能有效轉換 x86 指令之 RISC 指令集設計

Design of RISC-based Instruction Set For Efficient x86 Emulation

黃英哲　　許哲銘
Ing-Jer Huang, Jer-Ming Shiu
Institute of Computer and Information Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan
Email: {ijhuang, jmshiu}@cie.nsysu.edu.tw

摘要

在此研究中，我們先分析 x86 指令集的一些特性，並定義出以 RISC 為基礎的微指令來有效支援 x86 指令集的執行，為了更進一步改善執行的效能，新增加的有效指令能夠以結合多個微指令來合成一個單一的 RISC 指令。新合成的有效指令必需經由一個評估函式來決定，此評估函式包含指令執行的週期數，指令集個數的大小和硬體上的考量。研究結果，有效的新指令確實能減少 x86 指令的執行週期數並得以在指令集的層面上來獲得效能。

關鍵字：指令集設計，x86 架構，x86 模擬，RISC，CISC，微動作

Abstract

In this work, we first analyzed the features of x86 instruction set and defined micro operations as RISC-based instruction set to efficiently support x86 emulation. To further improve the performance, powerful RISC instructions are synthesized by compacting multiple micro operations into a single RISC instruction. The synthesized powerful instructions are evaluated by a cost function contained execution cycle, instruction set size and the hardware cost. As a result, the new powerful instruction set can execute the x86 instructions with reduced instruction cycle counts and gained the performance in instruction level.

Keywords: instruction set design, x86 architecture, x86 emulation, RISC, CISC, micro operation

1. Introduction

The x86 is a CISC style instruction set, and often one instruction can operate several complex micro operations. This caused the instruction execution cycles do not balance with instructions that executed in different functional unit.

Due to the benefit of the RISC instruction set, it is worth to find a new instruction set that has RISC style format to efficiently emulate the x86 instruction set. By using the new instruction set, we can gain performance in instruction level. So here we want to find a new RISC instruction set that supports x86 architecture efficiently, and with a fast, automatic way.

Viewing the x86 emulation as the dedicated application to be run of the RISC code, we can adopt the techniques used for application specific instruction set synthesis to systematically generate a RISC-based instruction set that best support the x86 emulation. First we construct basic RISC micro-operations (instructions) based on the analysis of the x86 instruction semantics. With these operations, we can synthesize new instruction set further. The new RISC-based instruction set consists of simple micro operations and powerful instructions consisting of multiple independent simple micro operations. The powerful instructions can be generated by compacting the simple micro operations.

The compaction can be treated in the three ways: 1. individual x86 instructions, 2. instruction pairs, 3. instruction window. In the next three figures, "X86_*I*" is the instruction of x86, the "MOP_*I*" is the mapped micro operation. In the right of the figures are the synthesized instructions.

Figure 1: Individual x86 instructions mapping

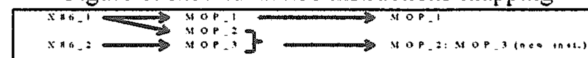

Figure 2: Instruction pairs mapping

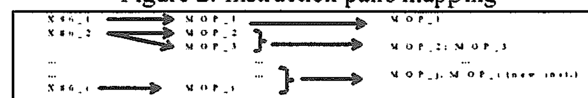

Figure 3: Instruction window mapping

Here we presented a way to systematically synthesize a new RISC-based instruction set for efficient x86 emulation from a given application benchmark or a sequence of basic blocks that can reflect the typical usage of x86 instruction set in real world. The technique is based on compaction of micro operations to form new instructions. The compaction result is evaluated by the cost function of cycle count, instruction set size and hardware cost that supported the new instruction set.

The rest of the paper is organized as follows. Section 2 describes the mapping of x86 instructions into RISC-based instructions (micro instructions). Section 3

[*] This work is supported by NSC, R. O. C. under contact number 85-2262-E-009-0102.

presents a systematic approach to search for powerful RISC instructions for efficient x86 mapping. Section 4 shows the results of section 3. Section 5 concludes this paper.

2. Mapping x86 Instructions to RISC Operations

The following sections described some micro operations that have special features for supporting x86 translation. These micro operations focus on integer instructions.

2.1 Decoupling Memory Operands

Most x86 ALU instructions support the operation with the operands from the memory location, or store the results into the memory location after execution. In RISC micro operations, this kind of instructions in x86 should be replace by load and store operations before and after execution ALU operation. The value loaded from memory first put in a temporarily register, then the function unit can operate the operand in the temporarily registers.

Because x86 instructions support different size of operand, the load or store micro operations need to specify the size that the data move from or to the memory. The possible size of data could be 8, 16, 32, 64, 80 bits. 64-bit data is for an instruction, "CMPXCHG8B", that compares the 64-bit value in EDX:EAX register with memory location. 80-bit data could be used for floating point instructions that need 80-bit extended real format or 80-bit packed decimal integer.

2.2 Instructions Without Modification to the Register Contents

There are two instructions that do not write back the results after execution. The "CMP" instruction compares two source operands but needs not to write back the result. Its operation is to subtract two source operands and updates the six flag status bits of EFLAGS register, then discards the result of subtraction.

Another instruction in the same situation is "TEST". Its operation is same as "AND", but not writing back the result. In RISC architecture, the register set often has a register called "R0" that it can not be modified and always has the value of zero. The concept of using "R0" as discarded register output is proper for the two instructions, but not for load and store micro operations that must modify the content of the register.

2.3 Flags Side Effects

X86 arithematic/logic instructions do not manipulate the flag bits in a uniform way . Therefore,many variutions of the arithematic/logic operations are necessary. For example, many ALU operations need to modify six flag status bits of the EFLAGS register, but "INC" and "DEC" just modify 5 bits.

"PUSH" and "POP" instructions need to decrease or increase the stack pointer register, ESP, to indicate the top location of the stack. But the decrement or increment of stack point register should not affect the six flag status bits of EFLAGS register. for this reason, we created new micro operations called "SUBIN and "ADDIN" that are suitable for this kind of instructions. These two micro operations are suitable for the other two instructions, "PUSHA" and "POPA".

2.4 EFLAGS Register Naming

In x86, it has 8, 16, and 32 bits general registers, and each part that used by the instruction for the operand has corresponding register names. Rather than provide separate storage for each, the shorter registers are overlaid on the longer ones, as shown in the Figure.4 . For example, a 32-bit register EAX has 8-bit part (AH, AL) and 16 bit part (AX) register name that can be assigned as operand for 8, 16, 32 bits operations. The registers of EAX, ECX, EDX, and EBX have 8-bit and 16-bit register names for short, while the registers of ESP, EBP, ESI, and EDI just have 16-bit names as illustrated in Figure.4.

But in EFLAGS register, it is a 32-bit register. Unlike general registers, it does not have additional names for 8-bit and 16-bit part. It will be fine if the instruction set does not need the small parts of the 32-bit register as operand. On the contrast, it does need. The instruction "LAHF" (Load AH register into flags) copies the lowest byte of the EFLAGS register into the AH register. The instruction "SAHF" (Store AH register into flags) copies the contents of the AH register into the *active* flags in the lowest byte of the EFLAGS register. Since the bit 1, bit 3, and bit 5 of the EFLAGS register can not be modified, the instruction just stores five bits values from the AH register into the lowest byte of the EFLAGS register. The instruction "PUSHF" pushes the contents of the low order WORD (16-bit) of the EFLAGS register onto the stack. The instruction "POPF" removes the top WORD from the stack and stores it into the low order WORD of the EFLAGS register.

Flag Control Instructions	Micro Operations Mapping
CLD	ANDIN EFLAGS,EFLAGS,0xFFFFFBFF
CLI	ANDIN EFLAGS,EFLAGS,0xFFFFFFFD
STC	ORIN EFLAGS,EFLAGS,0x00000001 MOVI temp1,0x00000001 ORN EFLAGS,EFLAGS,temp1
STD	ORIN EFLAGS, EFLAGS,0x00000400 MOVI temp1,0x00000400 ORN EFLAGS,EFLAGS,temp1
STI	ORIN EFLAGS, EFLAGS,0x00000200 MOVI temp1,0x00000200 ORN EFLAGS,EFLAGS,temp1

Table 1: Flag control instructions mapping

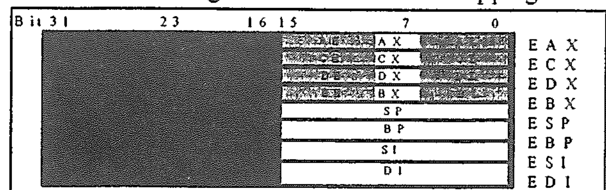

Figure 4: The general registers naming

As those instructions mentioned above, the operand needs additional names for indicating the 8-bit or 16-bit partial data of the EFLAGS register. There are two ways to solve this problem. One is to add the 8-bit and 16 bit register names for the EFLAGS register like the eight general registers. The second is to define a micro operation that can move register data by indicating the field of the register that needs for moving. Here we defined a micro operation called "MOVM" (move with mask) to implement the partial register names of the EFLAGS register. Table 2 lists the four instructions mapping using "MOVM" micro operation.

The micro operation "MOVM" needs five fields for operand.

MOVM Destination, Source, D_Size, S_Size, Mask

The first and second fields, Destination and Source, defined as the destination and source registers that the move operation needs. The third and fourth field defined for the size of the destination and source registers. The last field, Mask, defined the mask value for moving. It is a 32-bit value that indicates the bits need to move to destination register. For example, the instruction "LAHF" moves the lowest byte filed of the EFLAGS register to the register AH. The mask value, Mask, is defined as "0x000000FF" that can describe the bits need to move clearly.

x86 inst.	RTLs	Micro Operations Mapping
LAHF	AH ← EFLAGS$_{[7..0]}$	MOVM AH,EFLAGS,8,32,0x000000FF
SAHF	EFLAGS$_{[7..0]}$ ← AH	MOVM EFLAGS,AH,32,8,0x000000FF
PUSHF	SP ← SP - 2 temp1 ← EFLAGS$_{[15..0]}$ Mem[SS:SP] ← temp1	SUBIN SP,2 MOVM temp1,EFLAGS,16,32,0x0000FFFF ST16 temp1,SS,R0,SP,0,16
POPF	temp1 ← Mem[SS:SP] EFLAGS$_{[15..0]}$ ← temp1 SP ← SP + 2	LD16 temp1,SS,R0,SP,0,16 MOVM EFLAGS,temp1,32,16,0x0000FFFF ADDIN SP,2

Table 2: "MOVM" micro operation

This micro operation has another purpose for assignment different size from source to destination operands. For example, the instruction "LES" (Load pointer into ES and a register) loads the far pointer stored at the memory location into the registers of "ES" and the another one specified by the operand. It can access memory twice for the data writing into the two registers or access once because the memory addresses of the two data are consecutive. After load from memory with a lager temporarily register (32 bits) and then assign the data to the two registers by the micro operation of "MOVM" that can specify the part of the temporarily register for assignment.

2.5 Summary of the Basic Micro Operations

There are 112 integer of micro operations and some of the micro operations can operate on operands of 8, 16, or 32 bits. In this section, we summaries the integer related micro operations, which can be classified into eight sub-classes: (1) register transfer, (2) memory & I/O, (3) arithmetic, (4) logic, (5) rotation & shift, (6) bit manipulation, (7) branch, and (8) miscellaneous micro operations, as listed in Table3. In order to simplify the table, please note the micro operations with the * superscript have many variations with different of types or operations.

3. Synthesis of Application Specific Instruction Set

To find an automatic way to synthesis new instruction set, we need a tool to help us finding a solution that can support the x86 architecture efficiently, particularly the x86 instruction set is complex and large in size. With the help of the tool, the synthesized new instruction set can create with different constraints decided by the parameters. When changing the parameters, the tool can create another new instruction set in an automatic and systematic way in less time.

This section describes the steps of design flow of synthesis instruction set. Before the steps, some background knowledge of the x86 instruction set should have. It includes the instructions encoding formats, the addressing modes, the operation modes, and the behavior of each instruction.

Micro Operations	Description
Register transfer	
MOV* S1,S2	move operation
MOVcc R1,R2	conditional move
MOVM R1,R2,S1,S2,Mask	move with mask value
MOVSX R1,R2	move with sign-extention
MOVZX R1,R2	move with zero-extention
SETcc R1	conditional set 0 or 1
SWAPB R1,R2	byte swap operation
Memory & IO	
IN* S1,S2	data read from IN port
LD* (parameters)	Load from memory
LEO* (parameters)	Load effective address
OUT* S1,S2	data write to OUT port
ST* (parameters)	Store to memory
WRSEG R1,R2	move to segment register
Arithmetic	
AAA*	six BCD adjust MOPs
ADD* S1,S2	add operation
CMP* S1,S2	compare operation
DEC R1	decrement one
DIV* (parameters)	division operation
INC R1	increment one
MUL* (parameters)	multiply operation
NEG R1	negative operation
SUB* (parameters)	subtraction operation
Logic	
AND* S1,S2	logic and operation
NOT R1	logic not operation
OR* S1,S2	logic or operation
TEST* S1,S2	test operation
XOR* S1,S2	logic xor operation
Rotation & shift	
ROL* S1,S2	rotate left
ROR* S1,S2	rotate right
SHL* S1,S2	shift left
SHR* S1,S2	shift right
SHRA* S1,S2	shift arithmetic right
Bit manipulation	
BSF R1,R2	bit scan forward operation
BSR R1,R2	bit scan reverse operation
BT* S1,S2	bit test operation
Branch	
JI* Immed	jump immediate
JIcc* Immed	conditional jump immediate
JR* R1	jump register
JRcc* R1	conditional jump register

Miscellaneous	
CHKBND R1,R2	check bound
CPUID	return CPU information
EUD	undefined operation exception

Table 3: Summary of the basic micro operations

3.1 Benchmark Program

Here the benchmark program is binary execution code. It is hard to get source code just from execution file, so the software, "*SOURCER*"[8], is used for the purpose. Translation by "*SOURCER*", the output file is Assembly language listing file of original binary execution file. "*SOURCER*" can help us finding not only the instruction that the execution files used but also the *label* that referenced by the jump type instructions. The data definitions and other directives in the benchmark program are ignored. Only the x86 instruction lines are taken for consideration.

3.2 Mapping to Micro Operations

In this step, the x86 instructions are translated to micro operations. The rules of mapping are defined in library file by designer and can be modified for other reasons.

3.3 Synthesis of New Instruction Set

Given micro architecture, and hardware constraints, synthesis the new powerful instruction set that support the benchmark program we choose. The new instruction set is generated by *simulated annealing algorithm* [13], and its objective cost is evaluated by cost function that the designer can adjust for given intention.

The instruction word length of the new instruction set can be adjusted by the designer as parameters. The typical instruction word length are 32, 48, or 64-bit. Wider instruction word length can be compacted more micro operations but the hardware cost may be higher. To find the trade off, it is an impersonal way to evaluate it by a cost function.

The cost function is used to evaluate that add a new instruction is worthwhile. The function here contains execution cycle (`Cycle`), the final instruction set size result (`InstructionSetSize`), and the hardware cost (`HardwareCost`) that the instruction set used. The function can be describe as blew and is a parameter that can be adjusted.

$$Cost = \alpha * Cycle + \beta * InstructionSetSize + \gamma * HardwareCost.$$

α, β, and γ are the factors that can be adjusted by the designer. The higher value of the factor results in more emphatic about which parameters are taken more considerations.

The instruction cycle could be decreased when the instruction word width increases. That is large instruction word width can accommodate more micro operations. The new instruction set size could be increased by added these powerful instructions.

3.4 Reassemble the Benchmark Program

After the new instruction set is synthesized, we can use the new instruction set to reassemble the benchmark program. The new benchmark program should do the same function as the one using the x86 instruction. The new program assembled by the new instruction set has the spirit of RISC and has few execution cycles.

The whole design flow can be simply described in Figure 5.

4. Synthesis Results of Compact RISC Instructions

4.1 Synthesis From Individual x86 Instructions and Pairs

First, we give an example that the contains the TOP 25 x86 instructions, and TOP 15 x86 instruction pairs. Use TOP 25 instructions and TOP 15 instruction pairs as the application benchmarks for *ASIA* synthesis methodology.Each instruction and instruction pair in different basic blocks are divided by *labels*. Therefor, no data dependency will occur. The micro operations in the basic block also are the basic component of micro operations. Each basic block is assumed executed once.

```
label(top_1).
        mov_R16_R16(ax,bx).
abel(top_2).
        shl_R16_1(ax).
... ...
label(pair_1).
        mov_R16_R16(ax,bx).
        shr_R16_1(ax).
... ...
```

This benchmark program contains 25 different instructions originally and result is 22 RISC style instructions from the tool synthesized. The reduced instruction set size from 25 to 22 is that some x86 instructions, like string instructions, have same micro operations with other instructions. The result instruction set is one micro operation or compacted instruction with more than one micro operation. In this small benchmark program, one new parallel instruction is `<ADD, FWAIT>` appeared in the 15th instruction pair. "`FWAIT`" is a floating point instruction, and use different resource type with "`ADD`". This pair is used in our benchmark programs to execute next floating point instructions. The compiler adds a "`FWAIT`" instruction before a floating point instruction that follows an "`ADD`" instruction.

Another new instruction in the benchmark is *conditional add or sub* created from string instructions. All string instructions add or sub registers according to the flag "`DF`" (direction flag). If `DF` = 0, then one or two registers must increase. If `DF` = 0, then decrease. One or two registers must be changed is determined by the instruction itself.

The benchmark program executed 55 (25 + 15*2) instructions before using synthesized instruction set. With mapping into micro operations, it has 82 micro operations. The cycles of per micro operation are assumed one. After reassembled with new instruction set, it execution cycles became 72. The benefit is most

from simplify the string instructions using *conditional add or sub* instruction.

The hardware cost used by the benchmark program is 2 register read ports, 1 register write ports, 1 ALU, and 1 FPU. Because the benchmark program is very small, it used fewer hardware resources. The result is same with instruction length from 32 bits to 48 bits and 64 bits.

4.2 Frequent Execution Core of DOS and Win95 Applications

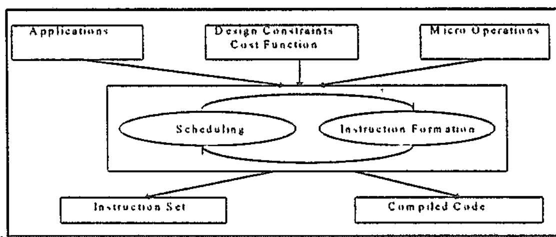

Figure 5: Design flow

It is hard to find all programs in the real world, so it is another way to find the most executed basic blocks for typical programming model. Nowadays, the program often executed in Windows mode and the Windows system executed some GUI routines for all Windows applications. On the other hand, Windows system uses Virtual Memory Management routines to support virtual memory. It is objective to find the representation slice of this kind of program as the benchmark input.

We took about 500 lines of assembly code as benchmark program. The number of basic blocks of the benchmark program is 131 with 396 x86 instructions, and this shows the fact that the benchmark program has many instructions that take branch, like `Jcond`, `CALL`, and `RET`. It results in a basic block contains a few instructions.

4.2.1 First experiment: optimization for performance/cost trade off

The original instruction set size is 69, contains 32 bits and 16 bits mode instructions. The result of instruction set size is 29. That is we take different size operation as same instructions, and the complex x86 instructions could be extracted into simple micro operations. The mapped micro operations number is 597. The average numbers of mapped form x86 instructions to micro operations are 1.45 (597/396). Assumed one micro operation executed in one cycle, so the original execution cycle counts is 597. After using the new instruction set synthesized, the execution cycle count is reduced to 574.

	Distinct Instructions	Execution Cycle Counts
Benchmark in x86	69	597
Micro operations	29	574

Table 4: First experimental result

The new instruction created in this benchmark program is <`JI, MI`> and <`JI, MR`> that shown in the Table 5. They are used for the program for 15 and 8 times respectively. "`JI`" is a jump instruction with relative address which is an immediate value. "`MI`" is a store instruction that writes an immediate value to memory, "`MR`" is also a store instruction but the value that writes to memory is from a register. With the two of new instructions, <`JI, MI`> gained performance improvement in reduced execution cycle counts as 2.51% and <`JI, MR`> as 1.34%, respectively. Totally, overall performance improvement is 3.85%.

The new instructions of <`JI, MI`> and <`JI, MR`> can be summarized as the Table 5. The RTL of the new instructions are the combinations from the simple micro operations. The symbol ";" in the RTL column means the two operations can be operated at the same time.

Micro operation	Operands	RTL
JI	addr(A)	reg(pc) ← addr(A)
MI	R1,Immed	mem(R1) ← Immed
MR	R1,R2	mem(R1) ← reg(R2)
<JI, MI>	addr(A), R1, Immed	reg(pc) ← addr(A); mem(R1) ← Immed
<JI, MR>	addr(A), R1,R2	reg(pc) ← addr(A); mem(R1) ← reg(R2)

Table 5: New instructions, <`JI,MI`> and <`JI,MR`>

The new instruction set used two register read ports, one register write port, and one ALU. If the designer reduces the factor of hardware cost in the cost function, then it is to minimize the effect of hardware resources. Let us try to reduce the factor and find what new instructions have produced.

4.2.2 Second experiment: optimization for performance

The input of second experiment is same as first, except the second using lower hardware cost constraint. That is, the synthesized new instruction set takes less importance on hardware cost, just considering the performance issue. Then the another result with lower hardware cost constraint summarized in Table 6.

	Distinct Instructions	Execution Cycle Counts
Benchmark in x86	69	597
Micro operations synthesized with lower hardware cost constrain	30	556

Table 6: Second experimental result

The new instructions created with lower hardware cost constraint that changed the hardware resources number compared with the previous result. It used *three* register read ports, and has new instructions <`MR, SUBIN`> and <`MR, SIGN`> that used for 30 and 1 times, respectively. "`SUBIN`" is a sub micro operation that one source operand is immediate value, and without changing corresponding flags (`CF, PF, AF, ZF, SF, OF`). It is mapped from the "`PUSH`" instruction in x86 for updating the stack top pointer register, `SP`. "`SIGN`" is a one to one mapping micro operation that from the "`MOVSX`" instruction in x86. It just appeared once in the benchmark program. As being compacted into

instruction with "MR", the instruction set created at this time does not contain the "SIGN" micro operation. The overall performance improvement with added hardware resources number has reached to 6.86%. The new instruction includes <MR, SUBIN> and <MR, SIGN> beside <JI, MI> and <JI, MR> mentioned at section 4.2.1.

Let us take more close look at the usage of the new instruction <MR,SIGN>. The original x86 instructions in the benchmark are written at the left column. The "PUSH" instruction mapped into 3 micro operations as <1>, <2>, <3> and MOVSX as <4>. Using the new instruction <MR,SIGN>, that is compacted from <3> and <4>, the result program slice reduced one instruction and reduced one execution cycle. The micro operation <MR> needs two register read ports and one memory unit. The micro operation <SIGN> needs one register write and one register read port. The new compact instruction, <MR, SIGN>, results in *three* register read ports that is due to the hardware cost constraint had reduced by designer.

Original x86 instructions	Mapped micro operations	Using synthesized new instruction set
PUSH [EAX+8]	1.RMD(temp32(1),eax,+8) 2.SUBIN(esp,esp,4). 3.MR(esp,temp32(1)).	1. ... 2. ... <MR,SIGN>(esp,temp32(1),eax,cl)
MOVSX EAX,CL	4.SIGN(eax,cl).	

Table 7: The new instruction <MR, SIGN>

5. Conclusions and Future Work

The instruction set design is a tedious work, especially for the x86 architecture. Not only the complex instruction formats and the addressing modes, but also the variety operations from ALU to system instructions. It needs group of work to cover all of the details in the instruction set and computer architecture.

We found a way to improve performance by designing a new internal instruction set of x86 emulation. The new instruction set has the spirit of RISC and still preserves all the operations of x86. The new instruction set is synthesized by compacting the micro operations that support the original x86 instructions. Here we presented an automatic way to synthesize the new instruction set and provided some parameters that can be adjusted by the designer such as the mapping rules from CISC to RISC, hardware cost constraints, instruction length (32 bits, 64 bits...). With the new powerful instructions, the program can be executed with fewer cycles that fully utilized the functional units provided by the given micro architecture. The result also gives us a suggestion to add additional functional units to support the new instruction set.

The design of instruction set not only need to consider the support of micro architecture and hardware resources but also need the benchmark programs to find the typical programming model. The future work contains:

(1) The collection of benchmark program that have used many types of x86 instructions like FPU and MMX unit.

(2) Define more precisely of the RTL descriptions of the micro operations, especial the system instructions.

(3) Define the hardware cost of each hardware resource in the cost function with reasonable parameters.

6. Reference

[1] Mike Johnson: "*Superscalar Microprocessor Design*", PTR Prentice Hall.

[2] "*Pentium Pro Family Developer's Manual*", Volume 2,3, Intel corporation 1996.

[3] David W. Wall: "*Limits of Instruction-Level Parallelism*", In Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operation Systems, April 8-11, 1991.

[4] "*Intel's MMX Speeds Multimedia*", Microprocessor Report, Vol. 10, No. 3, March 5, 1996.

[5] "*Pentium Family User's Manual, Vol. 3: Architecture and Programming Manual.*", Appendix B.

[6] "*SOURCER user's Manual*", V Communications, INC. 1993.

[7] "*Intel's P6 Uses Decoupled Superscalar Design*", Microprocessor Report, Vol. 9, No. 2, February 16, 1995.

[8] David A. Patterson: "*Reduced Instruction Set Computers*", Communications of ACM, Volume 28, Number 1, January 1985, pages 8-21.

[9] Ing-Jer Huang, Alvin M. Despain: "*Synthesis of Application Specific Instruction Sets*", IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, NO. 6, JUNE 1995.

[10] D.F. Wong, H.W. Leong, C.L. Liu: "*Simulated annealing for VLSI design*", Kluwer Academic Publishers.

[11] Andrew Wolf, John P. Shen: "*A Variable Instruction Stream Extension to the VLIW architecture*", In Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operation Systems, April 8-11, 1991.

[12] Monica S. Lam, Robert P. Wilson: "*Limits of Control Flow on Parallelism*", The 19th Annual International Symposium on Computer Architecture, 1992.

[13] Barry B. Bery: "*The Intel Microprocessors-8086/8088, 80186, 80286, 80386, and 80486-Architecture, Programming, and Interfacing*", 3rd ed.