

Precise Exception Handling in Discontinuous Control Flow Scenarios for Area-Constrained Systems

Young Hoon Kang and Jeffrey Draper

Ming Hsieh Department of Electrical Engineering / Information Sciences Institute
University of Southern California
Marina del Rey, CA/USA
{youngkan, draper} @ISI.EDU

Abstract— Exception handling is one of the most complicated issues in pipelined processors. Several incomplete instructions are in process in the pipeline at any instant in time, and an exception may cause a state change of the processor [5] at any such instant. Prior research efforts have proposed mechanisms for precise exception handling, but it is difficult to achieve precise exception handling in minimal area as required by embedded and processing-in-memory systems. This paper presents a correct and efficient exception handling scheme with a modest hardware resource. The presented idea maintains precise exception handling in the case of discrete control flow and has been implemented in 90nm technology.

I. INTRODUCTION

Exception handling has long been a major concern in processor design, and it has become even more complicated with advanced microarchitectures. Most of the difficulty arises from handling precise exceptions without losing any performance, and this difficulty caused some computer designers to relax requirements in noncritical cases [6]. Some old-fashioned computers ignored the problem and took imprecise exceptions, but this approach is not acceptable in current processors because of virtual memory and the IEEE floating-point standard [5]. The main concern of handling exceptions is to handle unexpected changes in control flow and restore the saved state to resume execution correctly once the exception handling is complete. To enable system software to easily resume the user program, a CPU must take a precise exception. A precise exception means that exceptions must be taken in program order, so that the prior instructions of the offending instruction can only be completed while following instructions including the faulting instruction are flushed. Exception handling is also a very important issue for embedded systems and processor-in-memory (PIM) systems because the exception handling capability must be accomplished with minimal area overhead. One such PIM system is the Data-IntensiVe Architecture (DIVA) design [1]. DIVA exceptions are handled in a precise mode, so it can be naturally implemented in single-issue in-order processors, like DIVA PIM processors. Although the work described in this paper targets the DIVA exception handling scheme, the idea can be applied for the precise exception mode of other processing platforms.

The Data-IntensiVe Architecture (DIVA) PIM project

focuses on achieving speedup on applications that are severely performance limited by memory bandwidth in conventional systems. The DIVA system architecture (Fig. 1) exploits efficient use of PIM technology while providing a smooth migration path for software. The DIVA WideWord Processor speeds up multimedia applications by use of data parallelism. It treats a 256-bit WideWord operand as a packed array of objects of 8, 16, or 32 bits in size [7]. DIVA PIMs support standard memory accesses and have been recently fabricated with SRAM.

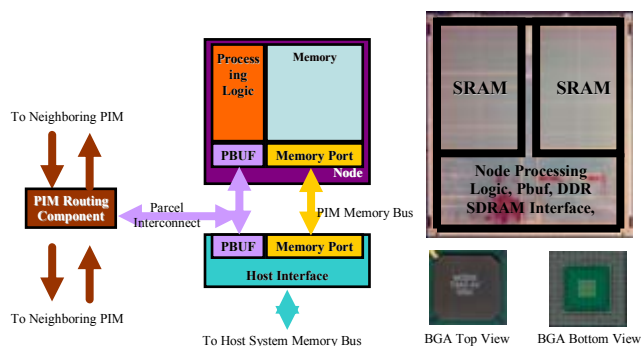


Figure 1. DIVA PIM chip organization

This paper presents exception handling mechanisms and precise state saving and recovery suitable for simple in-order processors like that of DIVA PIMs. Some prior research has been proposed for exception handling schemes for superscalar processors. One of the approaches is buffering the results of an operation until all the operations that were issued earlier are complete [5]. History files and future files [4] are also viable solutions to preserve precise exceptions. However, this work focuses on area-efficient precise exception handling targeted for area-constrained designs like embedded and PIM systems. Especially, this paper explains a more efficient mechanism to get the correct value of the return addresses when the exception is recognized. Even with this simplicity, the problem can still become difficult. A prior paper discussed a previous scheme for DIVA that placed too much burden on software to determine the return address [2]. Sometimes it is not possible to recover the processor state with a single program counter (PC) because of the discontinuous control flow with a branch instruction.

The presented idea in this paper gathers enough information with a minimum required hardware resource and carries out precise exception handling.

The remainder of this paper is organized as follows. Section II describes supported exception types in the DIVA system and general exception handling mechanisms. Section III explains precise exception handling in discontinuous control flow. Section IV gives the implementation results, and Section V concludes the paper.

II. EXCEPTION TYPES AND HANDLING MECHANISM

The exceptions in DIVA can be initiated in any pipeline stage but triggered only at memory stage in EDU (exception detection unit, Fig. 2). Multiple exceptions are handled based on a flexible priority assignment scheme [2]. Table I shows hardware-vectorized exceptions and corresponding vector addresses. According to the type of the detected exception, the exception handling starts from the specified vector address. Table II shows a list of the protected registers. All exceptions except reset and undefined instruction have an associated bit defined in the 32-bit exception source word, ESW, and corresponding bits in the exception-enable mask register, EMR, the exception set register, ESR, and the exception reset register, ERR. The Processor Status Word (PSW) is copied to the shadow PSW register (SSW) to record general state whenever exception handling commences. Also, the FADR and NADR registers record the addresses of a faulting instruction and next valid instruction, respectively, in the event of an exception. Table III shows exception sources which are recorded in ESW (exception source word) by classification.

TABLE I. HARDWARE-VECTORED EXCEPTIONS

Exceptions	Vector Address
RESET	0x08000000
Undefined Instruction (incl. BRK)	0x08000100
All other exceptions	0x08000200

Even though the term exception has been used to cover several types of events in this paper, each individual event has characteristics which can be classified into independent groups. If the cause of the exception was a particular instruction, the event is labeled synchronous. Page fault, unmapped instruction access, undefined instruction, floating-point arithmetic overflow and underflow fall into this category. Synchronous exceptions are carried to the memory stage until they are collected and aligned in ECU (ESW control unit, Fig. 2) based on the priority scheme. Asynchronous exceptions can occur any time and once such an exception is detected, all the instructions in the pipelines are flushed. External devices and hardware failures, such as interval timer expiration, I/O device request and hardware malfunctions, are asynchronous events. Other classifications can be seen in [5].

Once an exception or exceptions are detected, the corresponding bit(s) of ESW are set to 1 either by hardware

TABLE II. PROTECTED REGISTERS

Name	Number	Description
PSW	0	Processor Status Word
SSW	1	Shadow PSW, used in exception handling
Unused	2	Reserved
FADR	3	Faulting instruction address, used in exception handling
SCR0~SCR3	4-7	Supervisor-level scratch registers
ESW	8	Exception source word
EMR	9	Exception mask register
ESR	10	Exception set register
ERR	11	Exception reset register
MADR	12	Faulting memory address, used in exception handling
TIMER	13	Timer for programmable delay interrupts
RCL	14	Low-order bits of real-time clock
RCH	15	High-order bits of real-timer clock
NADR	16	Address of instruction after that of FADR, used in exception handling

in the event of a hardware exception, or by software setting corresponding bits in the ESR for software exceptions. Bits of the ESW are cleared to 0 by software setting corresponding bits in the ERR. The particular source of the exception is enabled by the mask register and globally enabled via an exception enable bit in PSW [2]. With the exception recognition, the mode is set to supervisor mode and the exception enable bit in PSW is automatically cleared. This means that nested exceptions are blocked at the beginning of exception handling. However, nested exceptions can be supported if the exception handler saves essential state, notably FADR, NADR and SSW, prior to re-enabling exceptions.

Fig. 2 shows the DIVA exception handling datapaths. All exception-related modules are placed at the memory stage because exceptions are recognized just before register files or memories are updated. PR (Protected registers) are placed at the instruction decode stage like GPR (general purpose registers), allowing consistent register transfers with the use of special instructions MFPR (move from protected register) and MTPR (move to protected register). The DIVA processor also includes some special instructions, ELO (encode leftmost one) and CLO (clear leftmost one) to aid in exception handling software. With these instructions, exception handlers can quickly process multiple exceptions, as captured by ESW, in a prioritized fashion. ELO can be used to quickly find the most significant bit set in ESW, corresponding to the highest-priority exception. The handler then processes that exception and performs a CLO to clear it before moving on to process the next exception by repeating the cycle.

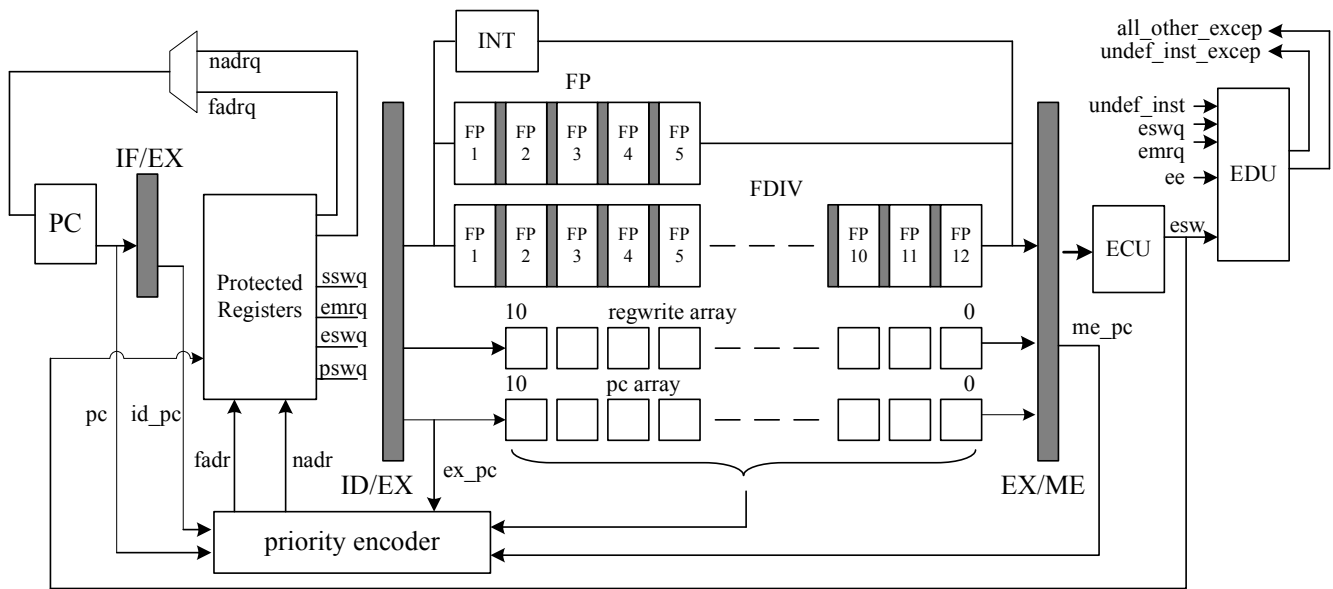


Figure 2. DIVA Exception Handling Datapaths

TABLE III. EXCEPTION SOURCE WORD

Initiator	Exception Name
Memory-access related exceptions	
HW	Unmapped Instruction Access, Invalid Instruction Access, Unmapped Data Access, Invalid Data Access
SW	Address Fault Fix-up
Execution related exceptions	
HW	Interval Timer, WW Not Available, FP Not Available, FP DZ (Divide by Zero), FP Invalid, FP Overflow or Underflow, FP Inexact, System Call, Privileged Instruction Violation, Scalar Integer ALU Exception, WW Integer ALU Exception,
SW	Context Swapper, Integer ALU Fix-up, WW ALU Fix-up, FP Fix-up, Lock Buzzer, Thread Rescheduler, Thread Dispatcher, Return to User Mode
Communication related exceptions	
HW	PBuf Interrupt
SW	Send Error Processing, PBuf doorbell Processing

III. PRECISE EXCEPTION HANDLING IN DISCONTINUOUS CONTROL FLOW

There is a special case where saving and recovering states can be tricky. If the cause of the exception was a delay slot of a taken branch, a target instruction following the delay slot can be lost without proper hardware support. To save the target instruction address, NADR is used to hold the address of the instruction that was issued after that pointed to

by the FADR. NADR keeps the flow of the execution even in the case that the flow is not sequential, and it is mentioned in the previous work [2]. We build on this concept to add robustness to the mechanism by which FADR and NADR record correct return addresses upon exception recognition in a discontinuous flow without software help.

In most cases upon exception detection, FADR records an instruction address of the memory stage and NADR records an instruction address of the execute stage. This works fine if both the memory and execute stages are occupied by valid instructions. However, due to the in-order nature of simplified pipelines, bubbles may occur in various stages because of pipeline stalls. So if FADR and NADR were hardwired to the memory and execute stages, it's possible that they would record obsolete addresses of instructions that had cleared the pipeline long ago if bubbles exist in either stage. In such a case, to guarantee precise exception handling, more burden falls upon the exception handling software to determine the return address used for returning from the exception after exception handling.

A more robust mechanism is to record valid instruction addresses in FADR and NADR whenever possible. For such precise exception handling, the first two valid instructions in the pipeline starting at the memory stage and searching backward toward the fetch stage need to be identified. A priority encoder which is shown in Fig. 2 searches and returns appropriate values to FADR and NADR. Fig. 3 shows a flowchart for what values get assigned to FADR and NADR. FADR gets the first valid instruction address and NADR gets the second if at least two instructions are valid in the pipeline. If only one instruction which is about to be fetched from an instruction cache at fetch stage is valid (meaning all other stages have bubbles), then FADR records the address of PC, and NADR records an internal value from the PC processing state machine according to the following

two cases. If the control flow was sequential without any branch instruction when the exception was taken, NADR would record ‘PC + 4’, which is the next instruction after FADR. However, if the last committed instruction was a branch, NADR would get a value that represents the resolved branch address from the PC processing state machine. This value may be ‘PC + 4’ if the branch is not taken or the branch target if the branch was taken.

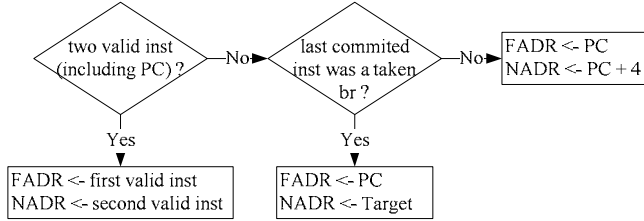


Figure 3. Flowchart for FADR and NADR

With these rules for setting FADR and NADR upon exception detection, a processor can easily resume normal processing upon returning from exception with minimal burden upon the exception handling software. This scheme guarantees that FADR and NADR always get correct return values for every possible case. It does not matter how many stages are bubbles or whether the control flow was not sequential. In the absence of such a hardware mechanism, exception handling software would incur at least 11 more cycles of execution time, based on a subroutine like that shown in Fig 4.

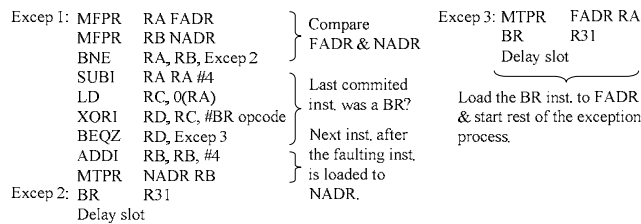


Figure 4. Return Address generation without hardware support

IV. IMPLEMENTATION

A corresponding exception handling unit has been specified in RTL-level behavioral VHDL code. The code was synthesized using Synopsys Design Compiler targeting the Artisan standard cell library for IBM 90 nm technology. Table IV summarizes the implementation results of the exception handling unit (priority encoder, ECU, EDU and protected registers) under three different timing constraints. The resulting timing and area are suited very well for the DIVA PIM as well as many other designs. The observed timing, area and power analysis are quite negligible compared to overall DIVA PIM measurements. The proposed exception handling mechanism of this paper can be adapted to any processors which are single-issue, in-order architectures, and it is especially useful to processors which require area- and power-efficient precise exception handling.

TABLE IV. SYNTHESIS RESULT

Delay (ns)	0.9	1.1	1.3
Area (sq μm)	22505.8	19013.1	17855.9
Dynamic Power (mW)	14.4405	8.8174	7.2798

The implemented exception handling units were tested as a part of the DIVA PIM design. Reset, Undefined Instruction and several Software-Vectored exceptions were tested through injected instructions. For floating-point arithmetic exceptions, the IEEE-754 standard [9] has been used. Proper exception handling actions were observed for each of the exception detections.

V. CONCLUSION

This paper described a precise exception handling mechanism for area-constrained systems such as embedded and PIM systems. It makes precise state recovering possible even in the case of discontinuity in the processor control flow. Some of the exception handling schemes in other architectures bound the performance by limiting the number of floating-point operations in the pipeline. However, the presented idea guarantees precise exception handling without any performance loss. The proposed mechanism achieves precise exception handling in a very small area, making it well suited to PIM systems like DIVA.

REFERENCES

- [1] Jeffrey Draper et al. “The Architecture of the DIVA Processing-In-Memory Chip.” In *Proceedings of the ACM International Conference on Supercomputing*, June 2002.
- [2] Sumit Mediratta et al. “A 0.18 μm CMOS Implementation of an Area Efficient Precise Exception Handling Unit for Processing-In-Memory Systems.” In *Proceedings of the 47th IEEE International Midwest Symposium on Circuits and Systems*, July 2004.
- [3] Wen-mei W. Hwu and Yale N. Patt. “Checkpoint Repair for Out-of-order Execution Machines.” In *Proceedings of the 14th annual International Symposium on Computer Architecture*, June 1987.
- [4] James E. Smith and Andrew R. Pleszkun. “Implementing Precise Interrupts in Pipelined Processors.” *IEEE Transaction on Computers*, Vol. 37, No. 5, May 1998.
- [5] J. Hennessy and D. Patterson. “*Computer Architecture: A Quantitative Approach*.” Morgan Kaufman, 3rd edition, 2002.
- [6] D. Patterson and J. Hennessy. “*Computer Organization & Design: The Hardware / Software Interface*” Morgan Kaufman, 3rd edition, 2007.
- [7] Jeffrey Draper, Jeff Sondeen and Chang Woo Kang. “Implementation of a 256-bit WideWord Processor for the Data-Intensive Architecture (DIVA) Processing-In-Memory (PIM) Chip.” In *Proceedings of the 28th European Solid-State Circuit Conference*, Sep. 2002.
- [8] Taek-jun Kwon, Jeff Sondeen and Jeff Draper. “Design Trade-Offs in Floating-Point Unit Implementation for Embedded and Processing-In-Memory Systems.” In *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 2005.
- [9] “IEEE Standard for Binary Floating-Point Arithmetic.”, ANSI/IEEE Standard 754, Aug. 1985.
- [10] Stephen W. Keckler et al. “Concurrent Event Handling through Multithreading.” *IEEE Transactions on Computers*, Vol. 48, No. 9, Sep 1999.