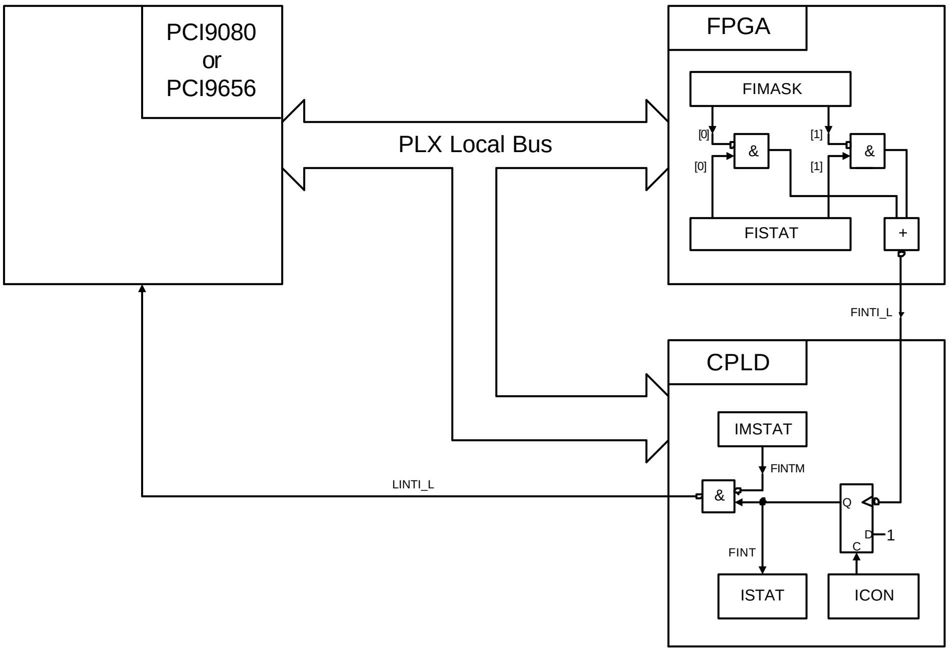
ADM-XRC Application Note – FPGA Interrupt Handling

This application note describes an implementation of a simple FPGA interrupt generation and handling mechanism for the ADM-XRC series of cards, with example application-level C code for the host. This mechanism permits an application running in the FPGA to generate interrupts which may be captured by an application running on the host.

The ADM-XRC series of FPGA PMC cards provide a reconfigurable computing resource coupled to the host by a fast PCI interface. In this document, “ADM-XRC” is used to refer to any ADM-XRC series card, such as the ADM-XRC, ADM-XRC-P, ADM-XRCII-L and ADM-XRCII.

FPGA Interrupt Logic on the ADM-XRC

The diagram below shows in simplified form, the interrupt logic in the ADM-XRC, plus the interrupt registers in a typical FPGA application that has two interrupt sources in the FPGA:



The FPGA contains the user-programmable logic, and has two registers, each 2 bits wide: FISTAT (FPGA interrupt status) and FIMASK (FPGA interrupt mask). If a bit in FIMASK is 1, the corresponding bit of FISTAT is masked and prevented from generating an interrupt on the FPGA’s FINTI_L pin. When a bit in FISTAT reads as 1, it can be cleared to 0 by writing 1 to the same bit of FISTAT.

The CPLD contains the IMSTAT register (interrupt mask), ISTAT register (interrupt status) and the ICON (interrupt control register). When a bit in ISTAT is 1, LINTI_L is asserted if the corresponding bit in IMSTAT is 0 (unmasked). Writing 1 to a bit in the ICON register clears the corresponding bit in ISTAT to 0.

Note the presence of the negative edge triggered flip-flop in the CPLD, meaning that only a high-to-low transition on FINTI_L can cause the FINT signal in the CPLD to be set to 1. In other words, the interrupt signal from the FPGA to the CPLD is negative edge sensitive. The reason for this flip-flop will become apparent later. The LINTI_L pin on the PCI9080/PCI9656 is active low and level sensitive, whereas the FINTI_L pin on the CPLD is negative edge sensitive.

The ADM-XRC Device Driver Interrupt Handler

Normally, a device driver will be responsible for handling interrupts from the ADM-XRC. The following code shows the algorithm used by the ADM-XRC device drivers provided by Alpha Data:

```
void admxrc_handle_interrupt(...)
{
      DWORD   intcsr;
      BOOLEAN fpga_interrupt = FALSE;

      /* Sample the PCI9080/PCI9656’s INTCSR register */
      intcsr = read_plx(PLX_INTCSR);

      /* Check other interrupts, eg. PCI9080/PCI9656 DMA engines */
      ...

      /* Check for local bus interrupt */
      if (intcsr & (1 << 15))
      {
            /* local bus interrupt is active (LINTI_L is asserted) */
            BYTE istat;

            /* read the CPLD interrupt status */
            istat = read_cpld(CPLD_ISTAT);
            if (istat & (1 << 0))
            {
                  /*
                  ** CPLD FPGA interrupt is active
                  ** Clear the CPLD FPGA interrupt
                  */
                  write_cpld(CPLD_ICON, 1 << 0);
                  fpga_interrupt = TRUE;
            }
            else
            {
                  /* We do not expect ever to get here! */
            }
      }

      /* Other interrupt processing? */
      ...

      /* See text below */
      if (fpga_interrupt)
            admxrc_handle_fpga_interrupt();
}
```

When the handler reaches the point where it calls `admxrc_handle_fpga_interrupt()`, the CPLD is no longer asserting LINTI_L, unless the FPGA has caused another high-to-low transition on FINTI_L.

The device driver's interrupt handler does **not** attempt to cause FINTI_L to be deasserted, as the driver cannot in general know how to deassert FINTI_L. However, this is will not immediately result in another host interrupt (unless FINTI_L transitions high to low again), as FINTI_L is an edge sensitive signal rather than a level sensitive signal.

In order for the user's application running on the host to be informed that an FPGA interrupt has occurred, the driver's `admxrc_handle_fpga_interrupt()` routine must perform some appropriate action to notify the application. Depending on the operating system, this could be:

- In VxWorks, signalling a user-created semaphore that the application has registered with the driver
- In Windows, signalling an user-created event that the application has registered with the driver by calling `ADMXRC2_RegisterInterruptEvent()`

Thus, the reason for making the FINTI_L signal level sensitive is to allow the application-specific handling of an FPGA interrupt to be deferred to application level.

The Application Level FPGA Interrupt Handler

In a Win32 application, the application level FPGA interrupt handler may simply be a thread that repeatedly waits on an event it has registered with the driver via `ADMXRC2_RegisterInterruptEvent()`. The following code fragment illustrates this (error checking omitted for clarity):

```
HANDLE            fpga_event;

/* Other code/declarations */
...

void handle_fpga_interrupt(ADMXRC2_HANDLE handle)
{
      DWORD fistat;
      DWORD fimask;

      fistat = fpga_read(handle, FPGA_FISTAT); /* ---(1) */
      fimask = fpga_read(handle, FPGA_FIMASK);

      /*
      ** Don't bother processing interrupts that are masked –
      ** this is optional and depends on the needs of the application
      */
      fistat &= ~fimask;

      /*
      ** Dismiss only the interrupts that we are going to process
      ** this time around
      */
      fpga_write(handle, FPGA_FISTAT, fistat);
      if (fistat & (1 << 0))
      {
            /* Code to process FPGA interrupt 0 */
            ...
      }

      if (fistat & (1 << 1))
      {
            /* Code to process FPGA interrupt 1 */
            ...
      }

      /* Rearm the FINTI_L signal ---(2) */
      fpga_write(handle, FPGA_FIMASK, 0xffffffffU);
      fpga_write(handle, FPGA_FIMASK, fimask);
}

DWORD WINAPI interrupt_thread(PVOID arg)
{
      ADMXRC2_STATUS status;
      ADMXRC2_HANDLE handle = (ADMXRC2_HANDLE) arg;
      DWORD          result;

      fpga_event = CreateEvent(NULL, FALSE, FALSE, NULL);

      status = ADMXRC2_RegisterInterruptEvent(handle, fpga_event);

      while (1)
      {
            result = WaitForSingleObject(fpga_event, INFINITE);
            handle_fpga_interrupt(handle);
      }
}
```

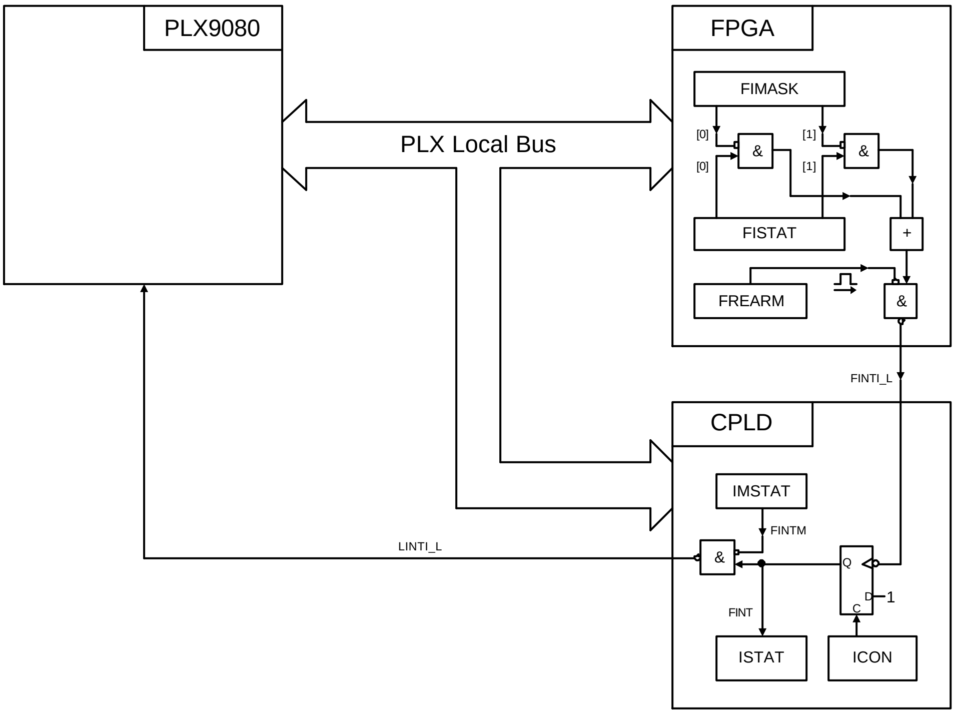
Why is it necessary to ‘rearm’ the FPGA interrupt as in (2), by first setting all bits of the FPGA’s FIMASK register to 1, and then restoring them to their previous state?

The reason is that in the time between executing the statement at (1) and the statement at (2), an interrupt **may** have occurred that was not seen when sampling the FPGA’s FISTAT register. **Thus, merely dismissing the interrupts in the FPGA that were pending at the time FISTAT was sampled is not guaranteed to cause FINTI_L to be deasserted**. By rearming FINTI_L, any FPGA interrupts that are still pending will cause a new local bus interrupt on LINTI_L and thus prevent a situation arising where the application waits for an interrupt that has actually occurred but is never seen.

Refinements To The Method

It may not be desirable to write to the FIMASK register in order to rearm FINTI_L, for a number of reasons. For example, in a multiprocessor system, another section of code running on a different CPU might alter the FIMASK register, and this would require the interrupt handler to coordinate its activity with other sections of code that potentially modify FIMASK.

Adding a FREARM register (FINTI_L rearm) to the FPGA removes the need for this coordination, simplifying driver design and increasing performance slightly. The FREARM register simply outputs a pulse of a single clock cycle when it is written to. The following diagram shows this refinement:



The application-level FPGA interrupt handler can now be modified as follows:

- Use a shadow for the FIMASK register. In other words, the application maintains a copy of the most recent value programmed into the FIMASK register so that a read from the FPGA FIMASK register can be avoided.
- The two writes to the FIMASK register in order to rearm FINTI_L become a single write to the FREARM register.

The `handle_fpga_interrupt()` function becomes:

```
DWORD fimask_shadow; /* Shadows the FIMASK register */

void handle_fpga_interrupt(ADMXRC2_HANDLE handle)
{
      DWORD fistat;

      fistat = fpga_read(handle, FPGA_FISTAT);

      /*
      ** Don’t bother processing interrupts that are masked –
      ** this is optional and depends on the needs of the application
      */
      fistat &= ~fimask_shadow;
```

```
        /*
        ** Dismiss only the interrupts that we are going to process
        ** this time around
        */
        fpga_write(handle, FPGA_FISTAT, fistat);

        if (fistat & (1 << 0))
        {
                /* Code to process FPGA interrupt 0 */
                ...
        }

        if (fistat & (1 << 1))
        {
                /* Code to process FPGA interrupt 1 */
                ...
        }

        /* Rearm the FINTI_L signal */
        fpga_write(handle, FPGA_FREARM, 0U); /* Value is irrelevant */
}
```

Conclusion

This application note has described, with reference to Alpha Data's ADM-XRC SDK, a mechanism for delivering interrupts from the FPGA to the application. The actual processing for the FPGA interrupts is deferred to application level instead of driver level, and the device driver need not know how to process interrupts from the FPGA. As long as the driver allows user-created events or semaphores to be registered and signaled when FPGA interrupts occur, this mechanism is applicable to any platform.

For further information, contact:

Alpha Data Parallel Systems Ltd.
58 Timber Bush
Edinburgh, EH6 6QH
UK

Te1 +44 (0) 131 555 0303
Fax +44 (0) 131 555 0728

Email : support@alphadata.co.uk
Web : www.alphadata.co.uk