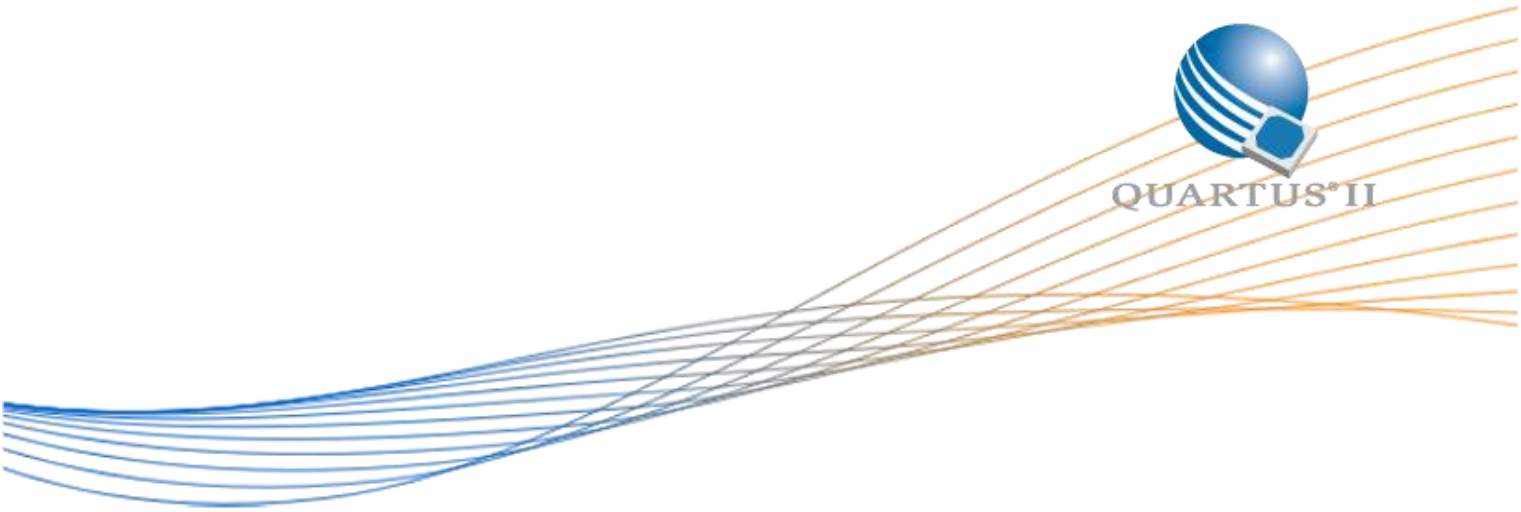

SPI Slave to Avalon Master Bridge on CVSX board design example

Date: April 2016

Revision: 1.0

Contents

Overview

This design example demonstrates how to use SPI slave to Avalon Master bridges, which provides a connection between host system and remote system for SPI transactions. This design example is running on ACDS 16.0 which is updated from previous design which is running on ACDS 9.1.

Tool Requirements

- Altera Cyclone V SoC Development Kit Rev D
- Quartus II software version 16.0
- Nios II Embedded Design Suite (EDS) 16.0
- Mini USB cable for programming the device

Hardware Design Specifications

Host system :

- Nios II/f
- Onchip memory : 156K
- System timer
- System ID
- JTAG UART
- SPI Core

Remote System

- SPI Slave to Avalon Master Bridge
- Onchip memory : 4K

Instruction for running the software

Before running the software, connect the USB blaster cable from PC to the on board connector which shown in Figure 1.

Figure 1: USB Blaster II connection location

1. Open the Nios II 16.0 Command shell and change the directory to project directory. Master image folder provided in the project directory contain the .sof and .elf file which is ready to be downloaded to the board

2. Program the configuration file (.sof) into the board by typing
   nios2-configure-sof –d 2 spi_bridge.sof
   The following messages are displayed when successfully programming the device

```
Searching for SOF file:
in .
  spi_bridge.sof

Info: ************************************************************************
Info: Running Quartus Prime Programmer
Info: Command: quartus_pgm --no_banner --mode=jtag -o p;./spi_bridge.sof@2
Info (213046): Using programming cable "USB-BlasterII on PG-TW00I-520.altera.priv.altera.com [USB-1]"
Info (213011): Using programming file ./spi_bridge.sof with checksum 0x0475F693 for device 5CSXFC6D6F31@2
Info (209060): Started Programmer operation at Tue Apr 26 10:03:12 2016
Info (209016): Configuring device index 2
Info (209017): Device 2 contains JTAG ID code 0x02D020DD
Info (209007): Configuration succeeded -- 1 device(s) configured
Info (209011): Successfully performed operation(s)
Info (209061): Ended Programmer operation at Tue Apr 26 10:03:16 2016
Info: Quartus Prime Programmer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 440 megabytes
    Info: Processing ended: Tue Apr 26 10:03:16 2016
    Info: Elapsed time: 00:00:06
    Info: Total CPU time (on all processors): 00:00:02
```

Figure 2: Successful programming the device

3. Download the .elf file and invoke the terminal by typing
   nios2-download -g spi_bridge_test.elf && nios2-terminal
   The software will perform write and read back a block of data to the on chip memory via the SPI slave to Avalon Master Bridge. The following messages are shown on the nios2-terminal for successful execution.

```
Creating random test data ...

Writing data to onchip memory ...
Write transaction successful

Reading data from onchip memory ...

Comparing data ...
Compare data completes error free
```

Figure 3: Messages shown for successful transaction

Note: Alternately, the software project can be imported into Nios II software Build Tools for Eclipse to run/debug the software.

Hardware regeneration

There are two Qsys system cpu_spi_core and spi_bridge_system in this design example. Hence, if you wish to recompile the hardware design in Quartus II, you need to regenerate two Qsys system.

By default, only one system.h file is created even if two Qsys systems exist in your project. To create two system.h files for two Qsys systems, you will need to manually create the second system.h file. In this example, spi_bridge_system.h is created for spi_bridge_system via the sopc-create-header-files command in the create-this-bsp script. This command will need an input parameter, which is the second Qsys system's .sopcinfo file and output paramter, xxx_system.h. For details, type sopc-create-header-files --help in the Nios II Command Shell.

Software regeneration

If you wish to rebuild the software, go into directory <project directory>/software example/app/spi_bridge_test and run the create-this-app script. A new .elf file will be generated.

If you wish to port the software in the alt_spi_to_avalon_bridge to another CPU, you will need to replace function in the spi command files with the driver function.

Software API

This section discusses the software API for the SPI slave to Avalon Master Bridge. When an external processor is used to send or receive SPI data to a SPI Slave to Avalon Master Bridge, the external processor can use the Altera supplied API to convert the SPI transaction to the protocol used by the bridge. For more information about the protocol, refer to embedded IP user guide here

The source code is located in the alt_spi_to_avalon_bridge folder. The files included are

- transaction_to_packet.c
- transaction_to_packet.h
- packet_to_byte.c
- packet_to_byte.h
- byte_to_core.c
- byte_to_core.h
- spi_command.c
- spi_command.h

Note: There are two type of memory allocation in the source code. Define DYNAMIC_MEMORY_ALLOC for each file if you are going to use the heap memory or STATIC_MEMORY_ALLOC for static memory. Memory will be allocated for send data as well as the expected response data. If you use STATIC_MEMORY_ALLOC, take note that the static memory allocated is sufficient for up to 1KB of transaction only. Change the buffer size defines (TRANSACTION_BUFFER_LENGTH, PACKET_BUFFER_LENGTH, BYTE_BUFFER_LENGTH) if you wish to change the static memory size.

Transaction to packet

```
unsigned char transaction_channel_write(unsigned int address, unsigned
int burst_length,unsigned char* data_buffer, unsigned char sequential)
```

This function will write a block of data to an address according to the following parameters:

- address : 32-bit address of a slave component connected to the bridge
- burst_length : length of data in bytes
- data_buffer : pointer the the data to be written for the write transactions
- sequential : flag to increment[1]/non-increment address[0]

If sequential is set to 1, 4 bytes of data will be written to the given address and the address will be incremented for the next 4 bytes of address. If sequential is set to 0, 4 bytes of data will be written to the given address and the address will not be incremented for the next 4 bytes. This means that the next 4 bytes will still be written to the same given address, replacing the first 4 bytes.

This function will append a transaction header to the block of data. If the return number of bytes is correct, this function returns TRUE.

```
unsigned char transaction_channel_read(unsigned int address, unsigned
int burst_length,unsigned char* data_buffer, unsigned char sequential)
```

The algorithm of transaction_channel_read() is similar to transaction_channel_write() except that it performs read operation and this function always return TRUE.

Packet to byte

```
unsigned char packet_to_byte_convert (unsigned int send_length,
unsigned char* send_data,unsigned int response_length, unsigned char*
response_data)
```

This function includes the following parameters:

- send length : size of send data
- send data : pointer to the send data
- response_length : size of response data
- response_data : pointer to the response data

This function will insert these special characters into send data, SOP(0x7a), EOP (0x7b), CHANNEL (0x7c) and CHANNEL ID (0x00). The special characters will be inserted with the following sequence :
SOP | CHANNEL | CH ID | .... Data ..... | EOP | Last byte of data |

If any byte of send data is a special character, this function will insert an ESC (0x7d), followed by the data XORed with 0x20.

Similarly, when this function finds any special characters in the response data, it will drop the special characters from the response data.

Byte to core

```
unsigned char byte_to_core_convert (unsigned int send_length, unsigned
char* send_data,unsigned int response_length, unsigned char*
response_data)
```

This function includes the following parameters:

- send length : size of send data
- send data : pointer to the send data
- response_length : size of response data
- response_data : pointer to the response data

This function will check for special characters, IDLE(0x4a) and ESC(0x4d).If a data byte is a special character, this function will insert an ESC (Escape) followed by the data XORed with 0x20.

Similarly, when this function finds any special characters in the response data, it will drop the special characters from the response data.

```
int spi_command(unsigned int base, unsigned int slave, unsigned int
write_length, const unsigned char * write_data, unsigned int
read_length, unsigned char * read_data, unsigned int flags)
```

This function is a mirror of the alt_avalon_spi_command(), which is the SPI driver of the SPI core. If you are using a different SPI driver, do replace this function with your own SPI driver function.

For more info on alt_avalon_spi_command(), refer to "SPI Core" chapter here

Revision History

Revision	Description
1.0	Updated design example to CVSX