figure 2.5(b) is the schedule of figure 2.5(a) by using MSAS. Assuming the binary strings in Figure 2.5(c) are the machine codes of these instructions. Then the switching activities is 9. But in figure 2.5(d) and figure 2.5(e), we can find that another schedule will cost less switching activities. We can observe that instruction A and instruction B have the same

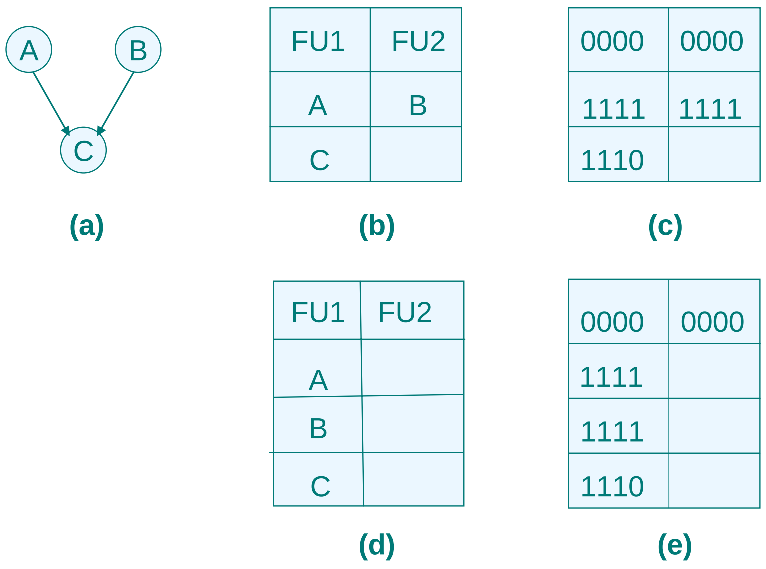


Fig 2.5 (a) A DAG ; (b) the schedule of (a) with MSAS ; (c) switching activities = 9 (d) other schedule of (a) ; (e) switching activities = 5

machine code in this case. If we use MSAS to schedule it, we lost the benefit of scheduling B instruction after A instruction. In addition, in actual VLIW architectures, different functional units perform different functions. For example, a load instruction can be executed in specific unit and so is a multiply instruction. Hence, it is not a good way to find min-cost maximum bipartite matching in an architecture while the functional units are not identical. Therefore we try to find a better way to schedule the applications to reduce power consumption in real VLIW architectures.

Another important motivation is that how to assign registers to reduce switching activities in MSAS or other algorithm is not considered. We will also try to find a better way to assign registers to achieve further reducing the switching activities. Moreover, finding a min-cost maximum bipartite matching takes too much time. The complexity of MSAS is $O(|V|*(N+|V|)^3)$, where N is the number of functional units and V is the number of nodes. Thus, we will also consider of reducing the complexity of our scheduling algorithm.

In the next chapter, we will describe the basic concepts and principles of our method in some detailed.

Chapter 3. Greedy Switching Activities Scheduling

In this chapter, we will first introduce our experimental machine architecture. Then our proposed method, GSAS (Greedy Switching Activities Scheduling), will be presented. We will use an example to explain our GSAS in some detail. We will also analyze our algorithm compared with MSAS[12].

3.1 Machine Architecture

The architecture of our experimental VLIW processor is based on TI TMS320C6K processor, since it is commonly used in image processing, multimedia, wireless security, etc[8-9]. There are multiple functional units executed simultaneously in these architectures, power consumption becomes one of the most important issues to be considered with the concern of performance. Thus, we choose TI TMS320C6K processor to be the base of our experimental model. We will first introduce the architecture of TI TMS320C6K processor. Then we will introduce our proposed machine for our experiments.

3.1.1 Architecture of TM320C6K[8-9]

The architecture of TMS320C62X/C64X/C67X processor is shown in Figure 3.1. The CPU contains program fetch unit, instruction dispatch unit, advanced instruction packing(C64 only), instruction decode unit, two data paths and each with four functional units, 32 32-bit registers or 64 32-bit registers (C64 only), control registers, ontrol logic, test, emulation and interrupt logic.

The program fetch, instruction dispatch, and instruction decode units deliver up to eight 32-bit instructions to the functional units every CPU clock. The processing of instructions

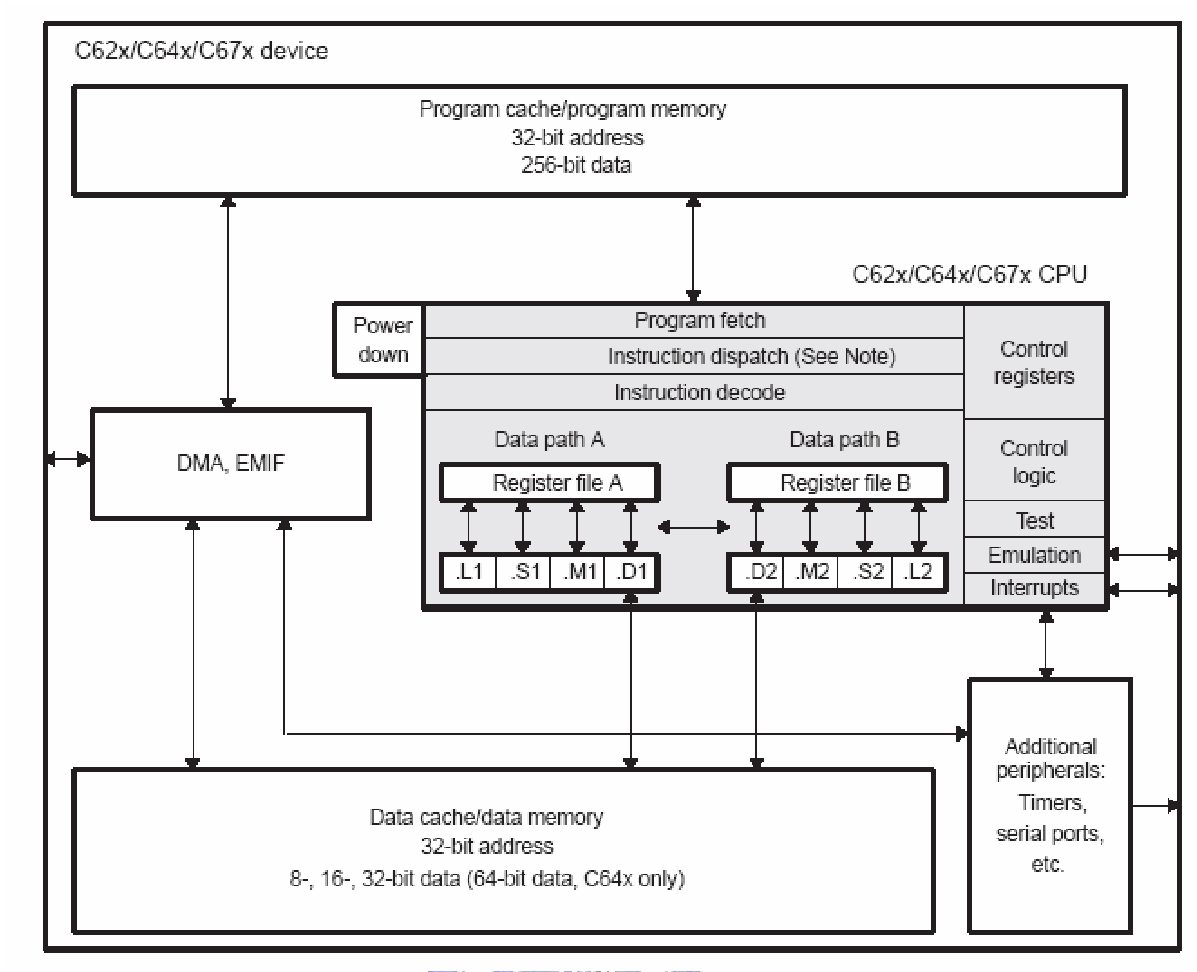
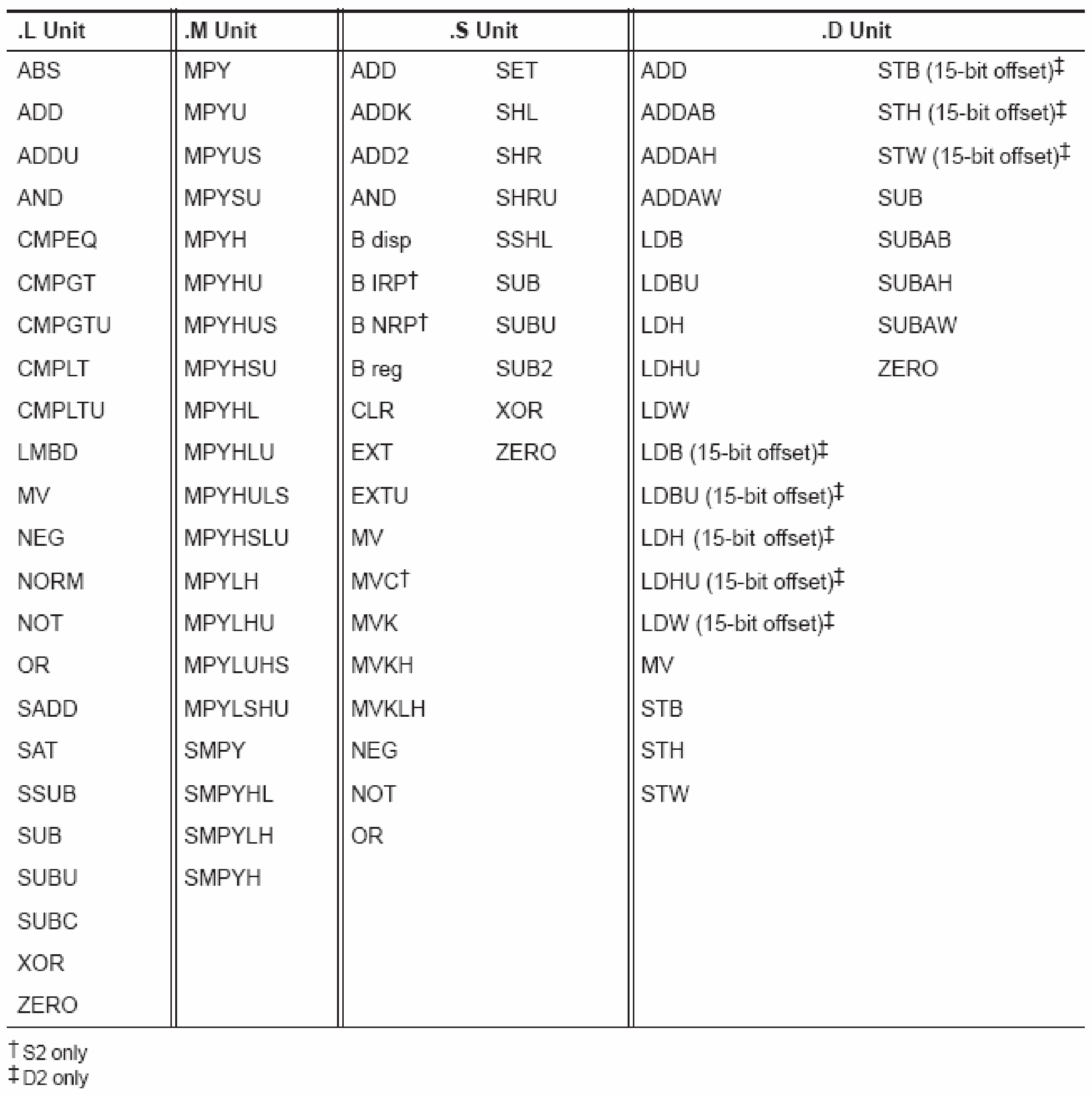


Fig 3.1 TMS320C62X/C64X/C67X Block Diagram

occurs in each of the two data paths (A and B), each of which contains four functional units and 16 32-bit general-purpose registers for the C62X/C67X and 32 32-bit general-purpose registers for C64X.

As for functional units, there are eight functional units in the architecture. The eight functional units in the C6000 data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The functional units are named .L, .M, .S, and .D. Figure 3.2 shows the mapping between instructions and functional units.



.L Unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset)‡
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset)‡
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset)‡
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SSHL	LDB	SUBAB
CMPGT	MPYHU	B IRP†	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP†	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB (15-bit offset)‡	
MV	MPYHULS	EXTU		LDBU (15-bit offset)‡	
NEG	MPYHSLU	MV		LDH (15-bit offset)‡	
NORM	MPYLH	MVC†		LDHU (15-bit offset)‡	
NOT	MPYLHU	MVK		LDW (15-bit offset)‡	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

† S2 only
‡ D2 only

Fig 3.2 Instruction to functional unit mapping

In figure 3.1, we can see that there are two general-purpose register files (A and B) in the C6000^{TM} date paths. For the C62X/C67X DSPs, each of these files contains 16 32-bit registers (A0-A15 for file A and B0-B15 for the file B). The general-purpose registers can be used for data, data address pointers, or condition registers. The C64X DSP register file doubles the number of general-purpose registers that are in the C62X/C67X cores, with 32 32-bit registers (A0-A31 for file and B0-B31 for file B).

Each functional unit read directly from and writes directly to the register file within its

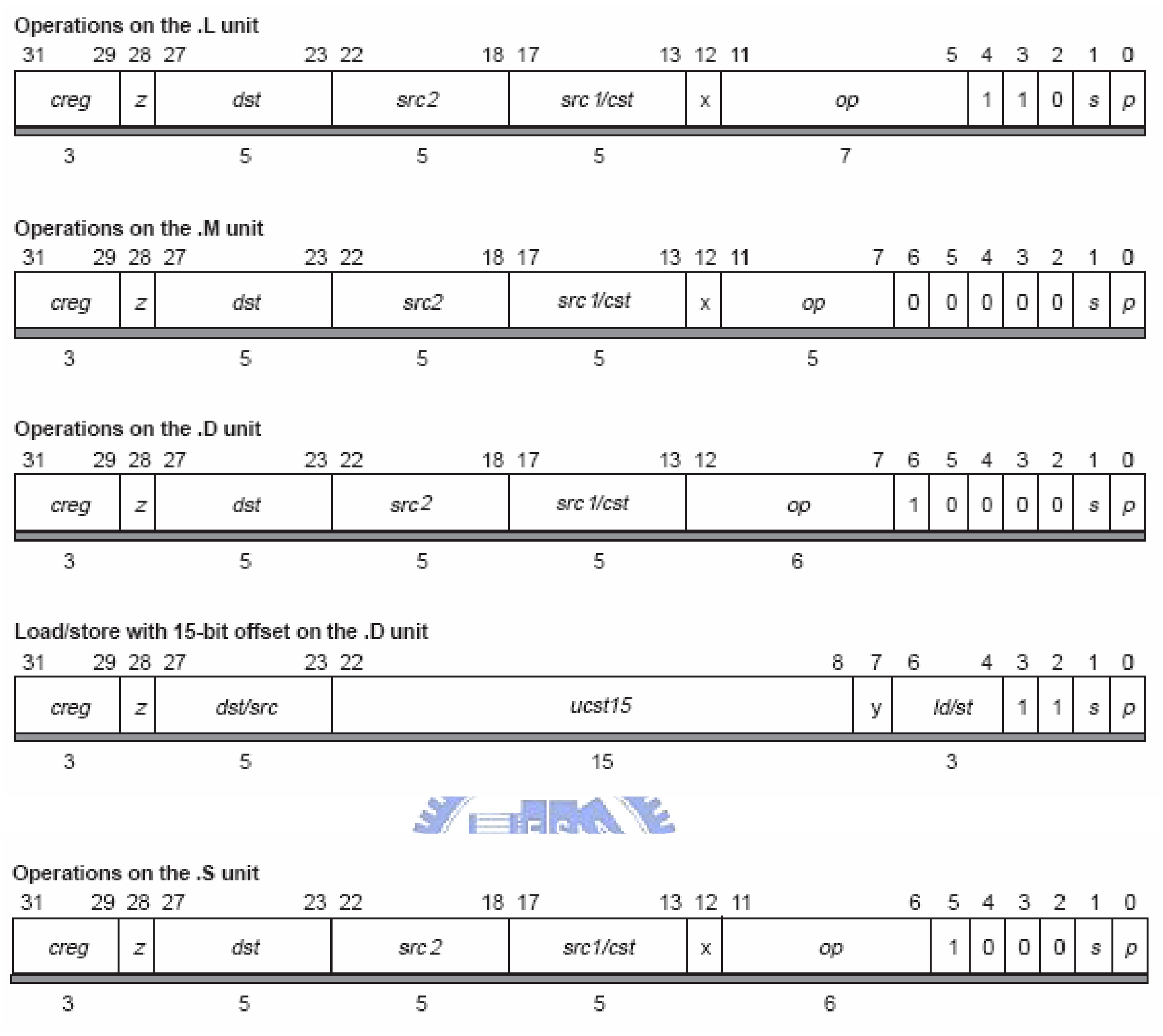

Operations on the .L unit
31 29 28 27 23 22 18 17 13 12 11 5 4 3 2 1 0
creg z dst src2 src1/cst x op 1 1 0 s p
3 5 5 5 7
Operations on the .M unit
31 29 28 27 23 22 18 17 13 12 11 7 6 5 4 3 2 1 0
creg z dst src2 src1/cst x op 0 0 0 0 0 s p
3 5 5 5 5
Operations on the .D unit
31 29 28 27 23 22 18 17 13 12 7 6 5 4 3 2 1 0
creg z dst src2 src1/cst op 1 0 0 0 0 s p
3 5 5 5 6
Load/store with 15-bit offset on the .D unit
31 29 28 27 23 22 8 7 6 4 3 2 1 0
creg z dst/src ucst15 y ld/st 1 1 s p
3 5 15 3
Operations on the .S unit
31 29 28 27 23 22 18 17 13 12 11 6 5 4 3 2 1 0
creg z dst src2 src1/cst x op 1 0 0 0 s p
3 5 5 5 6

Fig. 3.3 TMS320C62X/C64X/C67X Opcode Map

own data path. That is, the .L1, .S1, .D1, and .M1 units write to register file A and the .L2, .S2, .D2, and .M2 units write to register file B.

3.1.2 Proposed Machine Architecture

In our experimental machine architecture, we use a simplified model based TMS320C6000 described in previous section. We used the same functional units and data

Functional Unit	.L1 .L2	.S1 .S2	.M1 .M2	.D1 .D2
Operation	Addition Subtraction	Addition Subtraction	Multiplication	Load / Store Addition Subtraction

Table 3.1 Functional units of proposed machine architecture

	Machine Code Field			
Load	op(10000)	memory address (10bit)		dst (5 bit)
Store	op(10100)	memory address (10bit)		src (5 bit)
Addition	op(01000)	src1 (5 bit)	src2 (5 bit)	dst (5 bit)
Subtraction	op(01100)	src1 (5 bit)	src2 (5 bit)	dst (5 bit)
multiplication	op(00000)	src1 (5 bit)	src2 (5 bit)	dst (5 bit)

Table 3.2 Machine code of each instruction

paths, but with the simplified instructions. Based on the opcode of the instruction set[8] in the Fig 3.3, we can find that the most important factors affecting the machine code are the destination, source1, source2, and operate fields. The symbol *creg* and *z* represent the conditional registers. In most case, the conditional registers are not used. The symbol *s* means select side A or B for destination. In the counting of switching activities, it won't cause switching on the same side. The symbol *p* represents the parallel execution. We can't decide its' value until the schedule have been done. Hence, we delete those fields and preserve the fields of destination, source1, source2, and operate.

In most DPS benchmarks, the additions and multiplications are executed frequently. Hence, we preserve the instructions of addition, multiplication, subtraction, load, and store in our experiments. Table 3.1 shows the mapping between instructions and functional units. Table 3.2 shows the machine codes of each instruction. The machine codes of all instructions in our proposed architecture are 20 bits binary strings. The symbol *src* represents the source and *dst* represents the destination and they are 5 bits binary strings. The registers can be used in the fields of *src* or *dst*. In our proposed architecture, 32 registers (A0-A15 for file A and B0-B15 for the file B) can be used. It is the same as C62X/C67X processors. The machine codes of register file A are from 00000 to 01111, and the machine codes of register file B are from 10000 to 111111. The machine codes of memory address are from 0000000000 to 1111111111.

3.2 Greedy Switching Activities Scheduling

In this section, we will introduce our algorithm, GSAS (Greedy Switching Activities Scheduling), which is an effective method to reduce the switching activities in scheduling applications. GSAS consists of two phases. The first phase is to arrange the schedule. In other word, the first phase will decide what location and cycle the sub-instructions should be placed. After deciding the schedule, the second phase can be executed. The goal of second phase is to re-assign the registers to reduce the switching activities.

3.2.1 Phase One of GSAS

In this section, we will introduce phase one of our algorithm, GSAS (Greedy Switching Activities Scheduling). The word "greedy" is the main principle of our method. As we mention in the section 2.4, the drawback of MSAS is that it may neglect some situation which will cause less switching activities. Moreover, it takes much time to find a min-cost maximum weight bipartite matching. Hence, we use a greedy method to choose the sub-instruction which will cause the minimal switching activities.

The figure 3.4 shows the phase one of GSAS. The input of the algorithm is a DAG and the functional unit set. The output of the algorithm is a schedule with minimal switching

```
Input : DAG G = (V, E , Bit_String(u) ), FU_SET
Output : A schedule with minimal switching activities

1.  Height ← the number of nodes of the longest path in G ;
2.  for i = 1 to Height do
3.      $L_{RD}$ ← All nodes in Level(i) ;
4.      while $L_{RD}$ != $\phi$ do
            /*  when the nodes in ready list are not empty, construct the bipartite matching
                between the nodes and functional units   */
5.          Construct $G_{Bm}$ = ($V_{Bm}$ , E, W) ;
6.          Find E( Fi , u) in $G_{Bm}$ that has the minimal weight among all edge in $G_{Bm}$ ;
7.          Schedule Node u to functional unit Fi ;
            /*  update the content of instruction bus and the ready list   */
8.          Bit_String(Fi) ← Bit_String(u) ;
9.          Remove u from $L_{RD}$ ;
10.     end while
11. end for
```

Fig. 3.4 The algorithm of phase one of GSAS

activities. We will introduce it in the following.

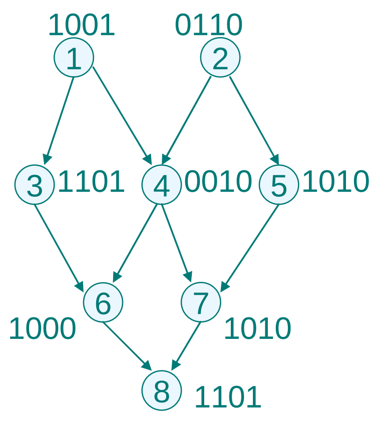
We first define some symbols in our algorithm. We use Level(i) to represent the set of nodes, where i represents the critical path of the nodes according to data dependency. For example, in Figure 3.5 (a), node 1 to 2 belong to Level(1) since these nodes have no parent. Node 3, Node 4, and Node 5 belong to Level(2) since these nodes couldn't be executed until some nodes in Level(1)have been executed. Hence, node 6 and node 7 are in Level(3) and so on. Height represents the number of the nodes of the critical path of the DAG. For example, in Figure 3.5 (a), the Height is 4. L_{RD} represents the nodes ready to be scheduled.

As we mention in our motivation, we focus our working on reducing switching activities as much as possible, and it may take more control steps to complete the application. We schedule the DAG level by level. It can assure that the nodes in high level will be scheduled first and avoid creating schedule with too long length.

In this algorithm, we schedule the nodes level by level. In line 3, we first assign the nodes in Level(1) to the ready list. When the ready list is not empty, we construct the weighted bipartite matching $G_{Bm} = (V_{BM}, E, W)$, where $V_{BM} = FU\_SET \cup L_{RD}$ where $FU\_SET = < F_1, F_2,\ldots,F_N>$ is the set of all functional units and L_{RD} is the set of all nodes in ready list . For each functional unit $F_i \in FU\_SET$ and each node $u \in L_{RD}$, an edge $e(F_i, u)$ is added into E and the weight of $e(F_i, u)$, $W(F_i, u) = WHD(\beta, Bit\_String(u), Bit\_String(F_i))$ and $Bit\_String(F_i) = Bit\_String(v)$ where v is the last node executed on F_i. $WHD(\beta, X, Y)$ is the weighted hamming distance function and is the same as $WHD(X, Y)$ in equation 3 besides a new parameter β. We use the parameter β represents the first β bits in the binary string. In the $WHD(\beta, X, Y)$, we only calculate the hamming distance of the first β bits between two binary strings. We do this because we will re-assign the register used by the destination of the sub-instructions in phase two. Therefore, we only consider the fields of *op*, *src1*, *src2*, and *memory address* of the machine codes of the sub-instructions in Table 3.2 and the value of β will be 15. It will increase the probability of operand sharing. The value of β will be 20 if we only use phase one and do not re-assign the registers. We will use an example to demonstrate our method and show the benefit of our algorithm.

After constructing the G_{BM}, we use a greedy method to choose the minimal weight edge and schedule the node. After scheduling the node, then we update the nodes in the ready list and the content of the instruction bus. We repeatedly choose the node from the ready list until all the nodes in the ready list are scheduled. When the ready list is empty, we assign the nodes in the next level to the ready list and repeat the step we did it before until all nodes in the DAG are scheduled.

Let's use a simple example to demonstrate our method. Since the machine codes of the sub-instructions are 4 bits, we calculate all the bits in hamming distance and β is 4 in



(a)

Step	FU1 (0000)	FU2 (0000)
1	1 (1001)	2 (0110)
2	3 (1101)	4 (0010)
3	5 (1010)	6 (1000)
4	7 (1010)	
5	8 (1101)	

(b)

Step	FU1 (0000)	FU2 (0000)
1	1 (1001)	2 (0110)
2	3 (1101)	4 (0010)
3	6 (1000)	5 (1010)
4		7 (1010)
5	8 (1101)	

(c)

Step	FU1 (0000)	FU2 (0000)
1	1 (1001)	2 (0110)
2	3 (1101)	4 (0010)
3		5 (1010)
4		7 (1010)
5		6 (1000)
6	8 (1101)	

(d)

Fig. 3.5 (a) A DAG ; (b) list schedule with total SA = 14 , SL = 5 ; (c) MSAS with total SA = 11 , SL = 5 ; (d) GSAS with total SA = 8 , SL = 6

WHD(β, X, Y). Figure 3.5(a) is a DAG. According to the algorithm of the phase one of GSAS, node 1 and node 2 will be in the L_{RD} first since they are in Level(1). Then we construct the edges between the ready nodes and the functional units and weight of the edges are both 2. We schedule node 1 on FU1 first and there is only node 2 in $L_{RD.}$ Hence, we construct the edges between node 2 and functional units. The weight of the edge connected to FU1 is 4 and the weight of the edge connected to FU2 is 2. Thus, we schedule node 2 on FU2. By the same way, we can schedule the nodes in Level(2) and so on until all the nodes are scheduled. Figure

3.5(b)~(d) are the schedules of different algorithms. In those figures SA stands for switching activities and SL stands for schedule length. From the result, we can see that phase one of GSAS has the better performance in reducing switching activities.

Let’s compare the schedule of MSAS with the schedule of phase one of GSAS. We can see that both algorithms have the same schedule in the first two control steps. In the control step 3, MSAS has to find the min-cost maximal weight bipartite matching, so it has to schedule the node 6 at (3, 1) and schedule the node 5 at (3, 2). Hence, in the control 4, MSAS only can schedule the node 7. But when we use GSAS to schedule the nodes, we can only schedule node 5 at (3, 2) in the control step 3 since it causes the minimal switching activities. Hence, in the control step 4, GSAS can choose the best option from node 6 and node 7 which can be scheduled after node 3 or node 5. Obviously, scheduling node 7 at (4, 2) causes the least switching activities. Then, in the control step 5, scheduling node 6 at (5, 2) causes the least switching activities and is better than that of scheduling it at (3, 1). Therefore, GSAS can avoid the disadvantageous arrangement caused by schedule more than one node in one iteration.

Moreover, the complexity of MSAS is $O(|V|*(N+|V|)^3)$, where |V| is the number of the sub-instructions and N is the number of functional units. But the complexity of phase one of GSAS is $(|V|*(|V|*N))$, since instead of finding the min-cost maximal weight bipartite matching, we just finds the only one node in one iteration which needs at most $O(|V|*N)$ to be completed. Hence, the phase one of GSAS saves more time in comparing with MSAS.

```
Input : DAG G = (V, E , Bit_String(u)), schedule S
Output : A new DAG G' =(V, E, Bit_String'(u))

1.   all src and dst in V ← NULL;
2.   Height ← the number of nodes of the longest path in G ;
3.   for i = 1 to Height do
4.       $L_{RA}$ ← all the nodes in Level(i) ;
5.       while $L_{RA}$ != $\phi$ do
6.           u ← the node has minimal time(v) for each v∈ $L_{RA}$;
7.           unlock(all registers);
            /*  lock all register if assigning it to the u.src will cause data hazard  */
8.           lock(v.src) for each v∈ V and time(v) > time (u) ;
9.           find the register $A_i$ will cause the minimal switching activities
10.          when assigning it at dst of u
                 /*  from all the registers which are not locked */
11.          u.dst ← $A_i$ ;
12.          re-assign src of all children of u ;
13.      end while
14.  end for
```

Fig. 3.6 The algorithm of phase two of GSAS

3.2.2 Re-assign the Registers by the Phase Two of GSAS

In this section, we will introduce the phase two of GSAS. The main goal of phase two is to re-assign the registers used by the *src* and *dst* in each sub-instruction to reduce the switching activities of the schedule created by phase one of GSAS. The meaning of symbol *src* and *dst* have been introduced in section 3.1.2.

In previous work of[11-13], they didn't consider about how to assign the registers to reduce the switching activities. Actually, the register assignment is an important factor affecting the component of the machine codes of sub-instructions. Thus, we use the phase two of GSAS to re-assign the registers to reduce the switching activities.

Figure 3.6 shows the phase two of GSAS. The input of the algorithm is the DAG and the schedule produced by the phase one of GSAS and the output of the algorithm is the DAG with different registers used by each sub-instruction. The DAG with new machine codes reduces the switching activities of the schedule produced by the phase one of GSAS.

We first define some symbols in our algorithm. In the algorithm, the meanings of Level(i) and Height are the same as that of the phase one. L_{RA} represents the list of nodes to which the registers are re-assigned. u.*dst* represents the register of destination of u and u.*src* represents the register of source of u. We use two functions lock() and unlock() to decide the state of the registers. If the state of the register is locked, then it couldn't be used; otherwise it can be re-assigned. With the lock() and unlock(), data hazard can be avoided. The function time() returns the control step of the node in schedule S.

In the algorithm, Line 1 sets the registers in all sub-instructions to be null in the first. Line 3 re-assigns the registers of the nodes level by level. In this sequence, data dependence won't be violated. In the first, the nodes in Level(1) will be assigned to $L_{RA.}$ Then we choose the first node to re-assign register to it in line 6 and u represents the node. We will assign the register which will cause minimal switching activities to *dst* of u. Before we choose the register, we should unlock all registers and then we lock all the registers which may cause data hazard in line 7 and line 8.

From line 9 to line 11, we scan the register file and find the register to cause the minimal switching activities when assigning it at *dst* of u. That is, assigning it to *dst* of u will make the switching activities between the field of destination of u and the previous sub-instruction on the same functional unit minimal. Figure 3.7 show the detail of how to assign the register.

Line 12 updates the *src* of the children of u. according to the DAG. Then, the first iteration of the algorithm is done. According to input of the schedule, we can find the next node. After we have re-assigned registers to all the nodes in Level(1), we can consider the

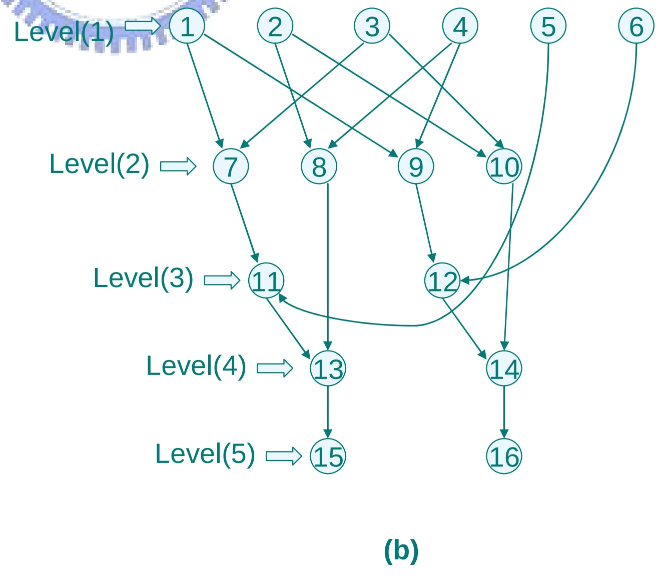
```
Input : current sub-instruction u and last sub-instruction v on the same functional unit
Output : register $A_i$

1.  R ← number of registers , Min ← $\infty$ ;
2.  for  i = 0 to R-1  do
3      if  $A_i$ is not locked  then
4.          if  WHD(Bit_String(u.$A_i$), Bit_String(v.dst)) < Min ;
5.          then  Reg ← $A_i$ ;
6.          end if
7.      end if
8.  end for
9.  return Reg ;
```

Fig. 3.7 The algorithm of register assignment

```
1 ld (Ar) A0
2 ld (Ai) A1
3 ld (Br) A2
4 ld (Bi) A3
5 ld (Cr) A4
6 ld (Ci) A5
7 mul  A0   A2   A6
8 mul  A1   A3   A7
9 mul  A0   A3   A8
10mul  A1   A2   A9
11add  A6   A4   A10
12add  A8   A5   A11
13sub  A10 A7   A12
14add  A11 A9   A13
15 st (Dr)  A12
16 st (Di)   A13
```

(a)



(b)

Fig 3.8 (a) Assembly code of complex_update ; (b) DAG of complex_update

.S1	.L1	.M1	.D1
			3
			4
			1
		7	2
		9	5
11		8	6
12		10	
13			
14			15
			16

(c)

```
 1 ld (Ar) A3
 2 ld (Ai) A2
 3 ld (Br) A0
 4 ld (Bi) A1
 5 ld (Cr) A3
 6 ld (Ci) A1
 7 mul   A3   A0    A4
 8 mul   A2   A1    A4
 9 mul   A3   A1    A5
10mul    A2   A0    A0
11add    A4   A3    A8
12add    A5   A1    A9
13sub    A8   A4    A1
14add    A9   A0    A1
15 st (Dr)  A1
16 st (Di)  A1
```

(d)

Fig 3.8 (c) schedule of the phase one of GSAS ; (d) Assembly code of complex_update after phase two

nodes in Level(2) and so on.

Let's use an example to demonstrate our algorithm of phase two. This example performs a complex update of the form D = C + A*B where A, B, C and D are complex numbers. We first assign the *dst* of the nodes in a common way: from A_0 to A_{15}. Figure 3.8 (a) shows the initial assembly code of complex update. Figure 3.8 (b) is the DAG of complex update. Figure 3.8 (c) is the schedule created by the phase one of the GSAS. In the phase one, the value of β in WHD (β, X, Y) is 15 since we will re-assign the register of the destination of a node. We only consider about the first 15 bits of the binary string. In figure 3.8 (c) we can find the benefits of using the phase one of GSAS. Node 7, 8, 9, 10 are scheduled in this sequence "7, 9, 8, 10". They are scheduled in the sequence since we choose the node which will cause minimal switching activities to schedule. That will increase the probability of operand sharing.

After the phase one of GSAS, we can re-assign the registers. According to the DAG, node 1 to 6 will be considered first. Node 3 will be re-assigned first since its control step is 1 and we assign A_0 to the *dst* of node 3 since A_0 cause the minimal switching activities. Then we will re-assign register to *dst* of node 4. Before we re-assign the register we should lock some register. According to the DAG and the schedule, the *src* of node 7 and node 10 will use the register A0 and the numbers of control steps of them are more than that of node 4, therefore we lock A_0 and A_0 can not be assigned to *dst* of node 4. A_1 will be assigned to the *dst* of node 4. By the same way, we can re-assign all the nodes and the new registers will reduce the total switching activities. Figure 3.8(d) shows the assembly code of complex update after we re-assign the registers.

The complexity of the phase two of GSAS is O(|V||*(|V|+R), where |V| is the number of nodes and R is the number of the registers. It needs O(|V|) to lock registers in line 8 and O(R) to find the register in one iteration. It needs |V| iteration to re-assign all nodes. Hence, the complexity is O(|V||*(|V|+R).

In this chapter, we have demonstrated our method in some detail. In the next chapter, we will see the experimental results and some analyses about our experiments.

Chapter 4. Preliminary Performance Evaluations

In this chapter, we will demonstrate our experimental results. The machine architecture of our experiments has been introduced in chapter 3. The benchmarks of our experiments are from DSPstone[23]. We will first compare the results of phase one of GASA with the results of other methods. Then we will evaluate the results of phase one of GSAS with the results of phase one and two of GSAS.

4.1 The Experimental results for phase one of GSAS

In this section, we will illustrate our experimental results of phase one of GSAS. In Table 4.1, it shows the schedule length and switching activities of each benchmark for each method. We can find that list schedule has the shortest schedule length in comparing with other methods. Our method, the phase one of GSAS, has the least switching activities in each benchmark although it may have the longer schedule length.

From the section 2.2, we found that schedule length and switching activities are two important factors affecting the power consumption. Two constant P_{base} and α play important role in calculating the total energy consumption in equation (4). Therefore, we change the value of P_{base} and α to consider different situations. When P_{base} is bigger than α, it means that schedule length plays the most important role in energy consumption. Conversely, when α is bigger than P_{base}, it means that switching activities play the most important role in energy consumption

Table 4.2 to 4.6 show the total energy and reduction compared with list schedule. We use different value of P_{base} and α to calculate the total energy. We can find that the more P_{base} is bigger than α, the less reduction in our method. But when α is bigger than P_{base}, we have a better performance in reducing total energy consumption since switching activities play an

		List Schedule	MSAS	Phase I
complex_multiply	SL.	9	9	11
	SA.	50	44	**40**
dot_product	SL.	7	7	8
	SA.	41	32	**28**
real_update	SL.	5	5	5
	SA.	33	27	**27**
complex_update	SL.	9	10	12
	SA.	80	69	**65**
biquad_one_section	SL.	13	15	14
	SA.	111	97	**84**
fir	SL.	12	12	13
	SA.	59	51	**48**
mat1X3	SL.	21	21	23
	SA	137	113	**102**
convolution	SL.	8	8	8
	SA.	66	58	**58**

Table 4.1 The comparison on the schedule length and the switching activities

important factor in affecting power consumption.

In Figure 4.1, the row is (P_{base}, α) and the column is the percentage of total energy based on list schedule. From figure 4.1 (a) to (h), we can see the reduction in total energy for each method clearly. In all figures, GASA have more reduction in total energy than that of MSAS when α is bigger than P_{base}. But in figure 4.1 (a), (b), (d), (f), we can find that phase one of GASA is worse than MSAS when (P_{base}, α) = (5, 1). For the phase one of GSAS, it has the longer schedule length than that of MSAS while P_{base} is much bigger than α. Hence, GSAS is not as good as MSAS. In figure 4.1 (c), (h), MSAS and the phase one of GSAS have the same results since they have the same schedule. Thus, we can find that our method has the better benefit of reducing switching activities and reduced more power while α is bigger.

		List Schedule	MSAS	Phase I
complex_multiply	E.	59	53	51
	Reduction	--	**10.2**	**13.6**
dot_product	E.	48	39	36
	Reduction	--	**18.8**	**25**
real_update	E.	38	32	32
	Reduction	--	**15.8**	**15.8**
complex_update	E.	89	79	77
	Reduction	--	**11.2**	**13.5**
biquad_one_section	E.	124	112	98
	Reduction	--	**9.7**	**21.0**
fir	E.	71	63	61
	Reduction	--	**11.3**	**14.1**
mat1X3	E.	158	134	125
	Reduction	--	**15.2**	**20.9**
convolution	E.	74	66	66
	Reduction	--	**10.8**	**10.8**

Table 4.2 The comparison on energy when $P_{base} = 1$ and $\alpha = 1$

		List Schedule	MSAS	Phase I
complex_multiply	E.	68	62	62
	Reduction	--	**8.8**	**8.8**
dot_product	E.	55	46	44
	Reduction	--	**16.4**	**20.0**
real_update	E.	43	37	37
	Reduction	--	**14.0**	**14.0**
complex_update	E.	98	89	89
	Reduction	--	**9.2**	**9.2**
biquad_one_section	E.	137	127	112
	Reduction	--	**7.3**	**18.2**
fir	E.	83	75	74
	Reduction	--	**9.6**	**10.8**
mat1X3	E.	179	155	148
	Reduction	--	**13.4**	**17.3**
convolution	E.	82	74	74
	Reduction	--	**9.8**	**9.8**

Table 4.3 The comparison on energy when $P_{base} = 2$ and $\alpha = 1$

		List Schedule	MSAS	Phase I
complex_multiply	E.	109	97	91
	Reduction	--	**11.0**	**16.5**
dot_product	E.	89	71	64
	Reduction	--	**20.2**	**28.1**
real_update	E.	71	59	59
	Reduction	--	**16.9**	**16.9**
complex_update	E.	169	148	142
	Reduction	--	**12.4**	**16.0**
biquad_one_section	E.	235	209	182
	Reduction	--	**11.1**	**22.6**
fir	E.	130	114	109
	Reduction	--	**12.3**	**16.2**
mat1X3	E.	295	247	227
	Reduction	--	**16.3**	**23.1**
convolution	E.	140	124	124
	Reduction	--	**11.4**	**11.4**

Table 4.4 The comparison on energy when P_{base} = 1 and α = 2

		List Schedule	MSAS	Phase I
complex_multiply	E.	95	89	95
	Reduction	--	**6.3**	**0**
dot_product	E.	76	67	68
	Reduction	--	**11.8**	**10.5**
real_update	E.	58	52	52
	Reduction	--	**10.3**	**10.3**
complex_update	E.	125	119	125
	Reduction	--	**4.8**	**0**
biquad_one_section	E.	176	172	154
	Reduction	--	**2.3**	**12.5**
fir	E.	119	111	113
	Reduction	--	**6.7**	**5.0**
mat1X3	E.	242	218	217
	Reduction	--	**9.9**	**10.3**
convolution	E.	106	98	98
	Reduction	--	**7.5**	**7.5**

Table 4.5 The comparison on energy when P_{base} = 5 and α = 1

		List Schedule	MSAS	Phase I
complex_multiply	E.	259	229	211
	Reduction	--	**11.6**	**18.5**
dot_product	E.	212	167	148
	Reduction	--	**21.2**	**30.2**
real_update	E.	170	140	140
	Reduction	--	**17.6**	**17.6**
complex_update	E.	409	355	337
	Reduction	--	**13.2**	**17.6**
biquad_one_section	E.	568	500	434
	Reduction	--	**12.0**	**23.6**
fir	E.	307	267	253
	Reduction	--	**13.0**	**17.6**
mat1X3	E.	706	586	533
	Reduction	--	**17.0**	**24.5**
convolution	E.	338	298	298
	Reduction	--	**11.8**	**11.8**

Table 4.6 The comparison on energy when P_{base} = 1 and $\alpha = 5$

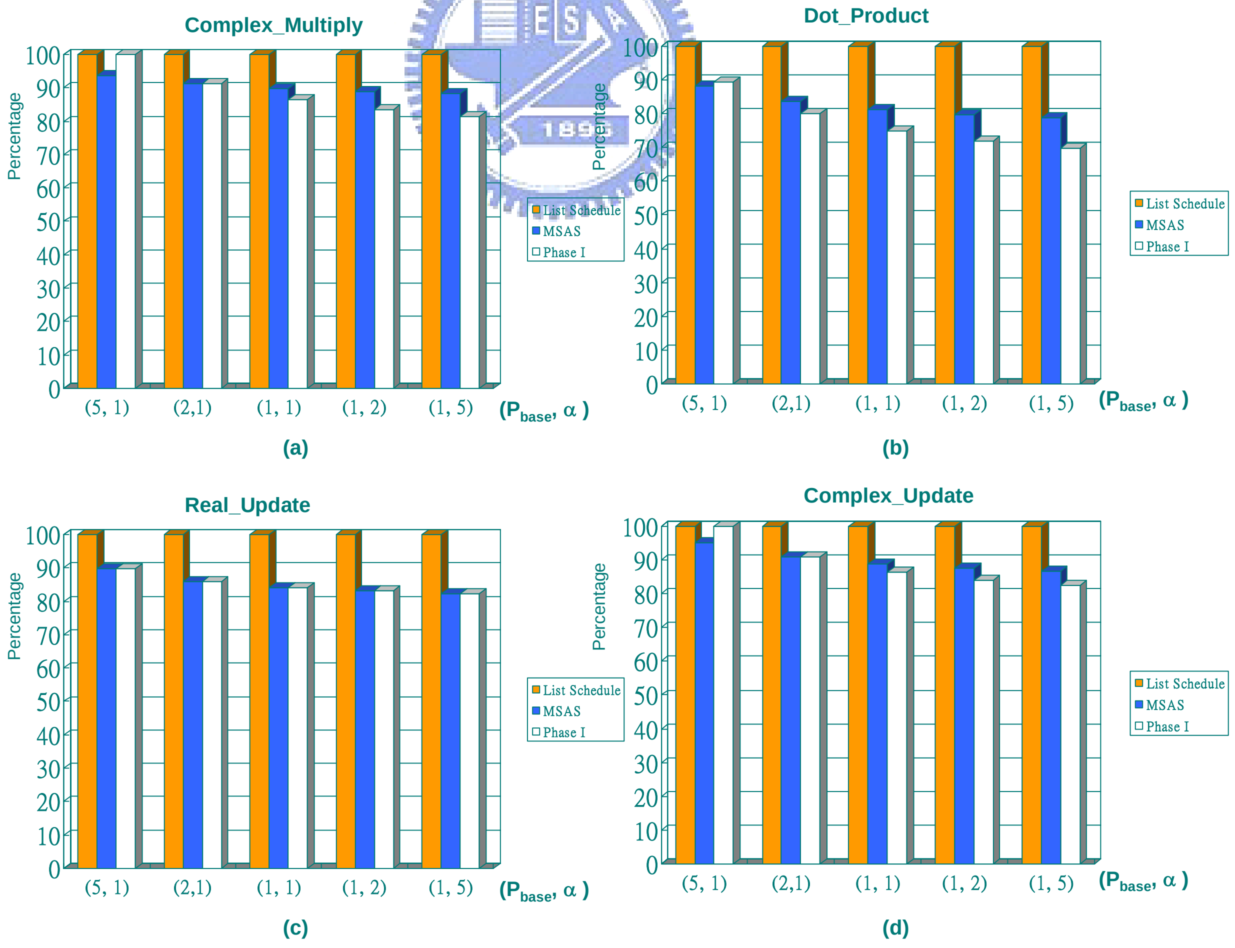
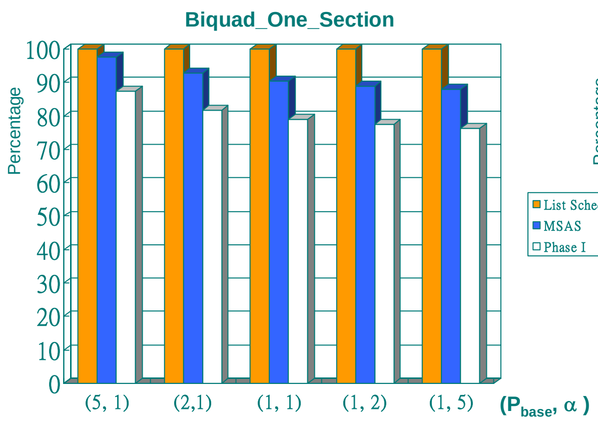
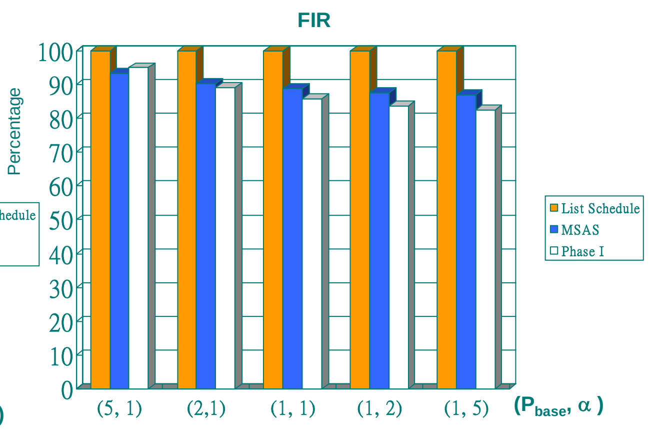


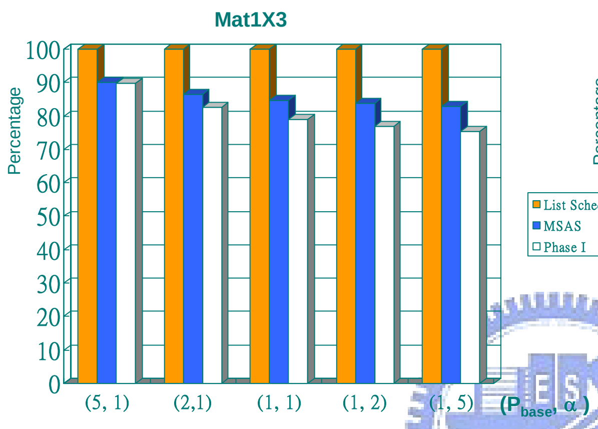
Fig. 4.1 The percentage of energy with different (P_{base}, α) (based on list schedule)



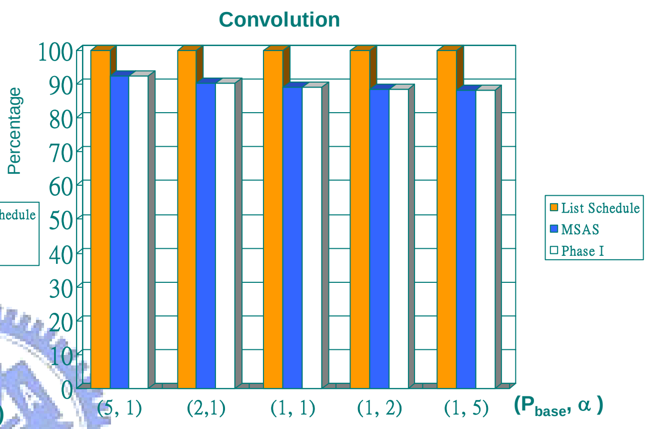
(e)



(f)



(g)



(h)

Fig. 4.1 The percentage of energy with different (P_{base}, α) (based on list schedule)

4.2 The Experimental results for phase two of GSAS

In this section, we will demonstrate the experimental results of phase two of GSAS. In Table 4.7, it shows the schedule length and switching activities of phase one and two of GSAS. We can find that it has better performance in reducing switching activities than that of only using the phase one of GSAS.

Figure 4.2 is similar to Figure 4.1. The row is (P_{base}, α) and the column is the percentage of total energy based on the phase one of GSAS. We can find that in every benchmark, re-assigning registers can reduce the total energy further.

		(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
Phase I + II	SL.	10	8	5	10	14	13	23	8
	SA.	38	22	22	50	68	41	91	45

Table 4.7 The result of Phase I + II of GSAS

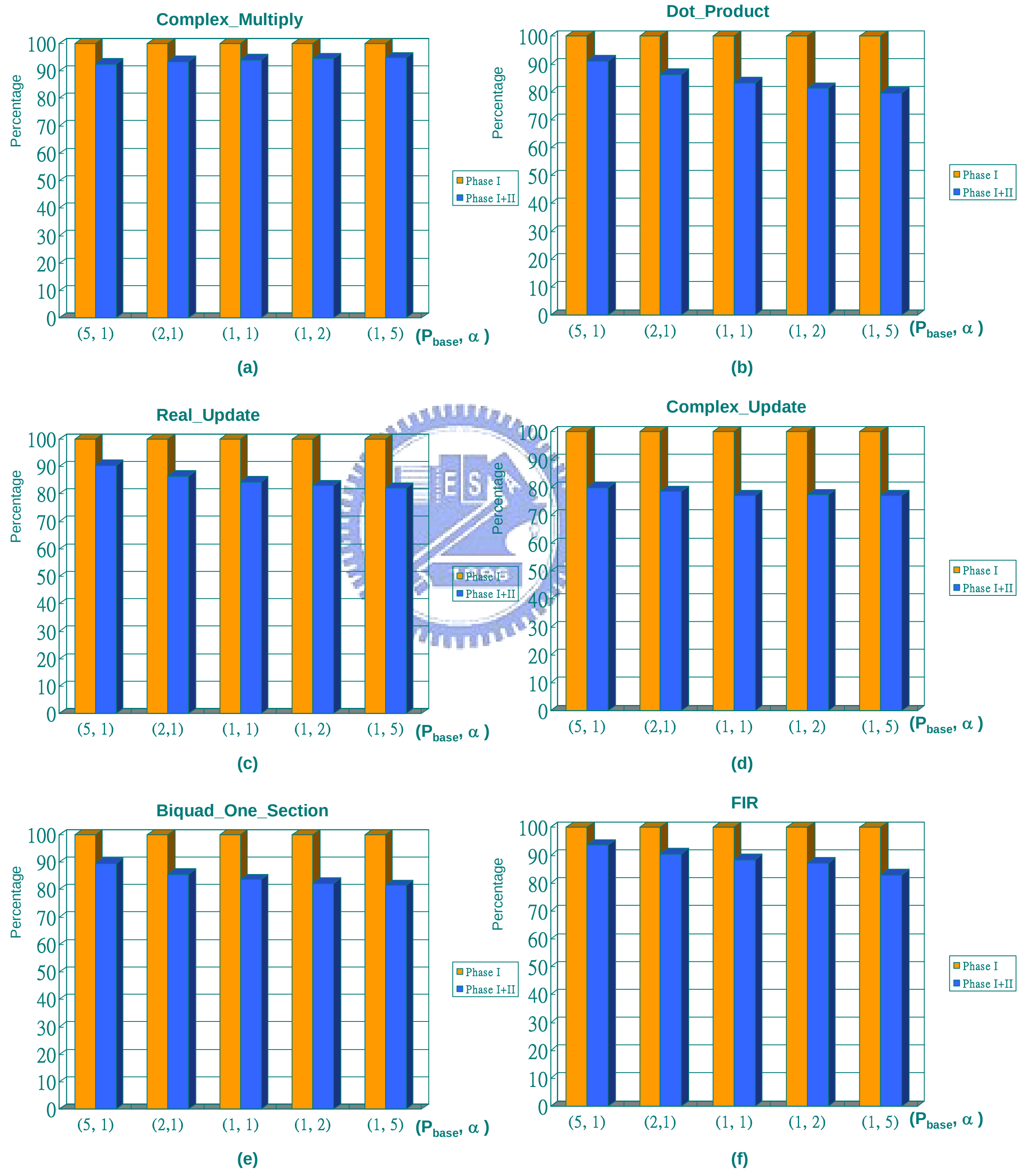


Fig. 4.2 The percentage of energy with different (P_{base}, α) (based on Phase I)

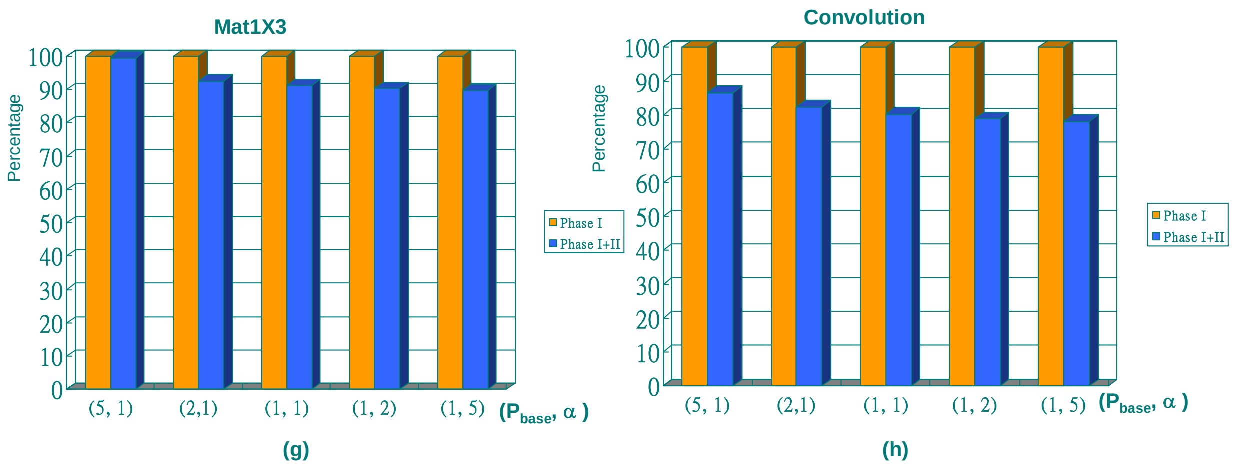


Fig. 4.2 The percentage of energy with different (P_{base}, α) (based on Phase I)

In summary, the phase one of GSAS has the better benefit of reducing switching activities and reduced more power while α is bigger. The phase two of GSAS can reduce the total energy further by re-assigning the registers.

Chapter 5. Conclusion and Future Work

In this thesis, we propose a method named *Greedy Switching Activities Scheduling* (GSAS) which comprises two phases. The phase one of GSAS schedules the DAG. The phase two of GSAS re-assigns the registers to reduce the switching activities. The experimental results have shown the effectiveness of our method. Finally, we will conclude our thesis and propose some future work for our research.

5.1 Conclusion

Portable devices, such as cellular phone, digital camera, PDA have become so popular and are used widely in the world. Hence, the power reduction in VLIW DSP becomes a more and more important problem. Due to buses consume a significant fraction of total power dissipation in a processor, so we propose a method, GSAS, to reduce the switching transitions on the instruction bus. In summary, we give the following conclusions:

(a) The phase one of GSAS uses a greedy method to schedule the DAG and reduce the switching activities. According to the experimental results, the more power caused by each bit switch the more energy our method can save. That is, when the power coefficient α representing the consumed power per transition is big, then we can save more power in switching activities.

(b) The time complexity of the phase one of GSAS is $(|V|*(|V|*N))$, where $|V|$ is the number of the sub-instructions and N is the number of functional units. It don't need to find the min-cost maximal weight bipartite matching and just finds the only one node in one iteration which needs at most $O(|V|*N)$ to be completed. But the complexity of MSAS is $O(|V|*(N+|V|)^3)$. Hence, the phase one of GSAS saves more time in comparing with MSAS.

(c) The phase two of GSAS can improve the results of the phase one of GSAS by re-assigning the registers. According to the experimental results, we can observe that when the phase one collocates with the phase two, it can save more power than only using the phase one. We can find that the register assignment is an important factor affecting the total switching activities. The phase two uses a greedy method to re-assign the registers and it can reduce the total switching activities of the schedule created by the phase one.

5.2 Future Work

There are still many things we can do in the future.

(a) In our experiments, we only use simplified machine of TI TMS320C6000. In the future, we can try to do our experiments with different machine architectures to see if our method works in other architectures.

(b) The phase two of GSAS can be only collocated with the phase one of GSAS. In the future, we will try to find a better way to re-assign the registers to reduce the switching activities and we will make it collocated with all other algorithms.

(c) Our method is not designed specially for the loop applications. We don’t do the optimization for the organization of the loop body. In the future, we can focus our research on the scheduling for the loop applications to reduce the schedule length and switching activities of a loop.

(d) Our method only consider about the self-transitions. There are some researches trying to reduce the coupling-transitions [24-25]. In the future, we can consider about both self-transitions and coupling-transitions and try to reduce more power.

Bibliography

[1] V. Tiwari, S. Malik, and M.Fujita, “Power analysis of embedded software: A first step towards software power minimization,” in *Proceedings of the IEEE.ACM International Conference on Computer Aided Design*, Nov. 1994, pp. 110-115.

[2] N. Chang, K. Kim, and H. G. Lee, “Cycle-accurate energy measurement and characterization with a case study of the ARM7TDMI,” *IEEE Tran. On VLSI Systems*, vol. 10, no. 2, pp.146-154, Apr. 2002.

[3] L. -F. Chao, Andrea LaPaugh, and Edwin H. -M. Sha, “Rotation Scheduling: A Loop Pipelining Algorithm”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 16, Issue 3, pp. 229-239, March 1997.

[4] Nelson L. Passos and Edwin H. -M. Sha, “Achieving Full Parallelism using Multidimensional Retiming”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 11, pp. 1150-1163, Nov. 1996.

[5] Nelson L. Pasos and Edwin H. -M. Sha, “Scheduling of Uniform Multidimensional Systems under Resource Constraints”, *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 6, Issue 4, pp. 719-730, Dec. 1998.

[6] Mike Tien-Chien Lee, and Vivek Tiwari, and Sharad Malik, and Masahiro Fujita, “Power Analysis and Minimization Techniques for Embedded DSP Software”,*IEEE Transactions on VLSI Systems*, Vol 5, no1, pp. 123-133, March 1997.

[7] M. J. Irwin. Tutorial: Power reduction techniques in SoC bus interconnects. In *1999 IEEE International ASIC/SOC* Conference, 1999.

[8] Texas Instruments, Inc. TMS320C6000 CPU and Instruction Set Reference Guide 2000

[9] Texas Instruments, Inc. TMS320C6000 Peripherals Reference Guide

.

[10] Aili Shao, Qingfeng Zhuge, Youtao Zhang, and Edwin H. -M. Sha, “Algorithms and Analysis of Scheduling for Low-power High-performance DSP on VLIW Processors”, accepted in *International Journal of High Performance Computing and Networking*.

[11] Zili Shao, Qingfeng Zhuge, Edwin H. -M. Sha, and Chantana Chantrapornchai, “Loop Scheduling for Minimizing Schedule Length and Switching Activities”, *Proc. of International Symposium on Circuits and Systems*, Vol. 5, pp. 109-112, May 2003.

[12] Zili Shao, Qingfeng Zhuge, Edwin H. -M. Sha, and Chantana Chantrapornchai, "Analysis and Algorithms for Scheduling with Minimal Switching Activities", *Proc. of 45th Midwest Symposium on Circuits and Systems*, Vol. 1, pp. 372-375, Aug. 2002.

[13] C. Lee, J. -K. Lee, and T. Hwang, "Compiler Optimization on Instruction Scheduling for Low Power", *Proc. of International Symposium on System Synthesis*, pp. 55-60, Sep. 2000.

[14] K. Choi and A. Chatterjee, "Efficient Instruction-level Optimization Methodology for Low-power Embedded Systems", *Proc. of International Symposium on System Synthesis*, pp. 147-152, Oct. 2001.

[15] Markus Lorenz, Rainer Leupers, Peter Marwedel, Thorsten Drager, and Gerhard Fettweis, "Low-energy DSP Code Generation using a Genetic Algorithm", *Proc. of International Conference on Computer Design*, pp. 431-437, Sep. 2001.

[16] E. Musoll and J. Cortadella, "Scheduling and Resource Binding or Low Power", *Proc. of International Symposium on System Synthesis*, pp. 104-109, April 1995.

[17] Suvodeep Gupta and Srinivas Katkoori, "Force-directed Scheduling for Dynamic Power Optimization", *Proc. of IEEE Computer Society Annual Symposium on VLSI*, pp. 68-73, April 2002.

[18] Daehong Kim, Dongwan Shin, and Kiyoung Choi, "Low Power Pipelining of Linear Systems: A Common Operand Centric Approach", *Proc. of International Symposium on Low Power Electronics and Designs*, pp. 225-230, Aug. 2001.

[19] Zili Shao, Qingfeng Zhuge, Edwin H. –M. Sha, *Meilin Li and Bin Xiao*, "Switching-Activity Minimization on Instruction-level Loop Scheduling for VLIW DSP Applications", *Proc. of 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Pages224 – 23, Sept. 2004 .

[20] H. Saip and C. L. Lucchesi, "Matching algorithm for bipartite graphs, Tecn. Rep.DCC-93-03 (Departamento de Cincia da Computao, Universidade Estudal de Campinas), March 1994.

[21] C. E. Leiserson and J. B. Saxe, Retiming synchronous circuity. *Algorithmica*, 6:5-35, 1991.

[22] M. J. Irwin. Tutorial: Power reduction techniques in SoC bus interconnects. In *1999 IEEE International ASIC/SOC Conference*, 1999.

[23] http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html

[24] Chun-Gi Lyuh, Taewhan Kim, Ki-Wook Kim, "Coupling-Aware High-level Interconnect Synthesis for Low power", *Proc. of the 2002 IEEE/ACM international conference in Computer-aided design*, Page609 - 613, Nov. 2002.

[25] Yan Zhang, John Lach, Kevin Skadron, Mircea R. Stan. "Odd/Even Bus Invert with Two-Phase Transfer for Buses with Coupling", *Proc. of the international symposium on Low power electronics and design*, Page 80 – 83, Aug. 2002.