ACC, a Next Generation CAN Controller

Reinhard Arlt, esd electronic system design gmbh
Andreas Block, esd electronic system design gmbh
Tobias Höger, esd electronic system design gmbh

Most standalone CAN controllers available today are connected to the host system by eight or sixteen bit wide parallel buses. Write and especially read accesses to such peripheral devices are very slow compared with the cycle time of modern CPU's. This paper discusses the resulting performance bottleneck and shows a solution, using a CAN controller implemented in an FPGA, that can use bus master DMA.

Introduction

There are still no CAN controllers integrated into most high end microprocessor chips and a common way to add CAN using these processors is simply to connect one or more standalone CAN controllers to such systems.

My co-workers and I have done a performance analysis on an embedded system, as we have found out, that the performance of this system does not scale well with increased CPU power.

Unfortunately we found out, that one of the bottlenecks was a CAN interface build around a SJA1000.

There are numerous eight bit accesses to the chip necessary to handle one CAN frame and most of these accesses are done in the interrupt routine with each access taking quite a while.

The Performance Bottleneck

The reason for this is that most available standalone CAN controllers are designed to work with relative low power microcontrollers.

The consequences are rather slow 8 or 16 bit wide interfaces. Interfacing such a device to state-of-the-art CPU's results in

a performance bottleneck, as the CPU and the system bus may be blocked for several thousand cycles by a single access to one register of the CAN controller.

The introduction of serial system buses like PCI Express makes the situation even worse, as these bus systems are optimized for streaming large blocks of data from the device to the memory of the host CPU and vice versa, while normal standalone CAN controllers rely on single byte accesses. The resource “System Bus Interface” is blocked for the other threads in the CPU core, that is used by the CAN process, as well as for all other CPU cores. Therefore you have to minimize “read” accesses to the device, and all “write” accesses should be posted, so the CPU does not need to wait for the completion of the accesses.

As an extreme example, the read access to an 8 bit register of a CAN device connected by PCI Express, may take up to 2000ns, and a 3 GHz CPU has to wait here for 6000 clock cycles.

Today, as even PCI slots may be connected with a PCI Express to PCI bridge to the host CPU, a single byte access to a PCI device is even slower due to the additional delay of the PCI Express to PCI bridge. Please note, while the access time to the real device is less than 100ns long, the time is spent sending for example a 160 bit long request packet to the PCI Express Bridge, then waiting the 100ns device access time, and for replying a 160 bit long answer packet from the PCI Express Bridge to the CPU.

A First Approach

At the time we detected this bottleneck, about 10 years ago, we decided to use a very small FPGA to poll the data from the SJA1000 CAN controller, present them in a 32 bit wide register within the FPGA to the CPU and to transfer a CAN message from 32 bit wide registers in the FPGA to the SJA1000.

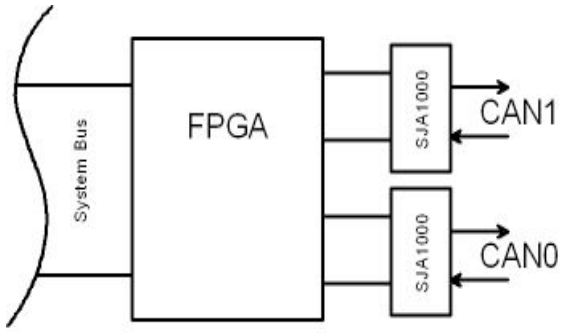

Figure 1: Supporting SJA1000 CAN Interfaces with a small FPGA

As an additional feature, the CAN-ID, DLC and data fields of the CAN frame were now arranged in a way that is much more convenient for a 32 bit processor. It was possible now to handle the data transfer on task level and not within the interrupt service routine. Even this project was only partially successful due to the limitation of the affordable FPGA resources at that time. The idea was born to implement a high performance CAN controller in an FPGA.

Technology used by Ethernet

Even computer systems with slower CPU's can handle several Ethernet interfaces without a significant CPU load, so why not have a look, how a typical Ethernet controller is build?
There are, for example, only very few Ethernet controllers available today, intended for the use in low end and low performance applications, that rely on data transfer done by the host CPU.

The majority uses an at least 32 bit wide interface, and transfers the data directly to and from the memory of the host CPU.

On the other hand, there are only a few Ethernet controllers, that use a local CPU to transfer data.

Why not use this well established technology for a CAN controller, too? Perhaps the market for a high end CAN ASIC or custom chip is too small.

CAN Interfaces with Local CPU

In the past and still nowadays, these problems are often mitigated by using “active” CAN interfaces, where the CAN controllers are connected to a local microcontroller, or even are an integrated part of that chip.

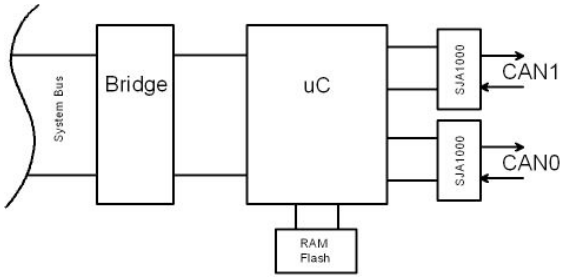

Figure 2: CAN Interfaces with a local CPU

If the local microcontroller of those implementations has the possibility to access the memory of the host CPU, it is possible to mitigate the issue with long access times quite well, but there is a new problem: As these processors used here are typically 10 times slower than the host CPU's, they introduce a noticeable delay between the transmit command of the host CPU and the appearance of the CAN frame on the CAN bus, and from the physical reception of the CAN frame until it is available to the host. This will increase the turnaround time for protocols, that rely on a direct answer from a device on that CAN bus, for example when doing an CANopen SDO transfer.

Use a CAN Controller with SPI Interface?

SPI (Serial Peripheral Interface) is a four wire serial interface used for the connection of slower peripheral devices to a host CPU. An additional wire for an interrupt is typically needed to connect a CAN controller.

Many high end microprocessors that do not have a CAN interface on chip provide very powerful SPI interfaces with very short access times. But due to slow SPI clocks, known SPI CAN controllers like Bosch CC750/CC770 or Microchip MCP2515 use only 8 MHz or 10 MHz, there will be again a significant delay introduced for the transfer of the CAN message.

The transfer of an 8 byte CAN frame with the SPI interface may take more than 10 microseconds. Depending on the SPI controller, there might be an interrupt for every byte transferred to the CAN controller, or the CAN driver has to poll for the completion of the transfer of each byte.

The only remaining advantage is, that the system bus is not blocked for other CPU's.

Develop our own CAN Core?

Before implementing a CAN core from scratch, it should be good engineering practice to check, if there are CAN cores available, that meet the requirements and the budget limits of the project.

With the knowledge gathered from the development of CAN drivers for many different CAN controllers, different operating systems, host system architectures and numerous CAN applications, we had to accept, that there is no CAN core available, that fulfills all requirements, or where the supplier is willing to modify his core for a reasonable price and within the given timeline.

Again, we had to find out, that most of the CAN cores in the market are only designed for the use in simple CAN I/O nodes.

Requirements

Our new CAN core must be compliant to ISO 11898-1 (CAN 2.0A and CAN 2.0B) and must support all relevant baud rates.

It must have a 32 bit wide interface, but it should be possible to synthesize it for 16 bit or even 8 bit wide bus interfaces.

It must provide 64 bit wide timestamps with a resolution of less than 1 microsecond.

There must be deep enough FIFO's for the transmit and receive path, to allow the usage within non real time systems.

The CAN core shall be able to generate 100% bus load. It must be able to receive a long stream of CAN messages with zero bytes of data without losing a message.

A “transmit done” or “transmit aborted” indication must be placed in the receive FIFO. This is necessary to inform the driver and application software of the exact sequence of the CAN frames on the bus.

It must be possible to abort any transmit request, that has not already won the bus arbitration and to get an indication for this event. This requirement is for example necessary to implement protocols that allow the transmission of a frame in a predefined time slot only (e.g. ARINC825).

It must have a seamless interface to a busmaster unit.

It must be possible to evaluate the CAN baud rate of an active bus without a significant delay and without any disturbance of the CAN messages.

An interrupt status register has to be implemented in a way, that allows a minimum number of accesses in the interrupt service routine, ideally no reads and as minimal as necessary write accesses.

It must be modifiable and extendable, the HDL source code must be available, and there should be no limitations in the use of the core.

The core should be written in VHDL.

The core should support a second local bus interface. This is necessary to allow a host CPU and a local CPU to access the same CAN bus. Although it would be possible to use two distinct CAN cores for the same pysical CAN interface, one for

the local and one for the host CPU, such an implementation would make it impossible to send messages with the same identifier from both CPUs.

The Basic CAN Core

The basic CAN core is built up from a CAN bit stream engine, two FIFO memories and a register file.

The bit stream engine constructs the stuffed CAN message, arbitrates for the bus, sends the message, adds the CRC and framing bits and checks for retransmission. It also receives the data from the bus, does the CRC check, and sends ACK and ERROR flags. The handling of the CAN error states is implemented here, too.

The register file is the interface to the driver and it is connected to the bit stream engine by the transmit and the receive FIFO. It also provides the bit timing parameters for the bit stream engine and reports the status information from the bit stream engine to the driver.

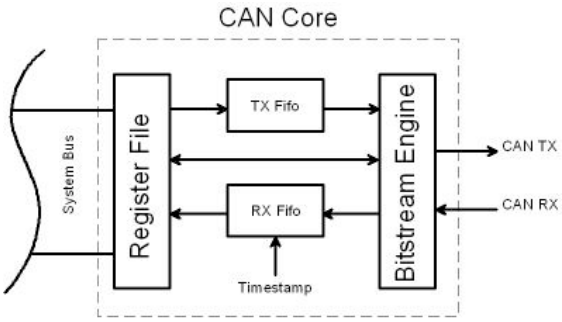

Figure 3: Basic CAN core

The timestamp generator is an input to the CAN core, because all instances of the core must use the same timestamp.

A global register file provides the number of cores in this FPGA to the driver, generates the timestamps for all cores, and informs the driver about the cores that needs attention.
With this basic implementation, it is possible to send a CAN Frame and start its transmission with only four write access cycles to the controller, and to read a frame within six cycles, including two cycles for a 64 bit timestamp.

An FPGA with this basic CAN core connected to a low latency local bus of PowerPC microprocessor is a very good solution for many embedded applications.

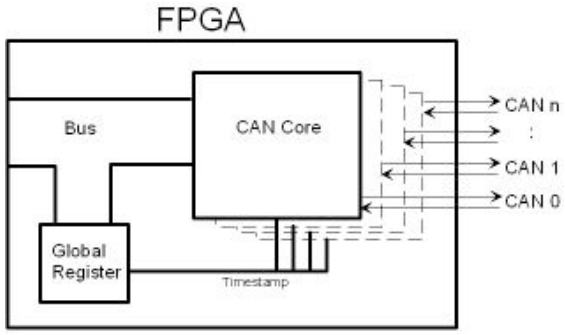

Figure 4: CAN core in an FPGA

Busmaster

The bus master unit transfers CAN frames and “transmit done” or “transmit aborted” indications from the cores into the memory of the host system without the help of the host CPU.

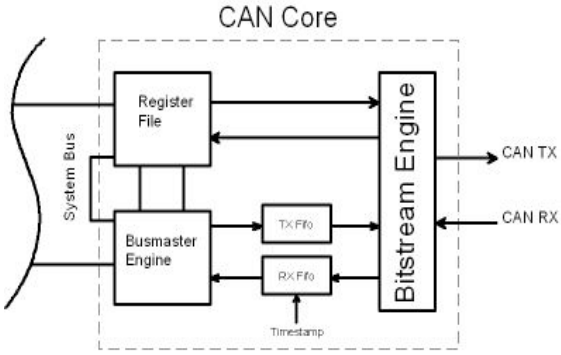

Figure 5: CAN core with bus master engine

A Local CPU

A local CPU can be very helpful for a CAN interface, that needs to cyclically send a lot of data. Adding a “soft” CPU to the core does not use too much resources in the FPGA, and it is usually much easier to implement more complex actions in a “C” program than in VHDL. Additionally changing the software for this CPU does not require the revalidation of the whole FPGA.

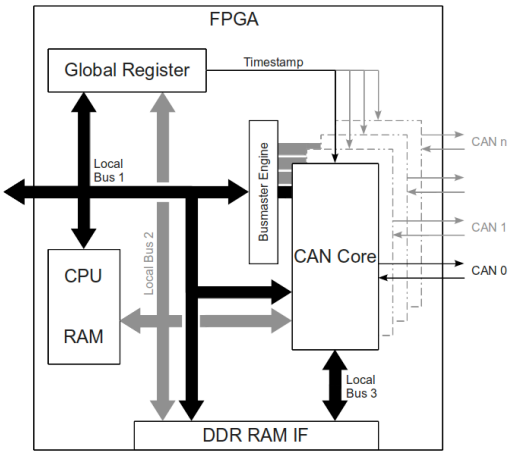

Figure 6: CAN core in a FPGA

Adding a DDR-RAM Interface

There are many possible applications for a large RAM within a CAN interface. It can be simply used as memory for a “soft” CPU or even as a “perfect” filter for all 29 bit CAN identifier.

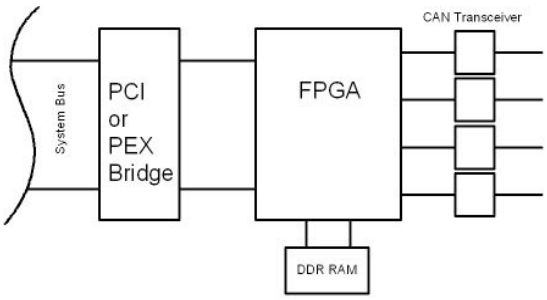

Figure 7: Typical CAN interface

PCI and PCI Express Interface

It is easy to add a PCI or PCI Express core to an FPGA, to get a “CAN chip” with such an interface. As most state-of-the-art FPGA's are no longer 5 volt tolerant, and the PCI slots in modern PC's are still coded as 5 volt slots, this is no solution for a general market interface, but is a good solution for an embedded system. PCI Express interfaces do not have such restrictions.

Resources Used in an FPGA

The CAN core is used mostly in Xilinx FPGA's at the moment. It is possible to implement more than 12 CAN core's with bus master support in a Spartan XC3S1600E chip. Depending on the number of CAN interfaces needed, there are many resources left for a soft CPU and other advanced features. Implementations in combination with error injection or IRIG-B have already been realized.

Field Proven

Nowadays, we are using the CAN core in many different products. The family of CAN/400 boards is build up of a PMC, Compact PCI, PCI and PCI Express board. These boards use the additional DDR RAM interface and also the PCI bus master engine. On the AMC-CAN board, there is no external RAM. The CAN core combined with a PCI Express endpoint in the FPGA is used on a custom board with an Intel ATOM CPU and a XMC board with a QorIQ P2041 quad core PowerPC microprocessor. Finally the basic configuration is used on our VME-CPU and other custom boards.

Next Steps

An error injection unit has already been added as an option to the CAN core.

Future directions may be more diagnostic tools, like monitoring and recording of the CAN bit stream.

Summary

This paper describes a CAN controller, that is implemented as an IP core in an FPGA and overcomes the long access times of standalone CAN controllers by implementing a 32 bit register interface and streaming the data from the CAN bus into the memory of the host CPU by bus master DMA. Deep FIFO sizes for reading and writing, precise timestamps and the ability to abort a CAN frame accurately, even if it is in the transmit FIFO (as needed for more sophisticated CAN protocols like ARINC825) and a register model optimized for the needs of CAN, are additional features. Depending on the selected features, up to twelve ACC CAN cores fit into an Xilinx Spartan XC3S1600E FPGA.

References:

[1] ISO 11898-1, (December 2003): Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signaling.

[2] ISO 11898-2, (December 2003): Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit.

[3] Etschberger, Konrad (2002): Controller-Area-Network. Grundlagen, Protokolle, Bausteine, Anwendungen. München: Hanser.

[4] Voss, Wilfred (2005): A comprehensible guide to controller area network: Copperhill Technologies Corporation.

[5] Phillips Semiconductor (1997): SJA1000. Stand-alone CAN Controller.

[6] Robert Bosch GmbH (2000): CC750 SPI-CAN.

[7] Robert Bosch GmbH (2009): CC770 Stand Alone CAN Controller.

[7] Microchip Technology Inc. (2010) Stand-Alone CAN Controller With SPI Interface.

[8] PCI-SIG (November 2010): PCI Express Base Specifikation Revsion 3.0.

[9] Höger, Tobias (2002): Entwurf, Aufbau und Test eines busmasterfähigen CAN-Interfaces für CompactPCI-Systeme.

[10] Webermann, Hauke (2011): Entwurf, Implementierung und Test einer Funktionseinheit zur Injektion von Fehlern in CAN-Busse.

Reinhard Arlt
esd electronic system design gmbh
Vahrenwalder Str. 207
D-30165 Hannover
+49-511-37298-0
+49-511-37298-68
reinhard.arlt@esd.eu
www.esd.eu

Andreas Block
esd electronic system design gmbh
Vahrenwalder Str. 207
D-30165 Hannover
+49-511-37298-0
+49-511-37298-68
andreas.block@esd.eu
www.esd.eu

Tobias Höger
esd electronic system design gmbh
Vahrenwalder Str. 207
D-30165 Hannover
+49-511-37298-0
+49-511-37298-68
tobias.hoeger@esd.eu
www.esd.eu