

How to interface FPGAs to microcontrollers

Rocendo Bracamontes Del Toro, Atmel - July 30, 2008

As many as half of all embedded designs have an FPGA next to a microcontroller. The FPGA can be used to implement anything from glue logic, to custom IP, to accelerators for computationally intensive algorithms. By taking on some of the [processing](#) tasks, FPGAs help to improve system performance, thereby freeing up the MCU from cycle-intensive tasks. FPGAs also provide excellent performance characteristics and lots of flexibility to accommodate changing standards.

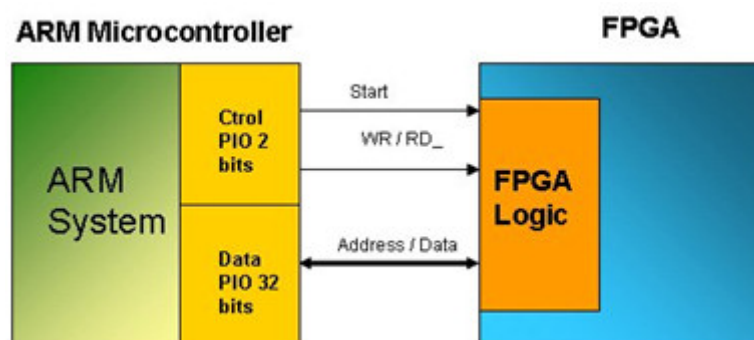
There are two basic implementations of MCU-plus-FPGA designs: putting a soft MCU core into the FPGA logic structure or using a standard product MCU with a discrete FPGA. Putting a soft core into the FPGA can be effective, but it can also be an expensive and power-hungry way to implement a microcontroller when compared to a standard product. This is especially true when using a [32-bit](#) ARM-based core. As a result, only about one-third of FPGA-plus-MCU designs are implemented with an MCU core inside the FPGA logic. The remaining two-thirds consist of a standard product microcontroller next to a discrete FPGA.

Neither standard product microcontrollers nor FPGAs were developed to communicate with each other efficiently. They even use different languages. Thus, interfacing the two can be a challenge. FPGAs do not have any dedicated logic that communicates with microcontrollers. This logic module must be designed from scratch. Second, the communication between the microcontroller and FPGA is asynchronous. Special care is needed to resynchronize the MCU to the FPGA [clock](#) domain. Finally, there is an issue of bottlenecks, both at the [interface](#) and on the MCU [bus](#). Transferring information between the MCU and the FPGA usually requires cycles on the MCU [bus](#) and usually ties up the resource (PIO or EBI) used to effect the transfer. Care must be taken to avoid bottlenecks with external [SRAM](#) or [Flash](#) and on the MCU bus.

There are basically three hardware options for interfacing the FPGA to the MCU: programmable [I/O](#) (PIO); external bus interface (EBI), if available; and, finally, a dedicated interface built into the MCU between the advanced high-speed bus (AHB) and the FPGA. Which approach to use depends on the end application and the desired result.

PIO Interface

Interfacing the MCU to the FPGA via the PIO is relatively simple for simple data transfers, consisting of the transfer of 32 bits of address, 32 bits of data, and some control signals for control. This requires a 32-bit PIO and an additional 2-bits on another PIO (*Fig 1*).



1. PIO interface to FPGA.

(Click this image to view a larger, more detailed version)

For a transfer of data to the FPGA, the direction of the bidirectional buffers in the PIO must be set to output. The software [algorithm](#) that transfers data to the FPGA is as follows:

```
PIO_DATA = ADDRESS; // Pass the address to write
PIO_CTRL = START | WR; // Send start of address cycle
PIO_CTRL = CLEAR; // Clear PIO ctrl, this ends the address cycle
```

```

PIO_DATA = DATA; // Set data to transfer
PIO_CTROL = START; // Data is ready in PIO
PIO_CTROL = CLEAR; // This ends the data cycle

```

Reading from the FPGA is similar. Again, the direction of the [buffer](#) on the PIO must first be set to output and then change directions to input to read the data from the FPGA, the following code is executed:

```

PIO_DATA = ADDRESS; // Set the address to read
PIO_CTROL = START | RD; // Send start of address cycle
PIO_CTROL = CLEAR; // Clear PIO ctrl, this ends the address cycle
PIO_DATA_DIR = INPUT; // Set PIO-Data direction as input to receive the data
DELAY(WAIT_FOR_FPGA); // wait for the FPGA to send the data
DATA_FROM_FPGA = *PIO_DATA; // Read data from FPGA

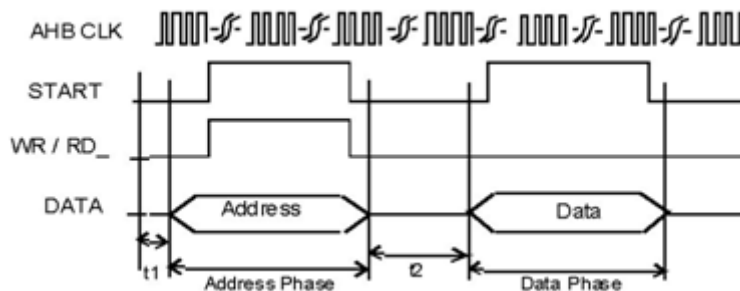
```

The above algorithms are for a basic transfer, a more sophisticated algorithm is necessary to establish a proper communication between the ARM microcontroller and the FPGA. Special care is necessary to ensure the acknowledgment of data, e.g. no data has been lost due to speed or wait cycles on each side.

The access time is calculated as the sum of:

$$T_{\text{Access-PIO}} = t_1 + \text{address phase} + t_2 + \text{data phase}$$

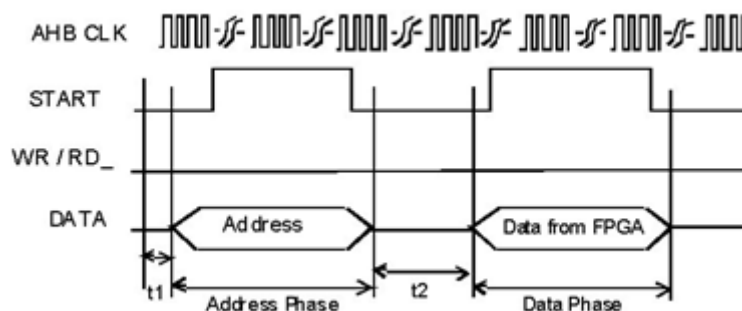
Using the GCC compiler with maximum optimizations, the system takes approximately 55 AHB cycles to perform the write operation to the FPGA (Fig 2).



2. PIO write to FPGA.

(Click this image to view a larger, more detailed version)

Assuming t_2 (wait for FPGA response ready) is also around 25 AHB cycles, the system takes approximately 85 AHB cycles for a read operation from FPGA (Fig 3).



3. PIO read from FPGA.

(Click this image to view a larger, more detailed version)

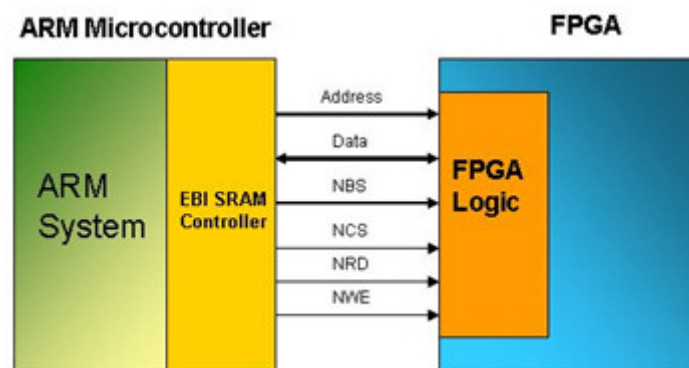
The interface from the MCU itself is fairly simple and straight forward. However, special logic must be implemented in the FPGA to decode all the traffic generated by the PIO. In the majority of cases, the traffic from the microcontroller is completely asynchronous. As a result, the FPGA must be able to oversample the control signals from the microcontroller; otherwise the FPGA will miss the time window and the traffic will not arrive at the final destination inside the FPGA.

Since the processor is the one in charge of keeping the PIO busy, there is an overhead of processing time. While the CPU is engaged in data transfers, it cannot do anything else. Thus, this solution has the potential to bog down system processing. DMA is not possible using a PIO interface, so the software programmer must limit data bandwidth to allow for other communication with the MCU. For example, if there is a routine that demands 100% of the processors cycles running concurrently with a serial (SPI, USART or TWI) transfer to or from the FPGA, then one of these two processes must wait. If the data going to or coming from the FPGA is not buffered on time, it will probably be overrun by the next byte/word of data. In essence, the embedded processor becomes a glorified data mover.

Interfacing through the External Bus Interface (EIB)

Many 32-bit microcontrollers have an external bus interface (EBI) module, which is designed to transfer data between external devices and the memory controllers on ARM-based devices. These external memory controllers are capable of handling several types of external [memory](#) and [peripheral](#) devices, such as SRAM, PROM, EPROM, EEPROM, flash, and SDRAM.

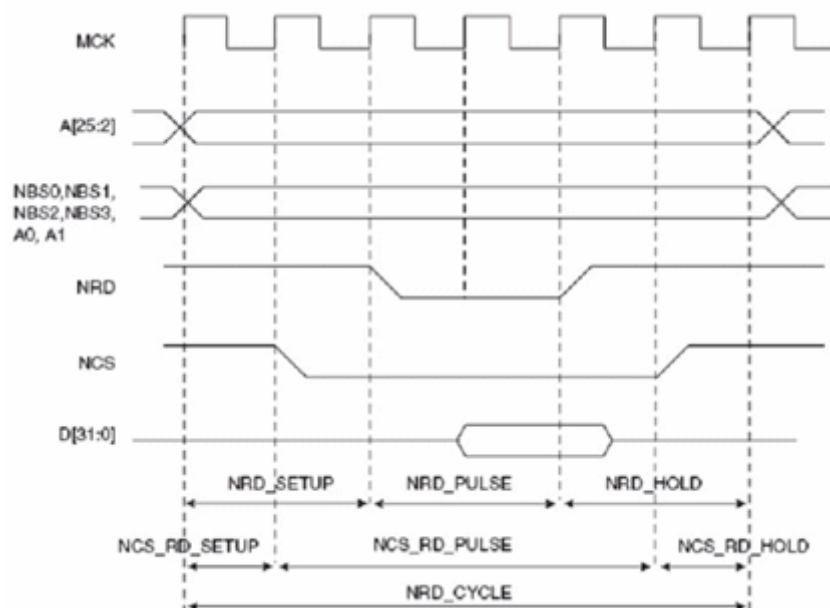
The EBI can also be used to interface to FPGAs as long as the FPGA can handle the predefined memory interfaces. Using the static memory interface (SRAM) on the EBI is preferable for FPGA communication because it is simple and most designers are familiar with it. As with the PIO interface, the FPGA will have to include a module that understands the SRAM timing and is able to produce a response back to the microcontroller (*Fig 4*).



4. EBI-SMC interface.

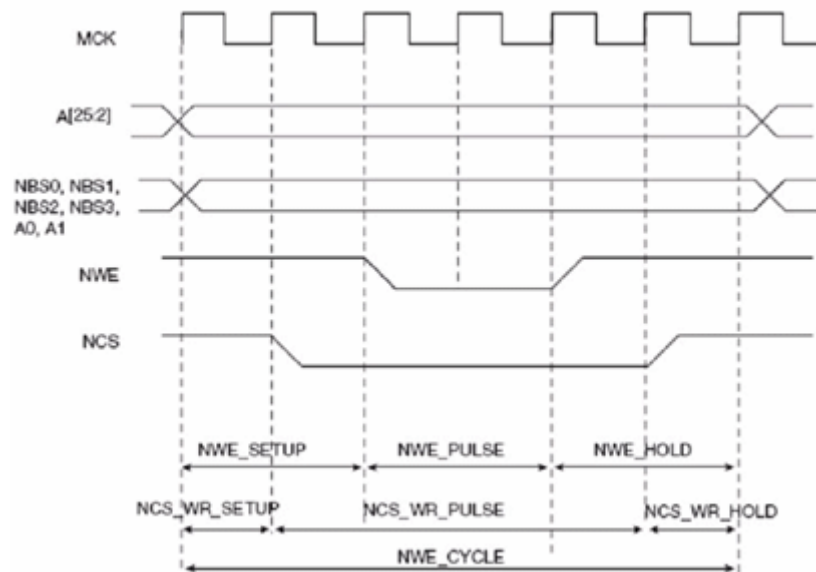
(Click this image to view a larger, more detailed version)

Fig 5 shows the standard read timing for the EBI using the SMC memory interface, while *Fig 6* shows the standard write cycle.



5. EBI-SMC read cycle.

(Click this image to view a larger, more detailed version)



6. EBI-SMC write cycle.

(Click this image to view a larger, more detailed version)

Note: These timing waveforms are the [default](#) for SMC specification. All parameters shown are programmable based on the speed of the external device.

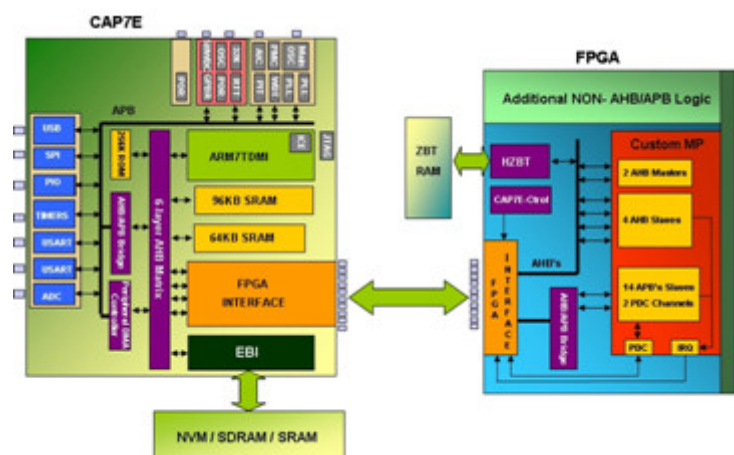
An EBI interface is faster than PIO because the EBI has its own I/O in place and most of the signals are parallel. However, if the external device is slow or introduces wait states, the EBI's speed advantage could be compromised.

Like the PIO interface, the EBI interface must be driven by the processor or another AHB master. Thus, the achievable bandwidth on the EBI is also dependent on [software](#) and depends on how much processor time is available to it. Certainly there could be a [bandwidth](#) limitation. This again potentially limits the embedded [processor](#) in carrying out other system functions that it was designed to do.

Using a dedicated FPGA interface on the MCU

ARM7-based microcontrollers are available with a special interface that allows the FPGA to directly access to the MCU's internal AHB bus with [DMA](#) access via two AHB masters and four AHB slaves. One extra AHB slave may be used to re-map the ROM using an external ZBT [RAM](#) through the FPGA with a programmable [ROM](#) re-map feature at startup.

The interface also gives the FPGA access to fourteen advanced peripheral bus (APB) slaves, two DMA full duplex channels, up to thirteen priority encoded interrupts (IRQs), two non-encoded IRQs for DMA transfers and 32-bits of shared programmable I/Os. The FPGA interface accesses the AHB through the pre-defined masters and slaves on the microcontroller (*Fig 7*).



7. MCU with dedicated FPGA interface.

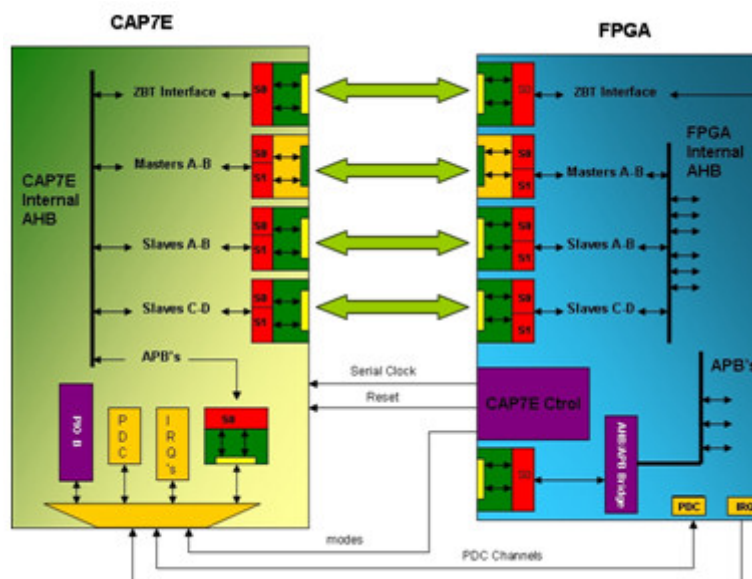
(Click this image to view a larger, more detailed version)

The FPGA interface is based on several serializers that encode and decode all the traffic between the microcontroller and the FPGA. In order to have a proper communication and synchronization between both devices, the following requirements must be in place:

- The FPGA must be capable of handling skew clock balancing and latency cancellation. In the Xilinx FPGA families, the usage of DCM's (Digital Clock Manager) is mandatory to handle all the latency cancellation and required clocks generation. Altera devices require the use of PLL's. The FPGA must also provide the configuration modes and reset to the microcontroller's built-in interface. It must provide the serial communication clock to the microcontroller, which may have a frequency of up to 100 MHz for commercial range devices. The ratio between the internal ARM7 clock and the serial clock should be in the order of 0.8 or lower (ARM Clk / Serial Clk).
- The FPGA-interface is based on a set of modules that encode and decode the internal AHB transactions. The encoded/decoded data is transferred through MPIO's using dedicated serializers for each master and slave. Due to the large number of bits to be transferred, a single transfer will take several AHB-Bus clock cycles. The specific number of clock cycles depends on the ratio between the AHB-Clock in ARM7, the serial clock, and the AHB-clock in the FPGA. Since the AHB-clock on the microcontroller is independent from the AHB-clock in the FPGA, the FPGA and microcontroller can run at different frequencies. Even masters or slaves inside the FPGA can run at different frequencies.
- Each serializer block on the microcontroller and FPGA has a complementary finite state machine (FSM) that understands and talks to the AHB bus. Thus, the interface can handle simultaneous transfers from either side eliminating the common bottle neck seen in others interface topologies such as EBI or PIO.

Microcontrollers with a direct FPGA interface also have DMA channels on nearly every peripheral, with several DMA channels dedicated to the FPGA interface. The DMAs are supported by a multi-channel peripheral DMA [controller](#) (PDC) that off-loads from the CPU the job of transferring data between the FPGA, the peripherals and the memories as in the other two methods. This avoids the bandwidth limitations of a conventional ARM7, which can be completely monopolized by data transfers at a data rate of only 4 Mega bits per second (Mbps).

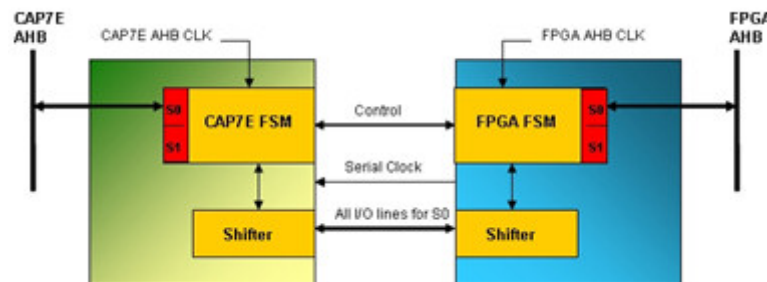
By off-loading this task from the CPU, the PDC can achieve a 12 Mbps data rate with 85% of the CPU's cycles available for processing. Multiple DMA channels are dedicated to the FPGA interface to connect multiple application-specific peripherals and interfaces to the PDC, without any intervention from the microcontroller. Using the dedicated DMA channels in the PDC frees the ARM processor to focus on processor heavy tasks and increases overall system performance and data bandwidth (*Fig 8*).



8. ARM7 to FPGA interface.
(Click this image to view a larger, more detailed version)

The serializer module handles all the AHB and serial communications. It consists of a finite state machine (FSM) and a shifter. The finite state machine understands and talks to the AHB. When a master initiates a transfer (read/write operation) the FSM introduces wait states using HREADY to comply with the AHB protocol. The number of introduced wait cycles is automatically handled by the FSM based directly on the ratio between the AHB clocks and the serial clock. The smaller the ratio is, the fewer the number of introduced wait cycles.

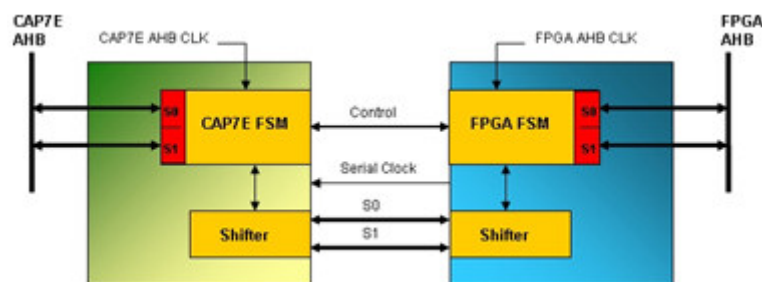
The FSM controls the shifter, which handles all the data shifting (serializing) between the microcontroller and the FPGA, transferring data at two bits per cycle. If the serial clock rate is set at 100 MHz, the shifter transfers 200 Mbps. The serializer modules that handle the masters A/B, slaves-A/B and slaves C/D are programmable at reset time through the "modes" module in the FPGA to maximize the number of available I/O. The designer can choose to use all 10 I/O lines for a single serial configuration, in which case the serial module will handle only one AHB interface. For example, if the user wants to use only AHB master A, the [serial](#) module handling the masters should be configured as a "serial single configuration". This configuration will improve the number of bits transferred between shifters, thereby speeding up the transactions between the microcontroller and FPGA (*Fig 9*).



9. Serial single configuration.

(Click this image to view a larger, more detailed version)

The other option is to configure the serial module to handle two AHB interfaces in "serial dual configuration", in which the 10 I/O lines are shared between the two AHB (masters/slaves). In this case, the data transfer rate between the microcontroller and FPGA is lower, but the data bandwidth is higher because two AHB interfaces are enabled. The dual configuration re-uses half of the dedicated I/O for another AHB interface (*Fig 10*).



10. Serial dual configuration.

(Click this image to view a larger, more detailed version)

FPGA interface logic

When implementing an FPGA interface through the EBI or PIOs, the engineer must write the RTL code that allows the FPGA to communicate with the MCU. Vendors of MCUs with a direct FPGA interface provide all the RTL needed to establish proper encoding and decoding communication with appropriate constraints for each specific FPGA vendor. This logic module produces a reset and provides the different modes under reset conditions. The RTL provided by the vendor lets the user decide what functionality to choose. By default all mode bits are zeroes (*Table 1*).

Mode -Bit	Description	0	1
0	Internal ROM select	Use internal ROM	Use external ZBT
1	Masters mode select	Use only Master A	Use Master A and B
2	Slaves A-B mode select	Use only Slave A	Use Slave A and B
3	Slave C-D mode select	Use only Slave C	Use Slave C and D
4	PIOB mode select	Use PIOB	Use IRQ's, PDC, APB's
5	CAP7 in ARM MODE used for emulation of CAP7 only	CAP7E mode	CAP7-ARM emulation mode
6	Disable Pullups	Use Pull-Ups	Disable-Pullups
7	ADC / LVDS Select used for emulation of CAP7 only	Use ADC	Used LVDS

Table 1. Mode bits.

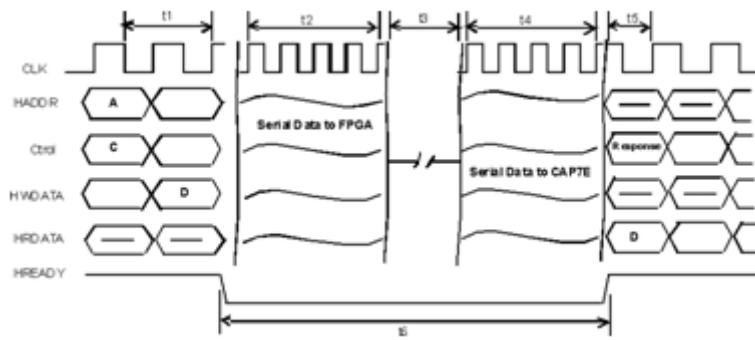
A vendor-supplied template can be used to instantiate the AHB masters and slaves to the FPGA interface. Specific examples are provided. In the FPGA template, a module called "Custom MP" requires the least amount of effort to integrate AHB/APB peripherals. The usage of this template gives the designer the option of migrating the two-chip MCU-plus-FPGA implementation to a single-chip customizable microcontroller with virtually no extra engineering effort since the logic in the FPGA will have been proven in the system.

The external ZBT-RAM and NVM/SDRAM/SRAM are optional, based on applications and system requirements.

Designers also may add non-AHB logic to the FPGA, providing the flexibility to add other functions unrelated to the AHB bus.

As mentioned before, the transfer speed depends from the ratio between all clocks related to the communication between the microcontroller and the FPGA and whether it is in single or dual serial mode.

In single configuration mode, it takes four AHB-Clock cycles to transfer all the AHB information for a single-AHB interface from the microcontroller to the FPGA and vice versa (*Fig 11*). In dual configuration mode, it takes eight AHB-Clock cycles to transfer all the AHB information for a dual AHB interface from the microcontroller to the FPGA and vice versa.



11. Timing for read/write transfers with direct FPGA interface.
(Click this image to view a larger, more detailed version)

The timing related to a transfer happening between the ARM7 MCU and the FPGA is as follows:

- **t1:** Time for a standard 2 cycles AHB
- **t2:** Time to transfer the request to FPGA (4 cycles single AHB interface, 8 cycles dual AHB interface)
- **t3:** Time for FPGA-Peripheral response
- **t4:** Time to transfer response back to CAP7E (4 cycles single AHB interface, 8 cycles dual AHB interface)
- **t5:** Time to read back the response/data from FPGA to the internal CAP7E AHB bus
- **t6:** Time for introduced wait cycles

The following formula is used to approximate the access time, from the ARM to the peripherals in the FPGA:

$$T_{\text{access}} = t1 + t2 + t3 + t4 + t5$$

Note: t1 and t5 are AHB cycles that could be ignored for comparison to PIO and EBI timings.

Conclusion

In situations where the data rate is low, such as a dot matrix LCD, an interface the MCU to the FPGA through the PIO or the EBI is sufficient. However, a high data rate between the FPGA and the MCU, or between some other peripheral and the memories, could monopolize CPU cycles and create bottlenecks on the peripherals. For example, a TFT LCD graphics color will require substantial amount of data to be transferred from the frame buffer to the LCD that would most likely completely overwhelm both the CPU and the EBI. This kind of application will perform better with direct interface from the MCU to the FPGA, while allows the LCD data to be transferred over the DMA, keeping the processor free for processing and the EBI free for other data transfers, such as the main application software running from flash, or in the case of the TFT LCD the usage of SDRAM for single or multiple frame buffers.

In addition, the logic defined in the FPGA on the AHB appears to the MCU as if it "inside" the MCU. This makes migration to a customizable microcontroller an easy future design path.

Development time is also shorter with a direct FPGA interface because the interface is already defined inside the microcontroller and the logic module for the FPGA is provided by the MCU vendor. The designer does not need to write any interface RTL. A microcontroller with a dedicated FPGA interface is likely to improve overall system performance and ease of design.

Rocendo Bracamontes del Toro received his bachelor's degree in electrical engineering from Tecnológico Cd. Guzman, Jalisco Mexico in 1997 and his master's degree in [computer](#) and electrical engineering from John Hopkins University, Baltimore, Maryland in 2003.

Rocendo can be contacted at rocendo.bracamontes@atmel.com.

