AVR-SoC

An AVR-based System-on-a-Chip

Mario Mauerer
mauerer.m@gmail.com

Jan 2011

Contents

Chapter 1

Introduction

The AVR-SoC was designed to provide a simple softcore-platform that can easily be implemented on an FPGA.

The System features an 8-Bit AVR CPU as main component. A Wishbone-Interface provides an interconnect to various other components. The system can be extended and modified to individual needs with only simple modifications.

To program the CPU, a standard AVR-toolchain such as AVR-GCC can be used, just like when working with a ,,normal" AVR. Using the Xilinx ,,data2mem" tool, the system does not have to be re-implemented every time when the firmware of the AVR has been altered. The ,,data2mem" tool simply creates a new bitstream where the program code is included. This speeds the entire process of developing firmware for the SoC massively up as it can quickly be transferred to the FPGA.

This basic system was developed on a ,,Xilinx Spartan-3E Starter Kit", therefore, it uses some specific hardware-components of that board to demonstrate the system. The LEDs, sliding switches and the UART are used in this simple form of the SoC.

You can find the entire sourcecode and other documentation along with this project.

If you have any questions or comments concerning this project, feel free to mail me: mauerer.m@gmail.com

Chapter 2

Used Tools

The following tools and projects have been used to develop the SoC and the firmware:

- Xilinx Spartan-3E Starter Kit:
  http://www.xilinx.com/products/devkits/HW-SPAR3E-SK-US-G.htm
- Xilinx ISE 12.4:
  http://www.xilinx.com/support/download/index.htm
- WinAVR 20100110:
  http://winavr.sourceforge.net/
- Atmel AVR Studio 4:
  http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
- AVR Core:
  http://opencores.org/project,avr_core
- UART 16550 Core:
  http://opencores.org/project,uart16550
- Wishbone Bus:
  http://opencores.org/opencores,wishbone
- Wishbone Conbus:
  http://opencores.org/project,wb_conbus

Chapter 3

Illustration of the SoC

This SoC is a combination of some open-source projects and own modules. In the following, the main components will be explained.

Figure 3.1 shows the Block diagram of the AVR-SoC as it is implemented in this project.

3.1 AVR Core

The AVR Core was taken from opencores.org (http://opencores.org/project,avr_core). It implements the core of an Atmel AtMega103 Controller. However, this is only the CPU of this microcontroller, there are no peripherals like timers. As the core is the only component responsible for executing the program code, it can be programmed like any other AtMega103.

3.1.1 Program Memory

The program memory is provided by some of the Spartan-3E's Block Rams. 16 BRAMs are used in the ,,RAMB16_S1"-configuration, therefore providing 32k Bytes of program memory for the CPU. The data width is 16 bits.

The BRAMs are initialized upon configuration of the FPGA. The initialization-constants are generated by the ,,data2mem"-tool and stored in the ,,progMemInit.vhd"-file.

The ,,prog_mem.bmm"-file specifies the layout of the program memory. With this file, the ,,data2mem"-tool knows how to put the data into the memory.

The program memory's clock is 180 degree phase shifted that that of the AVR. This is needed because the AVR wants to read the data from the memory at the next clock cycle from the beginning of the query. As there is a FlipFlop in the Address-path of the BRAMs, this wouldn't be possible if the BRAMs had the same clock than the AVR. The desired result can be achieved by inverting the clock of the memory so that the memory can deliever the data when the AVR needs it. From the AVR point of view, a

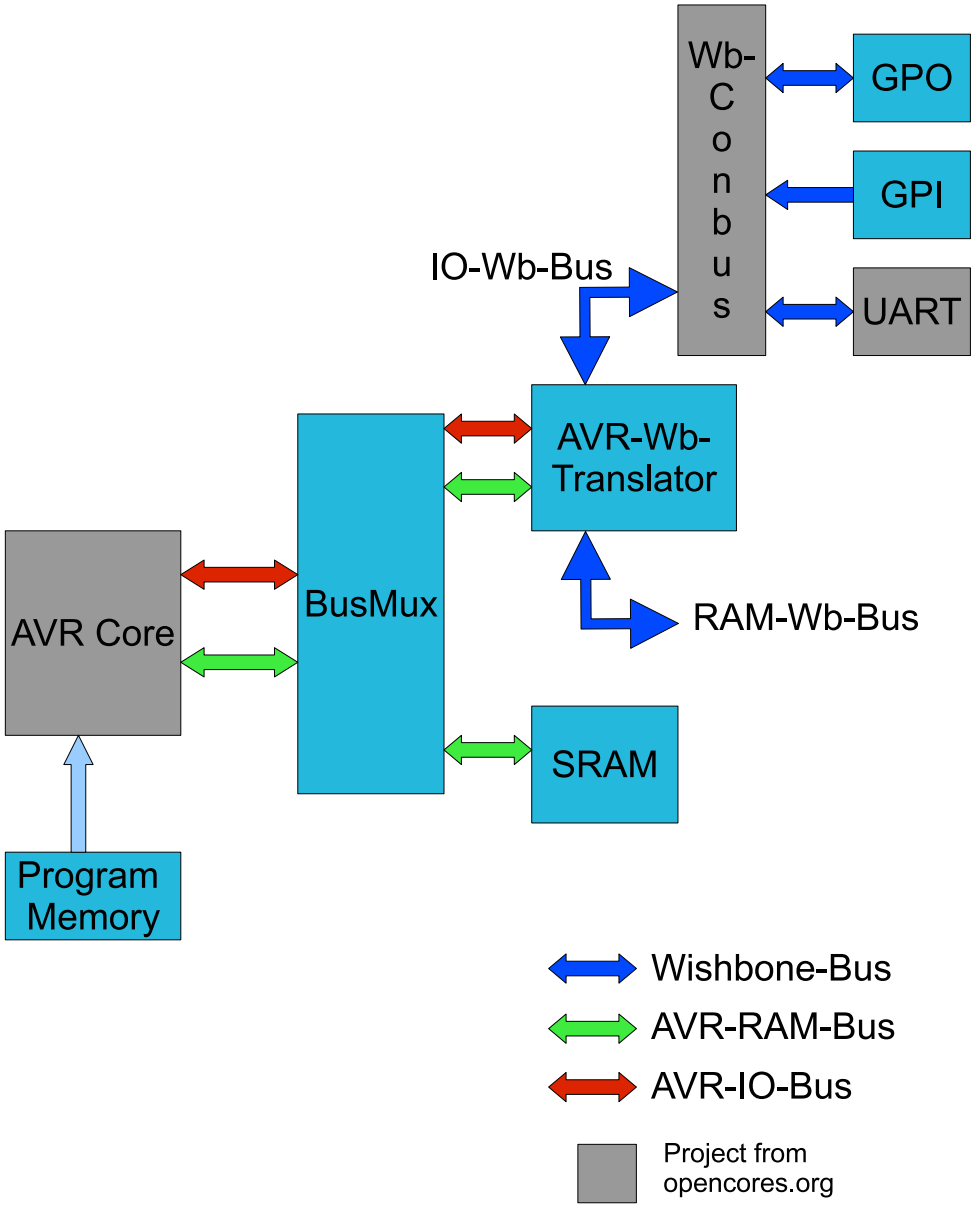

Figure 3.1: Block Diagramm of the SoC

fully ,,combinatorial" memory is provided in this way, the data are ready at the next clock cycle.

The address of the program memory is simple the program counter.

3.1.2 SRAM

The SRAM is also BRAM-based. It is 8k Bytes large, using 4 BRAM's in the ,,RAMB16_S2"-configuration. The data bus width is 8 bits.

Here, the same trick as with the program memory had to be done - inverting the clock in order to have the data ready at the next cycle.

3.1.3 AVR Busses

The AVR CPU has two busses: an IO-Bus and a RAM-Bus. However, they are not completely separated; they share the same datapath. Only the Read-, Write- and Address-Signals are distinct.

IO-Bus

The IO-Bus has an address width of 6 bits and is therefore capable of addressing 64 Bytes.

RAM-Bus

The RAM-Bus has an address width of 16 bits and is thus capable of addressing 65536 Bytes. The SRAM is attached to this bus, using the lowest 8192 Bytes (Addresses 0x00 to 0x1FFF).

3.2 BusMux

The BusMux is a Multiplexer, it diverts the datastreams to the correct locations. Accesses to the lowest 8192 Bytes on the RAM-Bus (Addresses 0x00 to 0h1FFF) are directed to the SRAM. The rest of the accesses is forwarded to the AVR-Wishbone Translator.

Accesses on the IO-Bus are almost completely forwarded to the AVR-Wishbone Translator. However, the AVR CPU presents accesses on some internal system registers (like the stack-pointer) on the IO-Bus. These won't be forwarded, they are simply ignored as they are of no use. (Addresses 0x34 to 0x3F)

One speciality of the AVR core is the fact that the outgoing data, if there is an access on the RAM-Bus, have to be delayed by one clock cycle. This is achieved by inserting a flipflop into the datapath. However, this may not affect accesses on the IO-Bus! When such an access is performed, the data may not be delayed, the flipflop is then bypassed.

3.3 Wishbone-Bus

The Wishbone-Bus is an open source hardware computer bus intended to let the parts of an integrated circuit communicate with each other. It's specification can be found at http://opencores.org/opencores,wishbone. This bus is used to let the AVR talk with its peripheral devices. Therefore, the accesses on the IO- and RAM-Bus have to be translated into Wishbone-Bus-Accesses.

3.3.1 AVR-Wishbone-Translator

This entity allows our AVR softcore to communicate with the rest of our SoC. As our on-chip bus is a Wishbone bus, this entity converts the data-interaction of the AVR to Wishbone-bus interactions.

The entity forms two Wishbone-interfaces, as we have to connect two busses from the AVR to our System; the IO-bus and the data-bus.
Usually, when the AVR initiates a transfer on one of its busses, the behavior of the modules attached is deterministic, for example, each write-action on the IO-bus will take one cycle. But with the many different modules we have attached to our system and to the CPU, this isn't the case anymore. We don't know how long a transaction will take place and how long it will take until the slave has finished its operations. Therefore, we have to stall the CPU during a Wishbone-access. Luckily, the AVR softcore we're using offers this functionality. The CPU has a ,,clock-enable"-input which can halt the entire cpu.(The cpuwait-input of the core can't be used!) This is exactly what we need because now, as soon as the AVR initiates a transfer on one the two mentioned busses, the entity stalls the AVR until the transfer on the wishbone-side is finished (the slave will send e.g. an ,,ACK" back, see the wishbone-specification.).
With this system, the AVR doesn't notice at all that there is more behind its busses, for the softcore, everything looks still the same.

Another concern was the fact, that the CPU has only one 8-bit datapath for both busses. Therefore, the translator has to multiplex the incoming data to this input, depending on which bus the transaction took place. The outgoing data are simply forwarded to both busses.

The AVR IO-Bus has a 6-bit address and the data-bus a 16-bit address. Thus, both wishbone-interfaces have each the same address-widths. Because the AVR has only one 8-bit datapath, both wishbone-interfaces have an 8-bit data bus. The rest of the wishbone-implementation is pretty simple, made according to its standard and can be read in the Wishbone documentation.

With this principle, we can completely isolate the AVR from the rest of the system, the CPU isn't affected at all because it can be halted during a wishbone-transaction.

3.3.2 Wishbone Conbus

The Wishbone Conbus is a Wisbhone-bus arbiter. Using this, one can connect several slaves and masters using the Wishbone-bus. The AVR-Wishbone-Translator is the only master in this SoC. The rest (in this case: general purpose output/input and the uart) are slaves on the bus.

3.4 Memory Map

The following figure 3.2 shows the memory map of the IO-Bus, as it is used in this project:

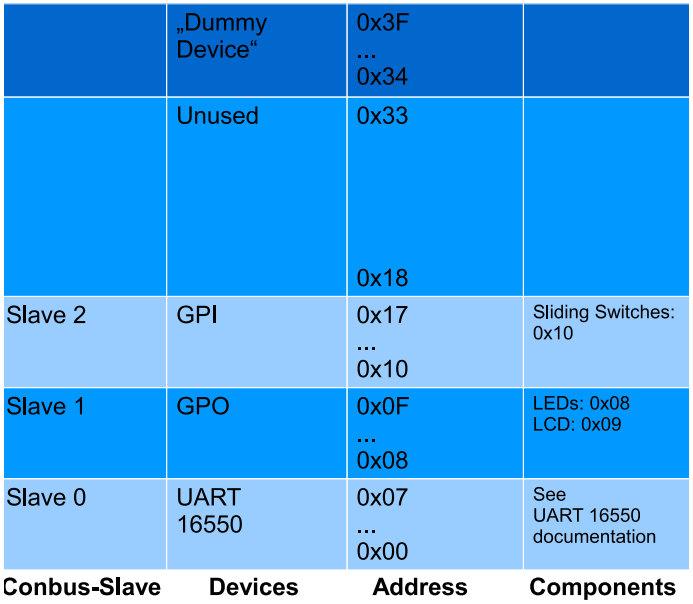

Figure 3.2: Memory Map of the IO-Bus

Figure 3.3 shows the memory map of the RAM-Bus. So far, only the SRAM is attached. But further components can be attached with no problems to the Wishbone-RAM-Bus.

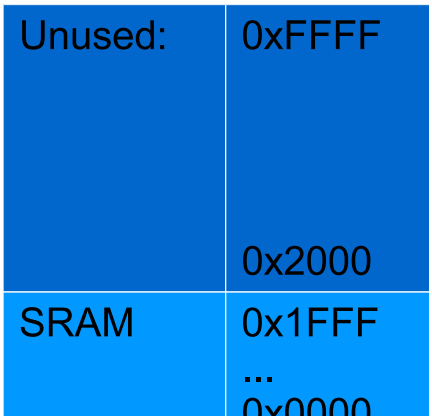

Figure 3.3: Memory Map of the RAM-Bus

Chapter 4

Firmware

The sample program provided with this project has been developed under WinAVR 20100110 and the Atmel AVR Studio 4.
The CPU can be treated exactly like any other AtMega103. See the ,,AVR_Softcore_main.h"-file how to link IO-addresses with variables. A similar scheme can be used to link RAM-Addresses with variables (_SFR_MEM8(adr) instead of _SFR_IO8(adr))

The sample program lets the 4 upper LEDs of the Starter-Kit blink whereas the lower 4 LEDs can be controlled with the sliding-switches on the board. The LCD displays ,,MARIO". The UART writes continuously ,,Hello World".

4.1 Creating the Bitstream

In order to pack a new program into the program memory of the AVR, the ,,data2mem"-tool is used which creates a new bitstream. All it needs is an existing bitstream (the one from the implementation of the SoC) and a .elf-file created by the C-compiler.

Under Microsoft Windows, firstly check the paths in the ,,gen_modified_Bitstream.bat"-file in the ,,data2mem" folder of this project. Adjust them to your system. Then, run this script. You will get the ,,avr_soc_top_fw.bit" which is the final bitstream to be programmed into the FPGA.

When a new firmware has to be programmed, simply run the ,,gen_modified_Bitstream.bat"-file and program the new bitstream into the FPGA.

,,data2mem" also works under Linux. Simply adjust the script to your system and it should work.

4.2 Simulation

The AVR core can also be simulated. You'll find a simple testbench along with this project. In order to simulate a firmware, the ,,data2mem"-tool can be used as well. Just like creating a bitstream described above, adjust the file ,,gen_Init_Mem.bat"-file to

your system and run it. You will get a ,,progMemInit.vhd"-file. Now, put this file in the ,,fpga"-folder of this project and replace the old one. Now, the SoC can be simulated and analzyed and the AVR will run the program code in the simulator.

Chapter 5

Peripherals

In principal, every peripheral providing a wishbone-interface can be connected to the AVR-SoC. In the example project, a UART-core along with general purpose input/output modules are attached to the AVR to show the functionality.

5.1 UART 16550

The UART 16550 core has also been taken from opencores.org (http://opencores.org/project,uart16550). It offers a wishbone-interface and can therefore very easily be accessed with the AVR and it's wishbone-translator. The UART is configured with the firmware, see the example firmware that comes along with this project.

5.2 General Purpose Output

This is a simple output-interface which can be accessed with the wishbone-interface. In the basic SoC, the 8 LEDs and the LCD of the Starter-Kit are connected to the AVR with this entity. The status of the outputs can also be read by the AVR, e.g. LEDS —= (1¡¡5) works.

5.3 General Purpose Input

This is a simple input-interface that can also be accessed by the AVR with the wishbone-bus. It just forwards the logic levels of the various input vectors to the AVR. In the SoC that comes along with this project, the 4 sliding switches of the Starter Kit are read using this entity.

5.4 Clocking and Reset

A single DCM generates the 30MHz clock of the AVR as well as the 180 degree phase shifted 30MHz clock. A pushbutton of the starter kit serves as an external reset-input.