# Decoding of CISC instructions in superscalar processors with high issue rate

R.-M.Shiu, J.-C.Chiu, S.-K.Cheng and J.J.-J.Shann

**Abstract:** The paper examines the design issues of decoders, including the primitive operation (POP) translation strategies and the decoding rules, for CISC superscalar processors to exploit a higher degree of parallel execution. Attention is focused on the x86 instruction set because of its popularity. There are two different approaches regarding POP translation strategies: one is to merge the address generation into load/store operations, and the other is to translate the isolated address generation operations. Simulation results show that, in high issue-rate decoders, the latter strategy improves the performance by 20 to 25%. Furthermore, considering the tradeoffs between the hardware cost and performance, a cost-effective decoding rule suitable for current commercial programs is recommended.

## 1 Introduction

The CISC instruction sets, such as x86 [1]. VAX [2], and Java bytecode [3], possess individualities like variable length, complex format, and complex semantics. For those CISC processors that use modern superscalar techniques to achieve higher performance via dynamic scheduling and out-of-order execution containing multiple instruction in parallel [4–6], their decoding units (decoders) have to translate the instructions into primitive operations (POPs) using fixed instruction length, regular formats, and simple functions to simplify the pipeline design and dynamic scheduling mechanisms [7, 8].

We examine the design issues for the decoders of CISC superscalar processors to exploit a higher degree of parallel execution in conjunction with the POP translation strategies and the decoding rules. The x86 instruction set is selected as an example because of its popularity. The x86 superscalar processors, such as Intel Pentium and Pentium II/III, AMD K5/K6, Cyrix M1/M2 [1, 9–14], have dominated the PC market for years and now require more powerful decoding techniques to achieve higher issue rates in their next generation's outlook.

The main difference of the strategies to translate the x86 instruction is that some decoders merge the address generation into load/store operations, and others translate isolated address-generation operations. The current commercial x86 processors are using the former one entirely. In this work, we observed that, in current issue rate, merging the address generation into load/store operations can achieve higher performance since it saves on the number of POPs translated. When higher issue rate is required, however, the method of translating isolated address-generation operations to exploit a higher degree of parallel execution becomes important.

Decoding rules used by current x86 superscalar processors are focal points that we want to improve as well. The decoding rule decides what permutations of x86 instructions can be decoded in one clock cycle, which can limit the fetch rule [15]. In this paper, we examine different decoding rules and suggest a cost-effective one suitable for current commercial programs.

© IEE, 2000

*IEE Proceedings* online no. 20000450

*DOI:* 10.1049/ip-cdt:20000450

Paper first received 21st December 1998 and in final revised form 14th March 2000

The authors are with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 30050, R.O.C.
E-mail: jjshann@csie.nctu.edu.tw

## 2 Decoding rules of current x86 superscalar processors

The instructions of x86 can be classified into four different types as

(i) complex: must be decoded by micro-ROM.
(ii) general ($G$): is not necessary to be decoded by micro-ROM and will be translated up to 4 POPs.
(iii) type 1 simple ($S_1$): can be translated to only one POP.
(iv) type 2 simple ($S_2$): can be translated to 1 or 2 POPs.

A $S_1$ instruction can be classified as $S_2$, and a $S_2$ instruction can also be classified as G.

Since the complex instructions are decoded by the micro-ROM and cannot be contained in the decoding rules, we denote a set of decoding rule by listing the maximum value for of each type of instructions that can be decoded in one cycle and by listing the maximum numbers of x86 instructions being decoded ($I$) along with POPs being generated ($P$) in one cycle. Assume that the decoder can decode up to $m$ x86 instructions and translate them up to $n$ POPs. Within these $m$ instructions, $j$, $k$, and $r$ of them can be type G, $S_2$, and $S_1$, respectively ($m = j + k + r$). Then, the decoding rule is denoted as "$m$I:$j$G:$k$S$_2$:$r$S$_1$:$n$P."

Likewise, the decoding rules of the current x86 processors using the notation described can be summarised as follows. Pentium II/III is "3I:IG:0S$_2$:2S$_1$:6P", AMD K5 is "4I:4G:0S$_2$:0S$_1$:4P", K6 is "2I:0G:2S$_2$:0S$_1$:4P or 1I:1G:0S$_2$:0S$_1$:4P", and K7 is "3I:0G:3S$_2$:0S$_1$:6P or

1I:1G:0$S_2$:0$S_1$:6P". As for other x86 superscalar processors that do not provide out-of-order execution, such as Pentium, M1/M2, and Rise m6, they are unnecessary to translate instructions to POPs. Moreover, Pentium II/III translate store operations into two POPs by putting store address and data separately; and AMD K6/K7 merge load and store operations into one instruction to become a load-store POP. In this work, however, such details of translating POPs are neglected for simplicity.

## 3 Primitive operation translation strategies

The major function of the decoder with respect to an x86 superscalar processor is to translate x86 instructions into POPs. In this Section POP translation strategies are introduced.

### 3.1 Translating X86 instructions to primitive operations

Current x86 superscalar processors translate x86 instructions to POPs by using the same POP translation strategy as the one that merges address generation into load/store operations. Another strategy is to translate address generation to an isolated operation. We define these two POP translation strategies as

*NoAGU-POP:* The generation of the address for a load/store is included in the load/store POP (*LD/ST*).
*AGU-POP:* The generation of the address for a load/store is kept apart from the load/store POP and, therefore, it becomes an address generation POP (*AG*).

For example, the instruction *Add mem[BX+SI], AX* can be translated by these two strategies

(a) NoAGU-POP:

1. $LD\ temp1,\ mem[BX+SI];$ // $temp1 \leftarrow mem[BX+SI];$
2. $ADD\ temp2,\ temp1,\ AX;$ // $temp2 \leftarrow temp1 + AX;$
3. $ST\ mem[BX+SI],\ temp2;$ // $mem[BX+SI] \leftarrow temp2;$

(b) AGU-POP:

1. $AG\ temp1, BX, SI:$ // $temp1 \leftarrow [BX+SI]:$
2. $LD\ temp2, mem[temp1];$ // $temp2 \leftarrow mem[temp1];$
3. $ADD\ temp3, temp2, AX;$ // $temp3 \leftarrow temp2 + AX;$
4. $ST\ mem[temp1], temp3;$ // $mem[temp1] \leftarrow temp3;$

According to these two strategies two corresponding load/store unit (LSU) mechanisms are shown in Fig. 1. In Fig. 1*a*, the address generation function (AGF) is combined with the LSU. In Fig. 1*b*, the separated AGU calculates the addresses and forwards the addresses to the reservation station (RS) of the LSU.

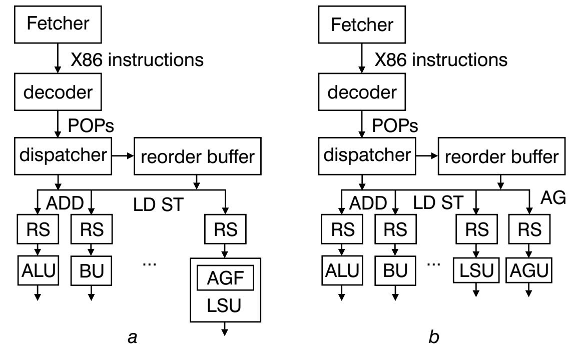


**Fig. 1** *Block diagrams of two LSU mechanisma*
*a* LSU with address generation (NoAGU-POP)
*b* Isolated AGU (AGU-POP)

### 3.2 Mechanisms of load/store units

Two different LSU mechanisms can be derived as a result of the two POP translation strategies mentioned. In regard to NoAGU-POP, the execution of an LD/ST requires three stages

(i) calculates the address of the LD/ST
(ii) checks the dependency between the LD/ST and the other LD/STs in LSU
(iii) access the cache data for the LD/ST.

As for AGU-POP, the execution of an LD/ST requires two cycles to execute stage (ii) and (iii). We find that enhancing the store buffer with the ability of snooping result buses is important for high issue-rate decoding units. Therefore we decided to build four LSU models: NoAGU_NoSNP, AGU_NoSNP, NoAGU_SNP, and AGU_SNP, for combining the strategies of POP translation (AGU or NoAGU) with the snooping ability of store buffers (SNP or NoSNP).

#### 3.2.1 Load/store units without snooping ability:

In Fig. 2 two LSUs are demonstrated with no snooping ability in the store buffers. In Fig. 2*a*, an *LD/ST* is dispatched to the RS and the RS snoops the result buses for the operands of the *LD/ST* POPs in the RS. An *ST* must

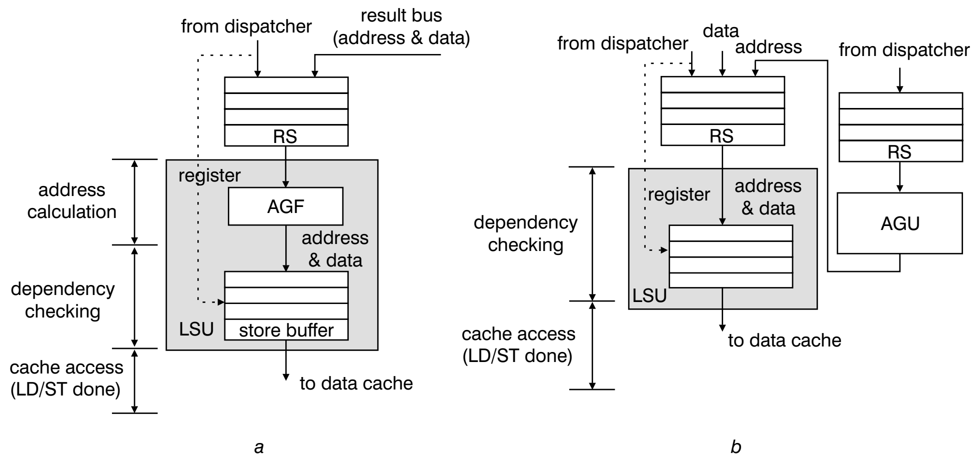


**Fig. 2** *LSUs in which store buffers have no snooping ability*
*a* NoAGU_NoSNP
*b* AGU_NoSNP

register in the store buffer to maintain the original order in the program. When all the operands of an *LD/ST* are ready, the *LD/ST* is issued to LSU. In Fig. 2*b*, an AGU is isolated to execute *AGs* and has its own RS. After the address of an *LD/ST* is calculated by AGU plus all its data operands are ready, then the *LD/ST* is issued to LSU.

Figs. 3*a* and 3*b* illustrate the execution flows of stream (a) and (b) mentioned in Section 3.1 for NoAGU_NoSNP and AGU_NoSNP, respectively. With respect to NoAGU_NoSNP, since both the data of the *ST* cannot be snooped by the RS of the LSU until cycle 5 and the store buffer cannot get the address for dependency checking until cycle 6, the next *LD* from other instruction can only be issued at cycle 7, if the same port is used. In contrast, for AGU_NoSNP, the store buffer can get the address of the *ST* for dependency checking at cycle 5, thus the next *LD* from other instruction can be issued at cycle 6. The execution path from an *LD* to the next *LD* is the critical path for the most part. Therefore such reduction of the cycles on this path can improve the performance significantly.

*3.2.2 Load/store units with snooping ability:* Figs. 4*a* and 4*b* depict NoAGU_SNP and AGU_SNP, respectively. In Fig. 4*a*, the store buffer snoops the result buses for the data operands of *STs* in the buffer. Therefore there is no need for RSs to wait for the data operands of *STs* and the RSs can soon pass the *STs* to AGU when all their operands for address calculation are ready. The dependency checking can be started earlier as well without waiting for the data operands. In Fig. 4*b*, the store buffer snoops the result buses for the address calculated by AGU. Therefore the store buffer can get the address without waiting for the data operands.

Figs. 5a and 5*b* show the execution flows of the stream (a) and (b) mentioned in Section 3.1 for NoAGU_SNP and AGU_SNP, respectively. In both situations, the store buffer can get the address far ahead in cycle 2, and thus the next *LD* can be issued early in cycle 3 for the same port. Since the execution path from an *LD* to the next *LD* is along the critical path, such cycle reduction can improve the performance significantly. In AGU_SNP, AG may be released from the critical path, and thus achieve higher performance.

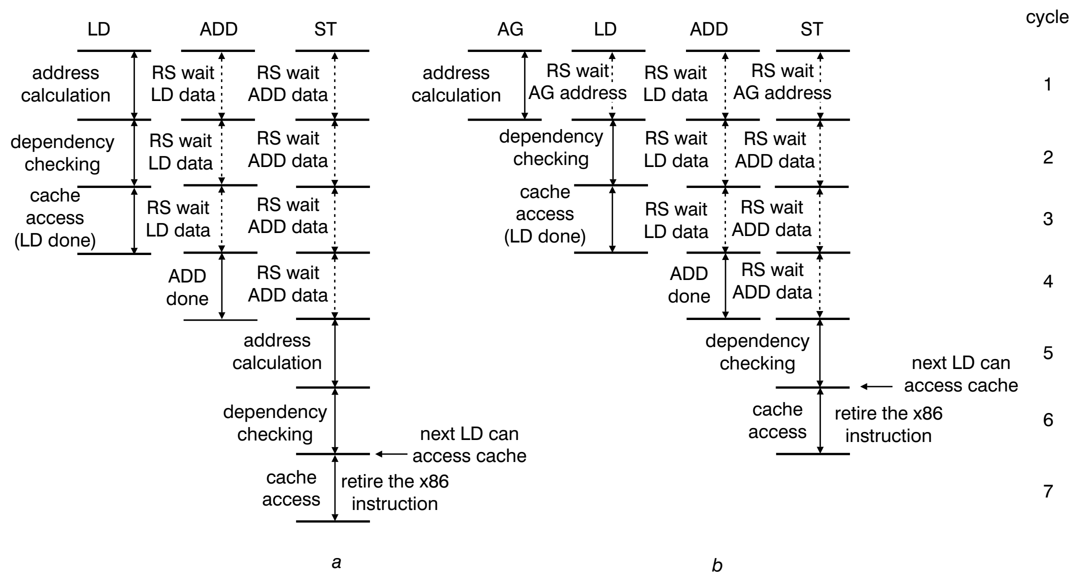


**Fig. 3** *Execution flows of instruction Add mem[BX+SI], AX for LSU models in which store buffers have no snooping ability*
*a* NoAGU_NoSNP
*b* AGU_NoSNP

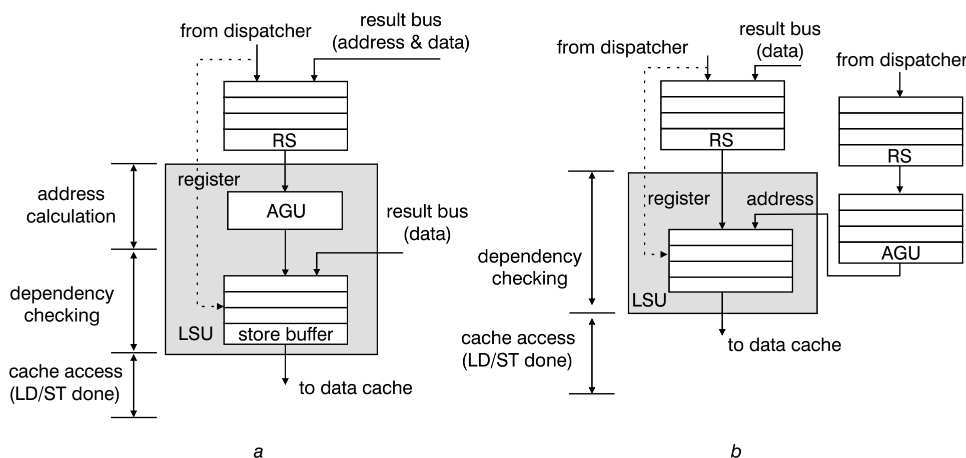


**Fig. 4** *LSU models in which store buffers have snooping ability*
*a* NoAGU_SNP
*b* AGU_SNP

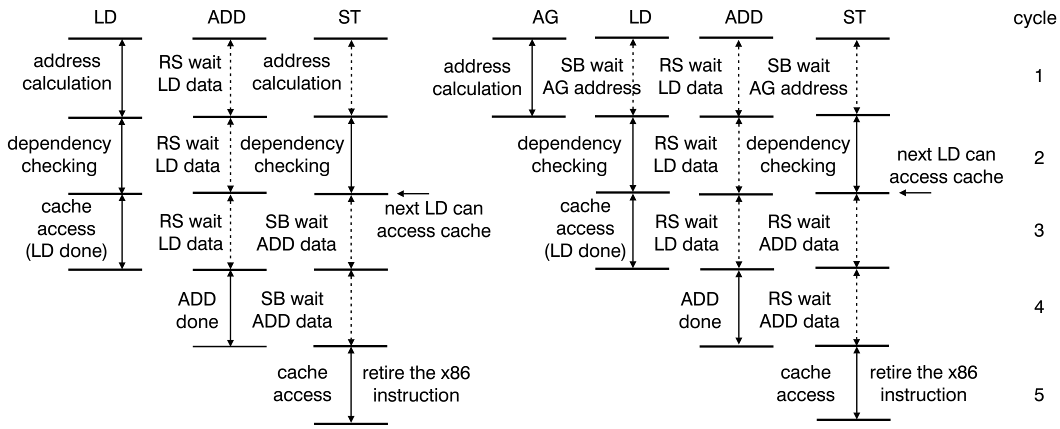


**Fig. 5** *Execution flows of instruction Add mem[BX + SI], AX for LSU models in which store buffers have snooping ability*
*a* NoAGU_SNP
*b* AGU_SNP

## 4 Performance evaluation and analysis

This Section examines the design issues of the decoder through simulation. First, we describe the simulation environment and assumptions presumed in this work. After that, the simulation results are analysed and demonstrated in conjunction with some suggestions.

### 4.1 Simulation environment and simulation model

We build our own trace-driven simulator whose environment is shown in Fig. 6. The SPECint95 benchmarks [14, 16] are compiled by GCC with their default optimisation flags and executed on Linux OS. Then the system call "ptrace" is used to get the trace files. We have run each benchmark until it terminates, and the buffers in the simulator are flushed every 500 thousand instructions to simulate the situation of context switches. The comparison of our simulation results are based on the total execution cycles of all the benchmarks.

Our simulation models used in this work are based on a superscalar processor containing six pipeline stages as shown in Fig. 7; the block diagram of the processor is shown in Fig. 8. The POPs are dispatched to the distributed RSs orderly for out-of-order issue. In addition, the POPs are orderly stored in the reorder buffer (ROB) and orderly retired.

The assumptions made in our simulator are as follows:

- the accuracy of branch prediction is assumed to be 100%.
- If the fetcher fetches a branch instruction and the branch is being taken, the succeeding instructions will not be issued to the decoder until the next cycle is reached.
- The size of ROB and each RS is unlimited.
- The number of execution units is unlimited.
- The number of load/store ports to L1 cache is unlimited.
- The latency of ALU, AGU and BU is one cycle.
- We adopt data forwarding for *LD*s in LSU.

Four LSU models described in Section 3.2 have been simulated. To clarify various parameters of our simulator, we set three simulation conditions and summarise them in Table 1. To compare the performances of these LSU models. NoAGU_NoSNP with four POPs is regarded as the baseline illustration.

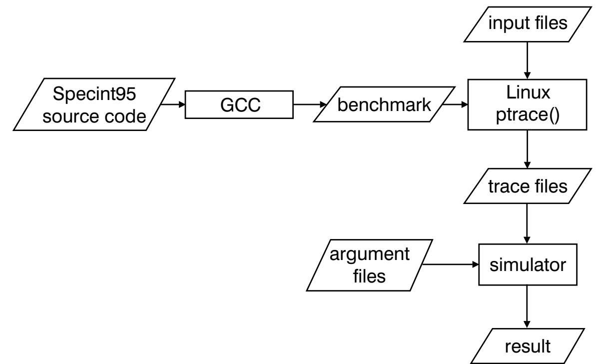


**Fig. 6** *Simulation environment*

fetch: fetch instructions from 1-cache

decode: decode predecoded x86 instructions into POPs

dispatch: schedule and dispatch POPs

RS: register POPs into reservation stations (RSs)

execution: execute POPs

retire: retire POPs

**Fig. 7** *Stages of pipeline for simulator*

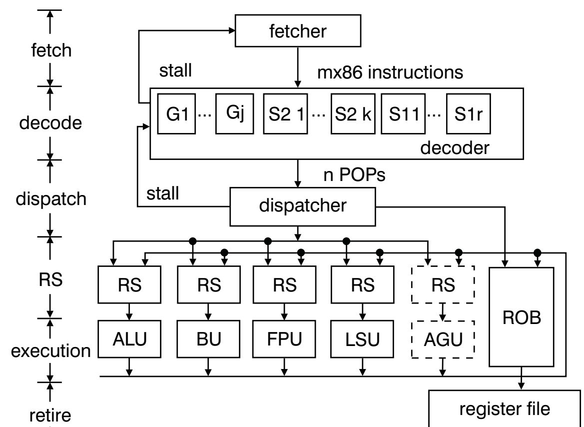


**Fig. 8** *Block diagram of simulator*

**Table 1: Parameter constrains under three different simulation conditions**

| Parameter | Unlimited condition | Fixed POPs and variable x86 instr. | Fixed POPs and x86 instr. with variable decoding rule |
|---|---|---|---|
| Strategy of POP translation | NoAGU-POP and AGU-POP | NoAGU-POP and AGU-POP | AGU-POP |
| Maximum x86 instr. fetching rate | unlimited | variable ($m$) | 5 |
| Maximum POP decoding rate | variable ($n$) | 8 | 8 |
| Maximum POP dispatching rate | $n$ | 8 | 8 |
| Maximum POP retiring rate | $n$ | 8 | 8 |
| LSU snooping ability | SNP and NoSNP | SNP | SNP |
| Maximum number of general instructions | unlimited | unlimited | variable ($j$) |
| Decoding rule | $n$I:$n$G:0$S_2$:0$S_1$:$n$P | $m$I:8G:0$S_2$:0$S_1$:8P | 5I:$j$G:0$S_2$:(5 − $j$)$S_1$:5P<br>5I:$j$G:(4 − $j$)$S_2$:1$S_1$:8P |

### 4.2 Determining number of POPs

We would like to find the saturation number of POPs generated by the decoding unit under the unlimited condition. The simulation results are shown in Fig. 9. The performance of the models in which the store buffers have snooping ability is better than those that have no such ability. Essentially, AGU_SNP has the best performance. However, as the maximum number of the decoded POPs is less than five, the performance of NoAGU_SNP is better than that of AGU_SNP. This is due to the fact that the total number of POPs with respect to AGU_SNP has increased abruptly (about 42%), thus restraining the speedup capability because of the limited of decoder's bandwidth. Based on these results, our recommendation is that AGU_SNP is suitable for high issue rate and it can be saturated at eight POPs under unlimited condition.

### 4.3 Variable number of X86 instructions

We would like to determine the saturation numbers of x86 instruction under eight POPs. Fig. 10 shows the speedup for different numbers of x86 instructions decoded in these four LSU models. It can be seen that AGU_SNP is saturated at five instructions and others are saturated at three.

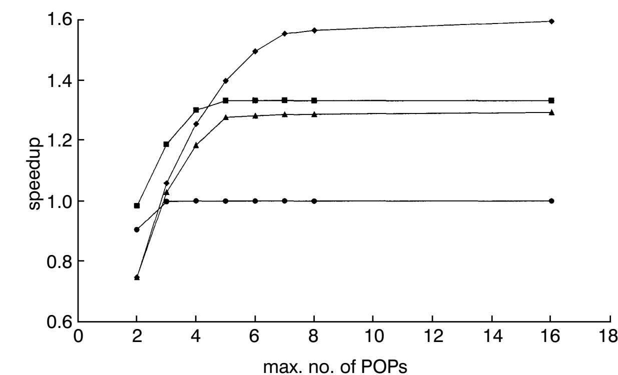


**Fig. 9** *Performances of four LSU models under unlimited condition*

–◆– AGU_SNP
–■– NoAGU_SNP
–▲– AGU_NoSNP
–●– NoAGU_NoSNP

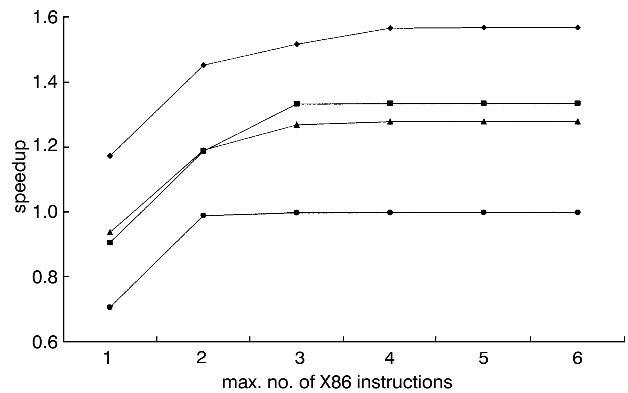


**Fig. 10** *Speedup for different numbers of x86 instructions with 8 POPs generated*

–◆– AGU_SNP
–■– NoAGU_SNP
–▲– AGU_NoSNP
–●– NoAGU_NoSNP

### 4.4 Determining decoding rules

In the previous simulation the decoders use the loosest decoding rules. In this Section we examine the different decoding rules under five x86 instructions and eight POPs using AGU_SNP, which are suggested in previous Sections. We focus on two decoding rule sets: "5I:$j$G:0$S_2$:(5 − $j$)$S_1$:8P" as rule set 1 and "5I:$j$G:(4 − $j$)$S_2$:1$S_1$:8P" as rule set 2. In both rule sets, the main parameter to be evaluated is the number of general instructions $j$ allowed to be decoded. Besides the general instructions, rule set 1 and rule set 2 allow a sequence of $S_1$ and $S_2$ type instructions to be decoded, respectively.

Fig. 11 shows the performance of the decoding rules for different numbers of G instructions with AGU_SNP. Notice that the performance is saturated at two G instructions. This result arises from the fact that one-POP and two-POP x86 instructions appear most frequently (more than 80%) in our traces, as shown in Fig. 12.

In Fig. 13 two block diagrams originated from decoding rules "5I:2G:2$S_2$:1$S_1$:8P" and "5I:1G:3$S_2$:1$S_1$:8P" are illustrated to compare the costs of decoders needed to allow one against two general instructions. Since the ratio regarding the sizes of translation table 1, 2, 3, and 4

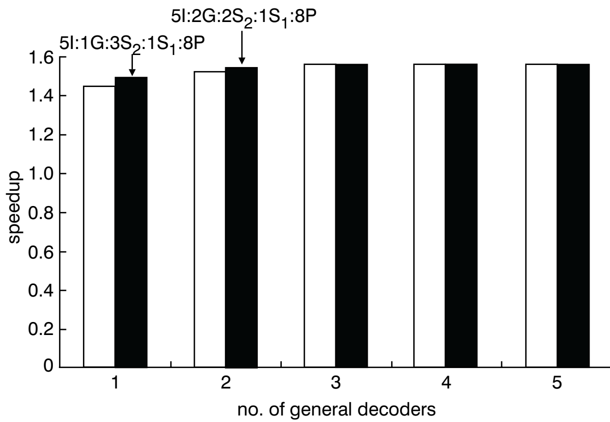


**Fig. 11** *Number of general instructions for decoding rules of AGU_SNP*

☐ 5I:$j$G:0S$_2$:(5-$j$)S$_1$:8P
■ 5I:$j$G:(4-$j$)S$_2$:1S$_1$:8P

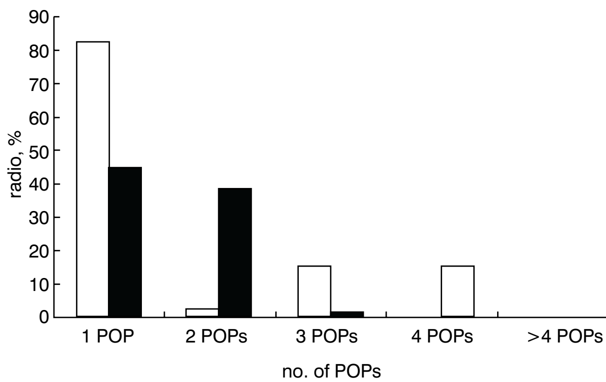


**Fig. 12** *Ratio of POP number*

☐ NoAGU
■ AGU

is about 11:9:8:2, the total sizes of the tables for the decoder in Fig. 13*a* is about twice as those in Fig. 13*b*. On the other hand, the ratio of the crossbar-area between the fetcher and the decoder in Fig. 13*a* and 13*b* is about 1.5:1. Thus we need about 100% extra translation tables and 50% extra crossbar area to achieve this additional 5% improvement. Considering the hardware cost, we suggest the decoding rule of "5I:1G:3S$_2$:1S$_1$:8P" in AGU_SNP.

## 5 Conclusions

We have examined two important issues, the POP translation strategies and the decoding rules, for the design of decoders in x86 superscalar processors. It is concluded that separating the address generation from a load/store operation to become another POP can achieve higher performance in high issue-rate microprocessors. And the capability for store buffers to snoop the result buses can further exploit a higher parallel execution degree. Furthermore, a cost-effective decoding rule suitable for high issue-rate microprocessors is recommended.

The CISC instruction sets possess good application compatibility and code density. Due to the recent advancement in silicon technology, sophisticated decoding units are capable of handling the complexity of instruction set. Prudent design of the decoding units has become very critical as the issue rate of a superscalar processor is increased.

In this research we work on the development of exploiting the machine parallelism for the next generation ×86 microprocessors without the support of compiler techniques. If the instruction-level parallelism can be improved by using compiler techniques, the POP translation strategies we proposed will become more important, and the decoding rules will be combined in conjunction with a software scheduling rule to achieve higher performance.

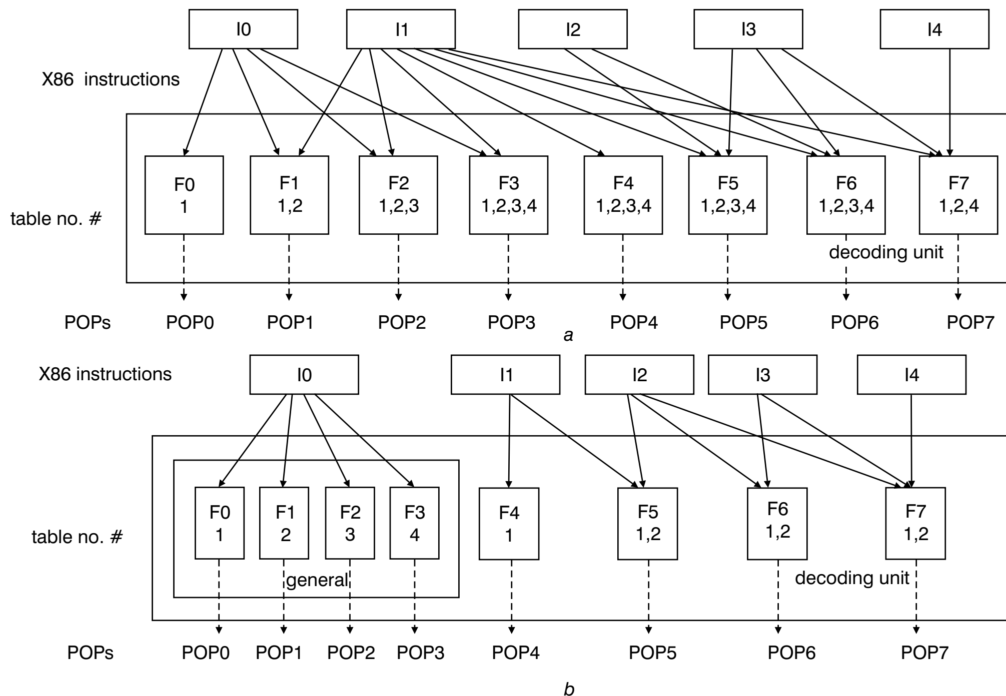


**Fig. 13** *Design schemes of two decoding rules under AGU_SNP*

10, 11: general instructions
12, 13: S$_2$ instructions
14: S$_1$ instructions
*a* 5I:2G:2S$_2$:1S$_1$:8P
*b* 5I:1G:3S$_2$:1S$_1$:8P

## 6 Acknowledgments

This paper presents partial result of a long-term research project financed by both the NSC of R.O.C. under contract NSC 86-2622-E-009-009 and the industry.

## 7 References

1 INTEL. 'Pentium II processor family developer's manual'. Oct. 1997
2 WILSON, J.E., MELVIN, S.W., SHEBANOW, M.C., HWU, W.M., and PATT, Y.N.: 'On tuning the microarchitecture of an HPS implementation of the VAX'. Proceedings of the 20th international symposium on *microarchitecture*, MICRO-20, 1987, pp. 162–167
3 LINDHOLM, T., and YELLIN, F.: 'The Java virtual machine specification' (Addison-Wesley, 1997)
4 JOHNSON, M.: 'Superscalar microprocessor design' (Prentice Hall, 1991)
5 SHRIVER, B., and SMITH, B.: 'The anatomy of a high-performance microprocessor: A systems perspective' (IEEE Computer Society Press, 1998)
6 INTRATER, G., and SPILLINGER, I.: 'Performance evaluation of a decoded instruction cache for variable instruction-length computers'. Proceedings of 19th international symposium on *Computer architecture*, ISCA '92, 1992, pp. 106–113
7 SMOTHERMAN, M., and FRANKLIN, M.: 'Improving CISC instruction decoding performance using a fill unit'. Proceedings of 28th international symposium on *Microarchitecture*, MICRO-28, Nov. 1995, Ann Arbor, Michigan, USA, pp. 219–229
8 MELVIN, S.W., SHEBANOW, M.C., and PATT, Y.N.: 'Hardware support for large atomic units in dynamically scheduled machines'. Proceedings of 21st international symposium on *Microarchitecture*, MICRO-21, 1988, San Diego, California, USA, pp. 60–113
9 PAPWORTH, D.: 'Tuning the Pentium Pro microarchitecture', *IEEE Micro*, 1996, **16**, (2), pp. 8–15
10 SLATER, M.: 'AMD's K5 designed to outrun pentium', *MicroProcessor Report*, 1994, **8**, (14)
11 CHRISTIE, D.: 'Developing the AMD-K5 Architecture', *IEEE Micro*, 1996, **16**, (2), pp. 16–26
12 AMD INC.: AMD-K6 MMX Enhanced Processor data sheet, 1997
13 CYRIX: Cyrix 6 × 86MX Processor, July 15, 1997
14 CASE, B.: 'SPEC95 Retires SPEC92,' MicroProcessor Report, 21 Aug. 1995
15 WALLACE, S., and BAGHERZADEH, N.: 'Modeled and measured instruction fetching performance for superscalar microprocessors', *IEEE Trans. Parallel Distrib. Syst.*, 1998, **9**, (6), pp. 570–578
16 POSTIFF, M.A., GREENE, D.A., TYSON, G.S., and MUDGE, T.N.: 'Limits of instruction-level parallelism in SPEC95 applications'. Presented at the international conference on *Architectural support for programming languages and operating systems*, ASPLOS-VIII, Oct. 1998