DOS/Win95 x86 指令集之分析及其對微架構之影響
Analysis of x86 Instruction Usage in DOS/Win95 Applications and it's Implication on Microarchitecture Design

黃英哲 彭子謹
Ing-Jer Huang, Tzu-Chin Peng
Institute of Computer and Information Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan
Email: {ijhuang, tcpeng}@cie.nsysu.edu.tw

摘要

根據 Amdahl's laws，要改進微處理，最有效率的作法是先改進最常用的指令因此而需要知道指令集在應用軟體我使用的狀況。而這些資料也有助於我們去改進微處理機的架構而利於執行這些常用的指令。本篇論文包括設計一個監視系統，以監視 DOS 及 Win95 下應用軟體對於指令集、指令配對、定址模式、運算原的型態及大小的使用情況，並以此資料分析出對於超純量及精簡指令集核心之 X86 電腦架構設計上的一些建議。

關鍵字: 複雜指令集、精簡指令集、超純量、指令集、指令配對、定址模式

Abstract

x86 is a popular microprocessor family based on the CISC architecture. It has an abundant instruction set, and complex addressing modes. To enhance the microprocessor's performance, according to Amdahl's laws (make the common case fast) [6], it is more efficient to improve the execution of most used instructions. So we first need to understand the run time behavior of the instruction set including the most frequently used instructions and instruction pairs in application programs. The information is helpful to optimize the microarchitecture and compiler for efficient instruction execution. This thesis presents a monitoring system to obtain the usage information under the MS-DOS and MS-Windows95 environment, including instruction set usage, instruction pairs, addressing mode, and operand types and sizes. The implications of the DOS/Win95 analysis on the x86 superscalar architecture design are investigated.

Keywords: CISC, RISC, superscalar, instruction set, instruction pair, addressing mode

1. Introduction

x86 is a popular microprocessor family based on the CISC architecture. It has an abundant instruction set, and complex addressing modes. To improve the x86 execution performance, there are three technique might be used in the design of a superscalar x86, including " out-of-order microinstruction issue", "overlapping microinstruction sequences", "superscalar execution of RISC Core instruction set".

In order to determine the new microarchitecture to enhance the microprocessor's performance, according to Amdahl's laws (make the common case fast) [6], it is more efficient to improve the most used instructions. So we first need to understand the run time behavior of the instruction set that include the most frequently used instructions and instruction pairs in application programs. The information is helpful to optimize the microarchitecture and compiler for efficient instruction execution.

We have developed a monitoring system to obtain the usage information of DOS application and use Intel's tool "VTune" [5] to get instruction trace of Windows95 applications.

After the benchmarks were traced, we analyze the data about instruction (list in appendix) and find out the implications related to four kinds of issues including (1). issues related to the decoders. (2). issues related to the function units. (3). issues related to the microarchitecture. (4). issues related to pairing of x86 instruction. It will be mentioned in section 8.2.

In section 2, we discuss about the relative research about this topic, and mention the drawback the their research.

Our analysis tools will be discuss in section 3 ,4 ,5 and 6. The "monitor program" will be discussed later in section 3. The "VTune" will be discussed in section 4. The "x86 instruction analyzer" will be discussed in section 5. The integrated environments will be discussed in section 6.

In section 7, we introduce the benchmark we have traced and discuss the data we trace and list the data.

In section 8, we introduce the superscalar architecture model for x86 and mention about the implication on x86 architecture design. And then we introduce the instruction pairing mechanism in pre-decoding.

In section 9, we summarize the implication and list in a table to make conclusion. And discuss our future work.

2. Related Work

Some papers have been released about this topic. In [3], it mentions about the implementation technique of the software instruction counter for debugging and profiling, but works only when source code is available. In [4] they also analyze the usage of x86 instruction set on MS-DOS, but they do not analyze the detail size of

[*] This work is supported by NSC, R. O. C. under contact number NSC85-2262-E-009-0102

and type of their operand.

3. Monitoring in DOS environment

Under the DOS environment, there is a very convenient way to do this job. We can take advantage of single-step mode of x86 family CPUs. And according to whether the source is available, our monitor includes two approaches—"with source code" , and "without source code".

3.1 With source code

When the source is available, there are many approaches also available. Our approach is to take advantage of the single-step mode, so we need not to add any additional function before each instruction. We can monitor any pieces of target program by just adding a small code before the program to open the T-flag and adding another code after the of program to close the T-flag. While T-flag is opened, the x86 CPU is set to the single-step mode. As Figure 1, we rewrite and set the ISR of `int1` before spawning the benchmark program. In the benchmark program, after each instruction, program will trap to the ISR of `int1` when T-flag is set. In the ISR, we can get the op code the instruction that will execute later, and record related data. After benchmark program is terminated, information will be arranged and reported.

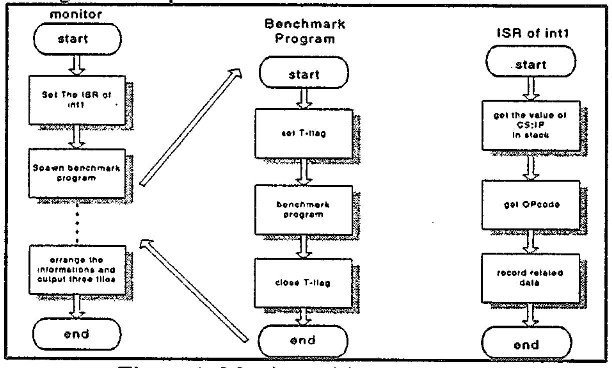


Figure 1: Monitor with source code

3.2 Without source code

When only executable (binary) files are available, i.e. source code is not available, we have to use a special skill to open the T-flag.

We replace the lowest byte of the first instruction that will be executed in the executable file with 0xCC (that means `int3`). Then rewrite the ISR of `int3`. In the modified ISR of `int3`, (as Figure 2), we open the T-flag and change back the first inst.ıction of modified program and let the program counter point to first instruction of benchmark program. After the ISR, the benchmark program can execute as if its first instruction is in its original location. Then we can start tracing the benchmark program from the beginning of the benchmark program.

Once the modified benchmark program is loaded and executed, `int3` would be executed first and then set the T-flag before the executing of benchmark. After T-flag is open, the benchmark program can be traced the same as "with source code" approach.

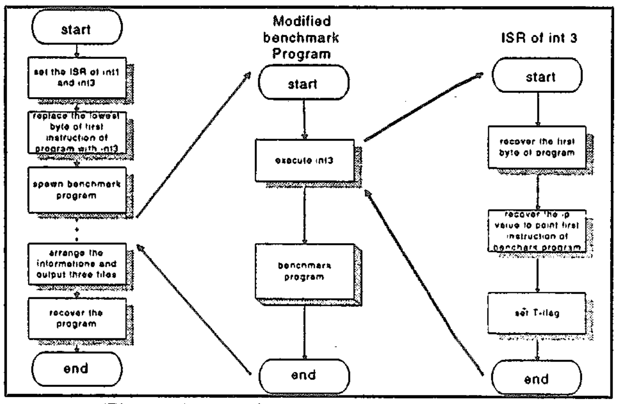

start
set the ISR of int1 and int3
replace the lowest byte of first instruction of program with int3
spawn benchmark program
arrange the informations and output three files
recover the program
end
Modified benchmark Program
start
execute int3
benchmark program
end
ISR of int 3
start
recover the first byte of program
recover the ip value to point first instruction of benchark program
set T-flag
end

Figure 2: Monitor without source code

4. Monitoring in Windows95 environment

On Windows95 environment, we use Intel's performance tuning tool – VTune to help us to get sample of instruction trace. We use two passes to get instruction traces (as Figure 3). First, we sample the executing instruction at regular interval (0.1ms). In the sampling data, we will know find the program blocks that have been executed. In second pass, we use simulate each program block that have been sampled.

In the first pass, VTune saves the sampled machine state by using interrupt. The interrupt was triggered at regular interval. After the program terminated, VTune will list all the address and times which instruction was executed.

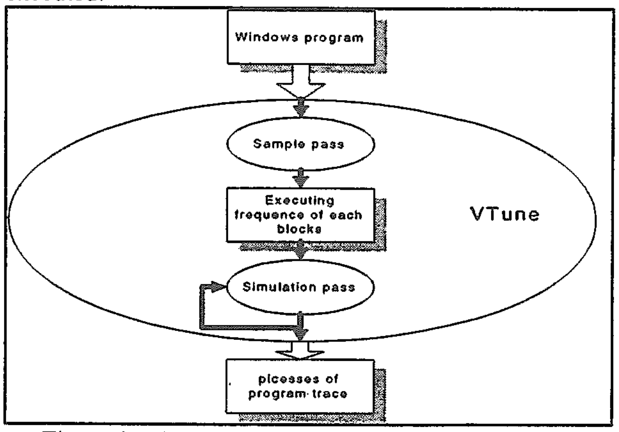


Figure 3: The VTune approach under Windows95

Because VTune cannot simulate whole program, we divided program into blocks. We instruments VTune to simulate each block individually. After simulating each block, we will get instruction trace of each block. The analyzer will read these instruction traces and output the instruction's frequency file and the instruction pairs file.

"VTune" is not design to produce instruction trace. We have to repeatedly operate the simulation pass manually in order to get the traces of all program blocks that was sampled by sample pass. This is really a time consuming job, and the user should stand by while program is simulating. So we will use WinBatch program to run simulation pass automatically and repeatedly.

5. The Integrated Environment for DOS/Win95 Analysis

In order to operate this system easily, we write a user interface under EXCEL macro. At first, our macro program will ask user Win95 or DOS benchmark will be trace. If users select DOS, our monitor program will be called and start monitoring. If users select Win95, VTune and Winbatch will be invoked. After user choice the sampled file and decide which program to simulation, Winbatch will operate the VTune (mentioned in section 4) automatically and output the trace file. After the trace file was generate, EXCEL macro will invoke analyzer to analysis the trace and output x86 instruction using frequency file. EXCEL macro will automatically draw a curve of x86 instruction usage according to the x86 instruction using frequency file. After this information showing, it will ask user to input the x86-to-MOP mapping table to generate the using frequent of MOP.

Our mapping program will be called after the x86 instruction using frequency file was generate and output MOP using frequency file with x86 to MOP mapping table and MOP executing cycle table is available. Once this file was generated, EXCEL macro will automatically draw the statistic curve of MOP using frequency according to the file.

After the MOP using frequency curve drawing, the EXCEL macro will ask user whether to trace another benchmark. If user answer "yes", EXCEL macro will help user to trace another benchmark as first benchmark automatically. If user answer "no", EXCEL macro will merge all the x86 instruction using frequency files and draw the curves of all x86 instruction using frequency, pair using frequency, all MOP instruction using frequency, cycles using frequency, operand using frequency, addressing mode, and instruction classification.

WinBatch is the Windows Batch Language that user can use to write Windows 95 batch files to control almost every aspect of your machine's operation. Because VTune does not support user to write a script file to control VTune in batch, we use WinBatch to run VTune automatically.

6. Benchmark Analysis

6.1 DOS/Win95 Benchmark Suite

In order to obtain a better combination of programs, we use the *Standford Benchmark*, and 14 DOS programs and 4 Windows95 programs and one system kernel[1] (*kernel32*) as benchmark.

We calculate the percentage of the table list in this section by the equation

$$\frac{This\_kind\_of\_instruction * weight}{\sum_{all\_benchmarks} Total\_traced\_inst * weight}$$

The "weight" lists in appendix 1, 2, 11, 12, 13, 14.
The "total_traced_inst" lists in appendix 1, 2.

Dos benchmark program	Windows95 benchmark program
Stanford Benchmark ver 4.1, ARJ2.8, Chkdsk, Find, Format, Grep, Link, Mem, Telnet, Qsort, Primes, Bgidemo, Tedit, Cview, and Cedit	Netterm, MS-WORD 7.0, NETSCAPE 3.0, EXCEL 7.0, and Kernel 32

6.2 Frequent used instructions

We rank the instructions according to their usage frequencies in the Table 1. In these table, we list those instructions that cover over 90% of program execution.

In Table 1, the column of "# mop" means the number of "Micro Operation " (mop) that a x86 instruction can be mapped into. In order to distinguish detail information of operands, we describe operands as:

- `r`: represent "Register", `r16` means a 16 bits register.
- `m`: represent "memory",`m16` means a 16 bits memory block.
- `i`: means "immediate", `i16` means a 16 bits immediate data.
- `d`: means "displacement", it is next to "m", used to describe the displacement of the memory offset. For example,"`m16d16`" means a 16 bits memory block with 16 bits offset.

We notice that the frequencies of `push,` and `pop` are very close. Because the stack will always be in the state of balance, the rank "`push`" and "`pop`" instruction will very close. The same observation holds true for `calln` and `retn`.

6.3 The frequency of instruction pairs

The instruction pairs appear more frequently list in Table 2. The column of "# of mop" means the number of "Micro Operation (Mop)" contained within an x86 instruction.

Notice that, "`pop, pop`" and "`push, push`" pair appear in very high frequency. The must be a lot of small functions in these applications. And in some application such as Stanford Benchmark, it uses a lot of recursive function.

A complex instruction may be mapped into several mops. If these mops consume different hardware resources. The dispatch unit can dispatch these mops into properly functional units such as ALU, FPU, etc. The more functional units are utilized, the more instruction-level parallelism will be achieved. When these two complex instructions are combined to a pair. This pair may use the same functional unit. If happens, the second instructions of the pair will be delayed until the first one releases the functional unit [9].

6.4 The frequency of micro operation

In the experiment the most used micro operations are "`ld`" (load from memory) and "`st`" (store to memory). The usage of these two micro operations is about 30%. And the percentage of cycles used by MOP instructions. "`ld`" instruction used 32.6% of cycle

[1] The most use system kernel is VMM (Virtual Memory Manager). but this file is in W4 format VTune can not handle this format.

(almost 1/3 of cycles). Most cycles are used by some specific MOP instructions (such as "ld" (load), "st" (store), "shl" (shift left), and "mov" (mov-r-r)). It means most resources are use by these MOP instructions.

6.5 Average Instruction cycles

We have count total x86 instruction, total MOPs and total cycles execute of our all benchmark. The average cycle one x86 instruction needs to execute is 2.75 cycles ($\frac{Total\_cycles}{Total\_X86inst}$). The average cycles one MOP needs to execute is 1.84 cycles ($\frac{Total\_cycles}{Total\_MOPs}$). One x86 instruction will translate to 1.5 MOPs ($\frac{Total\_MOPs}{Total\_X86inst}$) in average(very close to the ROP of Nx686).

Rank	instruction	# of MOP	executed ratio
1	push r32	2	8.4%
2	mov r32 m32d8	1	7.1%
3	jz i8	1	5.7%
4	pop r32	1	4.2%
5	mov r32 r32	1	4.0%
6	inc r32	1	3.0%
7	mov r32 m32d0	1	2.9%
8	xor r32 r32	1	2.7%
9	jnz i8	1	2.7%
10	calln i32	?	2.2%
11	cmp r32 r32	1	2.2%
12	mov r16 m16d8	1	2.1%
13	test r32 r32	1	2.1%
14	retn i32	?	1.9%
15	jl i8	1	1.9%
16	mov r8 m8d8	1	1.7%
17	cmp r32 i32	1	1.6%
18	add r32 r32	1	1.5%
19	add r32 i8	1	1.3%
20	cmp m32d32 i8	2	1.3%
21	jz i32	1	1.3%
22	lea r32 m32d0	1	1.3%
23	lea r32 m32d8	1	1.3%
24	cdq	1	1.3%
25	mov m32d8 r32	1	1.3%
	TOTAL		68.1%

Table 1: Top 25 most used x86 instruction under Win95

rank	fst_inst	sec_inst	percentage
1	cmp_R32_I32	jl_I8	3.7%
2	cmp_R32_R32	jnz_I8	3.7%
3	cmp_m32D32_I8	jz_I8	3.7%
4	inc_R32	cmp_R32_I32	3.7%
5	movzx_R8_m8D0	inc_R32	0.9%
6	inc_R32	mov_R32_m32D8	0.9%
7	movsb	pop_R32	0.9%
8	mov_R32_R32	cdq	0.9%
9	sub_R32_R32	lea_R32_m32D0	0.9%
10	lea_R32_m32D0	xor_R32_R32	0.9%

Table 2: The most used x86 instruction pairs under Win95

7. Implications on x86 microarchitecture design

To speedup x86 instruction execution, we use a two level microarchitecture. The outer level accepts x86 instructions, and translates the x86 instructions into micro-operations or RISC-like instructions which the inner level of the microarchitecture accepts. The inner level adopts a superscalar microarchitecture to speed up the execution.

7.1 A superscalar architecture model for x86

Our target microarchitecture has two types of decoder, simple decoders, general decoder (as Figure 4). The simple decoders can handle only simple (register to register) instructions. On the other hand the complex instruction can only be decoded by the general decoder. The decoders translate x86 instructions into micro-operations (Mops). These Mops will log in reorder buffer to make sure that it can be retired in program order. Then these Mops wait in the reservation station until their source operands are all available. In each cycle, several Mops can be dispatched according to the number of function units available.

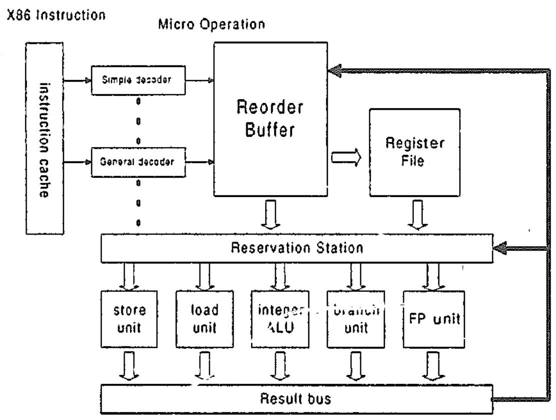


Figure 4: x86-to-RISC microarchitecture (simplified)

7.2 Implications of Benchmark Analysis

we analyze the instructions using frequency and find the implications related to four kinds of issues including (1). issues related to the decoders. (2). issues related to the function units. (3). issues related to the microarchitecture. (4). issues related to pairing of x86 instruction.

(1). Issues related to the decoders

One important design issue is to determine the right mix of x86 instruction decoders. Since general decoder is expensive to build, we assume that there is only one general decoder in any case. However, the ratio between simple and general decoders is yet to be decided.

In Pentium Pro and K5 processors, its simple decoder just can decode the simpie x86 instructions that will be translate to one MOP. The complex instructions that will be decoded to over 2 MOP are decoded by general decoder.

In the benchmarks, the usage ratio of simple instruction to complex instruction is 2.1:1. In addition, the complex instructions seldom appear as a pair in our instruction pair analysis.

Actually, Pentium Pro use two simple decoders and one general decoder. On the other hand, K5 want to have a better performance in decoding. K5 use three general decoders. But in the performance report, K5 does not have any benefit from this expensive design.

In NSC98 processor, the simple decoder is used to decode the x86 instruction that will be translate to one or two MOPs. The complex x86 instruction that will be decoded to over three MOPs are decoded by general decoder.

In NSC98 definition, the rate of x86 instruction use simple decoders to general decoder is 8:1. It is better to design simple and complex decoders to fit this rate.

Actually, NSC98 has 8 simple decoders and one general decoder. This design is fit to instruction rate. There is still a question about simple decoder design. Since the rate of one MOP to two MOPs in the benchmark is 3:1. In most case, there is just one MOP instruction was decoded by the simple decoder. It seems to spend too much cost to use such a simple decoder. But if we implement the decoders like Pentium Pro, the simple decoder of NSC98 just decode one MOP instruction per cycle. We may need three extra complex decoder to decode those x86 instruction that will be translate to over two MOP instructions. And these three extra complex decoders will cost much more than enhance the eight simple decoders to decode two MOP instructions per cycle.

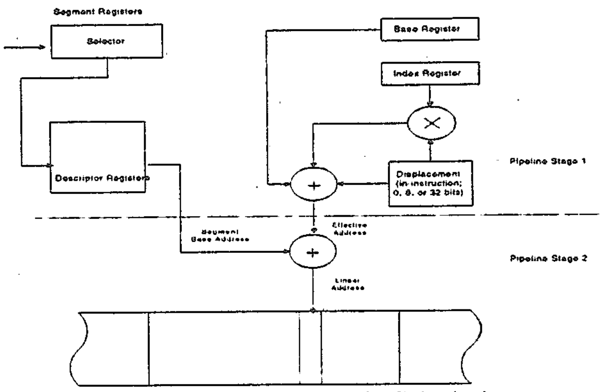
# of MOP a x86 inst. is mapped to	x86 Percentage
1	67%
2	22%
3	7%
4	3%
over 5	<0.5%

Table 3:Number of MOP a x86 instructions is mapped to

Some most used x86 instructions will be decoded to more than two MOPs. If this x86 instructions is very easy to implement in one micro operation. In order to have better performance we may add a new MOP that this x86 instruction can map one-to-one. The Pentium addressing mode calculation like below. Linear address will complete in two pipeline stages. First, to generate effect address (offset address) and select a base address in pipeline stage 1. Then the AGU could generate the linear address in pipeline stage 2.

In the case of NSC98, one of the most used x86 instruction - "PUSH"(5.7%) is decode to two micro instructions "SUBIN" and "ST". The Pentium addressing mode calculation like below. Linear address will complete in two pipeline stages. First, to generate effect address (offset address) and select a base address in pipeline stage 1. Then the AGU could generate the linear address in pipeline stage 2.

In the first cycle of AGU, it will add base register, index register, and displacement value in a three input adder. But when we access stack, there is just one input "sp" to this adder. We may substrate sp in this adder and output to result bus to update the SP value. Meanwhile, the SP value which is substrate by 2, will be send to next cycle to count the address of stack entry as original store MOP. There is almost no extra hardware to implement this MOP. It only needs a extra bus to result bus.



Pentium Addressing Mode Calculation

We may add a new micro instructions "PUSH" to micro instruction set and implement the instruction in the Load/Store unit. When we execute the x86 "PUSH" instruction we will reduce one cycle per instruction. The additional H/W in the AGU is minimal.

We calculate the improvement by the function $\frac{(reduced\_cycles\_per\_patten)*(pattern\_frequency)*(inst.\#)}{(inst.\#)*CPI} = \frac{1*5.7\%}{1.5} = 3.8\%$

Because P6 and NSC98 were full pipelined, so there is one MOPs executed every cycle in a function unit. The CPI is calculated by the function $\frac{all\_MOPs}{all\_X86\_inst} = 1.5$

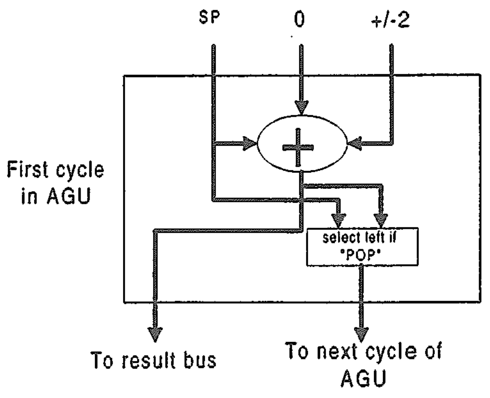
Because "substrate 2" is originally execute in ALU, implementation of this "push" MOPs will also reduce the usage of ALU by 7.1%

$(\frac{reduced\_MOPs}{MOPs\_use\_ALU} = \frac{(reduced\_X86inst\_ratio)*(X86inst.\#)}{(MOPs\_use\_ALU\_ratio)*(\#\_of\_MOP)} = \frac{5.7\%}{53.4\%} * \frac{1}{1.5})$ ●.

x86 instructions	Original MOPs	With new MOPs
PUSH AX	Subin SP,2 ➔ /*sp=sp-2 */ ST AGU(sp,0..).AX /*store AX to stack*/	PUSH AGU(sp,0,..).AX ➔ /*store AX to stack & sp=sp-2*/

Figure 5: MOPs of "push"

The other, one of the most used x86 instruction - "POP"(3.9%) is decode to two micro instructions "LD" and "ADDIN". It store the value to memory stack that was pointed by stack pointer and add the stack pointer by 2.



Implement push and pop in AGU

We may add this new micro instructions "POP" to micro instruction set and implement the instruction in the

● $(\frac{\#\_of\_MOP}{X86inst.\#} = 1.5)$ was discuss in section 7.10

Load/Store unit. When we execute the x86 "POP" instruction we will reduce one cycle per instruction. The additional H/W that complete in pipeline stage 1 in the AGU is minimal.

We calculate the improvement by the function $\frac{(reduced\_cycles\_per\_patten)*(pattern\_frequency)*(inst.\#)}{(inst.\#)*CPI} = \frac{1*3.9\%}{1.5} = 2.6\%$

x86 instructions	Original MOPs	With new MOPs
POP AX	➔ LD AX,AGR(sp,0...) /*load stack value to AX*/ ADDIN SP,2 /*sp=sp+2*/	➔ POP AGU(sp,0...),AX /*store AX to stack & sp=sp+2*/

Figure 6: MOPs of "pop"

Because "add 2" is originally execute in ALU, implementation of this "push" MOPs will also reduce the usage of ALU by 4.9% ($\frac{reduced\_MOPs}{MOPs\_use\_ALU} = \frac{(reduced\_X86inst\_ratio)*(X86inst.\#)}{(MOPs\_use\_ALU\_ratio)*(\#\_of\_MOP)} = \frac{3.9\%}{53.4\%} * \frac{1}{1.5}$)[■]. ALU is very busy function unit. If we implement both "push", "pop" will reduce the workload of ALU by 12.0% ($\frac{reduced\_MOPs}{MOPs\_use\_ALU} = \frac{9.6\%}{53.4\%} * \frac{1}{1.5}$).

(2). Issues related to the functions units

In different processor, there are different number of function units to execute MOPs (Table 4). We should make sure that there is enough function units.

Function unit	Number of function units in NSC98
ALU	4
Branch unit	1
Floating point unit	2
MMX unit	2
Load/store unit	2

Table 4: Number of function unit in NSC98

In the data we analyze, "ld" (19.2%) and "st" (10.2%) MOPs are used at very high frequency. These MOPs are executed in load/store unit. It is important to use enough Load/Store units to make these Mops executed without bottleneck.

In P6, it uses one load unit and one store unit. Because every cycle it will fire three MOPs from ROB to RS, we assume there are three MOPs will send to function unit per cycle. In average, there are 0.6 "ld" will be send to load unit and 0.3 "st" will be send to store unit. One load unit and one store unit are enough to handle "ld" (19.2%) and "st" (10.2%) in this case of three MOPs sent to function unit per cycle.

In NSC98, it use two load/store unit there are use 8 simple decoders and 1 complex decoder to make sure eight MOPs will be send to ROB per cycle. In average, there are 2.4 "ld" or "st" will be send to load/store units, but it just employ 2 load/store units. There are 0.4 "ld" or "st" have to wait function unit per cycle.

In NSC98, we may consider to add one load/store unit to make sure it can handle 3 (>2.4) "ld" and "st" per cycle. It will increase the number function unit from 10 to 11. We calculate the improvement by the function of $\frac{"LD/ST"\_used\_to\_delayed\_per\_cycle}{MOPs\_executed\_per\_cycle} = \frac{0.4}{8} = 5\%$

(3). Issues related to the microarchitecture

One of the most used instructions (11.1% in total MOP code, including of "movi") is "MOV R4, R3". It just copy the value of R3 to R4, is however like other complex instructions, it still needs to have a entry in ROB, RS, a function unit to execute it, and need to wait result bus to send data back or register file. We may simply this simple instruction in implementation.

Typically, this instruction was executed in the function unit as other instructions. Because the new architecture such as Pentium Pro and NSC98 have "Reorder Buffer(ROB)" to implement register renaming. And these register-to-register move instructions just copy the value of R3 to R4. We may take advantage of register renaming facility, and implement this instruction in the reorder buffer. This instruction will not need to stay in the Reservation Station to wait the function unit available. It just enters the reorder buffer after decoding, and waits in the reorder buffer until retired then write the value of the source register to the destination register.

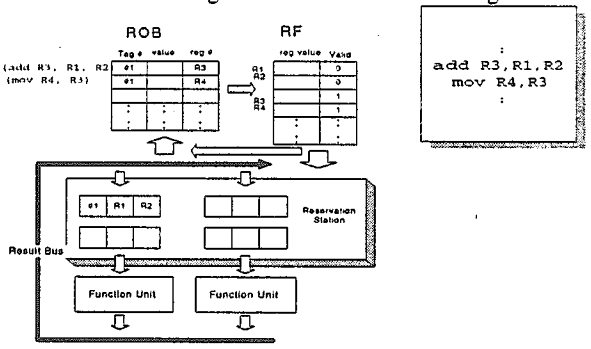


Figure 7 : Implement register-to-register MOV in ROB

For example (as Figure 7), After decoding, "MOV R4, R3" enters ROB, the value of source register R3 was fetched to ROB according to whether the value is available in the register file. If R3 is available in register file, then write R3 value to the "value" column of R4 row of this instruction in ROB and waiting to retire. When this instruction is in it's turn to retire. It will write the value of R3 to R4 in register file and finish the "MOV R4, R3".

If R3 is not available in the register file, then we write the tag identifier in the reservation station stands for R3 to the column value of this instruction. Once the tag value stands for R3 is calculated in function unit, the value will write back to ROB and replace the tag identifier. When this instruction is in it's turn to retire, it will write the value of R3 to R4 in register file and finish the "MOV R4, R3".

Since most of hardware needed to implement 'mov' operation within ROB has been available, we just need to add little control hardware. In the case of the value of source register is available, the only extra control hardware to implement this instruction in ROB is to make the value of source register write to ROB. The

[■] ($\frac{\#\_of\_MOP}{X86inst.\#} = 1.5$) was discuss in section 7.10

ROB to register file bus is exist (in order to implement renaming), but the bus is only one-way bus. We need to add a write port to register file and read port to ROB to make the value in register file can write to ROB. In order not to add a read/write port, we may add a new bus from write port of the register file to read port of ROB. The extra hardware will be very limited. In the case of the value of source register is not available, the extra hardware is to make the tag value write back the value which is calculated after function unit to possibly multiple destinations in ROB. But in design of ROB, it is support associate look up. So it may not need any change in ROB to support multiple destinations write back.

In the processor NSC 98, "MOV" instruction will be executed in ALU. Because this instruction is used at very high frequency (about 9.5%) and the arithmetic MOP instructions executed in the same ALU is also at very high frequency (53.4%). The possibility of waiting this function unit must be high. If we implement "mov" in ROB, the benefit will be significant. The "mov" instruction need not in reservation station to wait the ALU available. And the arithmetic MOP instructions executed will wait less time for ALU. It will reduce the 17.8% ($\frac{\#\_of\_mov}{\#\_of\_ALU\_MOP} = \frac{9.5\%}{53.4\%}$) of workload to ALU. It is over 1/4. In NSC98, we use 4 very complex ALUs, we may reduce one ALU with this implementation. The benefit is very large, because the ALU is very expensive.

8. Conclusions and future work

The x86 is a very popular and widely used microprocessor. We have written a monitor and used VTune to record the dynamic information about instruction usage. We used monitor to get the whole instruction trace of DOS applications and used VTune to sample the trace of Windows95 application then analysis it. This paper has reported the dynamic instruction frequencies and instruction pairs frequencies found in several programs running under the DOS and windows95.

We have noticed that 25 of the x86 mnemonic code constituted 90.1% of the benchmark execution. When considered op code, 25 of the x86 op code constituted 72.3% of the benchmark execution.

The most common instruction classified by functionality is "data transfer". The most common operand sizes are 16 bits in DOS and 32 bits in Windows95. The most common memory displacement size is 0 bits. The most common operand types are using a register operand and an immediate operand. The most common memory addressing mode are "bp + [disp]" of base addressing in 16 bits DOS code and 32 bits base addressing in 32 bits Windows95 code.

We have analyzed the benchmark and propose some suggestions to the microarchitecture design. Since every five instruction contains one branch instruction, it is necessary to employ a good prediction to avoid pipeline flush from branch prediction penalty. Too many x86 instructions use implicit registers as operand. It need use extra register to register data transfer instructions. We may make the implicit operand explicit. Three operands addressing mode is seldom use both in x86 instruction or Mops. Two operand addressing operands can be considered for the Mops in order to save the bit width of the Mops. The comparison instruction and condition jump usually appear in sequence. These two instructions may be combined into an instruction for the Mops. "ld" and "st" are used at very high frequency. It is need to enhance the memory interface to reduce memory latency. The better ratio of simple decoder to general decoder is 1: 2.

Our future work will automate the interfaces with instruction synthesis and instruction mapping tools. The instruction synthesis tool will create new instructions for the better performance. Our instruction analysis tools will recognize these new instruction and count the using frequency of the new instruction. The instruction mapping tool will create x86 to MOP mapping rule automatically and the MOPs using frequency for the new mapping rule will be generated in our integrated environment.

9. References

[1] Rakesh K. Agarwal, *80x86 Architecture & Programming* Volume II: Architecture Reference. Prentice Hall. 1991

[2] *Pentium Family User's Manual. Volume 3 Architecture and Programming Manual.* 1994

[3] J. M. Mellor-Crummey and T. J. LeBlanc . *A Software Instruction Counter.* ASPLOS-III Proceedings, 1989

[4] Thomas L. Adams and Richard E. Zimmerman. *An analysis of 8086 Instruction Set Usage in MS DOS Program.* ASPLOS-III Proceedings, 1989

[5] Mark Atkins and Ramesh Subramaniam. *PC Software Performance Tuning.* IEEE Computer, August 1996

[6] David A Patterson, John L Hennessy.*Computer organization & design the hardware/software interface.* Morgan Kaufmann

[7] David W. Wall. *Limits of Instruction-Level Parallelism.* ASPLOS-IV Proceedings, 1991

[8] Thomas Ball and James R. Larus. *Optimally Profiling and Tracing programs. ACM transaction on Programming Languages and System,* Vol.16, No. 4, July 1994

[9] Gurindar S. Sohi and Sriram Vajapeyam. *Tradeoffs in Instruction Format Design For Horizontal Architectures.* ASPLOS-III Proceeding 1989.

[10] Linley Gwennap. Intel's P6 Uses Decouple Superscalar Design. Microprocessor Report Vol. 9, No. 2, February 16, 1995