async_syscon guide

async syscon Pack : synthesizable cores Technical Reference Manual

Written by:

John Clayton
Klugwhallah FPGA design team

October 25, 2017

Table of Contents

1 List of Acronymns

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

2 Introduction

This document provides a description of the interface signals, internal structure and registers present in the synthesizable cores within the VHDL package file named “async_syscon_pack.vhd” Some of the cores may be intended for use at lower levels of a design, in hierarchical fashion. All cores in the VHDL package are synthesizable and have been tested via simulation and in hardware using a Xilinx “ARTY” Artix 7 FPGA development board, among others.

3 Description of Cores

3.1 Background

The original Opencores project, “rs232_syscon” was created back in September, 2001. The usefulness of this core has not diminished during the intervening years, and it continues to be one of the most basic and trusted modules upon which my hardware debugging efforts depend. During the 16 years of the existence of this core, very little has changed in principle. However, due to my own circumstances and work demands, it was translated from Verilog to VHDL, about nine years ago, and the VHDL version has gotten all of the maintenance and updates since then. It has also been renamed as “async_syscon_pack.vhd”

The “async_syscon_pack.vhd” VHDL package consists of two VHDL entities. For those who are interested in using their own UART designs, or who have a need for parallel ASCII interface, there is an entity called “ascii_syscon” which implements the design without a UART. The asynchronous serial version, “async_syscon” instantiates the ascii_syscon module, and adds a “uart_sqclk” module for handling the serial communication. Most often, I use the async_syscon module along with an automatic Baud rate generator. (See separate Opencores project by that name). An example of how to instantiate the async_syscon along with an automatic Baud rate generator is given in this project.

3.2 Summary of Cores In Package

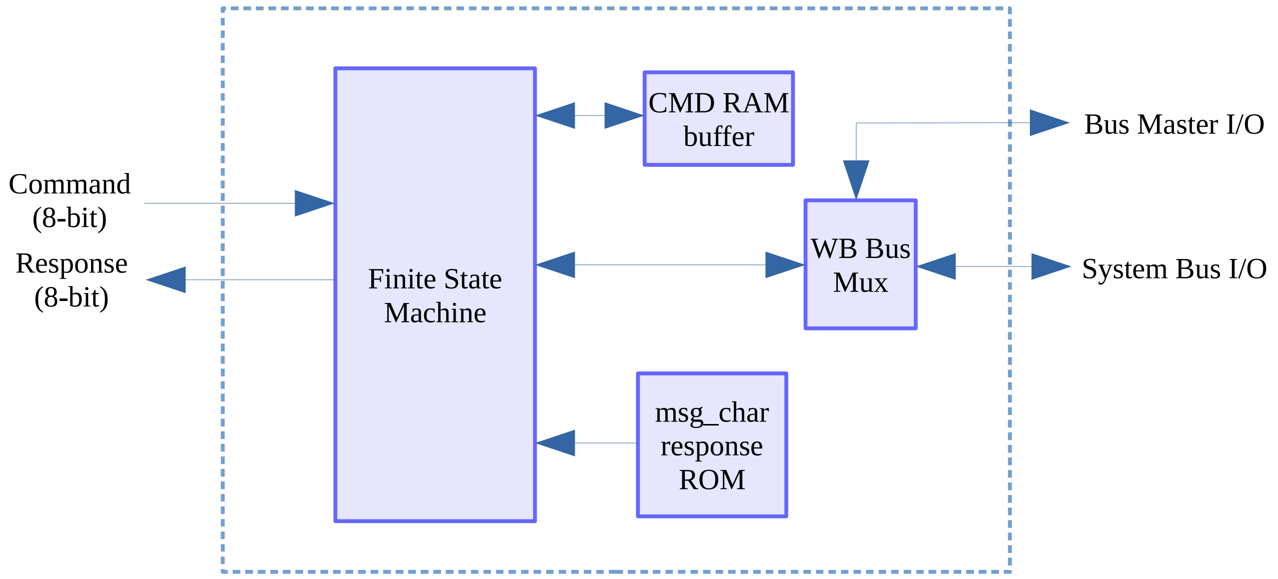
The cores present in the package file “async_syscon_pack.vhd” are shown in table Table 1. The ascii_syscon core is identical in function to the async_syscon core, except that the ascii_syscon lacks an actual UART. Therefore, ascii_syscon can be used with other interfaces if desired. In my case, I used it with a UDP/IP Ethernet based command interface.

Name	Description
ascii_syscon	ASCII based Wishbone bus controller
async_syscon	Asynchronous serial based Wishbone bus controller

Table 1: async_syscon_pack Cores

3.3 ascii_syscon

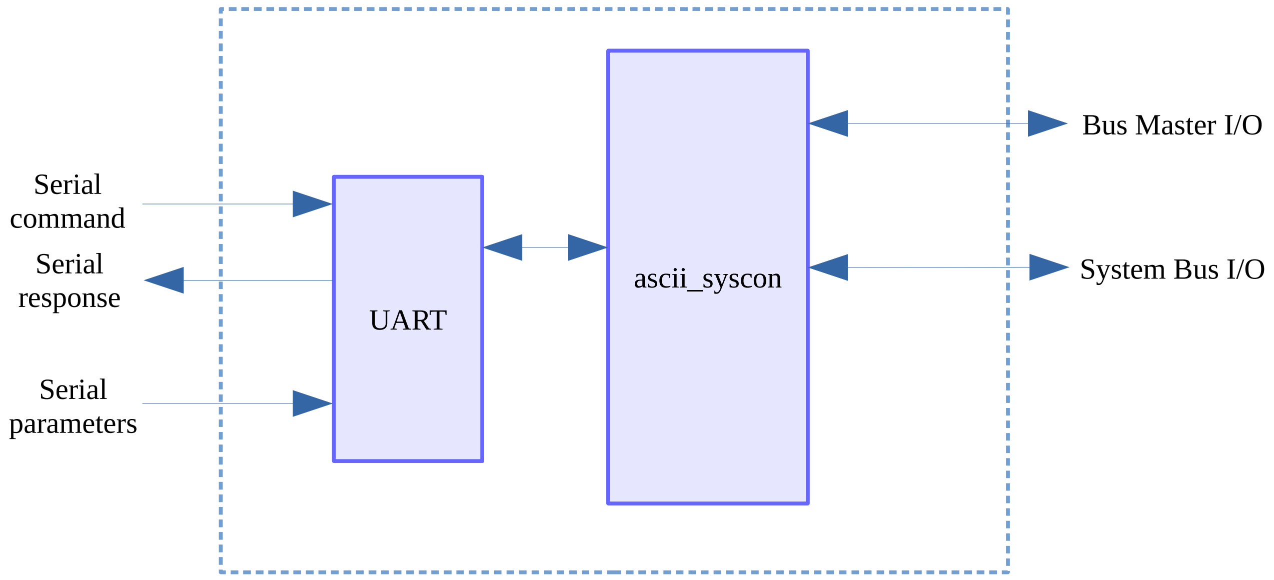
The ascii_syscon core implements the very simple set of commands and responses needed for exercising a Wishbone parallel bus.



The Bus Master I/O port is provided so that the ascii_syscon can be “inserted” between an existing bus master and the Wishbone bus. This allows, for example, the syscon to be able to run read and write cycles on a Wishbone bus that is normally under the control of a microcontroller. If there is no existing bus master, then those signals can be left unconnected and unused, thereby leaving the syscon as the bus master, so that the user can manually write to and read from the Wishbone bus. This capability is extraordinarily useful when testing designs within an FPGA, because it allows the user to change registers and memory contents at will, during troubleshooting and/or testing activities. Sometimes I use logic to halt the operation of the master during the manual accesses, and sometimes I use a bus arbiter, to coordinate the accesses. Also, the syscon has been used to load code and settings into RAM or FLASH memory, allowing setup and configuration of the overall device. A description of the command and response syntax is given later in this document.

3.4 async_syscon

This core is used much more often than the ascii_syscon, since it adds a UART to allow the commands and responses to be communicated serially. The block diagram of this core is:



The async_syscon module is what I like to call "fully parameterized," meaning that it includes settings to configure the operation of the module, which are evaluated at the time the design is synthesized. By adjusting the settings, the synthesis tools may end up optimizing out portions of the design, resulting in a more compact design. For this module, the parameters take the form of VHDL generics, which are described in the following paragraphs.

Parameter : ECHO_COMMANDS

When this parameter is set to anything non-zero, the module returns a copy or "echo" of each command character received. This is useful in many cases when using a serial terminal, such as PuTTY or hyperterminal to communicate with an async_syscon module.

Parameter : ADR_DIGITS

This parameter determines how many hexadecimal digits are used for address values. Address responses from the syscon contain exactly this many hexadecimal digits. When entering an address field as part of a command, if fewer digits than ADR_DIGITS are used, then the upper digits are padded with zeros. Similarly, if more digits than ADR_DIGITS are entered, then only the last ADR_DIGITS digits are taken for use by the syscon command processor.

Parameter : DAT_DIGITS

This parameter determines how many hexadecimal digits are used for data values. Data responses from the syscon contain exactly this many hexadecimal digits. When entering a data field as part of a command, if fewer digits than DAT_DIGITS are used, then the upper digits are padded with zeros. Similarly, if more digits than DAT_DIGITS are entered, then only the last DAT_DIGITS digits are taken for use by the syscon command processor.

Parameter : QTY_DIGITS

This parameter determines how many hexadecimal digits are used for quantity values. Read responses from the syscon many contain up to $2^{(4N)}$-1 data values, where N=QTY_DIGITS. When entering a quantity field as part of a command, if fewer digits than QTY_DIGITS are used, then the upper digits are padded with zeros. Similarly, if more digits than QTY_DIGITS are entered, then only the last QTY_DIGITS digits are taken for use by the syscon command processor.

Parameter : CMD_BUFFER_SIZE

This parameter determines the amount of RAM which is available for use in storing the command buffer. The command buffer consists of a single line of ASCII text, which is not evaluated until the user presses the “Enter” key in the terminal session. Actually, the Finite State Machine begins evaluation of the command line whenever the ENTER_CHAR (a constant) is encountered on the incoming command character bus, or whenever the command line buffer is full, whichever comes first. The ENTER_CHAR has been set to 0x0D, which is a carriage return character. Therefore, line feed characters (0x0A) are not strictly required, although they don’t harm anything when they are sent after the carriage return character.

Note that the backspace character, 0x08, will cause the previous command character to be dropped from the line buffer. If ECHO_COMMANDS is enabled, the backspace character’s echo will cause most terminal programs to move the cursor to the left by one space. However, the existing previous command character shown in the terminal is not erased. Simply overwrite the existing characters in the terminal, and proceed. Also, if too many backspaces are sent in, the command buffer will be empty, but the cursor in the terminal window will continue to move to the left, even passing to the left of the command prompt. I usually just hit “Enter” when I’ve done this, so that I get a new command prompt.

The line buffer for holding commands in implemented in distributed RAM, not block RAM. If that is a problem, please go ahead and modify the design to use a BRAM instead, if you wish. In most cases, the presence of a small distributed RAM does not consume an exorbitant amount of resources.

Parameter : WATCHDOG_VALUE

The Wishbone bus includes handshaking lines to acknowledge and terminate an open bus transfer cycle. This can present a hazard or difficulty if some elements in the design do not provide the “ack_i” at all, since the bus cycle can then extend indefinitely, preventing any further operation of the bus. To prevent this situation from becoming too annoying, a watchdog timer has been included in the syscon module. The WATCHDOG_VALUE parameter sets how many system clock cycles are going to occur before the syscon automatically terminates the bus cycle. When the cycle is terminated due to the watchdog timer expiring, an error message of ‘!’ is returned to the user.

Parameter : DISPLAY_FIELDS

This parameter determines how many hexadecimal data values are shown on each response line from the syscon. This setting has no effect on the command line, which can only hold the number of data values which can fit within the line buffer.

3.5 async_syscon serial settings

The asynchronous serial communication settings are configurable by input signals to the async_syscon unit. For the Baud rate, supply a squarewave of the desired rate to the module. Note that the baud_clk_i input signal does not need to be 4x, 8x or 16x the desired Baud rate, just 1x the desired rate. However, the frequency of the pulses provided on baud_clk_i **must** be less than 1/8 of the system clock frequency. In most modern systems, this is not an issue. Also, the duty cycle of the baud_clk_i input can be quite low, ranging all the way down to a narrow pulse just as wide as the system clock, if desired.

The parity settings are given in Table 2.

The baud_lock_i input is used to prevent any operation until the baud rate clock is stable and ready. Command characters can not be written into the syscon while baud_lock_i is low.

parity_i	Operation
“00”	No parity
“01”	Even parity
“10”	Odd parity
“11”	Odd parity

Table 2: async_syscon parity settings

3.6 async_syscon serial I/O

The cmd_i input is where the asynchronous serial command characters are sent into the syscon unit, and the asynchronous serial reply is transmitted out of the resp_o output.

The cmd_done_o output is not strictly related to the asynchronous serial I/O, but is was grouped along with cmd_i and resp_o because it just seems to make sense. At the completion of each command, when the next prompt is generated, a narrow pulse of one sys_clk duration is emitted on the cmd_done_o. One can use it for counting the number of completed commands, or for locking the bus arbiter until the given command is completed, or for whatever other purpose is desired.

3.7 Master bus I/O

These signals are fairly self explanatory, as they correspond to the normal Wishbone bus signals. The master_br_o signal indicates to the master that the syscon is requesting the use of the bus (br=bus request). The syscon waits for a positive handshake on master_bg_i (bg=bus grant) before proceeding to generate any bus cycles. If no bus master is present, simply connect master_br_o to master_bg_i. The syscon uses its watchdog timer when requesting the use of the bus from the master. If the watchdog timer expires without the master_bg_i handshake signal being given, the bus cycle is terminated, and the response “B!” is given in the terminal window.

3.8 System bus I/O

These are the regular Wishbone parallel bus signals. Note that the assertion of “err_i” can occur at any time (it does not need to be accompanied by ack_i). Asserting err_i during an active bus cycle causes the bus cycle to terminate, and the response ‘!’ is given in the terminal window, the same as if the watchdog timer had expired.

3.9 Command syntax

The command structure is quite terse and spartan in nature, this is for the sake of the logic itself. The command line buffer is small enough to be implemented without the use of dedicated BRAM memory blocks, and the menus and command responses were kept as small as possible. In most cases, the responses from the unit to the user consist of a "newline" and one or two visible characters, plus a new command prompt on its own separate line. The command structure consists of the following commands and responses:

Command Syntax	Purpose

`w aaaa dddd dddd dddd...`	`Write data items "dddd" starting at address "aaaa" using sequential addresses.`
	`(If the data field is missing, nothing is done).`

w0 aaaa dddd dddd dddd...	Write data items "dddd" at address "aaaa" without incrementing the address. (If the data field is missing, nothing is done).
f aaaa dddd xx	"Fill": Write data "dddd" starting at address "aaaa" perform this "xx" times at sequential addresses. (The quantity field is optional, default is 1).
f0 aaaa dddd xx	"Fill": Write data "dddd" starting at address "aaaa" perform this "xx" times at the same address. (The quantity field is optional, default is 1).
r aaaa xx	Read data starting from address "aaaa." Perform this "xx" times at sequential addresses. (The quantity field is optional, default is 1).
r0 aaaa xx	Read data from address "aaaa." Perform this "xx" times, using the same address. (The quantity field is optional, default is 1).
i	Send a reset pulse to the system. (initialize).
<COMMENT_CHAR>	"Single Line" type Comment token. Characters after the token are ignored until <ENTER>. This enables applications which send files to the unit to include comments for display and as an aid to understanding. The comment token is a constant, change it to be whatever makes sense!

Response from async_syscon	Meaning
OK	Command received and performed. No errors.
?	Command buffer full, without receiving "enter."
C?	Command not recognized.
A?	Address field syntax error.
D?	Data field syntax error.
Q?	Quantity field syntax error.
!	No "ack_i", or else "err_i" received from bus.
B!	No "bg_i" received from master.

3.10 Conclusion

This core is meant to be a helpful addition to any project where a human may wish to interact with an FPGA by accessing the Wishbone bus directly, using human (nerd) readable hexadecimal characters. It is supposed to help keep project checkout and operation fun and easy. Please enjoy this code, and I hope it is useful to you in your work.