The Schema of Interrupt Controller

Jih-Fu Tu

The Department of Electronic Engineering, St. John's University, Taiwan

Abstract

Exceptions or interruptions control is the most challenging aspect while designing a processor, and the hardest work of exception control is interruption among produces. In this paper, we embedded an event controller (EC) into an RISC architecture processor to handle when interruption occurring, then to reduce the latency time when context switch between user program and kernel program. To analyze the performance, we also compare the cost/performance（C/P） ratio and the C/P improved ratio of the proposed processor in different entry number of a reorder buffers.

Keyword: Event controller (EC), cost/performance (C/P), external interrupt, and interrupt priority comparator.

1. Introduction

Exceptions or interrupts are an event that causes an unexpected change in control flow and a temporary break in program execution, also. So the processor changes the normal flow of instruction execution to handle another chore. The interrupts commonly request I/O services or synchronize the processor with some external hardware activity. An exception is an unexpected event from within the processor; arithmetic overflow /underflow is an example of the exceptions. Exception handling is a mechanism to flexibly and with low implementation cost handle exceptional events in a way that doesn't impact the execution of the common case.

When an interrupt or exception occurs, which changes the flow of control of a program by means other than a branch instruction. They are triggered by the activation of interrupt signals, which denoted by *Int* [*i*], *i*=0,1,2,3… The activation of an *Int* [*i*] should result in a procedure call of a routine, this routine is called the exception handler for interrupt i and should take care of the program signaled by the activation *Int* [*i*].

The interrupts are initially created to handle unexpected events and to signal requests for service from I/O devices. Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a machine, which determines

the clock cycle time and thus performance [1,2].

When the interrupt call is made, the processor uses a dedicated interrupt stack. A new frame is created for interrupt on this stack and a new set of local registers is allocated to the interrupt procedure. The interrupted program's current state is also saved. Upon return from the interrupt procedure, the processor restores the interrupted program's state, switches back to the stack that the processor was using prior to the interrupt and resumes program execution.

Since interrupts are handled based on priority, requested interrupts are often saved for later service rather than being handled immediately. In this paper, we proposed an event controller, called EC. The machine for saving the interrupt is referred to as interrupt posting. On the EC, interrupt request may originate from external hardware sources, internal exceptions from software.

To simulate and prove the EC that is available, the EC is constructed in DLX pipeline architecture processor. Also, we compare the cost/performance of the processor with EC to the DLX processor without EC.

The rest of this paper is organized as; in Section 2 we provide a detailed procedure for exception processor, called EC. In section 3, we describe the algorithm of the interrupt priority for EC. In Section 4, we analysis the simulation result and compare the cost of the processor with EC to the processor without the CApro. Finally, we remark that the conclusion and future works in Section 5.

2. The Structure for the Exception Processor

We consider interrupts belong to classes. 1) Program interrupts, sometimes referred to as "traps", "fault", and "abort" result from exception conditions detected during fetching and execution of specific instruction which is shown as Figure 1, such as, numerical errors, i.e. overflow, and page faults. 2) External interrupts are not caused by specific instructions and are often caused by source outside the currently executing process which is shown as Figure 2, such as, I/O interrupts and timer interrupts [2].

The exception processor's primary functions are to provide a flexible, low-latency means for requesting and posting interrupts and to minimize the core's interrupt handling burden. The exception processor handles the posting of interrupts requested by hardware and software source. The exception processor acts independently from the core, compares the priorities of posted interrupts with the current process priority, off-loading this task from the core.

If we want to design an exception processor and prove that it works, we must to do the usual three things: 1) define what an exception mechanism is supported to do, 2) design the mechanism, and 3) show that it meets the specification. To manage and

prioritize all possible interrupts, the EC integrates an on-chip programmable interrupt controller.

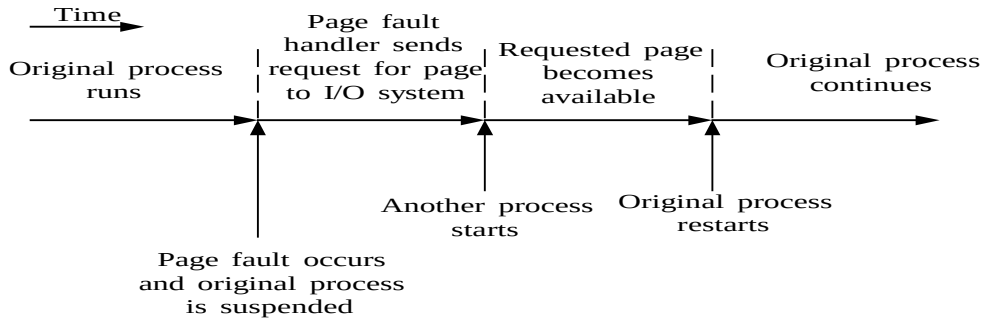

Fig. 1. Program interrupts processing [1]

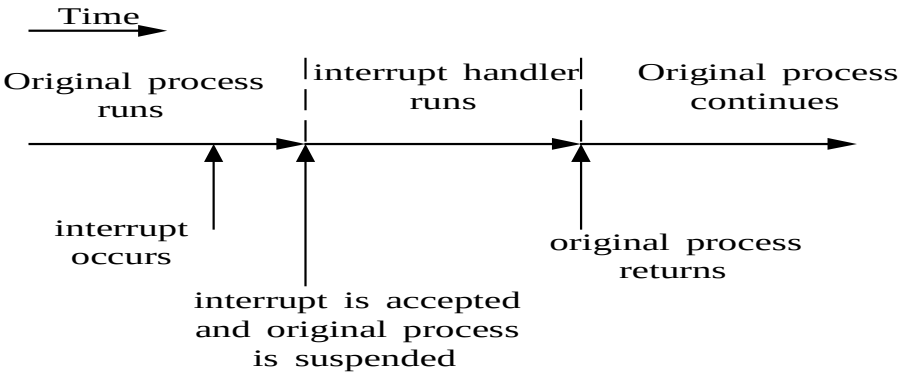

Fig. 2. Typical interrupt processing between instructions [1].

The EC architecture defines two data structure to support interrupt processing: the interrupt table and interrupt stack (shown in Figure 3) [3]. The interrupt table contains interrupt vectors for interrupt handling procedures and an area for posting software requested interrupts. The interrupt stack prevents interrupt handling procedures from overwriting the stack in use by the application program. It also allows the interrupt stack to be located in a different area of memory than the user and supervisor stack.

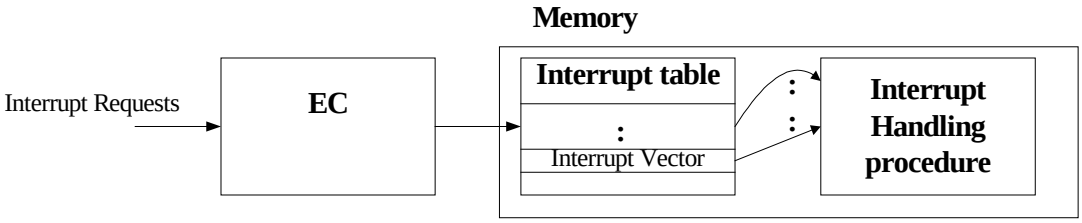

Fig. 3. Interrupt handling data structure for EC [3].

3. The Interrupt Priority for EC

To provide transparent prioritization of the all possible interrupts, each interrupt

vector has into different levels of priority. Every interrupt request is associated with interrupt vector in the interrupt table and has priority number. The lower vector number is assigned the higher priority. When multiple interrupt requests are pending at the same priority level, the highest vector number is serviced first.

The EC compares its current priority with the interrupt request priority to determine whether to service the interrupt immediately or to delay service through an interrupted comparator. The interrupt requests consist of external interrupt, i.e. hardware interrupt, and software exception. When an interrupt request is detected, then the interrupt is posted. Interrupt comparator compare the priority of the current procedure to the requested interrupt to decide which to service by the core. The interrupt request is serviced immediately if the interrupt request priority is higher than the processor's current priority. If the interrupt priority is less than or equal to the processor's current priority, the processor does not service the request then the lower priority interrupt or exception is fall into pent.

The EC is consisted of several registers, named and addressed of the registers are listed in Table 1, physical circuits, i.e., comparator and multiplex (shown in Figure 4). Those registers form the *SPR* [*0*] to *SPR* [*5*] of the special purpose register file.

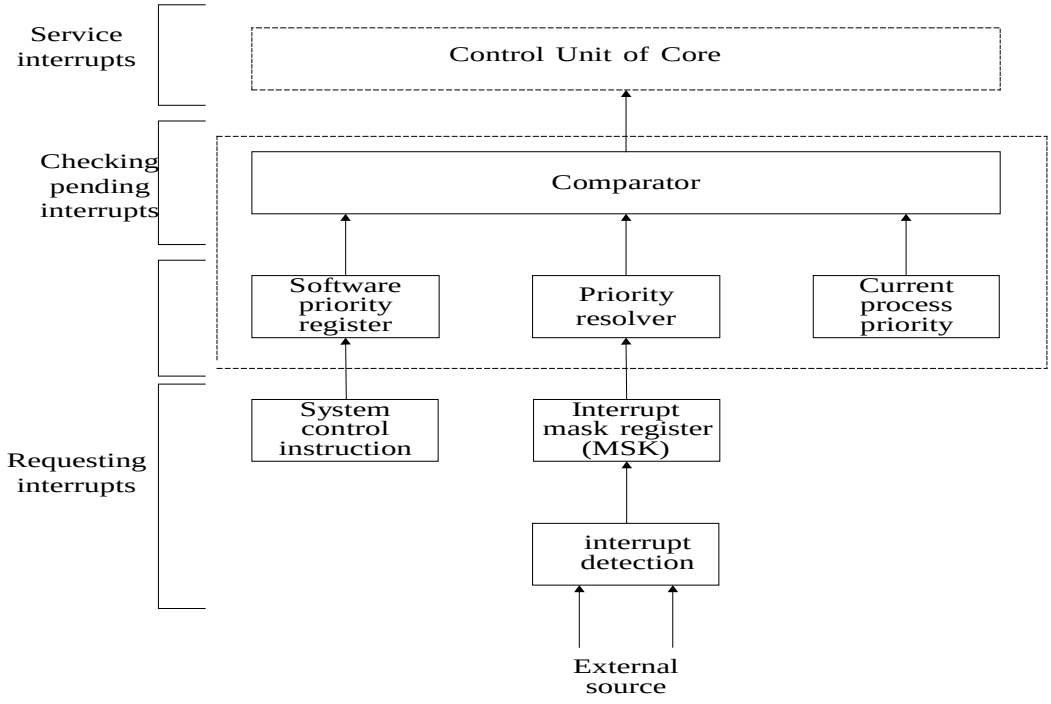

Fig. 4. The Schema of interrupts/exceptions procedure [3].

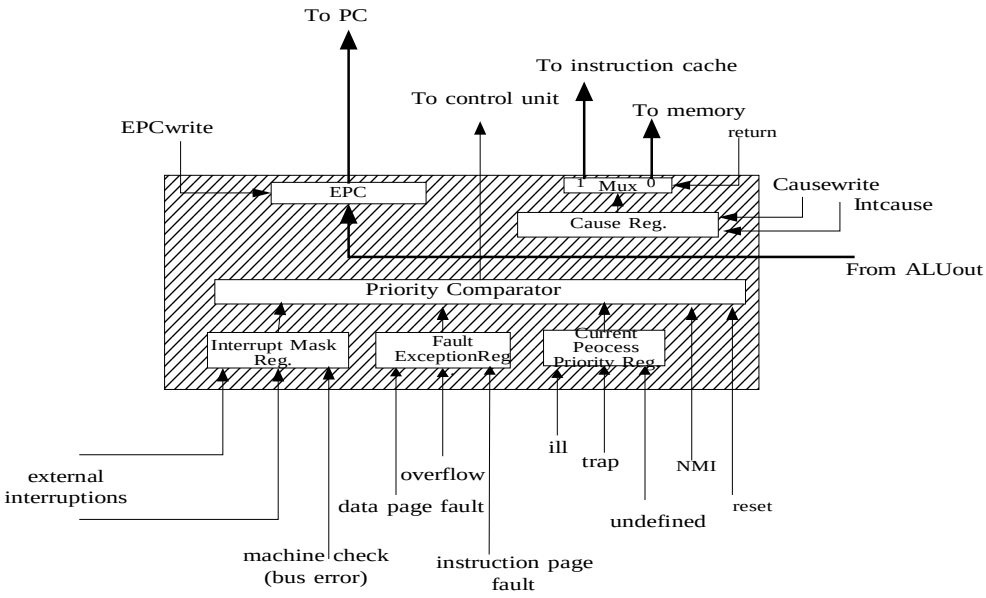

Fig. 5. The structure of the event controller (EC).

3.1 The produce of handling interrupts for EC

In this section, we design the interrupt hardware of the EC. The data paths get in the EC to collects the interrupt event signals and determines the interrupt cause.

The interrupt event signals are provided several pipeline stages; they are shown in Table 2. All kinds of interrupted events generated by the data paths and the control unit, but the stage in which a particular event is detected depends on the event itself. For the different kinds of events, after the interrupt handler completes that the user programs can be resumed in two ways: continue and abort.

When an interrupt, happened from external event or I/O device, or exception, caused by software, is occurred an event signal is posted from the interrupted source to the EC. When the EC receives this event signal, the following works are processed in interrupt routine.

1. checks this event whether mask or not?
2. compares and decide which has the highest priority,
3. saves (push) the states of the source instruction from GPR to SPR,
4. switches context to the interrupt handler, kernel model,
5. calls the exception or interrupt, routine and execute,
6. if the interrupt routine is completed, switch context from interrupt handler to user program,
7. returns (pop) the states of the source instruction from SPR to GPR, and
8. continues or aborts the original instruction.

TABLE 1. The definition special purpose register

Offset	Name	Meaning	Description
0	ESR	Exception status register	Saved the old masked interrupt routines
1	ECA	Exception cause register	Stored the masked interrupt cause
2	EPC	Exception program counter	Store the address of exception instruction
3	Edata	Exception data register	Stored parameter of the exception handler
4	MCA	Exception maskable register	Enabled the maskable
5	SR	Status register	Stored the maskable interrupts
6	CMP	Comparator	Compared the priority to the events

TABLE 2. The specific for all kinds of exceptions

Interrupt event	Signal	stage pipeline	Detected by which unit	resume
System clock	clock	Any stage	Kernel	Abort
System Reset	reset	Any stage	Kernel	Abort
Power failure	pw	Any stage	Kernel	Abort
Machine check	bus error	Any stage		Continue
Ill	ill	EXE	Control unit	Abort
Instr. misalignment	imal	IF	IMC	Abort
Data misalignment	dmal	MEX	DMC	Abort
Instruction page fault	ipf	IF	IMC	continue
Data page fault	dpf	MEX	DMC	continue
trap	trap	EXE	Control unit	continuer
Undefined instruction	undefined	ID	Control unit	Abort
Overflow	ovf	EXE	ALU	Abort
System call	call	ID	ALU	Continue
trace	trace	EXE		Continue
I/O device	external	Any stage	Interrupt processor	Continue
Thread interrupt*	ThInt	Any stage	Threaded dispatcher	Restart

Note: the threaded interrupt occurs for multithreaded processor when threaded slot has exception. For a threaded slot, the ThInt is defined an external interrupt for a threaded slot.

.

The above discussions of handler interrupt for the exception handler may be summarized in the following algorithm that is shown in Figure 6.

Algorithm Inthandler /handler interrupts/
Input: none
Output: none
{
detect interrupt types;
post interrupt;
save (push) current context layer;
determine interrupt source;
compare interrupt priority;
find interrupt vector;
call interrupt handler;
repeat:
check interrupt whether complete or not ?
if (full=1)
jump repeat;

else
if (restart=1 || continue =1)
jump original;
else stop;
original:
restore (pop) previous context layer;

Fig. 6. The algorithm of handling interrupts for exception handler.

4. Analysis the Results and Costs

4.1 Modeling the EC Use Petri Net

We design the interrupt hardware of the exception handler. The data paths get in the exception handler to collects the interrupt event signals and determines the interrupt cause. Figure 7 shows fundamental structure of the Petri net of interrupt programs and the user programs switch while occurs interrupt for processor. The symbol O represents place, where Information is store, and │ represents transition, which deals with transactions. The Petri net is a new feature that the weights called firing weights or introduced on the input arcs of the transitions and the signal called firing signal move with the token, is stand for ●, transfer [5].

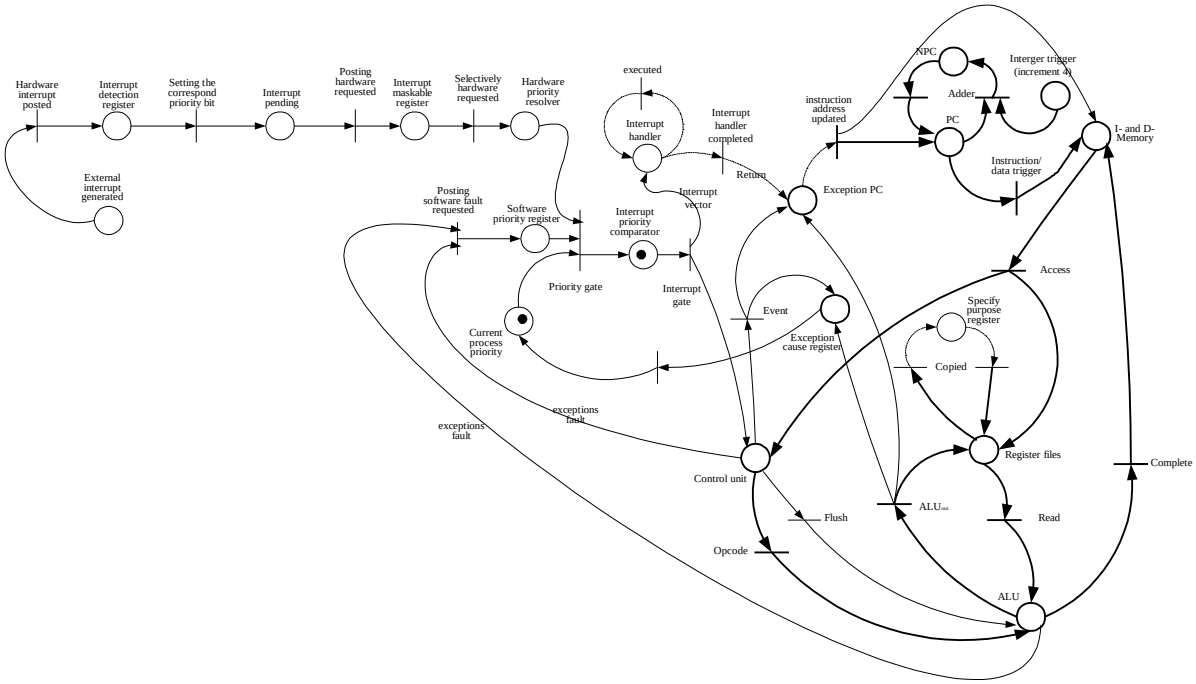

Fig. 7 The Petri net model of context switch between exception handler and user program.

4.2 Experiment the EC

In this section examines the performance benefits of using a dedicated entry to reduce overhead. To demonstrate the performance benefit of the exception handling for the EC, we experiments several exceptions and have implemented four kinds of entries using reorder buffer design logic. This research is performed using MIPS of the SimpleScalar tool kit [4], version 2.0. We chose SPEC CPU95 benchmarks, which are lists in Table 3, to compare the performance of different architectures. The produce of this simulation tools as following example.

Sim-outorder <command line switches> <benchmark binary> <benchmark command line>
Example : ***sim-outorder –config **.cfg –redir:sim ***.out /../go.ss 50 21 /../9stone21.in*** [7]

We obtained a simulate result via the ***.out file. If we want to progress the SPEC benchmark, we must have the SimpleScalar 2.0 and SPEC CPU95 or SPCE 2000 benchmark license. The benchmark binary is offers by SimpleScalar 2.0 and the benchmark command line is supplied by the SPEC benchmarks.

4.3 The IPC for Different EC Size

Figure 8 shows the IPC (Instructions per cycle) for four different event controller's entries in 4 thread slots across the benchmark suits. Refer to Figure 8, excepting the *perl* benchmarks, we obtain that the highest improve for IPC of each benchmark is happened in 8-entry. Though we increase the reorder buffer size of EC to 16-entry, the IPC does not be improved. This reason cause of the more entry number has the more copied time between general-purpose register (*GPR*) to specific purpose register (*SPR*) when an interrupt occurs and interrupt handler completed.

When the value of reorder buffers is small than 8-entry, the EC have expended long time in maintaining the exception data and status of exception data buffer (Edata), thus the CPU performance is reduced and the IPC does not be improved, too.

Obtaining the speedup ratio, the performance has clearly improved while the size of EC is 8-entry. For the benchmarks of *compress* and *ijpeg*, the IPC is up to 2.3.

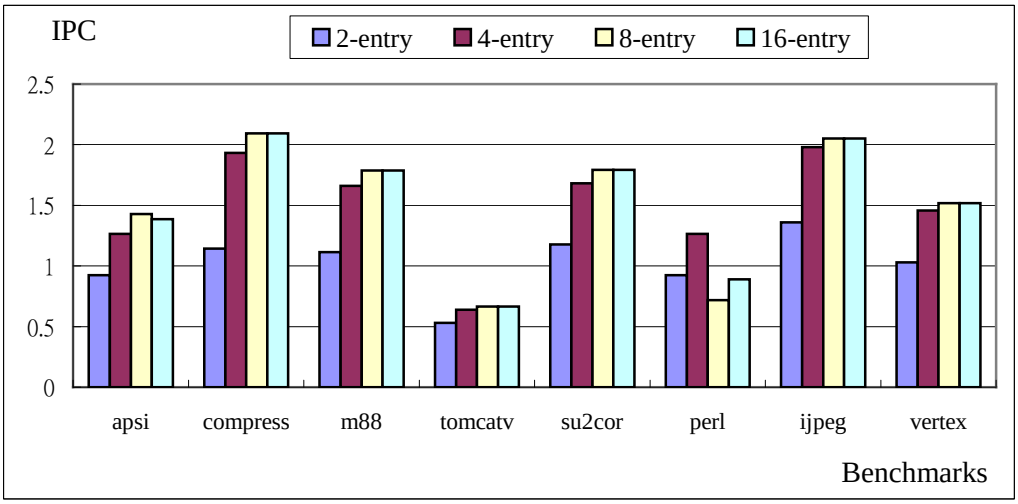

Fig. 7. The IPC when different entry's sizes in EC.

4.4 The Cost of EC

In this subsection, we compare the cost/performance (C/P) ratio for different entry number for the EC. The performance-improved ratio for each entry is defined: ***the latency time of 2^n-entry – the latency time of 2^{n+1}-entry***. The cost is defined the incremented entry number. In Table 3 illustrates the latency time of the EC for different entry numbers. Through the above formal calculates the C/P, which are listed in Table 4.

Observe to the benchmarks of Table 4, we found the best C/P ratio that occurred in 4 entries of EC. For *compress* benchmark, the maximum C/P is 1.3808 (referring to the third column of Table 4) and its C/P improvement ratio is 21% (the value of third column – the value of second column, i.e., (1.308-1.0957)/1,0957). The C/P ratio of other benchmarks, such as *apsi* is 73.3%, *m88* is 13.7%, *tomcatv* is 84%, *perl* is 16.8%, and *ijpeg* is 16.2%, respectively.

TABLE 3. The latency time of different entry for EC

Entries / Benchmark	1	2	4	8	16
Apsi	4.863	4.6515	2.7621	2.5177	2.5457
Compress	7.323	6.2273	3.6111	3.4389	3.4389
M88	4.672	4.2685	3.1868	2.6101	2.6105
Tomcatv	4.938	4.7864	2.7962	2.6808	2.6808
Su2cor	3.043	2.1578	1.6676	2.4112	2.4112
perl	5.156	4.6515	3.3165	2.2129	2.5751
ijpeg	3.845	3.5678	2.6681	2.7423	2.7428
vertex	2.946	2.6113	2.3095	2.5247	2.5247

TABLE 4. The cost/performance (C/P) ratio for EC

Entry increase	1increase to 2	2 increase to 4	4 increase to 8	8 increase to 16
aspi	0.2115	0.945	0.06	-0.0035
compress	1.0957	1.308	0.045	0
M88	0.4035	0.541	0.144	-0.0005
tomcatv	0.1516	0.99	0.026	0
Su2cor	0.8852	0.5	-0.19	0
perl	0.505	0.668	0.275	-0.045
ijpeg	0.278	0.45	-0.019	-0.0006
vertex	0.3347	0.151	-0.056	0

5. Conclusions and Future Works

In this paper, we introduce events control, EC, an architecture that increases event control performance. We evaluate this assumption control with SimpleScalar 2.0 simulation tool suits. Though the highest performance obtains from 8-entry EC architecture; the best cost/performance for different EC entry is 4-entry. The best cost/performance (C/P) is 1.308 that occurred in *compress's* benchmark and the best C/P improvement ratio is 84%, which occurred in *tomcatv's* benchmark.

For the future works, this topic has much rooms to work, such as, comparing the performance for different logical architectures, which are in-order method, history buffer, and future buffer, and explicit the EC architecture to control the exceptions among threads for multithreaded processor.

References

1. Silvia M. Mueller and Wolfgang J. Paul, "Computer Architecture Complexity and Correctness," Springer, 2000.
2. David A. Patterson and John L. Hennessy, "Computer Organization & Design the Hardware/Software interface," Morgan Kaufmann publishers, Inc. 1997.
3. Intel Corp., " Intel Pentium Processor User's Menu," Intel Corp. Lim., 1999.
4. D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1995.
5. T. Murata, "Petri nets: Properties, Analysis and Applications," *Proc. IEEE, Vol, 77, No. 4*, 1989, pp. 541-580.
6. E. Rotenberg, Q. Jacobesn, and J. Smith, "A Study of Control Independence in Superscalar Processors," *In proc. Of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.
7. SimpleScalar Mailing List Architecture, "Re: Inquiring about Spec95," http://ord.eecs.umich.edu/ss_archives/0093.html, pp. 1-4.