Technical Bulletin

Using Slow Peripherals with 29K Family Processors

EPD Systems Engineering August 1, 1994

Purpose

Usually, when a peripheral requests an interrupt from a 29K Family processor, the interrupt handling software clears the interrupt source and the peripheral de-asserts the interrupt line. However, some peripheral devices are so slow that the 29K Family processor is able to begin processing the interrupt again (as if it were a new interrupt) before the peripheral removes the original interrupt.

This technical bulletin describes how to work around the disparity in speeds between a slow peripheral and a 29K Family processor.

Affected Parts

The information in this bulletin affects the following parts:

Device	Revision
All 29K Family microprocessors and microcontrollers	All

Slow Peripherals

If an external peripheral device requests an interrupt from a 29K Family processor, the 29K Family-based part clears the interrupt by reading or writing the appropriate external bits from the interrupt handler. Then the interrupt handler performs an interrupt return sequence (IRET).

As part of the interrupt return, the processor looks for active interrupt sources. In the case of a peripheral device that is much slower than the 29K Family processor, the peripheral device still has not processed the interrupt-clearing LOAD or STORE, so the interrupt is still asserted. Consequently, the processor sees an active interrupt and begins to process it (for the second time).

After the processor determines that an interrupt is pending, and before the processor determines which interrupt is active, the slow peripheral device finally clears the interrupt condition. Now the processor is not able to determine which peripheral called for an interrupt, so it defaults to trap 0 (the invalid op code trap).

This situation is aggravated because the 29K Family pipeline allows load and store bypassing until there is a direct dependency situation. Therefore, simply moving the interrupt-clearing LOAD or STORE earlier in the handler may not help, as the LOAD or STORE is not guaranteed to actually execute any earlier.

Work-Around

The way to work around this situation is to force the LOAD/STORE to complete and guarantee the peripheral has de-asserted the interrupt before the processor executes the IRET. Adding a serializing instruction between a LOAD/STORE and an IRET forces a LOAD/STORE to complete before the IRET, allowing more time for the peripheral to de-assert the interrupt line before the processor executes the interrupt return sequence.

If the peripheral and processor are somewhat closely matched or if the interrupt handler is long, using a serializing instruction alone will often resolve the situation. However, if the speed disparity between the peripheral and the processor is significant, it may be necessary to guarantee that the peripheral has the time it needs to de-assert the interrupt line. This can be accomplished by using a busy loop or delay loop.

Using a busy loop is guaranteed to work even if the software is ported to a different-speed platform. However, for some applications the addition of a busy loop will degrade performance. Using a delay loop does not involve the bus overhead of the busy loop, but a delay loop needs to be tuned for each system configuration.

Serializing Instruction

Using a serializing instruction is generally an adequate work-around when the mismatch between peripheral and processor is on the order of several cycles.

A serializing instruction requires that all other instructions in the pipe complete before it executes. As explained earlier, adding a serializing instruction between a LOAD/STORE and an IRET forces a LOAD before the IRET, maximizing the amount of time the peripheral has to clear the interrupt before the processor executes the interrupt return sequence.

In the 29K Family of microprocessors and microcontrollers, the serializing instructions are as follows. (Note that for 29K Family microcontrollers, any access to an internal peripheral control or status register is also a serializing instruction.)

- Move to Special Register (MTSR)
- Move to Special Register Immediate (MTSRIM)
- Move to Translation Look-aside Buffer (MTTLB)
- Interrupt Return (IRET)
- Interrupt Return and Invalidate (IRETINV)
- Enter Halt Mode (HALT)

When adding a serializing instruction, most users choose to serialize using the "mtsr gr*xx*, lru" instruction. If the application does not use MMU, or all pages are mapped by the TLBs (page misses never occur), then serialization can be accomplished with this single instruction. Otherwise, the special register value must be saved with an MFSR instruction before serialization occurs with an MTSR instruction.

For example, an interrupt is cleared in an Am85C30 peripheral device by writing a zero to the SCC control register for that port. Adding the serializing instruction would look like the following:

```
msg_scc8530_tx_intr:
; reset tx interrupt.
    const  it1, SCC8530_CHA_CONTROL
    consth it1, SCC8530_CHA_CONTROL
    const  it0, 0
    store  0, 1, it0, it1
    mfsr it0, lru
    mtsr lru, it0

.    {rest of handler}
.
iret
```

Adding the serializing instruction simply ensures that the interrupt-clearing LOAD or STORE actually appears on the bus as soon as possible, maximizing the amount of time the peripheral has to clear the interrupt before the processor executes the IRET. The following sections explain how to guarantee the peripheral has adequate time to clear the interrupt before the processor executes the IRET.

Busy Loop

The work-around described above can be enhanced by following the interrupt-clearing LOAD/STORE with a STORE/LOAD loop that continues until the peripheral has had time to process the clearing of the interrupt.

Based on the same example from the previous section and using the busy loop approach, an interrupt handler that resets the transmit interrupt would look like the following

```
msg_scc8530_tx_intr:
; reset tx interrupt.
    const  it1, SCC8530_CHA_CONTROL
    consth it1, SCC8530_CHA_CONTROL
    const  it0, 0
    store  0, 1, it0, it1
bsy_loop:
    load 0, 1, it1, it0
    cpeq it0, it0, 0
    jmpf it0, bsy_loop
    nop
.
.    {rest of handler}
.
iret
```

Although this is a guaranteed work-around, remember that the busy loop can degrade system performance for some applications. For this reason, it is desirable to overlap as much of the peripheral's response time as possible with the interrupt handling. So, the busy loop in the above example could be moved after "{rest of handler}".

Delay Loop

An alternate way to enhance the work-around is by adding a delay loop. This requires the developer to know the exact hardware parameters of the system as each delay loop is tuned to the specific parameters of the design under development. The positive tradeoff for this loss of flexibility is that the delay loop involves less bus traffic than the busy-loop, which may be very important in the system. In addition, this approach minimizes the overhead to the minimum required.

The delay in the delay loop is determined by:

DELAY = 2 cycles + access_time + periph_time

The first two cycles are required to present the interrupt-clearing LOAD/STORE to the bus. The *access_time* value is the time in processor cycles that

is required for the peripheral to give a *RDY response to the LOAD or STORE. The *periph_time* value is the number of processor cycles required for the peripheral to process the LOAD/STORE and remove the interrupt request.

So, using the same example from the previous two sections, consider a system with a 3-processor-cycle access time to the peripheral, and a peripheral that requires two of its cycles (six processor cycles) to process and clear the interrupt. In this case, the delay loop implementation is the following:

```
msg_scc8530_tx_intr:
; reset tx interrupt.
    const  it1, SCC8530_CHA_CONTROL
    consth it1, SCC8530_CHA_CONTROL
    const  it0, 0
    store  0, 1, it0, it1
    mfsr it0, lru
    mtsr lru, it0
    const it0, DELAY
    jmpf it0, .
    nop
.
.    {rest of handler}
.
iret
```

The value of DELAY is calculated as:

DELAY = int(([2+3+6]-3) / 2;

where 2 is the access time, the first 3 is the *RDY response time, 6 is the process interrupt clear time, the second 3 is the number of cycles to set up the delay loop, and 2 is the number of instructions in the loop.

For More Information

For more information, see the appropriate processor user's manual.

If You Need Assistance

Product support for the 29K Family processors is available from our Embedded Processor Division (EPD) Technical Support Hotlines located in the U.S. and in the U.K.

Assistance is available in the U.S. from 9:00 A.M. to 6:00 P.M. central time, Monday through Friday (except major holidays). In Europe assistance is available during U.K. business hours. Contact us at one of the following numbers:

To reach the U.S. hotline

From	Call
U.S.	1-800-2929-AMD
U.K.	0-800-89-1455
Japan	0031-11-1163
Any other location	+1-512-602-4118†

†Toll applies.

To reach the U.K. hotline

From	Call
U.K.	(0)256-811101
France	0590-8621
Germany	0130-813875
Italy	1678-77224
Any other location	+44-(0)256-811101†

†Toll applies.