

Design of Instruction Address Queue for High Degree X86 Superscalar Architectures

JIH-CHING CHIU, MICHAEL JIN-YI WANG AND CHUNG-PING CHUNG

Department of Computer Science and Information Engineering

National Chiao Tung University

Hsinchu, 300 Taiwan

E-mail: chiujihc@ee.nsysu.edu.tw

A major hurdle of recent x86 superscalar processor designs is limited instruction issue rate due to the overly complex x86 instruction formats. To alleviate this problem, the machine states must be preserved and the instruction address routing paths must be simplified. We propose an instruction address queue, whose queue size has been estimated to handle saving of instruction addresses with three operations: allocation, access, and retirement. The instruction address queue will supply the stored instruction addresses as data for three mechanisms: changing instruction flow, updating BTB, and handling exceptions. It can also be used for internal snooping to solve self-modified code problems. Two CISC hazards in the x86 architectures, the variable instruction length and the complex addressing mode, have been considered in this design. Instead of the simple full associative storing method in lower degree (< 4) superscalar systems, the line-offset method is used in this address queue. This will reduce by 1/3 the storage space for a degree-5 superscalar x86 processor with even smaller access latency. We use synthesis tools to analyze the design, and show that it produces optimized results. Because the address queue design can keep two different line addresses in an instruction access per cycle, this method can be extended for designing a multiple instruction block issue system, such as the trace processor.

Keywords: address queue, ILP, superscalar processor, x86 architecture, multiple instruction issue

1. INTRODUCTION

In a superscalar microprocessor the instruction fetcher must provide multiple instructions to the decoders on each clock cycle. However, in the x86 architecture the variable instruction lengths and complex addressing system make fetching multiple instructions per cycle difficult [1]. In order to overcome these performance barriers, instruction predecoding is employed to help identify multiple instructions and assign duplicated instructions to decoders, as can be seen in Intel Pentium Pro [9], AMD K5 [7], K6 [6] and K7 [5] superscalar processors. Yet another difficulty is how to preserve in-order machine states. In von Neumann machines the program counter and memory contents define the machine states. Keeping the current instruction address until the instruction retires is necessary for such processors.

Instruction addresses are sometimes needed as the one of operands. Basically, the instruction addresses are so used in cases of

Received April 10, 2000; revised January 15, 2001; accepted May 22, 2001.

Communicated by Lionel M. Ni.

1. Change of instruction flow: Upon change of instruction flow, the instruction address is needed for calculating a new program counter value. For example, when a branch with relative addressing is executed or a branch misprediction is detected, the branch instruction's address is needed for generating the next fetch address.
2. Update of BTB: Whenever the Branch Unit executes a branch instruction, it must in the meantime update the BTB with the branch instruction address so that later branches may be predicted with higher accuracy [4].
3. Exception: When an interrupt or exception occurs, a corresponding Interrupt Service Routine (ISR) will be invoked. Upon completion of the ISR, the excepted instruction's address is needed for resuming normal execution [3].

While instruction addresses may be needed by any instruction, carrying the instruction address with the instruction itself through the pipeline is not a very good idea, since it will increase the routing complexity, and hence requires more silicon area. In a 32-bit n -issue superscalar microprocessor carrying the instruction address with the instruction requires an additional bus of $32*n$ bits wide. For example, if the superscalar degree is 4, the additional hardware cost is a $4*32$ -bit bus. And, unlike the operands, the instruction addresses are seldom used. So, when the superscalar degree is increased, reducing this hardware cost becomes important. On the other hand, we can store the instruction addresses in the instruction fetcher, which is at the front end of the pipeline, and when an address is needed, the functional units can acquire it from the instruction fetcher by a separate bus. With this design, the routing complexity can be greatly reduced. Moreover, because the x86 architecture allows self-modifying code [3], the addresses of the instructions in the pipeline must be checked when a write-to-memory operation occurs [4]. This can enhance the performance of a one-segment OS, such as OS2, whose instruction codes and data are stored in the same segment [4].

In RISC superscalar architectures instruction addresses can be easily expressed with an index of the instruction within the cache line [1] if the cache-line address is known. But in the x86 superscalar architectures, because of variable-length instructions, split-line problems and the complex addressing system, instructions handled in the same clock cycle may have different cache line addresses and lengths. Therefore, proper maintenance of the instruction addresses is an important task for high issue rate x86 processors. In this paper we propose such a design, called the Address Queue, for x86 superscalar processors. This design is used to store instruction addresses in the fetcher and to reduce the routing complexity for propagating instruction addresses in the pipeline.

2. BASIC DESIGN PHILOSOPHIES OF THE X86 ADDRESS QUEUE

One noticeable feature of the x86 architecture is that it allows self-modified code. The instructions that are buffered in the processor may have been modified in memory and, hence, be incorrectly executed. In order to solve memory accesses must be snooped to check if there are any write-to-memory operations that modify the code being buffered within the processor. Instruction addresses are also the information to be snooped and checked against.

In a 32-bit microprocessor, the address of an instruction is 32 bits wide. Carrying the address of an instruction together with the instruction itself increases routing complexity.

However, during program execution, the majority of executed instructions do not use their addresses. In addition in x86 microprocessors that employ the instruction translation technique, an x86 instruction may be converted into several ROPs, and the instruction address must be duplicated. This further deteriorates the problem. On the other hand, if instruction addresses are stored in a queue in the instruction fetcher, and only pointers to the queue are propagated with the instructions, the routing complexity can be reduced dramatically. Moreover, checking for self-modified code can simply be done in the Address Queue by comparing addresses stored in the Address Queue against the snooped write addresses. We use a queue to store the addresses because of its first-in-first-out feature. In order to enable precise interrupt in an out-of-order execution microprocessor, exceptions are handled just before the faulting instruction retires. Hence, the address must not retire before its instruction retires. Since the instruction retirement is in-order, the address retirement must be in-order, too. The FIFO property of a queue is appropriate for the in-order retirement. Finally, checking for the self-modified code can be done in the Address Queue by simply adding comparators in the Address Queue.

2.1 Address Space Problems With the x86 Architecture

The address space of the x86 architecture is manipulated with the segmentation mechanism. (The x86 architectures also implement the demand-paging mechanism, which is optional. The segmentation mechanism, however, may not be disabled [3]). This complicates the address storage problem. The address space before segmentation is called the “linear address space,” and the address space after segmentation is called the “effective address space”. While the L1-cache is accessed with the linear address, the internal program counter (EIP) is based on the effective address space. Therefore, the fetcher must translate addresses between these two address spaces. This translation is performed by adding (or subtracting) the Segment base address to (or from) the EIP (or the linear address). The result is the linear address (or the EIP). This process is shown in Fig. 1. Also shown in Fig. 1 is the paging scheme.

Although the translation for segmentation is not very complex, it may cause the start points of the two address spaces to be unaligned with each other, as Fig. 2 shows. The misalignment of two address spaces makes address storage more complex.

To reduce the space needed for address storage, several instructions in the same cache line can share the storage space of their line addresses. Only the offset of the instructions within the cache line must be stored separately. Ignoring processing of a split-line instruction, all the instructions processed in current clock cycle have the same cache-line address. And, if we process a split-line instruction, only the first instruction processed (which may be a split-line instruction) may have a different cache-line address while all other instructions processed in the current clock cycle have the same cache-line address. This is not the case in the effective address space. Two instructions in the same cache-line may be on different EIP lines. An example is shown in Fig. 2. Instructions A and B both are on the cache line labeled “Linear Line i ”, but instruction A is on “EIP Line j ,” while instruction B is on “EIP Line $j + 1$ ”.

Thus, the calculation of EIP-line address must take this fact into consideration. For simplicity the two EIP lines corresponding to the instructions on the same cache line are called the “EIP line 1” and “EIP line 2”. If the instruction’s offset (of the linear address)

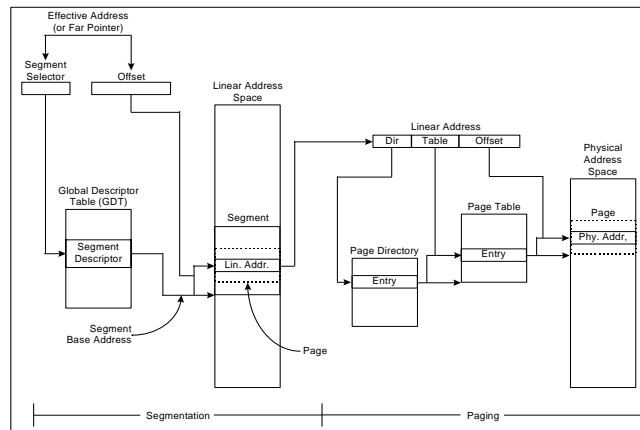


Fig. 1. Segmentation and paging.

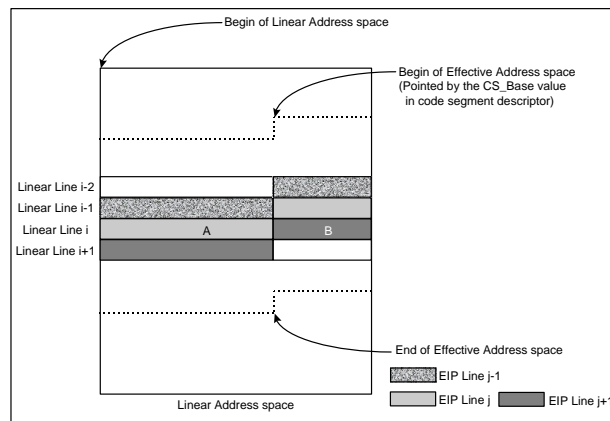


Fig. 2. Misalignment of linear and effective address spaces.

is less than the five LSBs of the CS_Base, it must be on the “EIP line 1” in the effective address space. Instruction A in Fig. 2 is such an example. Otherwise, if the instruction’s offset (of the linear address) is greater than or equal to the five LSBs of the CS_Base, it must be on the “EIP-line 2” in effective address space. And, Instruction B in Fig. 2 is such an example.

2.2 Address Queue Behavior and Characteristics

Before giving the detailed design of the Address Queue, we first describe the behavior and characteristics of the Address Queue. Although the addresses of instructions enter and leave the Address Queue in the order they are fetched in (i.e. the original program order), they may be accessed randomly. Some of the addresses may never be accessed, while others may be accessed several times. For example, an ALU instruction that does not cause an exception may never need its address, while a branch instruction that

causes an exception may need its address several times (when the target is calculated and when the exception is handled). Therefore, the Address Queue must support not only the basic first-in-first-out queue-style access. Four other operations on the Address Queue are defined:

1. **Allocate:** Allocate is the operation that en-queues the addresses from the instruction fetcher into the Address Queue. For each queued address, the Address Queue returns a pointer to the allocated queue entry. This pointer is passed to the succeeding stages of the pipeline together with the instruction. Since several instructions may be fetched during at each clock cycle, the allocation control of the Address Queue must be able to allocate several entries for the addresses in one clock-cycle.
2. **Access:** This operation reads the queued addresses from the Address Queue. The functional units requesting the addresses must send pointers pointing to the entry. Since different functional units may request instruction addresses at the same time, the access control of the Address Queue must have one port for each candidate functional unit. However, it is meaningless for a functional unit to access two instruction addresses in the same clock cycle. (The branch unit is limited to predict only one branch in a clock cycle, and only the earliest exception should be handled in order to maintain the precise interrupt). Thus, unlike the allocation and retirement operations, the access ports output only one address each time they are accessed.
3. **Retire:** Retire is the operation that de-queues the addresses in the Address Queue. The retire operation should take place in pace with the instruction's retirement in the reorder buffer. It is initiated by a signal from the reorder buffer. Since several instructions may be retired each clock cycle, the retirement control of the Address Queue must be able to retire several instruction addresses simultaneously.
4. **Snoop:** The Address Queue must compare the data write addresses against the addresses stored in it as described earlier.

Fig. 3 shows the operations with the related functional units.

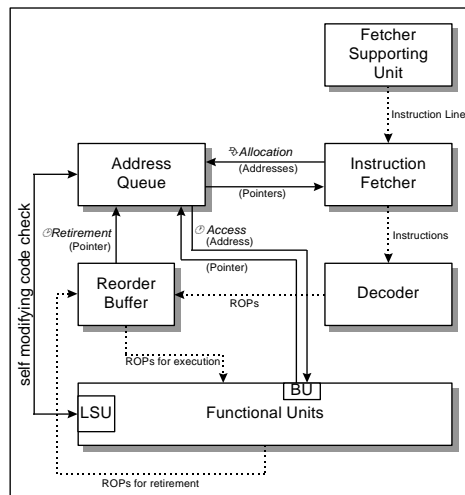


Fig. 3. Operations of the address queue.

3. THE ADDRESS QUEUE STRUCTURE

In the x86 superscalar architecture many instructions are fetched in one cycle, and their addresses will be saved to the Address Queue. To simplify the write-path control a set buffer is used to store these addresses each clock cycle. Because instruction issue rules limit the number of fetched instructions, often a fewer number than the maximum permissible number of superscalar-degree instructions are fetched. Some buffer space is wasted. For handling exceptions we must keep a strong sequential relation among instruction addresses. Hence, address queue entry $i + 1$ must help store the address of the succeeding instruction to entry i . However, allocating a variable number of entries makes the allocation operation very complex. It must first decide the number of entries to be allocated according to the results of the Information Maintenance unit. Then it must route the addresses to the selected entries. This requires a complex crossbar network. Such a complex operation is time-consuming and may form a critical path in the instruction execution pipe. For performance and cost tradeoffs, the Address Queue size and organization are analyzed in the following sections.

3.1 Analysis of the Address Queue Size

All instructions must store their addresses in the Address Queue before their retirement. When the Address Queue is full, the instruction fetcher must be stalled, otherwise the forthcoming instructions would not be able to retrieve their addresses when needed. As a result, the size of the Address Queue impacts the performance of the instruction fetcher significantly. If it is too small, it may frequently stall the instruction fetcher and become a bottleneck. Therefore, the Address Queue size needs to be considered carefully. Two parameters are related to Address Queue size, the number of sets in the Address Queue, and the number of entries in each set. The number of entries in each set is equal to the maximum number of instructions the fetcher can handle plus one.

To prevent the Address Queue from becoming a performance bottleneck in the microprocessor, it must be large enough so that it seldomly stalls the instruction fetcher. Since instruction addresses are stored when their corresponding instructions are fetched, the Address Queue must be able to store any instruction addresses unless no fetch can be issued from the instruction fetcher, i.e. when the reorder buffer is full, when reservation stations are full, or when a serializing instruction is being execution. In a superscalar environment, there is usually more than one instruction entering the reorder buffer each clock cycle. Therefore, if the number of sets in the Address Queue is equal to the number of entries in the reorder buffer, the utilization will be very low. A more reasonable design is needed. The Address Queue size can be expressed by a function of these factors. The conceptual estimation function is:

$$Q_{size} = \left\lceil \left[\frac{ROB_{size}}{Deg_{x86} \times ROP_{avg}} \right] \right\rceil \times (1 - RS_{stall}) + (F + D) \quad (\text{Eq. 1})$$

where

Q_{size} is the size of Address Queue (in # of sets)

ROB_{size} is the size of reorder buffer (in # of ROPs)

Deg_{x86} is the average superscalar degree of x86 instructions

ROP_{avg} is the average number of ROPs each x86 instruction corresponds to

RS_{stall} is the ratio of stall cycles caused by reservation station shortage

F is the number of pipeline stages the instruction fetcher uses

D is the number of pipeline stages the decoder uses

This function is expressed in three parts. The first part is in the first pair of parentheses, which is the reorder buffer size divided by the superscalar degree in number of ROPs. The second part is in the second pair of parenthesis, which is the fraction of clock cycles in which the reservation stations are available. The third part is the total number of pipeline stages for instruction fetching and decoding.

The first part gives the average number of clock cycles needed for filling the reorder buffer. This is the reasonable size for the Address Queue if only the reorder buffer is considered, because the reorder buffer will be full after this number of clock cycles if no instructions in the reorder buffer are retired. The reorder buffer will not fetch any further instructions and thus the Address Queue, full or not, will not cause a bottleneck. The second part takes the reservation stations into consideration. The reason for this is that in a clock cycle, if the reservation stations are full while the reorder buffer is not, then one stall cycle is incurred. If one-twentieth of the clock cycles are in this condition, then one-twentieth of the accessed instruction set will not be used, and hence, can be eliminated. Note that the RS_{stall} excludes stall cycles caused by reorder buffer full. The third part represents the instructions currently in the instruction fetcher and decoders, which are not yet logged in the reorder buffer. These instructions must still be considered separately.

The Deg_{x86} and POP_{avg} of the SPEC95 benchmark suite in our design, whose superscalar degree is 5 instructions, are 2.68 and 1.39, respectively. The RS_{stall} is very small such that the second term can be ignored. In our design both F and D equal 1. From equation 1 the number of sets in the Address Queue should be $[(reorder\ buffer\ size / 3.7252) + 2]$. Let the reorder buffer size be 64. Then the first term equals 17.18, and the resulting Address Queue set number is twenty.

3.2 Analysis of the Address Queue Organization

The organization of the Address Queue is shown in Fig. 4. It consists of two parts, the controller and the storage.

The controller has three functions, allocation selection, queue status, and retirement selection. Furthermore, it supports access control for other units with the queue pointer. The allocation selection function is invoked by an instruction address store request, and returns the queue pointer to each request. The queue status function maintains the age records of the allocated sets for the set retirement reference, and checks if the queue is full. The retirement selection function keeps the retirement order of sets. If a pointer is sent in for the retirement request, the older sets must be retired.

The storage stores EIP and the PC, and is divided into several sets. Each set can handle the storage of the instructions in one superscalar instruction fetching, and the number of sets can be evaluated by Eq. 1.

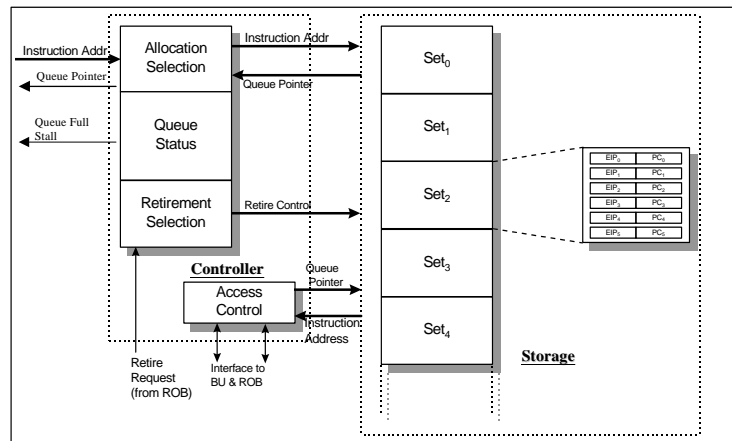


Fig. 4. Organization of the address queue.

4. X86 ADDRESS QUEUE IMPLEMENTATION

The basic design philosophy of the Address Queue is described in the previous sections. This section gives the implementation details of the x86 Address Queue with the six-address buffer set. The whole structure of the Address Queue implementation is shown in Fig. 5. We first introduce the Address Queue Controller. Then we give two implementations of the Address Queue storage.

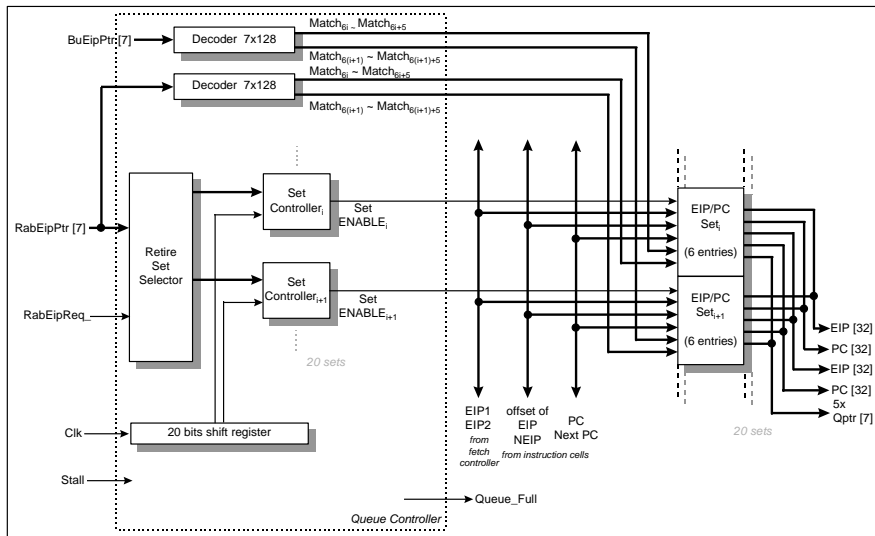


Fig. 5. Address queue implementation.

Assume an x86 superscalar degree of 5, and that the reorder-buffer has 64 entries, each of which can hold one ROP. From the discussion in previous section, there should be twenty sets of storage spaces in the Address Queue. Since at most five instructions are fetched on each clock cycle, each set contains six effective address entries and six linear address entries. As long as the instruction fetcher is not in stall state, a set will be used in each clock cycle.

Both the reorder buffer and the branch unit will access the Address Queue, and they will not communicate with each other before accessing the Address Queue. In order to avoid conflicts there are two independent output ports for the two units. Each port contains one 32-bit effective address output and one 32-bit linear address output.

4.1 Address Queue Controller

For each instruction fetched by the fetcher, both the linear address and the effective address must be stored in the Address Queue. Since the two addresses of the same instruction are always manipulated (en-queued or de-queued) at the same time, a common controller is adequate. Therefore, the Address Queue Controller, shown in Fig. 6, controls the effective address storage and linear address storage simultaneously. Its operations are described as follows:

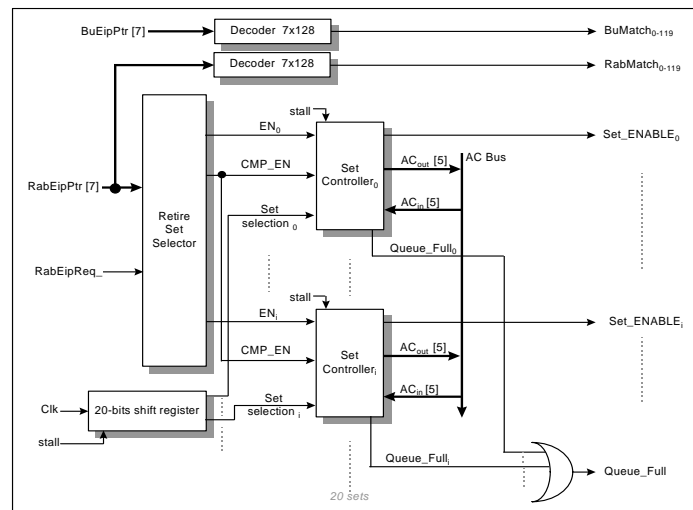


Fig. 6. Address queue controller.

1. Allocation: As described previously, when the instruction fetcher is not in the stall state, on each clock cycle one set of storage addresses is used. A 20-bit shift-register containing a single “1” and nineteen “0”s is used to indicate which set is to store addresses in current clock cycle. The 20-bit shift register corresponds to the twenty sets of Address Queue storage. This shift-register is initialized to “0...01”, meaning the first set is chosen. Each clock cycle, as long as the instruction fetcher is not in the stall state, the shift register rotates its content left one bit to

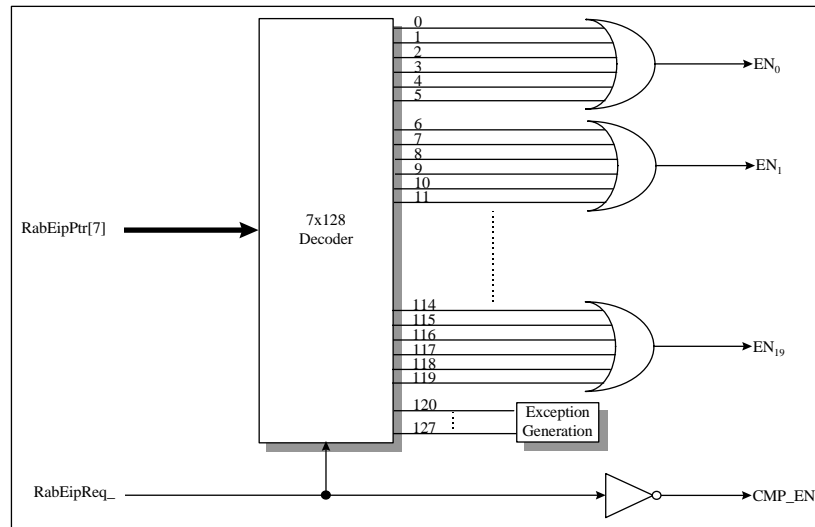


Fig. 8. Retire set controller.

There is an age counter in the Set Controller that increments as long as the valid bit is set and the instruction fetcher is not in the stall state. The count is the age of the instruction addresses (how many cycles they have been in the Address Queue) in the corresponding set. The age is also latched to avoid a race condition. As instructions are retired from the reorder buffer, the reorder buffer sends the address queue index of the last retired instruction and a retirement request signal (the RabEipPtr and RabEipReq_, respectively) to the instruction fetcher. The address queue index is given to the instructions when entries of Address Queue are allocated for them. This index will be sent to Retire Set Selector to generate the CMP_EN and EN_{*i*} signals. As the EN_{*i*} signal arrives in Set Controller_{*i*}, the Set Controller_{*i*} will output the content of its age latch to the AC_Bus. All Set Controllers will compare the coming data from AC_BUS against the content of its own age latch when the CMP_EN signal is activated. If a Set Controller finds that its age is greater, meaning the corresponding instruction is before the one being retired in the original program flow, its valid bit is cleared and this set becomes free. The comparison requires “larger than” rather than “larger than or equal to” because even if an instruction in a set is retired, a succeeding instruction in the same set may not be retired yet.

4.2 X86 Address Storage Schemes

In this section we give two x86 address storage schemes, Scheme 1 (original) and Scheme 2 (optimized). Scheme 1, storage organization is as simple as a full associative storage. There are twenty sets in the address queue. Each set can store six effective address and linear address pairs. The storage cells are numbered from 0 to 119. Set_{*i*} is composed of Cell_{6*i*} to Cell_{6*i*+5}. The structure of the Address Storage Scheme 1 is shown in Fig. 9.

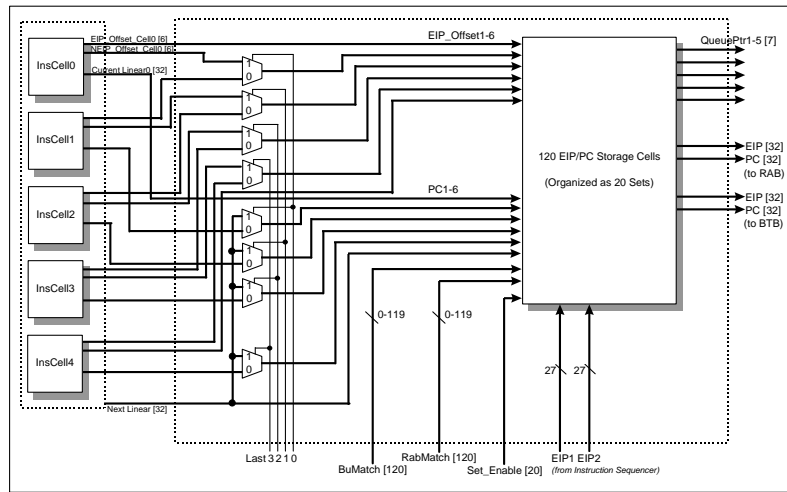


Fig. 9. Address storage scheme 1.

On each clock cycle a set is allocated for storing fetched instruction addresses by the Address Queue controller. The entries in a set are used in the following way: if set_i is selected and if there are x (x ≤ 5) instructions being fetched, the x instructions are stored in Cell_{6i} to Cell_{6i+x-1}, and the next sequentially fetch address is stored in Cell_{6i+x}.

Since there may be fewer than five instructions fetched on each clock cycle, some instruction cells may not contain valid addresses. Therefore, four multiplexers are used to select valid addresses that should be stored into the Address Queue. The four multiplexers are controlled by the fetcher, which indicates where the last delivered instruction is in this clock cycle. For a self-modified code check, the stored linear addresses are compared with the snooped data write addresses. The set organization and the Cell are shown in Figs. 10 and 11, respectively.

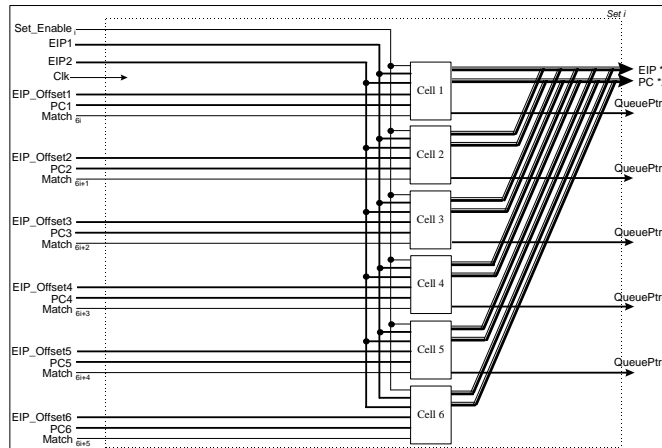


Fig. 10. Set organization of address storage scheme 1.

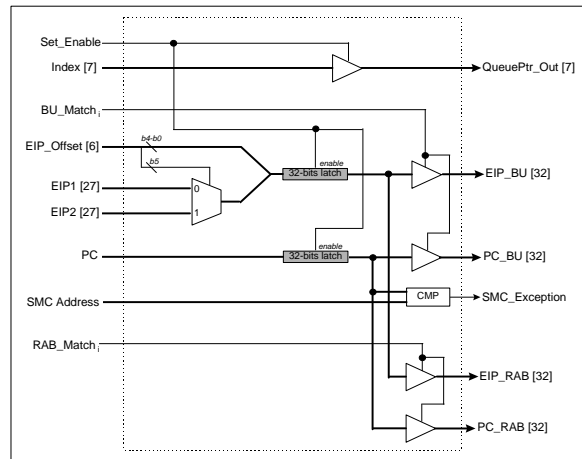


Fig. 11. Storage cell of address storage scheme 1.

Address Storage Scheme 1 is simple and easy to implement. However, it wastes a lot of storage space. As described earlier, because we do not implement split-fetching operations, the instructions we fetched in each clock cycle may come from at most two different cache lines. Therefore, the addresses may have 27 bits in common. As a result the common line addresses need be stored only once. Only the offsets of the instructions within a cache line need to be stored separately. And, to reduce the space the EIP addresses can also be stored using the “EIP-line” concepts as previous described. Therefore, a storage optimized address storage scheme, Scheme 2, is proposed. Its structure is shown in Fig. 12.

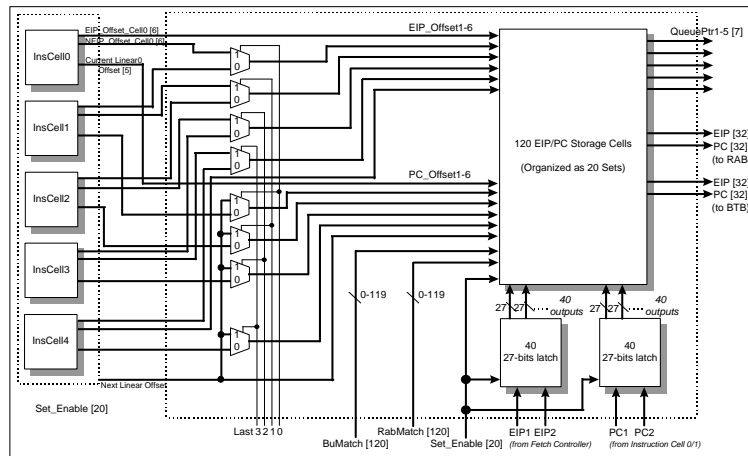


Fig. 12. Address storage scheme 2.

In this scheme the set organization remains the same. Comparing it to Scheme 1, in each set there are four separate 27-bit latches to store the effective and linear line ad-

addresses, while only the offsets are stored in the Address Queue Cell.

However, the split-line instructions once again complicate the problem. As Fig. 2 shows, instructions in the same cache line may be in different EIP lines. Therefore, when a split-line instruction is entirely fetched, its EIP-line address may be neither the “EIP1” nor the “EIP2” address calculated. For example, if the split-line instruction is in the Linear Line $i-1$ in linear address space and in EIP-line $j-1$ in effective address space of Fig. 2, when it is entirely fetched, the EIP1 is EIP-line j and the EIP2 is EIP-line $j + 1$. In this case the EIP-line $j-1$ address is lost! To solve this problem, note that the EIP-line $j-1$ has already been stored in the Address Queue as the EIP1 in the previous clock cycle. Therefore, we can retrieve it from the previous set. With this solution only a local 27-bit bus, but no additional storage, is needed. Thus, the first storage cell is different from other storage cells in each set. The self-modified code problem is handled in the same way as in Scheme 1. The set organization, the basic storage Cell, and the first storage Cell of each set are shown in Figs. 13, 14 and 15, respectively.

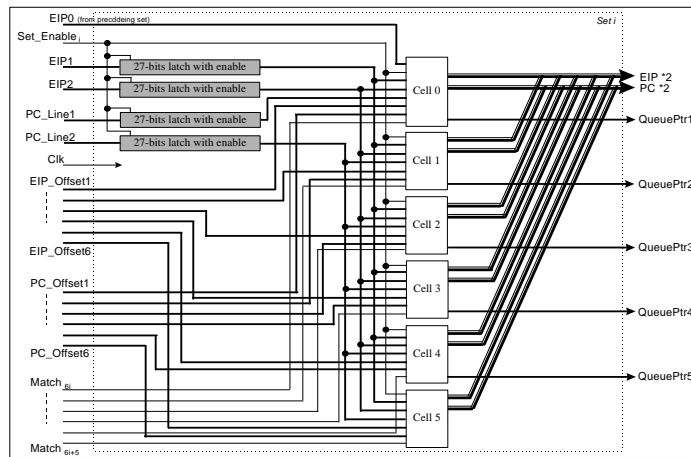


Fig. 13. Set organization of address storage scheme 2.

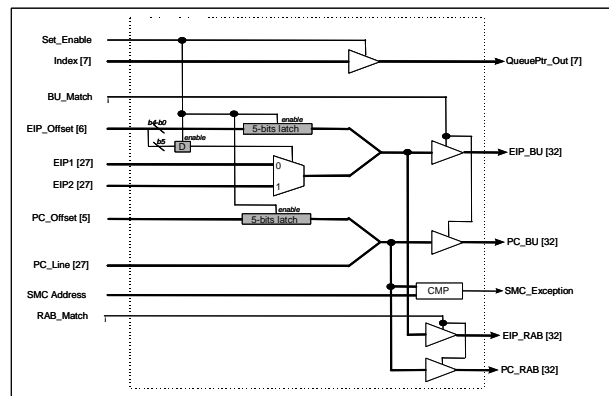


Fig. 14. Storage cell of address storage scheme 2.

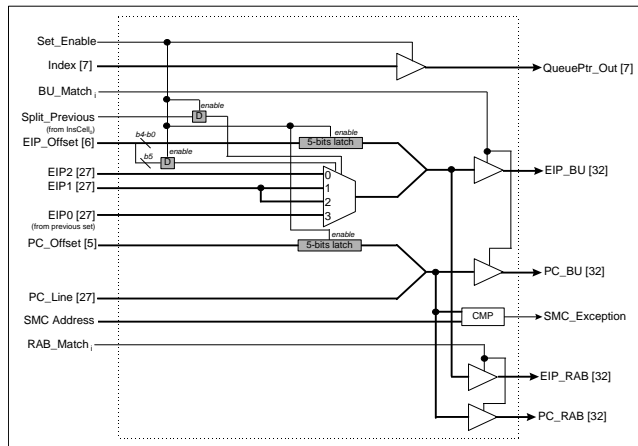


Fig. 15. First storage cell of each set in address storage scheme 2.

5. CIRCUIT SYNTHESIS

In this section we emphasize the gate count comparison when discussing the Address Queue synthesis results. The goal of using Address Queue is to reduce the complexity of the microprocessors. As long as the Address Queue is not on the critical path, its delay is not critical to the microprocessor performance. On the other hand, the gate count should be reduced as much as possible so that the Address Queue is cost effective.

We use the Synopsys synthesizer version 3.4b to synthesize the Address Queue, and the COMPASS 0.6-micron library version 2.31 for the synthesis. Both the structuring and flattening optimization are turned on to optimize the circuits, and the mapping effort is set to high. The Address Queue synthesis results are given below in Fig. 16.

	Controller		Storage Scheme 1		Storage Scheme 2	
	Alloc./ Retire	Access	Each Set	Other Parts	Each Set	Other Parts
Gate Count	4679	313	7371	842	5101	655
total (20 sets)	5305		148262		102675	
Delay	6.79	1.34	0.99	0.3	1.12	0.3

Fig. 16. Address queue synthesis result.

From Fig. 16 one can see that the Address Queue controller is very small compared to the Address Storage (< 5%). On the other hand, the controller takes more time than the address storage part. Hence, organizing the Address Queue storage as sets is the optimized choice since it simplifies the controller and reduces its delay.

The two address storage scheme results show a great difference. Obviously, Scheme 2 is much better than Scheme 1. The ratio of storage spaces needed for twenty sets is 1: 0.69. Therefore, the silicon area needed for Scheme 2 is only about two-thirds that for Scheme 1. This result is predictable from the estimated number of bits stored in each scheme. In Scheme 1 there are $32 \cdot 2 \cdot 6 = 384$ bits stored in each set and in Scheme 2

there are only $7 + (6*5) + 5*6 + 27*4 = 175$ bits stored. The ratio is $175/384$ or 0.456 . Taking into account some other necessary components in the storage cells and routing paths, the 0.69 ratios are very reasonable.

6. CONCLUSIONS

In this paper we have proposed a mechanism called "Address Queue" to effectively maintain the instruction addresses so that branches and exceptions can be handled more efficiently. With this Address Queue, the instruction address handling in the x86 microprocessors can be done as easily as it is in RISC microprocessors. This improvement has particular significance in designing a high degree x86 superscalar processor, since it reduces the routing complexity and saves space for storing instruction pointers. For properly handling the Address Queue, three operations are implemented. They are the allocation in set, the access by sequence and the retirement by aging. The Address Queue size can be well estimated by its behavior. We have run simulations to evaluate different design alternatives, and we have run synthesis to estimate the cost and performance of these designs.

Crossing basic block boundaries to fetch instructions has been proven important in ILP performance improvement [12]. The key ideas of our proposal, how to handle the split-line instructions and the complex address space problems effectively, hint that the group of instructions fetched by a high bandwidth superscalar fetcher, which may have different cache line addresses, can be stored, accessed and retired in one cycle.

Recent studies have proposed the use of a trace cache to enhance ILP. The Address Queue designed can be extended to handle machine states of trace-cache supported architectures. With this help, a technique for high bandwidth instruction fetching can be achieved at a much lower cost.

REFERENCES

1. D. Sima, "Superscalar instruction issue," *IEEE Micro*, Vol. 17, 1997, pp. 28-39.
2. M. Slater, "The microprocessor today," *IEEE Micro*, Vol. 16, 1996, pp. 32-44.
3. Intel Corporation, *Pentium Processor User's Manual Volume 3: Architecture and Programming Manual*, 1993.
4. T. Shanley, *Pentium Pro Processor System Architecture*, Mind Share, Inc., 1997.
5. AMD Corporation, *AMD-K7(TM) Technology Presentation*, 1998.
6. AMD Corporation, *AMD-K6 MMX Enhanced Processor Data Sheet*, 1997.
7. D. Christic, "Developing the AMD-K5 architecture," *IEEE Micro*, Vol. 16, 1996, pp. 16-26.
8. AMD Corporation, *AMD5K86 Processor Technical Reference Manual*, 1996.
9. L. Gwennap, "Intel's P6 uses decoupled superscalar design," *Microprocessor Report*, Vol. 9, 1995, pp. 1-7.
10. M. Slater, "K6 to boost AMD's position in 1997," *Microprocessor Report*, Vol. 10, 1996, pp. 1-2.
11. A. Yu, "The future of microprocessors," *IEEE Micro*, Vol. 16, 1996, pp. 46-53.
12. E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to

high bandwidth instruction fetching,” in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Micro-architecture*, 1996, pp. 24-34.



Jih-Ching Chiu (邱日清) received the B.S. degree in Electrical Engineering from National Sun Yet-Sen University in 1984, and M.S. degree in Electrical Engineering from National Cheng Kung University in 1986. He is now working towards the Ph.D. degree in Department of Computer Science and Information Engineering of National Chiao Tung University at Hsinchu from 1994.

He is a Lecturer of Electrical Engineering at the National Sun Yat-Sen University. His research interests include computer architectures, processor architectures, real-time operating systems and microprocessor-based systems.



Michael Jin-Yi Wang (王敬毅) received the B.S. and M.S. degree in Computer Science & Information Engineering from National Chiao Tung University in 1996 and 1998, respectively. Currently he is a teaching assistant of Department of Computer Science & Information Engineering at National Taiwan University. His research interests include computer architectures, microprocessor design and microprocessor-based systems.



Chung-Ping Chung (鍾崇斌) received the B.S. degree from the National Cheng Kung University, Taiwan, Republic of China, in 1976, and the M.S. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in Electrical Engineering.

He was a Lecturer of Electrical Engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao Tung University, Hsinchu, Taiwan, ROC, where he is a Professor. From 1991 to 1992, he was a Visiting Association Professor of Computer Science at the Michigan State University. Currently, he is a Consultant of Computer & Communications Research Laboratories (CCL), Industrial Technology Research Institute (ITRI). His research interests include computer architecture, parallel processing, and parallel compiler design.