

**LNIV**  
Faculty of Electrical Engineering  
Tržaška 25, Ljubljana, Slovenija  
<http://lniv.fe.uni-lj.si>

**MODIFICATIONS OF QDR MODULE**  
**Documentation**

**Matjaž Finc**  
[matjaz.finc@fe.uni-lj.si](mailto:matjaz.finc@fe.uni-lj.si)

Ver. 1.0a

Updated:

9.1.2004

## MODIFICATION 1.4 – CONTINUOUS TRIGGERED MODE

Memory Map @ 0x0010 (Control Register):

VME Offset	Bit	Description
0x0010	0	<b>VME Access Enable</b>
	[2..1]	<b>Gated (Synchronous) Acquisition</b> 00 → Disable Acquisition 01 → Enable Continuous Acquisition 10 → <u>Enable Continuous Triggered Acquisition</u> 11 → Enable Gated (Synchronous) Acquisition
	[4..3]	<b>Acquisition Mode Select</b>
	5	<b>Link Port</b>
	6	<b>FIFO Access Select</b> (1.6 – PAE/PAF) 0 → FIFO Stack Access 1 → PAE/PAF Offset Registers Access
	[9..8]	<b>DDC Data Format</b>

### Operation:

In the continuous triggered mode, the QDR remains passive until it receives an external gate trigger. After the triggering, the data is continuously delivered.

### Usage:

The data acquisition is enabled/disabled by bits [2..1] of the Control register (@ 0x0010). All three modes of acquisition are disabled by value '00' of bits [2..1]. Continuous triggered mode is enabled by value '10' of bits [2..1]. For direct acquisition, bits [4..3] have to be set to '01' (bypassing FIFOs).

### Architecture:

In order to control the single triggering, a D flip-flop `ext_gate_ff` is added, which detects the edge of the `gate_in`. The triggering of the flip-flop is controlled only by bits [2..1] of the control register. The basic functional block diagram of the architecture modification of the external gate triggering control is presented in Figure 1.

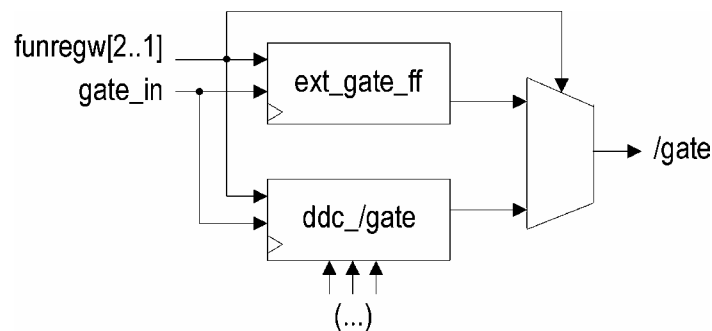


Figure 1 - Functional block diagram of the modified architecture

## MODIFICATION 1.6 – Programmable PAE/PAF Offset Registers

In order to access PAE/PAF Offset Registers, flags 6 and 7 @ 0x0010 Control Register are used. FIFO PAE/PAF register data read/write access is performed with the use of registers @ offset 0x0000, 0x0004, 0x0008 (see the QDR Technical Reference).

PAE/PAF register programming is done by enabling the `/ld` FIFO pins and enabling the `/wen` FIFO pins (enable FIFO write clock). On every positive edge of the FIFO write clock, the FIFO data input access changes between PAE and PAF (see the excerpt from the datasheet in *Appendix A*). Since programming the PAE/PAF FIFO registers is actually the same as a read/write operation to the FIFO stack, the most convenient way of accessing registers is to execute the appropriate read/write sequence in SW:

1. read once (PAE)
2. read once (PAF)
3. write once (PAE)
4. write once (PAF)

This sequence assures the registers to maintain appropriate values. For read sequence, use only steps 1 and 2. For write sequence, use all four steps in sequence, for easiest control over currently accessed registers. Otherwise, the type of the targeted register has to be carefully monitored, not to overwrite the wrong one.

Here is an example of a write operation (PAE = 0xAAAA) from the user perspective:

1. select FIFO PAE/PAF register access (flag 6 of Control Register = 1)
2. read data from FIFO (original PAE value)
3. read data from FIFO (original PAF value)
4. write data to FIFO (PAE = 0xAAAA)
5. write data to FIFO (original, previously read PAF value)
6. deselect FIFO PAE/PAF register access (flag 6 of Control Register = 0)

Strict SW flow is required, but it is transparent and not timing critical, since it is a part of system setup and is not executed in real-time critical processing tasks.

## MODIFICATION 1.6 – IRQ ROUTINE

### Memory Map:

VME Offset	Bit	Description
0x0100	0	<b>Fn Register</b> IRQ 1 Status Flag (highest priority)
	1	IRQ 2 Status Flag
	2	IRQ 3 Status Flag
	3	IRQ 4 Status Flag (lowest priority)
0x0104	[9..0]	<b>X1 Register</b> IRQ Flags for <b>FIFO Status Register</b>
0x0108	[9..0]: X	<b>M1 Register</b> 0 → Non-masked (Ignored) Flag 1 → Masked Flag (Enabled)
0x010c	[31..0]	<b>V1 Register</b> Programmable IRQ Vector
0x0110		/
0x0114	[9..0]	<b>X2 Register</b>
0x0118	[9..0]	<b>M2 Register</b>
0x011c	[31..0]	<b>V2 Register</b>
0x0120		/
0x0124	[9..0]	<b>X3 Register</b>
0x0128	[9..0]	<b>M3 Register</b>
0x012c	[31..0]	<b>V3 Register</b>
0x0130		/
0x0134	[9..0]	<b>X4 Register</b>
0x0138	[9..0]	<b>M4 Register</b>
0x013c	[31..0]	<b>V4 Register</b>
0x0140	0	<b>IE Register</b> IRQ 1 Enable
	1	IRQ 2 Enable
	2	IRQ 3 Enable
	3	IRQ 4 Enable

**X<sub>m</sub>** – value register for m-th IRQ

**M<sub>m</sub>** – mask register for m-th IRQ

**V<sub>m</sub>** – programmable interrupt vector register for m-th IRQ

### Operation:

When the FIFO status register equals X<sub>m</sub> register (masked with M<sub>m</sub>) the IRQ<sub>m</sub> bit of the Fn register is set to '1' and then, if the IRQ<sub>m</sub> bit of the IE Register is '1', the m-th IRQ is triggered and /lirq is asserted low. When /lden is received (see *Appendix C*), the m-th interrupt vector V<sub>m</sub> of the first triggered IRQ is put onto the data bus. If more IRQs are triggered simultaneously, the IRQ with the lowest IRQ number has the highest priority. Based on ROAK procedure (see *Appendix B, C*), the /lirq interrupt signal is released upon interrupt acknowledge (/lden activated).

By writing '1' to the IRQm bit of the Fn register, the IRQm is cleared and next IRQ triggering cycle is enabled. If not all the active IRQs are cleared with the aforementioned write operation or new IRQs are triggered before the operation, /lirq signal is triggered and the remaining interrupt vectors are treated based on IRQ priority. In this manner, all the IRQ events are taken into account and the /lirq signal is triggered only after the completion of the current interrupt service routine.

### Usage:

After an IRQ is triggered, the interrupt vector has to be read by SW. The SW interrupt service routine is assigned to each IRQ with the dedicated interrupt STATUS/ID vector (Vm registers). Interrupt vectors should be set during initialization. For each interrupt, the appropriate interrupt service routine is called, based on the interrupt STATUS/ID vector present on the data bus during the IRQ. At the end of the interrupt routine processing, the serviced IRQ flag has to be cleared with a '1' write operation to the appropriate bit in the Fn register. This enables the triggering of the next interrupts. If there are more IRQs or waiting in the queue (Fn register has multiple '1' bits – i.e. Fn = "1011"), after the Fn flag reset, the next IRQ cycle is triggered immediately. The remaining IRQs are serviced by priority, until all are serviced. If the interrupt queue is not empty, all the new incoming IRQs are also triggered by priority, until all the IRQs are serviced.

### Important notes:

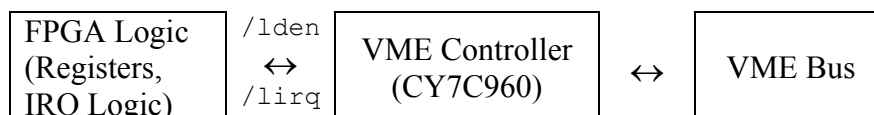
1. In order for the IRQ logic inside the FPGA to function correctly, after every reset of the QDR module and during the initialization of the SW, the IRQ logic must be re-initialized with write operation "1111" to Fn register.
2. For IRQ vectors (Vm), only even values in the range 0 – 255 can be used.

### IRQ Level

VME IRQ level can be set for each board individually only with use of a DIP switch (SW3) and cannot be set in the FPGA. The Cypress VME controller is configured to use the **IRQ level 4**. Therefore, switch SW3 must be set to level 4.

### Architecture:

#### Signal Flow – IRQ Handling (Handshaking)



/lirq – single IRQ line for highest priority IRQ to be driven onto VME Bus

/lden – enable interrupt vector onto the VME Bus

(pin must be connected physically to the VME Controller: TP7 ↔ FLEX\_162 or 163)

The functional overview of the architecture is illustrated in Figure 2. The irq\_vec\_hold detects the triggering of the first IRQ and makes a snapshot of Fn into the irq\_vec\_nr. The appropriate Vm interrupt vector is decoded by priority, and put onto the data bus when

/lden is activated low. Signal /lirq\_off enables/disables the active /lirq\_out onto the interrupt output line towards the VME controller.

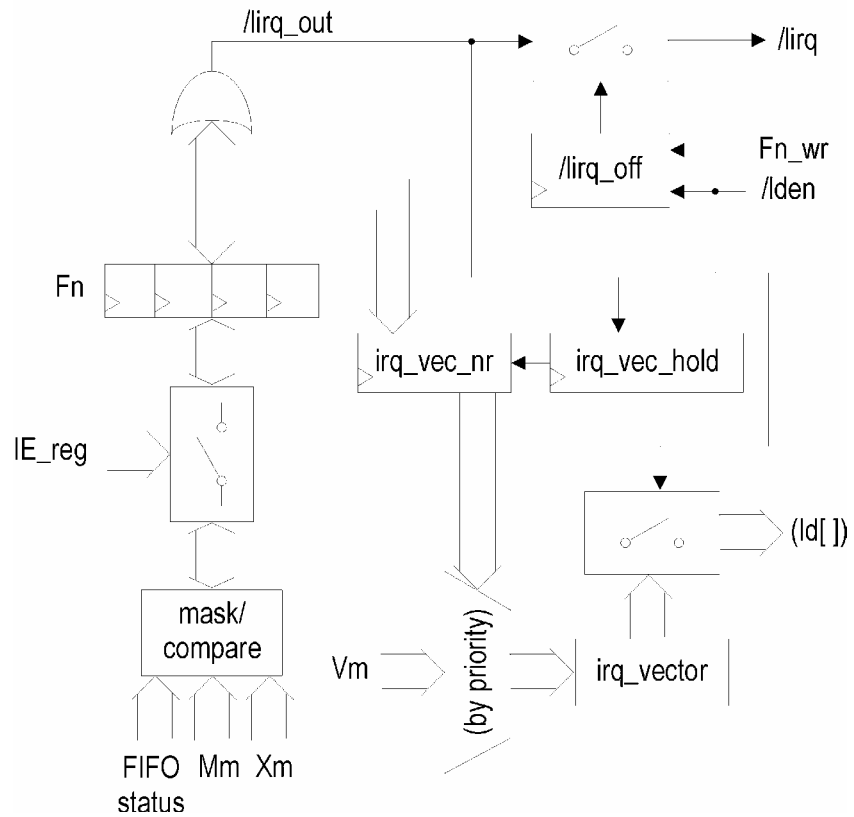
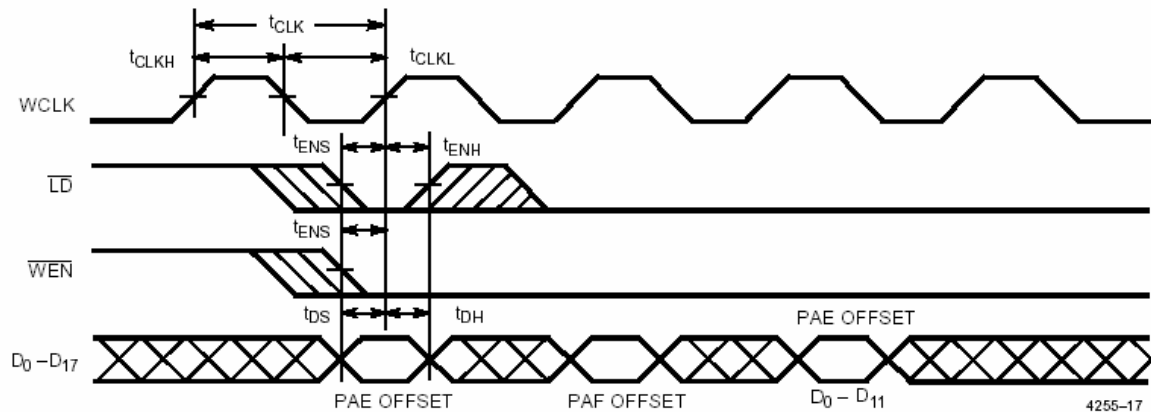


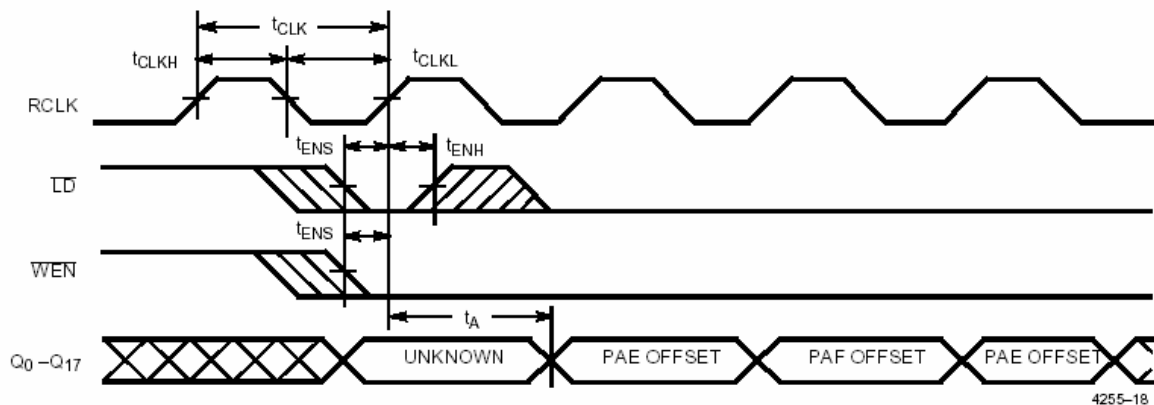
Figure 2 - Functional block diagram of the IRQ controller architecture

## Appendix A - Excerpt From Cypress CY7C4255 FIFO Datasheet

### Write Programmable Registers



### Read Programmable Registers



### Programming

The CY7C4255/65 devices contain two 14-bit offset registers. Data present on D0–13 during a program write will determine the distance from Empty (Full) that the Almost Empty (Almost Full) flags become active. If the user elects not to program the FIFO's flags, the default offset values are used (see Table 2). When the Load LD pin is set LOW and WEN is set LOW, data on the inputs D0 – 13 is written into the Empty offset register on the first LOW-to-HIGH transition of the write clock (WCLK). When the LD pin and WEN are held LOW then data is written into the Full offset register on the second LOW-to-HIGH transition of the write clock (WCLK). The third transition of the write clock (WCLK) again writes to the Empty offset register (see Table 1). Writing all offset registers does not have to occur at one time. One or two offset registers can be written and then, by bringing the LD pin HIGH, the FIFO is returned to normal read/write operation. When the LD pin is set LOW, and WEN is LOW, the next offset register in sequence is written.






The contents of the offset registers can be read on the output lines when the LD pin is set LOW and REN is set LOW; then, data can be read on the LOW-to-HIGH transition of the read clock (RCLK).

### Programmable Almost Empty/Almost Full Flag

The CY7C4255/65 features programmable Almost Empty and Almost Full Flags. Each flag can be programmed (described in the Programming section) a specific distance from the corresponding boundary flags (Empty or Full). When the FIFO contains the number of words or fewer for which the flags have been programmed, the PAF or PAE will be asserted,

signifying that the FIFO is either Almost Full or Almost Empty. See Table 2 for a description of programmable flags. When the SMODE pin is tied LOW, the PAF flag signal transition is caused by the rising edge of the write clock and the PAE flag transition is caused by the rising edge of the read clock.

**Table 1. Write Offset Register**

$\overline{\text{LD}}$	$\overline{\text{WEN}}$	$\text{WCLK}^{[36]}$	Selection
0	0		Writing to offset registers: Empty Offset  Full Offset
0	1		No Operation
1	0		Write Into FIFO
1	1		No Operation

--End of Excerpt--



## Appendix B - Excerpt From VME Specification (pp. 160)

### 4.3.7 Interrupt Request Release Capabilities

Many widely used peripheral ICs generate interrupt requests. Unfortunately, there is no standard method for indicating to these ICs when it is time for them to remove their interrupt request from the bus. Three methods are used:

- When the relevant processor senses an interrupt request from a peripheral device, it enters an interrupt service routine, and Reads a status register in the device. The peripheral device interprets this read cycle on its status register as a signal to remove its interrupt request.
- When the relevant processor senses an interrupt request from a peripheral device, it enters an interrupt service routine, and Writes to a control register in the device. The peripheral device interprets this write cycle to its control register as a signal to remove its interrupt request.
- When the relevant processor senses an interrupt request from a peripheral device, it reads a Status/Id from the device. The peripheral device interprets this read cycle as a signal to remove its interrupt request.

The VMEbus specification calls Interrupters that use methods a and b Release On Register Access (RORA) Interrupters, and those that use method c Release On Acknowledge (ROAK) Interrupters. Figure 4-8 shows how an ROAK Interrupter releases its interrupt request line when the Interrupt Handler reads its Status/Id and how an RORA Interrupter releases its interrupt request upon an access to a control or status register.

#### RULE 4.6:

An ROAK Interrupter **MUST NOT** release its interrupt request line before it detects a falling edge on DSA\* during the interrupt acknowledge cycle which acknowledges its interrupt, and it **MUST** release its interrupt request line within 500 nanoseconds after the last data strobe goes high at the end of the Status/Id read cycle.

#### RECOMMENDATION 4.2:

RULEs 4.5 and 4.6 have been retained for compatibility with previous versions of the VMEbus specification. The recommended practice for new designs is that the Interrupt REQUEST line be released as soon as possible but in all cases less than 50ns from data strobe high of the register access cycle (RULE 4.5) or the Status Id read cycle (RULE 4.6).

#### OBSERVATION 4.51:

The reason for speeding up the release time is to reduce the possibility of software race condition. 500ns can now be a large number of CPU instructions.

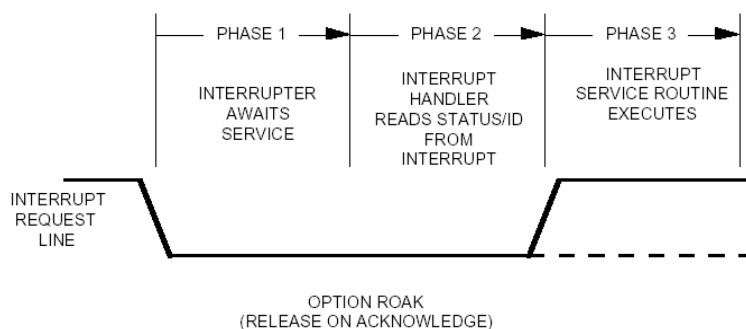


Figure 4-8

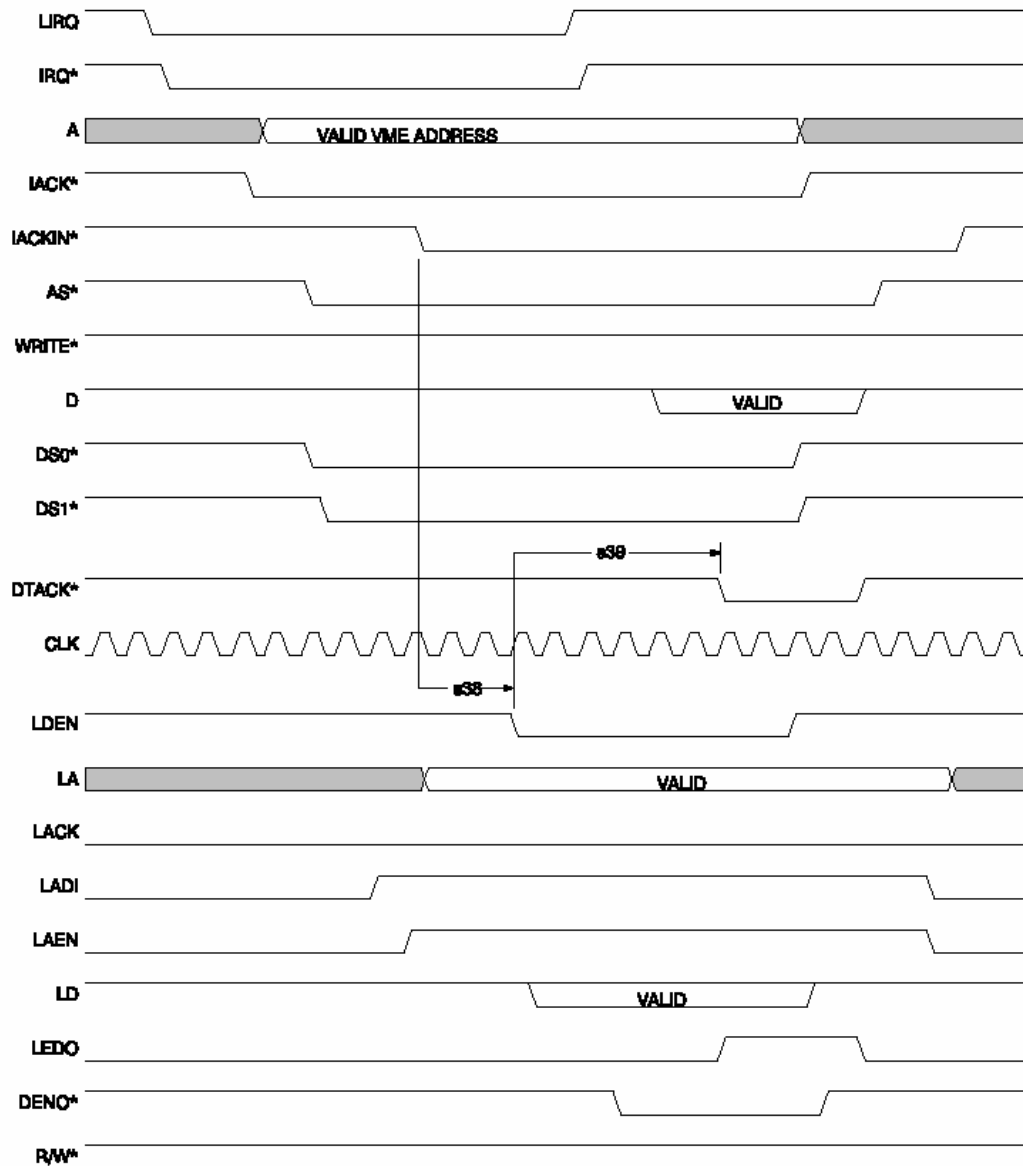
-- End of Appendix A --

### Appendix C - Handbook 3-1 (Waveform 7)



CYPRESS

AC Parameters



**Waveform 7. IACK CYCLE Self-timed**

-- End of Appendix B --

## Appendix D - Handbook 3-5 (pp.18)

### 3.5.11 Interrupt Cycle Support

There are six pins associated with interrupt requesting on the VMEbus: IRQ\*, IACK\*, IACKIN\*, IACKOUT\*, LDEN\*, and LIRQ\*. There are two ways an interrupt can be signalled on the VMEbus backplane: a local interrupt request or a reset.

A power-on, system reset, or local reset condition causes the device to implement its initialization protocol. Refer to sections 3.4.3 and 3.4.5 for details on initialization.

LIRQ\* is the input to the device from local circuitry demanding an interrupt cycle. If the LIRQ\* signal is asserted Low, the CY7C960's IRQ\* pin is asserted Low. The local circuitry should remove LIRQ\* after the interrupt has been acknowledged: The CY7C960 simply drives the state of LIRQ\* onto IRQ\*. *Figure 3-22* illustrates the relationship between LIRQ\* and IRQ\*. *s60* represents the assert and deassert delays.

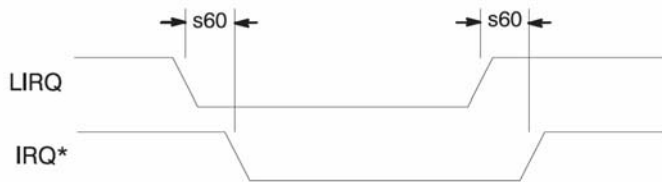


Figure 3-22. Interrupt Signal Timing

The CY7C960 actively drives the value of IACKIN\* to IACKOUT\* in conformance with the rules for VMEbus D8 Interrupt Acknowledge Cycles. If it is driving IRQ\* and IACK\* is sampled Low and IACKIN\* is sampled Low and AS\* is sampled Low, then LA[3:1] is compared with the programmed interrupt request level. If they are not equal then the Low IACKIN\* is driven to IACKOUT\*. If they are equal, then IACKOUT\* remains High and LDEN\* is driven Low to enable an external Status/ID vector onto the local data bus. This vector is in turn enabled onto the backplane and the VMEbus Interrupt Handler is acknowledged by a Low transition on the DTACK\* line.

It is important to note that, while the CY7C960 is a D8 interrupter, it drives all 32 bits of VMEbus data on the backplane with the assertion of DENO\* to the CY7C964s. If the user wishes to provide only 8 bits of status ID, then the value of the upper bytes should be set to FF hex to ensure full compliance with the VMEbus specification. If the user wishes to provide 16 or 32 bits of status ID, then the interrupt handler should be programmed to handle the provided width. The CY7C960 does not attempt to differentiate between 8-, 16-, or 32-bit interrupt acknowledge cycles.

### 3.5.12 Interrupt Handshake Support

The CY7C960 can be programmed at initialization to wait for a local acknowledge signal before terminating an IACK cycle. LACK is the local signal used to suspend completion of an IACK cycle until the local interrupter is ready to present a STATUS/ID vector to the VMEbus Interrupt Handler.

If the Handshake bit is set the CY7C960 will sample the state of the LACK pin three clocks after it samples a Low level on its IACKIN\* pin during a VMEbus IACK cycle. If LACK is High at this time, then the CY7C960 will wait until LACK is Low before terminating the cycle with a falling edge on DTACK\*.

If the Handshake bit is not set, the CY7C960 will ignore the state of the LACK pin and DTACK\* the VMEbus Interrupt Handler four clocks after sampling a valid IACKIN\*.

Handshake support is available immediately after local completion of the Combination Initialization Method or after full completion of the VMEbus or Local Initialization Methods. Refer to Chapter 3.4 for details on the three programming methods.

-- End of Appendix C --