High-bandwidth x86 instruction fetching based on instruction pointer table

J.-C.Chiu and C.-P.Chung

Abstract: Providing higher degree superscalar instruction fetching is a major concern in a high performance superscalar processor design. In x86 architectures, the variable-length instructions make fetching multiple instructions in a cycle difficult. A common practice is to use predecoded information to help in instruction fetching, while the complex instruction formats induce high redundancies in storing and processing the pre-decoded information in the cache. In the paper, the authors propose to use an Instruction Identifier to predict instruction length and store the instruction pointers as superscalar instruction group indicators. With this method, the difficulty of achieving a high instruction fetch degree (>3) can be overcome. Simulation results suggest that the Instruction Identifier with a 64-entry table is a good performance/cost choice. In the meantime, as the table size decreases, the prediction scheme becomes increasingly important. Moreover, simulation and circuit synthesis show that this design is feasible for high clock rate design.

1 Introduction

The goal of a superscalar design is to simultaneously issue, execute and complete as many instructions as possible [1], and the functions of an instruction fetcher are to identify instructions and to provide them to the decoders at a sufficient rate. Unfortunately, the variable-length instruction format, such as in the x86 architecture, makes it difficult to achieve this goal [2]. There are two approaches about instruction-fetch mechanisms which are proposed to improve the instruction fetch and decode bandwidth. The one is a dynamically scheduling instruction sequence to match the parallel decoding rules [3]. The other is enlarging the instruction window with trace prediction to fetch multiple contiguous basic blocks in a cycle [4]. But, in x86 architectures, we consider the basic points of the instruction fetch problem that are the variable-length instruction and the complex instruction format. These will cause high routing-path complexity for selecting a sequent instruction [2], which may be a barrier to approaching high clock-rate design. So, in current x86 processors, the parallel decoding rules are used to limit the fetched-instruction-sequence type and to simplify the instruction-selected routing-path complexity [5]. In the fixed-length instruction architectures, the design of trace cache microarchitecture will just consider the assembly problems of the noncontiguous-location instructions from alternative basic blocks [6]. But, in x86 architectures, each instruction is noncontiguous. That will cause the trace cache mechanisms to be hardly used in the design of x86 architectures.

IEE Proceedings online no. 20010456

DOI: 10.1049/ip-cdt:20010456

Paper first received 19th June 2000 and in revised form 10th April 2001

The authors are with the Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, 30050, Taiwan
E-mail: chiujihc@ee.nsysu.edu.tw

The variable-length instruction denotes that the next instruction cannot be decoded, or even fetched, until the length of the previous instruction is known. In a traditional x86 processor, a sizer is used to identify instruction lengths within the latency of a pipeline stage [7, 8]. Hence, identifying the starting points of the second and subsequent instructions in a cycle remains an issue in the x86 superscalar design. Some AMD series products implement instruction identification by using predecode information in its instruction cache [9–11]. Instead of instruction pointers, boundary bits are usually used to save cache space, and simple bit-vector scanning can be used to construct the instruction pointers and to identify each instruction. A large amount of time latency and hardware cost may be needed to implement these methods. Therefore, we intend to discuss about how to fetch multiple instructions quickly from a traditional instruction cache. The basic idea is that, by predicting instruction lengths and storing instruction pointers with a table, one can simultaneously provide a sequence of instruction pointers to the fetcher. This pointer storage is independent of the cache; hence its design and operation are independent as well. Instruction fetch parallelism can then be boosted without sacrificing the design flexibility or clock rate.

2 Characteristics of identifying instructions with instruction boundary bits

Most of the current x86 processors, such as P54C [12], Pentium II [13], K5 [9], K6 [10] and K7 [11], use boundary bits to identify individual instructions. This method is illustrated in Fig. 1 to show its hardware and time complexities. The lookup window selects w boundary bits out of the l instruction queue boundary bits. The boundary bit scanner scans these selected boundary bits and shows each instruction pointer with an instruction displacement. The number of displacements shown is equal to the superscalar degree, d, and they are for the

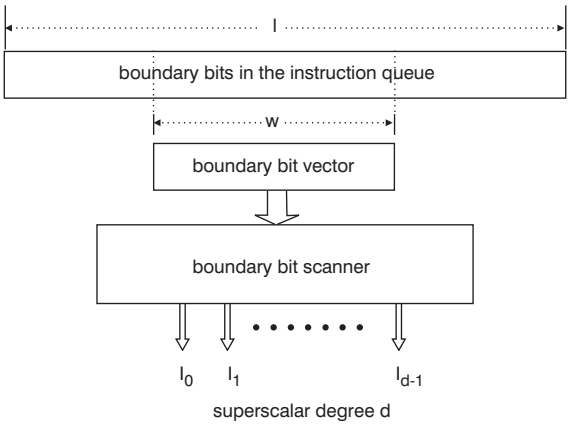

Fig. 1 *Architecture of identifying instructions with instruction boundary bits in a superscalar degree d fetcher*

next stage to identify d instructions and to pass them to the decoders. Two methods can be used to design this scanner. The first is the sequential scan method, and the other is the bit-lookahead method.

2.1 Sequential scan method

Fig. 2 illustrates the organisation of the sequential scan method. The selected boundary bits are input to the first priority encoder to find the second instruction displacement, and then a bit mask masks this boundary bit off and sends the remaining bit vector to the next priority encoder as the process repeats. The above procedures repeat until N instruction displacements are found. With the COMPASS 0.6 μm library version 2.31 in the synthesis, the delay time of an optimised priority encoder is 1.5 ns, which means at least 1.5 ns in a 0.6 μm process technology is spent to find its next instruction pointer using the priority encoder technique. Therefore, an N-way processor must spend $(N-1) \times 1.5$ ns to identify all N instructions. This latency can incur multiple stages of overhead in the fetch unit as well as making the higher recovery penalty in a high degree superscalar processor.

2.2 Bit-lookahead method

Fig. 3 illustrates the organisation of the bit-lookahead method. Where there is a large fan-out problem, as the number of fan-outs increases, several characteristics of interconnects, such as the propagation delay and the effective characteristic impedance, are affected [14]. With this method, each boundary bit will be sent to all the higher order bit positions to generate the instruction-selected bits simultaneously (ISB). Let ISB_{di} be the dth instruction-selected bit in the boundary bit-i position. The ISB_{di} is

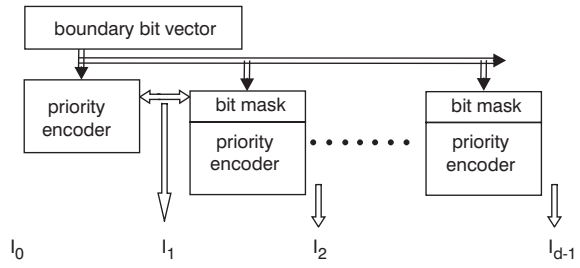

Fig. 2 *Organisation of sequential scanning method*

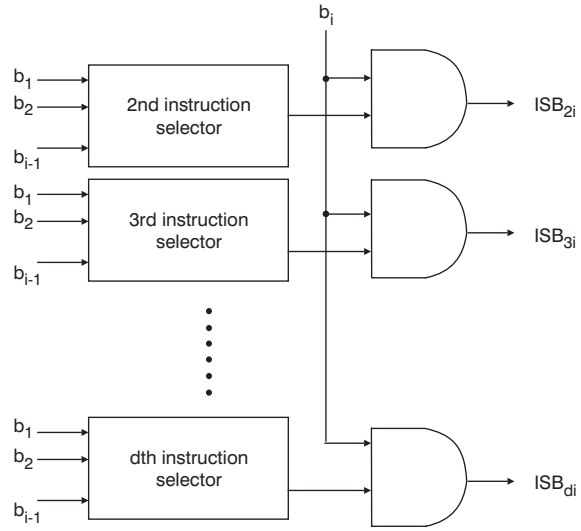

Fig. 3 *Organisation of bit-lookahead method*

described by the following Boolean equation for $w > i \geq d-1$ and $d \geq 2$:

$$\begin{aligned} ISB_{di} = (&NOR(\overline{b_1}, \overline{b_2}, \ldots, \overline{b_{d-2}}, b_{d-1}, \ldots, b_{i-1}) \\ &+ NOR(b_1, \overline{b_2}, \overline{b_3}, \ldots, \overline{b_{d-1}}, b_d, \ldots, b_{i-1}) \\ &+ \cdots + NOR(b_1, b_2, \ldots, \overline{b_{(i-1)-(d-2)}}, \ldots, \overline{b_{i-1}})) \cdot b_i \end{aligned}$$

The fan-outs of b_1, F_1 can be evaluated by the following equation:

$$F_1 = \sum_{j=2}^{d} \sum_{i=j-1}^{w-1} C_{j-2}^{i-1}$$

The distribution conditions are shown as Fig. 4. When the d and w are increased, the number of fan-outs of b_1, F_1 grows faster. This fact makes this method inadequate for use in the high degree superscalar instruction fetch.

3 Fetching multiple instructions using the Instruction Identifier

To achieve a high superscalar degree in variable instruction length architectures to conduce to instruction execution parallelism, we propose the idea using an Instruction Identifier. By means of predicting instruction lengths and recording instruction pointers to a table, our design can simultaneously provide a sequence of instruction pointers for the fetcher to fetch a group of N instruction cycle by cycle.

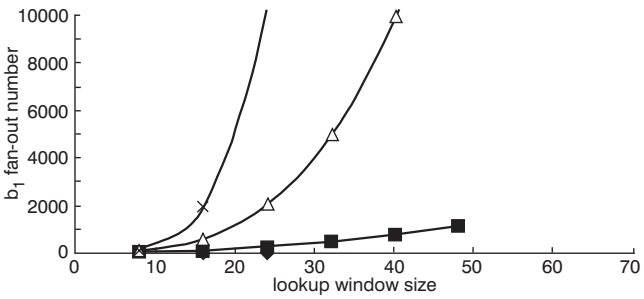

Fig. 4 *Distribution of number of bit-1 fan-outs*

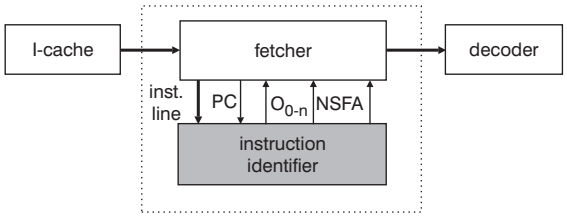

Fig. 5 *Role of Instruction Identifier*
O = offset (instruction pointer); NSFA = next sequential fetch address

3.1 Procedure

As previously described, key ideas of our design are as follows:

1. If a program counter (PC) is not found in the table, then we will predict its corresponding length to generate the next instruction pointer.
2. If a PC is found in the table, then we can simultaneously get multiple recorded instruction pointers, and predict the first unknown instruction length if an insufficient number of instruction pointers are available.

Fig. 5 shows that when the Instruction Identifier receives a PC from the fetcher, it responds with the instruction pointers and the next sequential fetch address (NSFA) to the fetcher. To implement the above-mentioned framework, the Instruction Identifier has five operations, namely the access, the prediction, the commitment, the placement and the address generation.

3.1.1 Access to Instruction Pointer Table: Access is the operation by which the instruction pointers are read from the table when a PC hits in the table. In each cycle, the Instruction Identifier will look up the table for the PC from the fetcher. If the PC hits in the table, a sequence of instruction pointers following the PC will be provided to the fetcher simultaneously.

3.1.2 Prediction of instruction length: This operation predicts the first unknown instruction length. Cases in which the prediction will take place are as follows:

- PC does not hit in the table, or
- PC hits in the table but the instruction length is unknown.

The instruction whose length needs to be predicted is named the predicted instruction P, and the instruction that follows the predicted instruction is named the speculative instruction S.

3.1.3 Commitment of speculated instruction length: In order to avoid fetch restart caused by length misprediction, the length verification is done in the Instruction Identifier, not the decoder. Commitment is the operation that will decide whether the speculative instruction pointer is correct by checking the actual length of the predicted instruction. If the length prediction is right, then the speculative instruction pointer is correct, and this instruction can be sent to the decoder when there is not any branch-taken instruction before it. In addition, this operation also checks the actual length of the speculative instruction at the same time. All of these results help to generate the next sequential fetch address.

3.1.4 Address generation: The next sequential fetch address is generated according to the results of access and commitment operations. For example, if the PC does not hit in the table and the instruction length is predicted correctly, then the next sequential fetch address is set to be the next sequential instruction pointer of the speculative instruction. This is the address from which the Instruction Identifier expects the fetcher to fetch next.

3.1.5 Placement of instruction pointers: Placement is the operation of storing the instruction pointers in the appropriate fields in the table. In this operation, we will allocate a new table entry under the following conditions:

1. PC does not hit in the table and is not equal to NSFA.
2. PC does not hit in the table and is equal to NSFA, but without empty field in the entry used last cycle.
3. PC is an address of a split-line instruction.

Fig. 6 shows the flowchart of our design algorithm in detail.

3.2 Instruction Identifier design

The Instruction Identifier consists of the Instruction Pointer Table (IPT), the Instruction Identifier Controller and the Speculation Commit Unit. Fig. 7 shows the block diagram of the Instruction Identifier.

3.2.1 Instruction Identifier Controller: The Instruction Identifier Controller is responsible for controlling all of the operations in the Instruction Identifier. These operations, presented in the previous section, include the following:

(i) *access:* looks up IPT
(ii) *prediction:* predicts the instruction length
(iii) *commitment:* notifies the Speculation Commit Unit to check the instruction length
(iv) *placement:* stores the instruction pointers to IPT
(v) *address generation:* generates the next sequential fetch address.

3.2.2 Instruction Pointer Table: The Instruction Pointer Table stores a sequence of instruction pointers. Each entry in the Instruction Pointer Table consists of N fields to store a PC field and $(N-1)$ offset fields, as Fig. 8 shows. Here N is the superscalar degree of the processor. In addition, each field needs two extra bits. One of them is the valid bit that indicates whether the instruction length is known. The other is the split-line bit that indicates whether the instruction is a split-line instruction.

From Fig. 9, the most common (or frequent) instruction length is 3 bytes, and nearly 65% of the instructions are 2 or 3 bytes. According to these results, we propose three prediction schemes, namely fixed_3, global and private predictions, respectively, to predict the unknown instruction length. The fixed_3 prediction scheme is a very simple scheme that always predicts an unknown instruction length to be 3 bytes. The global prediction scheme is similar to the 1-bit scheme that depends on whether the prediction was correct recently. The private prediction scheme is a 1-bit table scheme that uses a table, indexed by several least significant bits, to record the recent prediction results. We simulate the performance of these schemes and analyse them in the next section.

3.2.3 Speculation Commit Unit: For a fetched instruction with unknown length, the prediction mechanism will predict its instruction length and generate the subsequent instruction pointer. Two sizers in the Specula-

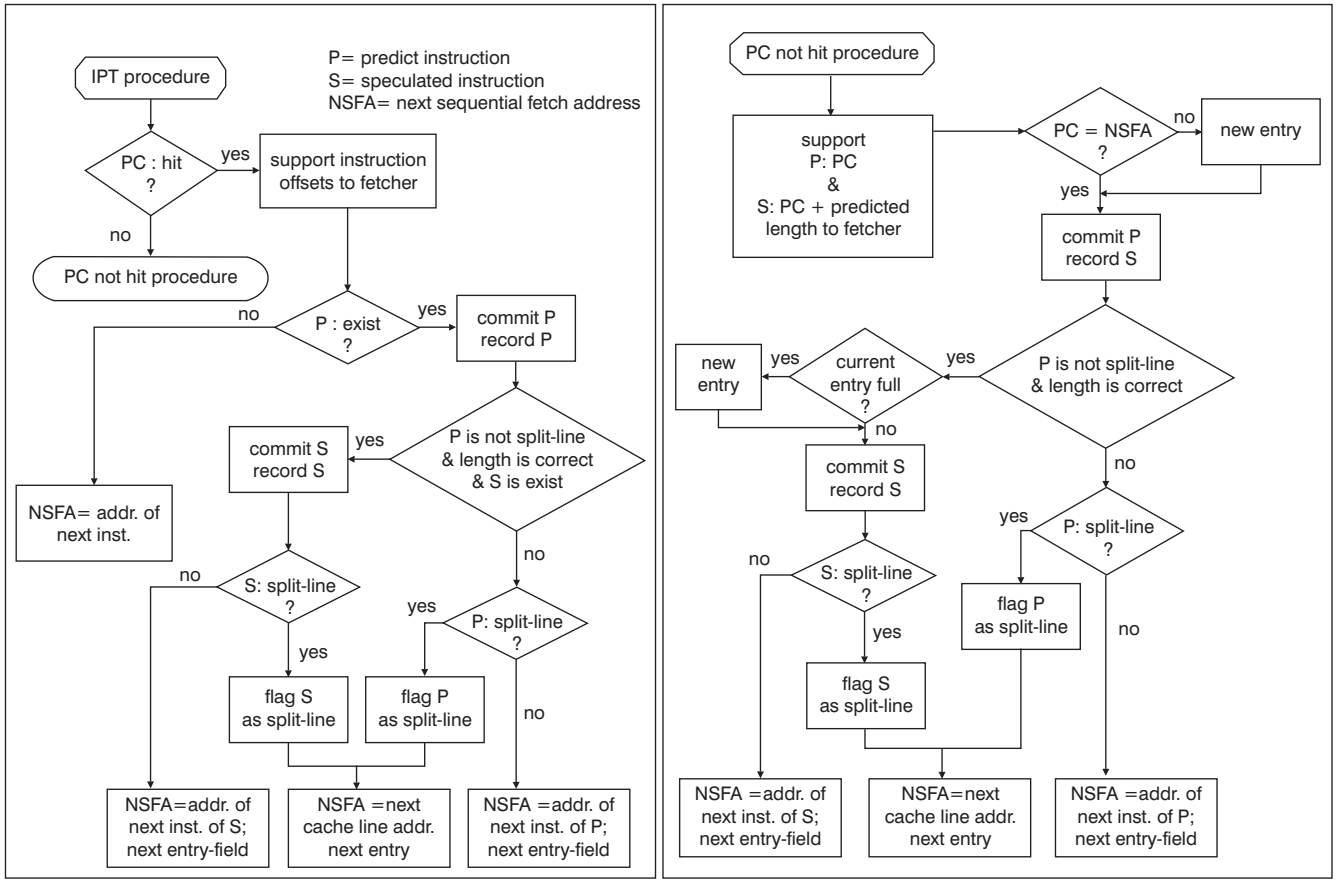

Fig. 6 *Flowchart of Instruction Identifier operating algorithm*

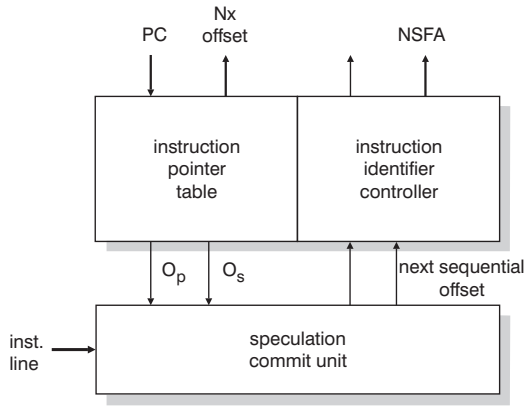

Fig. 7 *Instruction Identifier block diagram*

O_p = offset of predicted instruction; O_s = offset of speculative instruction; NSFA = next sequential fetch address

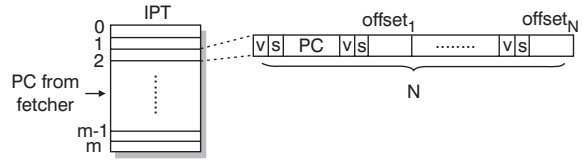

Fig. 8 *Structure of entry in IPT*

v = valid bit; s = split-line bit

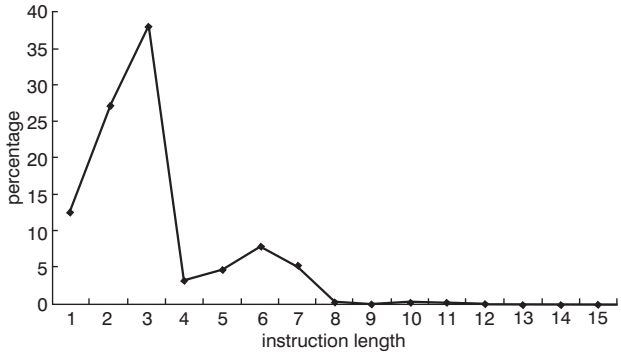

Fig. 9 *Distribution of instruction lengths of SPECint95*

tion Commit Unit are responsible for checking the lengths of the predicted and speculative instructions. The task of the Speculation Commit Unit is to check whether the speculative instruction pointer is correct by checking whether the prediction length is right. If the prediction length is right, the next sequential fetch address equals the next subsequent instruction pointer of the speculative instruction; otherwise it equals the next sequential instruction pointer of the predicted instruction.

4 System performance evaluation

In this section we present the simulation and synthesis results under our architecture. By the fetch rate view [2] we focus the performance evaluation only in the natural barriers, namely the branch instruction barrier and split-instruction barrier.

4.1 Simulation environment

The experiments to evaluate the performance of the Instruction Identifier are run through a trace-driven simulation. In this simulation, eight SPECint95 benchmarks, the *go, m88ksim, gcc, compress, li, jpeg, perl* and *vortex*, are used. When a benchmark is executed, we use a Linux-system-call 'ptrace' to extract the traces of all instructions,

store the traces in a file, then feed their trace files into our simulator.

The assumptions of the simulation model are as follows:

1. The accuracy rate of the branch prediction is 100%.
2. The miss penalty of the instruction buffer is zero cycle.
3. When a taken-branch instruction is encountered, it will be the last fetched instruction in that cycle.
4. The size of the reorder buffer is unlimited.
5. The instruction cache is perfect.

These assumptions will make the unbounded instruction issued-and-executed environments to focus on evaluating the effects of fetch mechanisms. For the following reasons, we let the assumption of perfect branch prediction that cannot affect our evaluation results.

1. The number of IPT entries is much greater than a basic block (= 5 instructions).
2. The replacement policy of the IPT is FIFO (first-in-first-out).

4.2 Prediction scheme analysis

There are tree schemes described in Section 3.2, namely the fixed_3, the global and the private predictions. The simulation results of these schemes are shown in Fig. 10. In this figure, the *x*-axis represents different numbers of entries in the IPT table and the *y*-axis represents the average fetch rate. Furthermore, 'perfect' means that the prediction is always correct, and 'worst' means that there is not any prediction scheme in our design. These two values are used as the performance upper and lower bounds.

From Fig. 10, two observations are made. One of them is that the performance of fixed_3 is better than other proposed schemes. So, in the following experiments, we use the fixed_3 prediction scheme to predict the unknown instruction length. The other observation is that the less the IPT entries are, the more important the prediction scheme becomes.

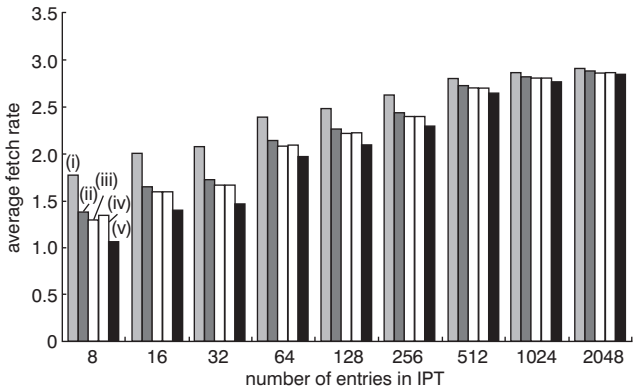

Fig. 10 *Performance evaluations of different prediction schemes*
(i) Perfect; (ii) fixed_3; (iii) global; (iv) private; (v) worst

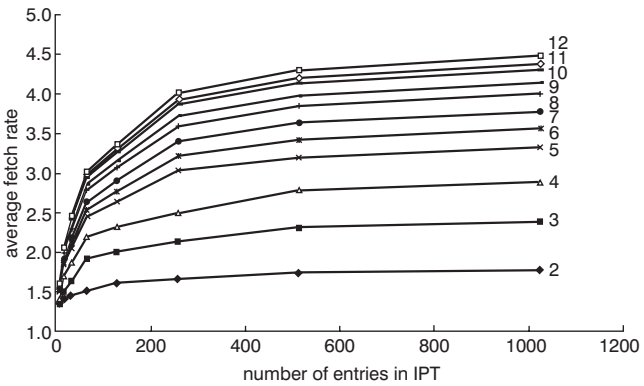

Fig. 11 *Performance evaluations of numbers of IPT entries*

4.3 Number of IPT entries analysis

In this section, we study the performance due to number of entries and superscalar degree. The results of the performance evaluations are shown in Fig. 11. In this figure, the *x*-axis represents different numbers of entries in the IPT table, ranging from 8 to 1024, and the *y*-axis represents the average fetch rate. The 11 lines represent different superscalar degrees, ranging from 2 to 12.

From Fig. 11 we find that, for any fixed number of IPT entries, when the superscalar degree increases, the average fetch rate also increases. Meanwhile, for any fixed superscalar degree, increasing the number of IPT entries increases the average fetch rate, which will eventually reach a saturation point. From the simulation results we find that a 64-entry IPT is a good choice under performance/cost consideration.

4.4 Instruction cache line size analysis

In this section, we study the impact of different instruction cache line sizes on performance. The performance evaluation results are shown in Fig. 12. In this figure, the *x*-axis represents different instruction cache line sizes, ranging from 16 to 2048 bytes, and the *y*-axis represents the average fetch rate. In this experiment, the number of IPT entries is 2048 and the superscalar degree ranges from 2 to 12.

In Fig. 12 it is obvious that, when the instruction cache line size increases, the average-fetch rate increases and eventually reaches a saturation point. The reason is simply that increasing the instruction cache line size gives the fetcher to fetch more instructions to the decoders at a time.

4.5 Delay time estimation

In discussing the delay time of a chip, the critical path is very important because it limits the maximum clock frequency. We use the Synopsys synthesiser to synthesise

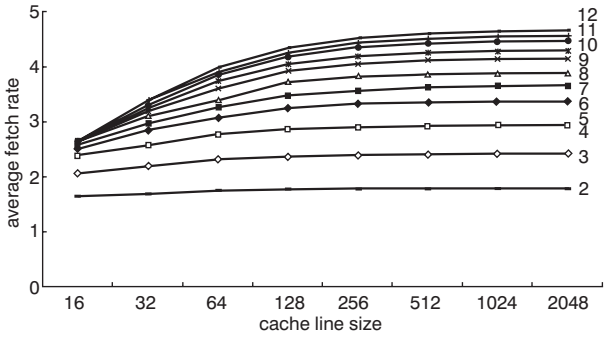

Fig. 12 *Performance evaluations on different instruction cache line size*

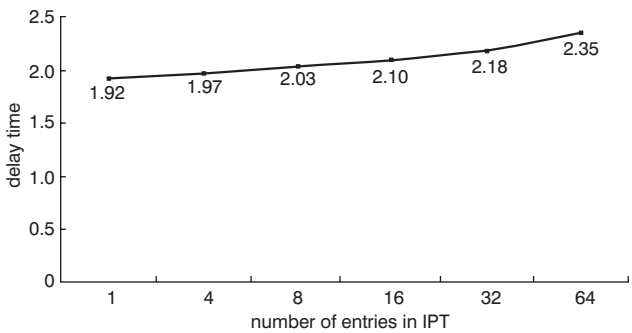

Fig. 13 *Access delay on different number of entries*

our design. This synthesiser uses the COMPASS 0.6 μm library to time the critical paths. Fig. 13 shows the access delay against number of IPT entries ranging from 1 to 64. When the IPT entry number is 64, the access delay is 2.33 ns. This delay is less than the latency of the sequential method or bit-lookahead method.

An instruction length is determined through a sequence of checks on the Prefix, Op1, Op2, Mod R/M and SIB. Thus, the longest path of the sizer is the path that includes 11 *Prefix*, 1 *Op1*, 1 *Op2*, 1 *Mod R/M* and 1 *SIB*. The estimated delay of the critical path is 1.58 ns. From these estimated synthesis results we find that our design can run at 200 MHz in a 0.6 μm process technology. Much higher clock rates can be achieved if more up-to-date technologies are to be used.

5 Conclusions

The goal of the superscalar microprocessor design is to simultaneously issue and execute the maximum allowable number of instructions as often as possible. But the variable-length x86 instruction encoding makes it very difficult for the instruction fetcher to efficiently fetch several instructions from a stream of raw instruction bytes at a time. Current x86 superscalar microprocessors employ several different strategies to handle this problem, but these strategies may all soon reach a superscalar degree limit. The idea of the Instruction Identifier to overcome this difficulty is hence proposed.

The simulation results show that, when the number of IPT entries increases, the average fetch rate increases. From the simulation results we find that a 64-entry table is a good choice under performance/cost consideration. The simulation results also show that, when the number of IPT entries decreases, the prediction accuracy becomes very important. In addition to the simulation results, we also present the synthesis results to illustrate that our design is feasible because it can run at least 200 MHz in 0.6 μm process technology—a very competitive speed in that technology.

6 References

1 SIMA, D.: 'Superscalar instruction issue', *IEEE Micro*, 1997, **17**, (5), pp. 28–39.
2 CHIU, J.-C., and CHUNG, C.-P.: 'The fetch mechanism issue of x86 superscalar processors with fetch rules'. Workshop on *Computer architecture*. Proceedings of the 2000 International Computer Symposium, December 2000, pp. 129–136
3 SMOTHERMAN, M., and FRANKLIN, M.: 'Improving CISC instruction decoding performance using a fill unit'. Proceedings of 28th annual international symposium on *Microarchitecture*, 1995, pp. 219–229
4 ROTENBERG, E., BENNETT, S., and SMITH, J.E.: 'Trace cache: a low latency approach to high bandwidth instruction fetching'. Proceedings of 29th Annual IEEE/ACM international symposium on *Microarchitecture*, MICRO-29, December 1996, pp. 24–34
5 SHIU, R-M., CHIU, J.-C., CHENG, S.-K., and SHANN, J.-J.: 'The design of the decoding unit with high issue rate for an x86 superscalar microprocessor', *IEE Proc., Comput. Digit. Tech.*, 2000, **147**, (2), pp. 101–107
6 ROTENBERG, E., BENNETT, S., and SMITH, J.E.: 'A trace cache microarchitecture and evaluation', *IEEE Trans. Comput.*, 1999, **48**, (2), pp. 111–120
7 CASE, B.: 'Intel reveals pentium implementation details', *Microprocess. Rep.*, 1994, **7**, (4), pp. 1–9
8 GWENNAP, L.: 'Intel's P6 uses decoupled superscalar design', *Microprocess. Rep.*, 1995, **9**, (2), pp. 1–7
9 CHRISTIE, D.: 'Developing the AMD-K5 architecture', *IEEE Micro*, 1996, **16**, (2), pp. 16–27
10 AMD Corporation: 'AMD-K6-III Processor Datasheet', 1999
11 AMD Corporation: 'AMD Athlon Processor Technical Brief', 1999
12 MINDSHARE INC., ANDERSON, D., and SHANLEY, T.: 'Pentium processor system architecture' (Addison-Wesley Developer Press, 1995)
13 MINDSHARE INC., and SHANLEY, T.: 'Pentium Pro processor system architecture' (Addison-Wesley Developer Press, 1997)
14 ZARKESH-HA, P., DAVIS, J.A., LOH, W., and MEINDL, J.D.: 'Stochastic interconnect network fan-out distribution using Rent's Rule'. Proceedings of IEEE 1998 International Interconnect Technology Conference, June 1998, pp. 184–186