PAPER

Fast Precise Interrupt Handling without Associative Searching in Multiple Out-Of-Order Issue Processors

Sang-Joon NAM[†], **In-Cheol PARK**[†], *and* **Chong-Min KYUNG**[†], ***Nonmembers***

SUMMARY This paper presents a new approach to the precise interrupt handling problem in modern processors with multiple out-of-order issues. It is difficult to implement a precise interrupt scheme in the processors because later instructions may change the process states before their preceding instructions have completed. We propose a fast precise interrupt handling scheme which can recover the precise state in one cycle if an interrupt occurs. In addition, the scheme removes all the associative searching operations which are inevitable in the previous approaches. To deal with the renaming of destination registers, we present a new bank-based register file which is indexed by bank index tables containing the bank identifiers of renamed register entries. Simulation results based on the superscalar MIPS architecture show that the register file with 3 banks is a good trade-off between high performance and low complexity.

key words: *computer architecture, precise interrupt, multiple out-of-order issue processors*

1. Introduction

Modern superscalar or VLIW processors exploit instruction level parallelism by issuing multiple instructions to functional units if there is no data conflict between the instructions. The performance of processors can be significantly enhanced by issuing multiple instructions out of order [1], [6], [10], [15].

However, the out-of-order issue can cause a serious problem at the time of interrupts, because it makes the process states different from those defined by the program sequence. In this paper, abnormal operations such as software traps, cache misses, page faults, and branch misprediction are all referred to as interrupts. When an interrupt occurs, the interrupted process state must be saved. The process state generally consists of the program counter, register contents, and memory contents. Interrupts are precise if the saved processor states are consistent with the in-order states defined by the program sequence where one instruction completes before the next begins. Since interrupts are not rare and the precise interrupts are essential to making known states at the time of interrupts, how efficient and how fast a processor can recover the in-order state is very crucial in multiple out-or-order issue processors.

Manuscript received February 9, 1998.
Manuscript revised August 24, 1998.

[†]The authors are with the Dept. of Electrical Engineering, Korea Advanced Institude of Science and Technology, 373–1, Kusong-dong, Yusong-gu, Taejon, 305–701, Korea.

1.1 Previous Works

In CDC 6600, *scoreboarding* [14] was applied in order to execute instructions out of order when there are sufficient resources and no data dependencies. The scoreboard keeps all dependencies between instructions, but does not handle the cases having data dependencies. For example, instructions dependent on the result of a preceding instruction must wait until the instruction writes its result to the register file because the result is never forwarded.

Tomasulo's dependency-resolution algorithm was first proposed for the floating-point unit of the IBM 360/91 [2], [16]. The algorithm operates as follows. An instruction whose operands are not available when it enters the decode and issue stage is forwarded to a *Reservation Station* (*RS*) associated with the functional unit to be used. The instruction can resolve its dependencies by monitoring the Common Data Bus, and waits in the RS until all the data dependencies have been resolved. When all the operands for the instruction are available, it is dispatched to the corresponding functional unit for execution. While this algorithm is straightforward and effective, it does not support the precise interrupt. An extension of this algorithm for the scalar unit of the CRAY-1 is presented in [10]. Patt and his colleagues have also described an extension of Tomasulo's algorithm called HPSm [4].

Several methods have been proposed to solve the precise interrupt handling problem. *Checkpoint repair* was proposed to recover processor states from mispredicted branches and interrupts [5]. The method needs a tremendous amount of logical space to save temporal process states at checkpoints, but the contents of these spaces differ by only a few locations, depending on the number of results produced between checkpoints. The storage complexity is the most serious disadvantage of the method.

Smith and Pleszkun have proposed three methods, *reorder buffer, history buffer, and future file*, to implement the precise interrupts in pipelined scalar processors [11]. The *reorder buffer* is used to keep the in-order sequence of instructions before they modify the process states. When an instruction is decoded, the instruction is assigned to a reorder-buffer location, and its destination register number is associated with the location.

This means the reorder-buffer location is used to rename the destination register. But the method requires associative tag-matchings in the reorder buffer when a following instruction refers the value stored in the reorder buffer, and takes a long time to recover the register contents at the time of interrupts. To reduce the number of bypass comparators needed and the amount of circuitry required for the multiple bypass checks, another read port is added in the *history buffer or future file* scheme.

Sohi has extended Tomasulo's dependency resolution algorithm and proposed *RUU* (*Register Update Unit*) to combine dependency-resolution and precise interrupts [12]. The register identifier sent to RUU consists of the register number appended with the LI (Latest Instance) counter. This guarantees that future instructions access the latest instance and instructions in the RUU receive correct data. Although there is no associative circuitry within RUU, the scheme needs the result storage and the bypass logic in RUU because the latest copies of the register contents are placed not in register file but in RUU.

The Metaflow architecture executes instructions out of order and speculatively via *DRIS* (*Deferred-scheduling, Register-renaming Instruction Shelf*) [8]. Entering an instruction into DRIS automatically and uniquely renames its destination register. The issue logic determines true data dependencies on preceding uncompleted instructions by searching DRIS. When the instruction's result becomes available at update time, the value is shelved adjacent to the instruction in DRIS. The unique destination register name for the result is used at update time to search DRIS for operands locked on the result. Since the execution results must be shelved in DRIS until the results of all preceding instructions are retired to the register file, the result shelves are needed to store the unretired results in DRIS as the reorder buffer is.

1.2 Outline of the Paper

In this paper we concentrate on register state recovery in multiple out-of-order issue processors, because the other states such as program counter and memory contents can be recovered by methods similar to [11]. We propose a fast precise interrupt handling scheme which does not require any associative searching. To remove the associative searchings which are inevitable in the previous approaches, the scheme renames the destination register to a register entry of another register bank in place of a reorder-buffer location. We also show that the scheme reduces the recovery time by using two register bank index tables.

This paper is organized into 4 sections. The proposed fast precise interrupt handling scheme is presented in Sect. 2. Sections 3 and 4 describe the experimental environment and results, respectively.

2. The Precise Interrupt Handling Scheme

We propose a new precise interrupt handling scheme which removes the associative searchings within the reorder buffer and takes only one cycle time at the time of interrupts to recover the in-order states in multiple out-of-order issue processors. Unlike the reorder buffer, history buffer, or future file scheme, this method efficiently implements precise interrupts by maintaining two bank index tables and a bank-based register file instead of using additional bypass paths, register ports, compare operations or associative searchings.

2.1 Model Architecture

As a basic architecture, we choose a register-register architecture where all memory accesses are through registers and all functional operations involve only registers as their sources and destinations. A parallel pipelined implementation for the architecture is shown in Fig. 1.

Our scheme consists of an instruction decoder, a bank-based register file, functional units, reservation stations, a reorder buffer, and a register identifier calculation unit. This architecture uses an instruction fetch/decode pipeline which processes instructions sequentially as specified in the program. Overall operation of the model architecture is described below:

a. Decode & Issue: The instruction decoder interpretes multiple instructions and places them and their operands into the reservation stations of the appropriate functional units. If there are uncompleted instructions having the same destination register, the entry of the recently-updated bank index table for the destination register is incremented. The destination register identifiers are calculated in the register identifier calculation unit using the recently-updated bank index table. The reorder buffer keeps the issue sequence to recover in-order states at the time of interrupts.
b. Execute: A functional unit executes an instruction out of order if all of its operands are valid and the functional unit is not busy. Operands may be valid during the decode stage or may become valid when the required operands are computed and forwarded to the reservation station. The instruction remains in the reservation station until all dependencies are released and the functional unit is available. After the functional unit executes the instruction, it announces its completion and forwards the result to all reservation stations.
c. Write-Back: The result of a functional unit is written directly to the bank-based register file using the destination register identifier in the reservation station, and the status of execution is written to

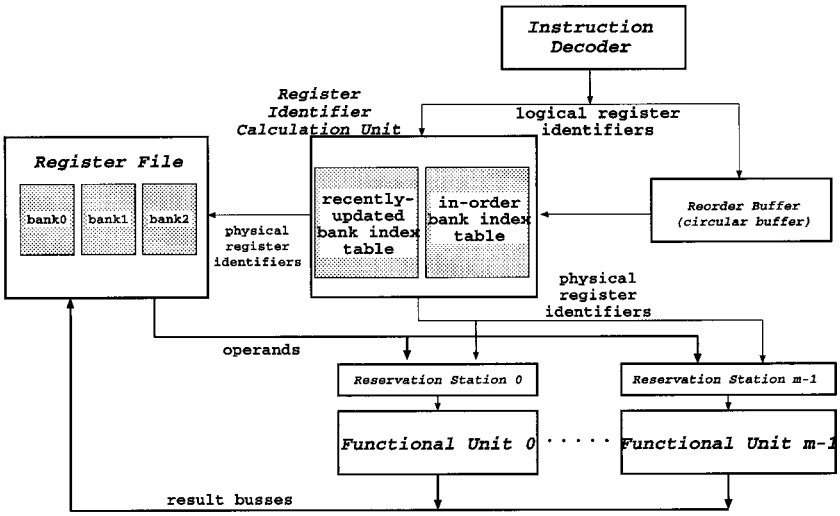

Fig. 1 Block diagram of model architecture.

Issue order	Operation	Completion order
1	**R3** ← R1 op R2	5
2	R4 ← R3 op R4	6
3	R5 ← R5 op R6	3
4	R8 ← R7 op R8	4
5	**R3** ← R9 op R10	1
6	**R3** ← R2 op R11	2

Fig. 2 Example of out-of-order issue instructions.

a reorder buffer entry pointed to by the tag field in the reservation station. If all instructions issued before the instruction are completed, meaning that the instruction reaches to the head point of the reorder buffer, checks are made to determine if there were any exceptions during the execution. In the case of exceptions, the precise states can be recoved by writing the in-order bank index table to the recently-updated bank index table.

For the sake of easy understanding, we will explain the scheme using an example shown in Fig. 2 throughout this paper. Although it would be better to include multiple instructions at each issue, we have used simple instruction at each issue for clear explanation. In addition, the number of instructions to be issued is dependent on the issue rate. In Fig. 2, the *issue order* column represents the issue sequence which is the same as the program sequence and the *completion order* column denotes the completion sequence.

Each instruction is decoded in the *issue order* in the instruction decoder and the decoded information is placed in a reorder buffer entry and a reservation station entry. According to the data dependencies and functional unit availability in the reservation station, the *completion order* may be different from the *issue order*. For example, *issue order 1, 5, and 6 instructions* write their execution results to the same register *R3*. So register renaming is necessary for solving WAW (write after write) and WAR (write after read) dependencies [3], [7]. Traditional register renaming schemes assign different registers to destination registers of *issue order 5 and 6 instructions*, while the proposed scheme assigns the same register entry in *different register banks* of the bank-based register file.

2.2 Bank-Based Register File

The configuration of a bank-based register file is similar to that of a general register file, except that it is partitioned into several banks as shown in Fig. 3. It has as many write ports as the number of instructions issued per clock, and twice as many read ports. There is no additional read port on the bank-based register file, unlike the history buffer or future file scheme that needs to write the current destination register content into a reorder buffer entry.

Every bank has the same number of register entries and can be accessed through physical register identifiers. In the above example, if the result of *issue order 1 instruction* is assigned to the fourth register (R3) of bank (i), that of *issue order 5 instruction* is written to the same register entry (R3) of bank ($i + 1$). And the result of *issue order 6 instruction* must be written to the fourth register (R3) of bank ($i + 2$). So whenever the register renaming is required, the result is written to the same entry of the next register bank. Its physical address is simply calculated by concatenating the bank index and the register index.

To identify the bank of each register to be allocated for renaming, we maintain two bank index tables which will be described in the next section. A problem of the

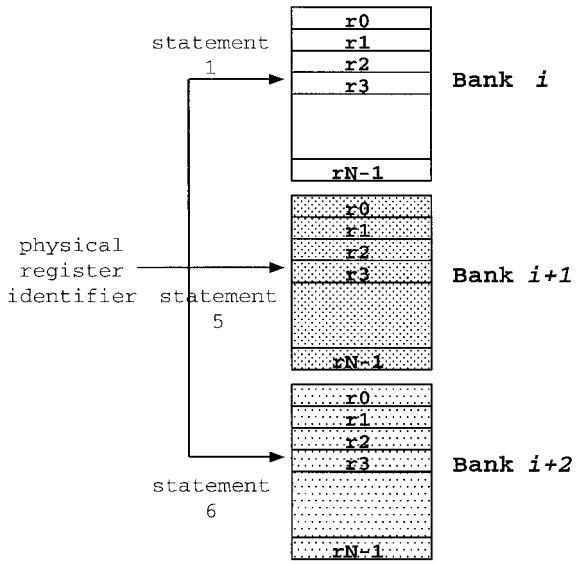

Fig. 3 Bank-based register file.

scheme is that the register file size must be increased according to the number of register banks. The number of register banks is dependent on how many register renaming happens continuously for a register. To minimize the register file size without sacrificing much performance, we must find the proper number of register banks.

2.3 In-Order Bank Index Table and Recently-Updated Bank Index Table

To record the in-order bank index and recently updated bank index of each register entry, there are two index tables, *In-order Bank Index Table* (*IBIT*) and *Recently-updated Bank Index Table* (*RBIT*). *IBIT* is so consistent with the sequential architectural state defined by program sequence, that it is updated when an instruction completes in order. On the other hand, an entry of *RBIT* is incremented by one whenever an instruction is issued. The instruction issue should be blocked if the incremented value of the *recently-updated bank index* is equal to the *in-order bank index*. The processor must stall in order not to loss in-order states until the corresponding *IBIT* is updated. We can avoid WAW (write after write) and WAR (write after read) hazards using the tables instead of the traditional register renaming schemes. *IBIT* and *RBIT* contain the bank pointer for each register entry as shown in Fig. 4. The total bit width of each table is given by

$$\log(\textit{number of register banks}) \times (\textit{number of register entries in each bank}).$$

Although a log (*number of register banks*)-bit counter is needed for each index table entry, the hardware complexity is negligible compared to the register file area.

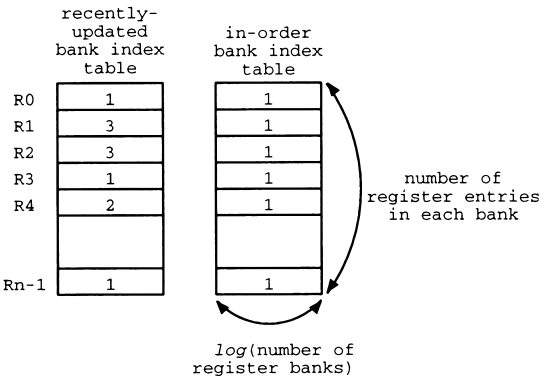

Fig. 4 In-order and recently-updated bank index tables.

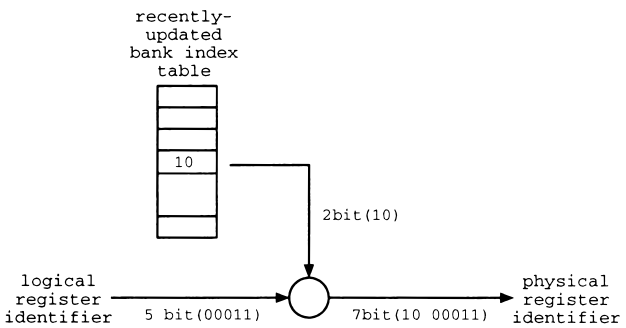

Fig. 5 Register identifier calculation method.

Overwriting *IBIT* to the *RBIT* is the only operation to deal with precise interrupts. After this operation is done, both IBIT and RBIT point to the in-order register bank states. Hence, our scheme achieves processor state recovery in one cycle as the future file scheme recovers correct states by clearing tag or validity bits. RBIT is accessed in the instruction decoding stage and in the correct state recovering. On the other hand, in the future file scheme, tags are accessed not only in the above stages but also in write-back stage.

2.4 Physical and Logical Identifier for Register File

As RBIT points to the bank having the most recent value of each register in out-of-order execution, the bank indices are used to calculate the register identifiers when the following instructions access registers. Figure 5 illustrates the register identifier calculation method, where we assume the register file has 4 banks, 32 entries per bank. In Fig. 5, the logical register identifier obtained from instruction field is concatenated with the 2-bit recently-updated bank index to make the 7-bit physical register identifier to be used to access the register file.

2.5 Reorder Buffer

The reorder buffer shown in Fig. 6 (a) is a circular buffer having head and tail pointers. Unlike the reorder buffer

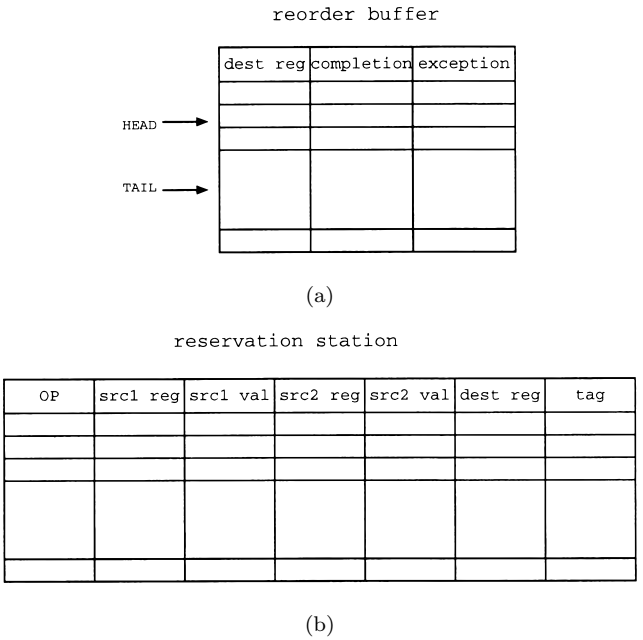

Fig. 6 Configuration of the reorder buffer (a) and reservation station (b).

proposed by Smith and Pleszkun [11] and the result shelves of DRIS [8], there is no need to include the *result* column in our reorder buffer, because the recently updated result value is directly written to the bank-based register file. Entries between the head and tail are considered valid. When an instruction is issued, the next available reorder buffer entry, pointed to by the tail pointer, is allocated to the instruction. The current value of tail pointer is used as a tag to identify the entry in the reorder buffer reserved for the instruction. When the instruction completes, the corresponding functional unit sends both completion and exception conditions to the reorder buffer. When the completion bit of the entry at the head of the reorder buffer is set, meaning that the instruction has finished in order, exceptions are checked. If any exception is detected, all entries of RBIT is made equal to those of IBIT to go back to the in-order processor state.

The source operands are obtained only from the register file, as execution results are written directly to the register file. There are no additional write paths from the register file to the reorder buffer, and no bypass paths from functional units to the reorder buffer.

2.6 Reservation Station

Reservation stations shown in Fig. 6 (b) partition the instruction window by the number of functional units [6], [16]. The following steps are taken to issue instructions from a reservation station:

a. Allocate the reservation station entry for a decoded instruction. The destination register of the instruction is assigned by the register identifier calculation unit using RBIT.
b. Identify entries containing instructions ready for execution. An instruction is ready when all of its operands are valid. Operands may have been valid during decode or may become valid when results are computed and forwarded to the reservation station through result busses.
c. If many instructions are ready for execution, select one instruction to execute.
d. Execute the selected instruction in the functional unit.
e. Deallocate the reservation station entry containing the executed instruction so that the entry can receive a new instruction. For the best utilization of the reservation station, the entry should be able to receive a new instruction from the decoder in the next cycle.
f. Monitor the result busses for matching source operand tags.

Our reservation station is similar to that proposed in [6] except the *tag* field. The tag is used to directly identify the reorder buffer entry needed to write completion and exception conditions, when the instruction is completed. Therefore, the associative searching within the reorder buffer can be completely eliminated. Even in the future file scheme that has no associative searching, a selection logic is needed to provide operands because register identifiers are applied to both the register file and the future file [6]. Hence, our scheme that does not require the associative searching and the selection logic makes the in-order state recovery fast.

2.7 Procedure

Figure 7 shows the changes of IBIT and RBIT after each instruction in Fig. 2 is issued. We assume the initial state of each bank index table entry is given as shown in Fig. 7 (a). Changed entries are highlighted with bold characters. On the next cycle RBIT must change the recently-updated bank index. Figure 7 (b) shows that *R3* entry in RBIT is changed from *1* to *2* after *issue order 1* instruction is issued. The value represents that the recently updated value of *R3* is located at register bank *2*. The value in IBIT is not changed, because the instruction is not completed at the time. The issue of *issue order 5 instruction* results in the change of *R3* entry in RBIT from *2* to *3* (Fig. 7 (c)). So does the issue of the *issue order 6 instruction* (Fig. 7 (d)).

If an interrupt occurs at the time of Fig. 7 (c) or (d), all entries of IBIT are overwritten to those of RBIT. So the processor can recover the initial state, i.e., before *issue order 1, 5 and 6 instruction* are issued, in one cycle as shown in Fig. 7 (e). Once *issue order 1 instruction* is completed in the program sequence, IBIT is updated to save the in-order state (Fig. 7 (f)).

Compared to the previous schemes, the proposed precise interrupt scheme has advantages described below.

- removal of result writing to the reorder buffer,
- unnecessity of the result column in the reorder buffer,
- unnecessity of write-back from the reorder buffer to the register file,
- simplicity of the reorder buffer organization,
- direct identification of the reorder buffer entry to write completion and exception conditions,
- complete elimination of associative searching in reading operands and writing a result,
- and removal of the operand selection logic.

On the other hand, there are also points to be investigated:

- the amount of register array increases in proportion to the number of banks,
- and depth of register renaming is limited to the number of banks, that can degrade performance qualitatively.

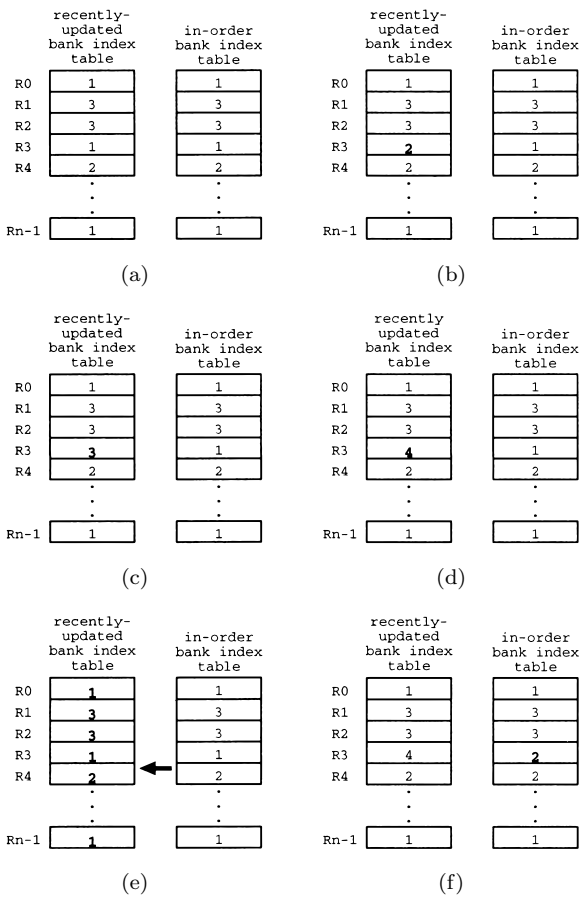

Fig. 7 Changes of RBIT and IBIT. (a) initial states, (b) after issue order 1 is issued, (c) after issue order 5 is issued, (d) after issue order 6 is issued, (e) interrupt handling when an interrupt occurs at (c) or (d), (e) after issue order 1 is completed without exception.

To check the points, now we have to determine the proper number of register banks which enables high performance and low hardware cost.

3. Experiment Environment

We have used *pixie*^{TM} [9], [17] and the simple superscalar trace-driven simulator [13] to decide the number of register banks. The instruction traces generated by using *pixie*^{TM} is simulated as shown in Fig. 8. The object code for a program is processed by *pixie*^{TM} that reads an executable program, partitions it into basic blocks, and annotates the object code at the target of each branch and at each memory reference. The annotated code has the same behavior as the original program. However, when the annotated code is executed, the instructions added by *pixie*^{TM} generate the dynamic trace streams as well as the normal outputs of the program. This trace stream consists of branch target addresses, the number of instructions up to the next branch, and the addresses referenced by loads and stores. The simulator takes the trace stream as input and generates the full instruction trace and various statistics data.

The trace-driven simulator can support both scalar and superscalar processors that are based on the MIPS architecture. By changing machine parameters and machine configuration, we can specify the machine to be simulated. The simulator collects the instruction and/or data trace of the specified MIPS processor.

Six realistic SPEC95 benchmarks shown in Table 1 were used to evaluate the effect of issue rate on the register renaming depth.

The reservation stations of the functional units do not have to be of the same size. Table 2 shows the machine configuration selected as a typical model. We limit the number of reservation station entries for simplicity. But stalling of instruction issues by the number

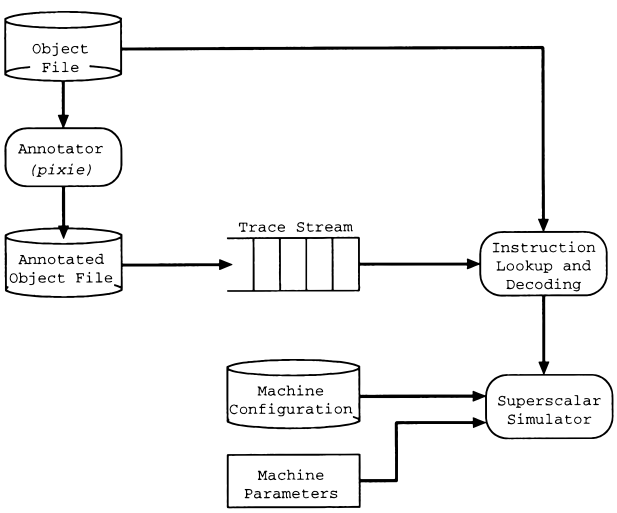

Fig. 8 Flow chart for simulation.

Table 1 Benchmark program description.

Program	Description
go	GO game
m88ksim	MC88100 instruction simulator
compress	file compression using Lempel-Ziv encoding
li	LISP interpreter
espresso	logic minimization
ijpeg	JPEG compressor and decompressor

Table 2 Configuration of functional units.

Functional Unit	Issue Latency	Result Latency	Reservation Station Entries
Integer ALU (x2)			4
-single	1	1	
Shifter			2
-single	1	1	
Branch			4
-single	1	1	
Load & Store			8
-single	1	2	
-double	2	3	
Float_Add			2
-single	1	2	
-double	1	2	
Float_Mul			2
-single	1	4	
-double	1	5	
Float_Div			2
-single	12	12	
-double	19	19	
Float_Conv			2
-single	1	2	
-double	1	2	

of reservation station entries is included in simulation model. We assume the register file has 32 entry per bank. The reorder buffer is assumed to have infinite entries to evaluate the maximal depth of register renaming. And we simulate these organizations with changing the issue rate from 2 instructions to 8 instructions.

4. Results

4.1 Effect of Register Renaming on Performance

Table 3 shows the effect of issue rate, ranging from 2 to 8, on the register renaming occurrence for the configuration shown in Table 2. In the table, register renaming of depth 0 means that register renaming is not needed. The other depths represent the number of register renamings required to achieve the best performance. The table indicates that about 96% of register renaming occurs within depth 2, but seldom occurs for above 2 (less than 4%).

Table 4 shows the effect of number of banks and reorder buffer size on performance for various issue rates. The future_xx column in Table 4 indicates the performance of the future file scheme with xx-entry reorder buffer. Compared to the future file with 32 reorder-buffer entries which has similar hardware complexity, 3-bank register file degrades the performance by 1.35% on average, but higher than the future file with 16 reorder-buffer entries. The performance is significantly decreased for 2-bank register file. The simulation re-

Table 3 Effect of issue rate on Register renaming for the configuration shown in Table 2 (unit: %).

Program	Issue rate	Register renaming depth				
		0	1	2	3	4+
go	2	77.71	8.18	13.53	0.48	0.09
	4	76.70	5.02	16.38	1.69	0.21
	8	76.35	5.53	15.36	1.80	0.95
m88ksim	2	92.53	4.41	3.04	0.02	0.00
	4	86.52	10.08	3.19	0.19	0.01
	8	86.31	10.18	3.29	0.21	0.01
compress	2	87.26	9.83	1.97	0.91	0.02
	4	83.92	11.71	2.08	1.28	1.01
	8	83.31	11.82	2.52	0.85	1.50
li	2	95.22	1.64	3.11	0.00	0.03
	4	92.95	3.84	3.18	0.00	0.03
	8	90.54	6.16	3.28	0.00	0.03
espresso	2	87.87	8.93	1.95	0.14	1.10
	4	83.32	10.53	2.21	1.42	2.52
	8	80.13	12.89	2.77	1.44	2.77
ijpeg	2	90.12	5.33	2.48	0.75	1.32
	4	81.37	12.13	4.39	0.67	1.43
	8	79.53	12.95	5.31	0.73	1.47

Table 4 Effect of the number of banks and reorder buffer entries on performance. Assume the performance of the configuration that has an 32-entry reorder buffer is 100%.

Program	Configuration	Issue rate		
		2	4	8
go	2 bank	85.89	81.72	81.89
	3 bank	99.42	98.09	97.25
	4 bank	99.91	99.78	99.05
	future_12	91.24	88.78	88.57
	future_16	93.54	91.69	91.56
m88ksim	2 bank	96.94	96.61	96.49
	3 bank	99.97	99.80	99.78
	4 bank	99.99	99.08	99.99
	future_12	97.71	97.35	97.01
	future_16	99.41	99.24	99.19
compress	2 bank	97.19	95.63	95.13
	3 bank	99.06	97.71	97.65
	4 bank	99.98	98.99	98.50
	future_12	95.54	93.04	91.46
	future_16	99.04	98.28	96.65
li	2 bank	96.86	96.79	96.69
	3 bank	99.97	99.97	99.97
	4 bank	99.97	99.97	99.97
	future_12	99.92	99.91	99.87
	future_16	99.96	99.96	99.95
espresso	2 bank	96.80	93.84	90.54
	3 bank	98.75	96.06	92.69
	4 bank	98.90	97.48	95.14
	future_12	97.78	95.25	91.97
	future_16	98.25	96.00	92.17
ijpeg	2 bank	95.45	93.50	92.48
	3 bank	97.93	97.89	97.79
	4 bank	98.69	98.57	98.53
	future_12	95.53	94.82	94.36
	future_16	97.34	96.96	96.43

Table 5 Hardware cost comparison.

	Future file	Our scheme	Gain
Equivalent gates	20,261	19,399	4.2%
Area (mm^2)	6.689	6.308	5.7%

sults show that 3-bank register file is proper for high performance. Thus we conclude that 3-bank register file can support fast precise interrupts with negligible performance degrade compared to the future file scheme that has almost the same hardware complexity. In this case, the processor must stall when more than 2 register renamings are needed for a register.

Register renaming may occur more frequently when issue rate increases, as the number of decoded instructions increases and a large number of instructions increases the chance to have the same destination registers. However, Table 3 shows that the increase of register renaming is not great for benchmarks even in the case of 8 issue rate.

4.2 Hardware Cost Comparison

The future file scheme consists of a traditional register file and a future file that has another read port. And validity and tag bits are associated with each entry of the future file and a selection logic is required to provide operands. On the other hand, our scheme requires three register file in the case of three banks. In addition, we can simplify the reorder buffer since there is no need to store results into the reorder buffer. The deleted hardware is almost equal to the size of one register file. However, a 2-bit counter is required for each index table entry in the case of three banks. To estimate the area required, both schemes are implemented using a $0.6\,\mu$m CMOS standard cell library. As shown in Table 5 our scheme saves hardware cost by 5.7% compared to the future file scheme. If we would count the removal of the path needed to write results to the reorder buffer and the associated routing area, the amount of hardware saving would be more. In addition, the control logic of our scheme is much simpler than that of other schemes.

Therefore, our scheme can achieve the fast precise interrupts with simple control logic and less hardware at the expense of a little performance degradation.

5. Conclusions

We have proposed a fast precise interrupt handling scheme based on a bank-based register file and two register file bank index tables. This scheme eliminates all the associative searchings which are inevitable in the previous reorder buffers. Instead of writing results into the reorder buffer, our scheme renames the destination register by using the same entry of another register bank. In addition, the tag associated with the reservation station guides the execution status directly to the corresponding reorder buffer entry. We also defined two register bank indices, in-order bank index and recently-updated bank index, to maintain in-order states and out-of-order states, respectively. When an interrupt occurs, the processor register contents can be recovered by simply copying the in-order bank index table into the recently-updated bank index table. This removes the register transfers in the reorder buffer and the operand selection logic in the future file. Hence, the removal of the associative searchings and the operand selection logic make the hardware structure of our precise interrupt scheme simple.

To evaluate the proposed scheme, simulations based on the superscalar MIPS architectures are performed for numerous benchmarks. The simulation results show that about 96% of register renaming occurs within depth 2. Therefore, 3-bank register file is an effective choice to implement fast precise interrupts. Compared to the register file with infinite banks, 3-bank register file degrades performance by 1.35% on average. Our scheme can save hardware cost by 5.7% than the future file scheme. Consequently, our scheme can support the fast precise interrupts with less hardware at the expense of a little degradation in performance.

Acknowledgment

The authors would like to thank the reviewers for their helpful suggestions.

References

[1] R.D. Acosta, J. Kjelstrup, and H.C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors," IEEE Trans. Comput., vol.C-35, no.9, pp.815–828, Sept. 1986.

[2] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, "The IBM system/360 Model 90: Machine philosophy and instruction-handling," IBM J. Res. Develop., vol.11, pp.8–24, Jan. 1967.

[3] J.L. Hennessy and D.A. Patterson, "Computer Architecture a Quantitative Approach," Morgan Kaufmann Publishers, Inc., 1990.

[4] W.-M.W. Hwu and Y.N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," Proc. 13th Annual Symposium on Computer Architecture, pp.297–307, June 1986.

[5] W.-M.W. Hwu and Y.N. Patt, "Checkpoint repair for out-of-order execution machines," Proc. 14th Annual Symposium on Computer Architecture, pp.18–26, June 1987.

[6] M. Johnson, "Superscalar Microprocessor Design," Prentice Hall, Englewood Cliffs, NJ, 1991.

[7] D.A. Patterson and J.L. Hennessy, "Computer Organization & Design: The Hardware/Software Interface," Morgan Kaufmann Publishers, Inc., 1994.

[8] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lighter, and D. Isaman, "The metaflow architecture," IEEE Micro, pp.10–13, 63–73, June 1991.

[9] MIPS Computer Systems, Inc., "MIPS Language Programmer's Guide," Sunnyvale, CA., 1986.

[10] R.M. Russel, "The CRAY-1 computer system," CACM, vol.21, pp.63–72, Jan. 1978.
[11] J.E. Smith and A.R. Pleszkun, "Implementation of precise interrupts in pipelined processors," IEEE Trans. Comput., vol.37, no.5, pp.562–573, May 1988.
[12] G.S. Sohi, "Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers," IEEE Trans. Comput., vol.39, no.3, pp.349–359, March 1990.
[13] M. Johnson and M.D. Smith, "ssim: A Superscalar Simulator," Stanford University, Stanford, 1993.
[14] J.E. Thornton, "Design of a computer—The control data 6600," Glenview, IL:Scott, Foresman, 1970.
[15] G.S. Tjaden and M.J. Flynn, "Detection and parallel execution of independent instructions," IEEE Trans. Comput., vol.C-19, pp.889–895, Oct. 1970.
[16] R.M. Tomasulo, "An efficient algorithm for exploring multiple arithmetic units," IBM J. Res. Develop., vol.11, pp.25–33, Jan. 1967.
[17] M.D. Smith, "Tracing with pixie," Technical Report CSL-TR-91-497, Stanford University, Stanford, CA 94305-4055, Nov. 1991.

Sang-Joon Nam received the B.S. and M.S. degrees in Electrical Engineering from KAIST (Korea Advanced Institude of Science and Technology), Korea in 1993 and 1995, respectively. He is currently pursuing the Ph.D. degree in Electrical Engineering in KAIST. His current research interests include multiple-issue microprocessor design and multimedia VLSI design.

In-Cheol Park received the B.S. degree in Electrical Engineering from Seoul National University in 1986, the M.S. and Ph.D. degrees in Electrical Engineering from KAIST (Korea Advanced Institude of Science and Technology), in 1988 and 1992, respectively. From May 1995 to May 1996, he worked at IBM T.J. Watson Research Center, Yorktown, New York as a postdoctoral member of the technical staff in the area of circuit design. He joined KAIST in June 1996 as an Assistant Professor in the Department of Electrical Engineering. His current research interest includes CAD algorithms for high-level synthesis and VLSI architectures for general-purpose microprocessors.

Chong-Min Kyung received the B.S. degree in Electronic Engineering from Seoul National University, Korea in 1975, and the M.S. and Ph.D. degree in electrical engineering from KAIST (Korea Advanced Institude of Science and Technology), Korea in 1977 and 1981, respectively. After graduation from KAIST, he worked at AT&T Bell Laboratories, Murray Hill, NJ, from April 1981 to January 1983 in the area of semiconductor device and process simulation. In February 1983, he joined the Department of Electrical Engineering at KAIST, where he is now a Professor. His current research interests include microprocessor design, VLSI CAD, computer graphics, and DSP chip design.