Integrated Device Technology, Inc.

INTERRUPT HANDLER FOR THE IDT79R3051 RISCONTROLLER™ FAMILY

APPLICATION NOTE AN-131

by Dean Smith

INTRODUCTION

The reader is encouraged to refer to Chapter 5 of the IDT79R3051 RISController Hardware User's Manual for a thorough description of IDT RISController exception handling. In addition, the MIPS Programmers Handbook illustrates two alternative methods for interrupt prioritizing. This application note illustrates a third much faster method specific to the IDT RISController Family, as detailed in the Appendix - 'R3051/2 Priority Based Nested Interrupt Handler'. The corresponding latency cycles for this example interrupt handler are quantified in Table 1.

R3051/2	Service Latency	Restart Latency
Priority 1	4	9
Priority 2	14	13
Priority 3	16	13
Priority 4	19	13
Priority 5	25	14
Priority 6	25	14

3158 tbl 01

Table 1. IDT79R3051/2 Interrupt Latency (in cycles)

The following assumptions apply to the latencies quantified in Table 1:

- The corresponding algorithm/code is detailed in the Appendix.
- Service Latency, Restart Latency are as defined in this application note.
- The code and stack are resident in the R3051 on-chip cache.
- The R3051 pipeline is in a 'run' state at the instant the interrupt is detected.
- A higher priority interrupt is not already in progress.
- Service is not interrupted by a higher priority interrupt.
- Service is not interrupted by any other type of exception.
- Only 1 register is needed by PRIORITY 1,2,3,4 service routines.
- Only 3 registers are needed by PRIORITY 5,6 service routines.

The interrupt handler detailed in the Appendix is specific to the R3051/2. However, much of the content detailed in this application note equally applies to the other RISController family members with only minor code modifications being required. Where applicable these differences in the family members are detailed.

R3051 EXCEPTION MODEL

External interrupts are just one class of R3051 exceptions. The R3051 implements a 'precise' exception model. By definition, precise exceptions imply that exact processor context and the cause of the exception are known. In addition, the current process does not advance state (ie. all subsequent instructions are aborted) until the corresponding interrupt is serviced.

The following automatically occurs when the R3051 detects an interrupt:

- The current process is halted.
- The Exception Program Counter is loaded with the return address for the current process.
- The Cause Register is loaded with exception cause information.
- The Status Register KUc bit is cleared (ie. enter 'kernel mode').
- The Status Register IEc bit is cleared (ie. disable subsequent interrupts).
- Execution is continued at the General Exception Vector.

These activities preserve the necessary processor context to implement a precise exception model. The R3051 processor makes no assumptions about an external interrupt cause or servicing techniques. For instance, R3051 registers are not automatically stacked upon detection of an interrupt since this often causes unnecessary service latency. Instead, the software designer is allowed to fine-tune response to the corresponding service requirements. This technique allows for extremely fast interrupt handling.

INTERRUPT SERVICE LATENCY

Interrupt Service Latency is defined as the cycle count from the assertion of an external interrupt to the beginning of the corresponding service routine. This latency includes three components;

1) pipeline latency to the General Exception Vector
2) exception type decode
3) preserving context.

PIPELINE LATENCY:

The R3051 pipeline must be in a 'run' state for an interrupt to be recognized. Thus, pipeline stalls caused by such events as cache misses and multiply/divide interlock cycles delay detection of an interrupt. Once an interrupt is detected, the address of the General Exception Vector will be the next instruction fetched.

The R3051 has two types of external interrupt pins; synchronous interrupts, and direct interrupts. The synchronous interrupts are internally synchronized and thus may be driven by an asynchronous source, with a corresponding pipeline latency to the General Exception Vector of two cycles. The direct interrupts are not internally synchronized by the processor, and thus must be externally synchronized. As a result, these interrupts have only a one cycle pipeline latency to the

General Exception Vector and are most useful for interrupting agents which operate off the R3051 SysClk output.

EXCEPTION TYPE DECODE:

The General Exception Vector is the start address for all types of R3051 exception handlers (except RESET and UTLB Miss exceptions) - interrupts being just one classification. Thus, the exact exception type must first be decoded before servicing can begin. This is typically accomplished by software interrogation of the R3051 Cause Register. The following example code details this procedure:

```
mfc0  k0,C0_CAUSE;       # k0 = CR(Cause Register)
sw    t1,t1_OFF*4(sp)    # use delay slot to stack gpr t1
and   k1,k0,EXC_MASK;    # isolate ExcCode field of CR
lw    v0,cause_table(k1); # fetch cause start address
and   k1,k0,IP_MASK;     # isolate IP field of Cause Register
j     v0;                # go to Exception handler start address
                         # (v0 = INT_EXTERN, if an interrupt)
sra   k1,k1,8;           # shift IP field 8 bits for word address

INT_EXTERN:
lw    v0,IP_table(k1);   # fetch service routine start address
sw    v1,v1_OFF*4(sp);   # use delay slot to stack gpr v1
j     v0;                # jump to corrresponding int(n) service
sw    t0,t0_OFF*4(sp);   # use delay slot to stack gpr t0
```

Even faster exception type decode can be achieved by using the R3051's BrCond(n) input pins. The MIPS ISA contains conditional branch instructions based upon the value of BrCond(n). These pins can be physically connected to interrupt pins for extremely fast decode. The following example code details this procedure:

```
bc0t  PRIORITY_1;        # int(0)?
sw    k0,EPC_OFF*4(sp);  # stack EPC (use branch delay slot).
bc1t  PRIORITY_2;        # int(1)?
sw    k1,SR_OFF*4(sp);   # stack SR (use branch delay slot)
bc2t  PRIORITY_3;        # int(2)?
sw    v0,v0_OFF*4(sp);   # stack v0 (use branch delay slot)
bc3t  PRIORITY_4;        # int(3)?
sw    t0,t0_OFF*4(sp);   # stack t0 (use branch delay slot)
```

The interrupt handler detailed in the Appendix is specific to the R3051/2 by making use of the four available BrCond(n) pins. Minor code modifications are required for the other RISController family members due to the different number of available BrCond(n) pins for each.

RISController Family Member	Number of BrCond(n) pins
R3051/2	four
R3071/81	three
R3041	two
	3158 tbl 02

PRESERVING CONTEXT:

Detection of an exception causes the R3051 to automatically disable subsequent interrupts. This makes it possible for immediate servicing of the interrupt without preserving Cause Register, Status Register, or Exception Program Counter context. Note that care must be taken by the software designer to ensure that execution of the interrupt handler and service routine do not generate any other type of exception. If 'nested' interrupts are allowed, then the Status Register and Exception Program Counter must be stacked. Otherwise the handling of the original interrupt can not be resumed. The IntMASK field of the Status Register can then be modified to re-enable higher priority interrupts. The following example code details this procedure:

```
bc0t  PRIORITY_1;        # int(0)?
sw    v0,v0_OFF*4(sp);   # use delay slot to stack gpr v0.

# PRIORITY 2,3,4,5,6 - must stack context for servicing of higher priority interrupts.
subu  sp,sp,exc_stack_sz; # Initialize Stack.
mfc0  k0,C0_EPC;         # k0 reserved for kernel processes
mfc0  k1,C0_SR;          # k1 reserved for kernel processes
sw    k0,EPC_OFF*4(sp);  # stack EPC.
mfc0  k0,C0_CAUSE;       # k0 = CR(Cause Register).
sw    k1,SR_OFF*4(sp);   # stack SR.
bc1t  PRIORITY_2;        # int(1)?

PRIORITY_2:
# Stack additional General Purpose Registers needed for servicing.
# re-enable int(0) - higher priority.
li    v0,x0000401;
mtc0  v0,C0_SR;
# PRIORITY 2 service here: . . .
```

Note that registers k0 and k1 are immediately available for interrupt handling. These registers need not be stacked since MIPS compiler and assembler conventions reserve k0 and k1 for kernel processes, and since subsequent interrupts are disabled during any interrupt handlerís use of these registers. However, the interrupt handler must stack any General Purpose Registers to be used for interrupt servicing. The number of registers required is of course interrupt service specific. The delay slots immediately following branch and load instructions are convenient locations to stack context without adversely affecting service latency.

Other features of the R3051 also help to minimize interrupt service latency. For instance, the on-chip cache is 'physically' indexed. This means that virtual-to-physical address transla-

tion is performed prior to cache addressing. As a result, cache flushing is not required on a context switch (ie. jump to interrupt service routine). Other processors implement virtually indexed caches thereby dramatically slowing context switch performance. Also of importance is the R3051 PID (Process ID) field associated with each entry of the TLB (Translation Lookaside Buffer). The 'Extended' memory management option uses an on-chip TLB as a hardware cache for software managed page tables. The PID is compared to the contents of each TLB entry at the time of address translation, thereby providing a mechanism for multiple processes to share the TLB even if identical virtual page numbers are encountered. As a result, TLB flushing is not required on a context switch.

INTERRUPT RESTART LATENCY

Interrupt Restart Latency is defined as the cycle count from the end of the interrupt service routine to the restart of the parent process. This latency includes two components;

1) context restore
2) pipeline refill.

Context Restore:

Any processor context stacked prior to interrupt servicing must be restored after servicing is complete. Then the stack pointer must be restored to its previous value. Finally, execution can then return to the parent process. The following example code details this procedure:

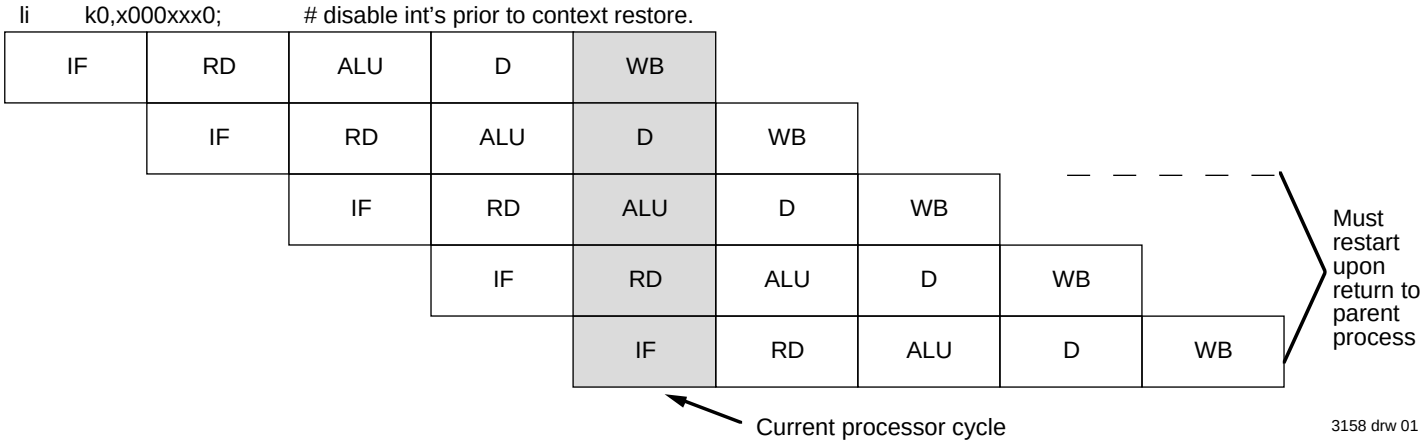
```
li      k0,x000xxx0;          # disable int's prior to context restore.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);      # restore gpr v0
lw      k0,EPC_OFF*4(sp);     # acquire parent process return address
addu    sp,sp,exc_stack_sz;   # restore stack.
mtc0    k1,CO_SR;             # restore SR(Status Register).
j       k0;                   # return to parent process
rfe;
```

Note that interrupts must be disabled prior to context restore. This is because k0 and k1 are not preserved prior to use by the interrupt handler. Otherwise, the context of these registers would be lost if another interrupt occurs during context restore for the current interrupt.

Pipeline Refill:

Figure 1 illustrates R3051 pipeline refill following an interrupt. Upon detection of an external interrupt, the three instructions less advanced than the ALU stage are aborted. These instructions must be restarted upon return to the parent process. This three cycle penalty must be considered when calculating the Interrupt Restart Latency.

Figure 1. IDT79R3051 instruction pipeline.



APPENDIX—R3051/2 PRIORITY-BASED NESTED INTERRUPT HANDLER

```
# This is an example R3051/2 priority based nested
  interrupt handler.
# Other RISController Family members  require minor code
  changes due to the different number
# of available BrCond(n) inputs
#       – prioritize up to four R3051/2 interrupts
#       – prioritize up to three R3081 interrupts
#       – prioritize up to two R3041 interrupts

# BrCond(n) is tied to corresponding int(n). This allows for
  fast interrupt decode:

# The following interrupt priority is assumed:
#       PRIORITY 1 = Int(5) = BrCond(0)
#       PRIORITY 2 = Int(4) = BrCond(1)
#       PRIORITY 3 = Int(3) = BrCond(2)
#       PRIORITY 4 = SIint(2) = BrCond(3)
#       PRIORITY 5 = SInt(1)
#       PRIORITY 6 = SInt(0)

# Exception causes execution to jump here:
  General Exception Vector.
```

```
.set    noreorder               # assembler directive–disable
                                pipeline scheduling.
bc0t    PRIORITY_1;             # PRIORITY 1?
subu    sp,sp,exc_stack_sz;     # use delay slot to Initialize
                                Stack.

# PRIORITY 2,3,4,5,6: Must stack CP0 context
  to allow for nested servicing.
sw      v0,v0_OFF*4(sp);        # stack gpr v0.
mfc0    k0,C0_EPC;              # k0 reserved for kernel processes
                                - no need to stack.
mfc0    k1,C0_SR;               # k1 reserved for kernel processes
                                - no need to stack.
sw      k0,EPC_OFF*4(sp);       # stack EPC.
mfc0    k0,C0_CAUSE;            # k0 = CR(Cause Register).
sw      k1,SR_OFF*4(sp);        # stack SR.
bc1t    PRIORITY_2;             # PRIORITY 2?
and     k1,k0,EXC_MASK;         # isolate ExcCode field of CR.
bc2t    PRIORITY_3;             # PRIORITY 3?
lw      v0,cause_table(k1);     # fetch exception cause start address.
bc3t    PRIORITY_4;             # PRIORITY 4?
and     k1,k0,IP_MASK;          # isolate IP field of Cause Register.

# PRIORITY 5,6: Evaluate Cause Register, jump to
Exception cause start address.
# (process already started by using Branch Delay Slots
above)
j       v0;                     # jump to Exception cause start
                                address.
sra     k1,k1,8;                # shift right 8 bits to create word
                                address.

# Exception cause start address = INT_EXTERN if an
interrupt.
INT_EXTERN:
lw      v0,IP_table(k1);        # fetch Interrupt routine start address
                                from IP_table.
sw      v1,v1_OFF*4(sp);        # use delay slot to stack gpr v1.
j       v0;                     # jump to PRIORITY_5 or 6, per IP
                                field of Cause Register.
sw      t0,t0_OFF*4(sp);        # use delay slot to stack gpr t0.

PRIORITY_1:
sw      v0,v0_OFF*4(sp);        # stack gpr v0.
# Stack any additional gpr's needed for PRIORITY 1
interrupt servicing.
# k0 & k1 are also available for PRIORITY 1 servicing.
# PRIORITY 1 service here.
        •
        •
        •
# Restore any gpr's used.
lw      v0,v0_OFF*4(sp);        # restore gpr v0.
# Restore Stack and return to parent process.
addu    sp,sp,exc_stack_sz;     # restore sp(Stack Pointer).
mfc0    k0,C0_EPC;
nop;
j       k0;                     # return from int svc.
rfe;

PRIORITY_2:
# Stack gpr's needed for PRIORITY 2 interrupt servicing.
# v0 already stacked.
# Re-enable PRIORITY 1 (higher priorty interrupt).
li      v0,x0008001;            # re-enable PRIORITY 1—Int(5). 2cycle
                                inst'n.
mtc0    v0,C0_SR;
# PRIORITY 2 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
process.
li      k0,0x00000000;          # disable interrupts prior to context
                                restore. 1 cycle inst'n.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);        # restore gpr v0.
nop
lw      k0,EPC_OFF*4(sp);
addu    sp,sp,exc_stack_sz;     # restore sp(Stack Pointer).
mtc0    k1,CO_SR;               # restore SR(Status Register).
j       k0;                     # return from int svc.
rfe;

PRIORITY_3:
# Stack gpr's needed for PRIORITY 3 interrupt servicing.
# v0 already stacked.
# Re-enable PRIORITY 1,2 (higher priorty interrupts).
li      v0,x000C001;            # re-enable PRIORITY 1,2 - Int(5,4).
                                2cycle inst'n.
mtc0    v0,C0_SR;

# PRIORITY 3 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
process.
li      k0,0x00000000;          # disable interrupts prior to context
                                restore. 1 cycle inst'n.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
```

```
 lw      v0,v0_OFF*4(sp);      # restore gpr v0.
 nop
 lw      k0,EPC_OFF*4(sp);
 addu    sp,sp,exc_stack_sz;   # restore sp(Stack Pointer).
 mtc0    k1,CO_SR;             # restore SR(Status Register).
 j       k0;                   # return from int svc.
 rfe;
PRIORITY_4:
# Stack gpr's needed for PRIORITY 4 interrupt servicing.
# v0 already stacked.
# Re-enable PRIORITY 1,2,3 (higher priorty interrupts).
 li      v0,x000E001;  # re-enable PRIORITY 1,2,3 - Int(5,4,3).
                       2cycle inst'n.
 mtc0    v0,C0_SR;
# PRIORITY 4 service here.
         •
         •
         •
# Restore SR, gpr's used, Stack, and return to parent
process.
 li      k0,0x00000000;        # disable interrupts prior to context
                               restore. 1 cycle inst'n.
 mtc0    k0,C0_SR;
 lw      k1,SR_OFF*4(sp);
 lw      v0,v0_OFF*4(sp);      # restore gpr v0.
 nop
 lw      k0,EPC_OFF*4(sp);
 addu    sp,sp,exc_stack_sz;   # restore sp(Stack Pointer).
 mtc0    k1,CO_SR;             # restore SR(Status Register).
 j       k0;                   # return from int svc.
 rfe;

PRIORITY_5:
# Stack gpr's needed for PRIORITY 5 interrupt servicing.
# v0,v1,t0 already stacked.
# Re-enable PRIORITY 1,2,3,4 (higher priorty interrupts).
 li      v0,x000F001;          # re-enable PRIORITY 1,2,3,4
                               - Int(5,4,3,2). 2cycle inst'n.
 mtc0    v0,C0_SR;
# PRIORITY 5 service here.
         •
         •
         •
# Restore SR, gpr's used, Stack, and return to parent
process.
 li      k0,0x00000000;        # disable interrupts—1 cycle inst'n.
 mtc0    k0,C0_SR;
 lw      k1,SR_OFF*4(sp);
 lw      v0,v0_OFF*4(sp);      # restore gpr v0.
 lw      v1,v1_OFF*4(sp);      # restore gpr v1.
 lw      t0,t0_OFF*4(sp);      # restore gpr t0.
 lw      k0,EPC_OFF*4(sp);
 addu    sp,sp,exc_stack_sz;   # restore sp(Stack Pointer).
 mtc0    k1,CO_SR;             # restore SR(Status Register).
 j       k0;                   # return from int svc.
 rfe;

PRIORITY_6:
# Stack gpr's needed for PRIORITY 6 interrupt servicing.
# v0,v1,t0 already stacked.
# Re-enable PRIORITY 1,2,3,4,5 (higher priorty
interrupts).
 li      v0,x0007F801;         # re-enable PRIORITY 1,2,3,4,5
                               - Int(5,4,3,2,1). 2cycle inst'n.
 mtc0    v0,C0_SR;
# PRIORITY 6 service here.
         •
         •
         •
# Restore SR, gpr's used, Stack, and return to parent
process.
 li      k0,0x00000000;        # disable interrupts - 1 cycle inst'n.
 mtc0    k0,C0_SR;
 lw      k1,SR_OFF*4(sp);
 lw      v0,v0_OFF*4(sp);      # restore gpr v0.
 lw      v1,v1_OFF*4(sp);      # restore gpr v1.
 lw      t0,t0_OFF*4(sp);      # restore gpr t0.
 lw      k0,EPC_OFF*4(sp);
 addu    sp,sp,exc_stack_sz;   # restore sp(Stack Pointer).
 mtc0    k1,CO_SR;             # restore SR(Status Register).
 j       k0;                   # return from int svc.
 rfe;

.set reorder                   # assembler directive - enable pipeline
                               scheduling.
```