Netherlands Institute for Radio Astronomy

I^{2}C SMBus Module Description

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Eric Kooistra	ASTRON	
Controle / Checked: André Gunst	ASTRON	
Goedkeuring / Approval: André Gunst	ASTRON	
Autorisatie / Authorisation: **Handtekening / Signature** André Gunst	ASTRON	

UniBoard

Doc.nr.:	ASTRON-RP-329
Rev.:	2.0
Date:	29 Feb 2012
Class.:	Public

Distribution list:

Group:	Others:
André Gunst Eric Kooistra Daniel van der Schuur Harm-Jan Pepping	Gijs Schoonderbeek

Document history:

Revision	Date	Author	Modification / Change
0.1	04-09-2009	Eric Kooistra	Creation from LOFAR RSP design documents
0.2	04-09-2009	Eric Kooistra	Added I^2C slave module from LOFAR RCU Minor edits.
1.0	08-09-2009	Roelof Kiers	Final v1.0
2.0	29-02-2012	Eric Kooistra	Added opcode OP_RD_SDA Added I2C commander for ADU I2C control Added I2C application examples used on UniBoard

Table of contents:

References:

[1] "RSP Firmware Design Description"", LOFAR-ASTRON-SDD-018, Rev. 4.0, 2009
[2] "RSP-MAC Interface Description", LOFAR-ASTRON-ICD-002, Rev. 7.0, 2009
[3] https://svn.astron.nl/UniBoard_FP7/UniBoard/ = $UNB
[4] http://www.opencores.org, "I^{2}C-Master Core Specification", Richard Herveille, rev 0.9, Jul 2003
[5] http://www.freemodelfoundry.com, HDL open source code
[6] http://www.smbus.org, System Management Bus Specification, Rev. 2.0
[7] http://www.nxp.com, Philips I^{2}C standard, I^{2}C slave IO expander chips
[8] http://www.maxim-ic.com, I^{2}C slave sensors chips

Terminology:

ADU	Analogue Digital Unit (APERTIF)
AVS	Avalon bus Slave (Altera bus)
HDL	Hardware Description Language (typically VHDL or Verilog)
I^{2}C	Inter-IC Interface
IC	Integrated Circuit
I/O	Input/Output
MISO	Master In Slave Out
MM	Memory-Mapped
MMS	Memory-Mapped Slave
MOSI	Master Out Slave In
RCU	Receiver Unit (LOFAR)
RSP	Remote Station Processing board (LOFAR)
RTL	Register Transfer Level
SCL	I^{2}C Clock line
SDA	I^{2}C Data line
SENS	Sensors module
SMBUS	System Management Bus
SOPC	System on a Programmable Chip (Altera tool)
T,V	Temperature, Voltage sensor
UNB	UniBoard
VHDL	Very High Speed IC HDL

1 Introduction

The I^{2}C SMBus controller module was originally designed for the LOFAR Remote Processing Board (RSP) firmware [1], [2] and has also been reused for the EMBRACE Center Board (ECB) firmware. For the RadioNet FP7 UniBoard development it is available again and described separately in this design document.

The I^{2}C SMBus controller is described in chapter 2. It consists of:

1. I^{2}C master core obtained from Open Cores [4] that drives the I^{2}C bus [7]
2. SMBus protocol processor that follows the protocols defined in [6]
3. Control interface

The SMBus controller can be used in several ways that differ in the amount of software or hardware control. These applications are described in chapter 3 by means of existing examples in the UniBoard firmware [3]:

1. One fixed protocol list in ROM → unb_sens in design unb_common
2. One programmable protocol list in RAM under software control → i2c_master in module Lofar/i2c
3. A set of protocol lists in RAM that can be issued as commands → i2c_commander in module Lofar/i2c

2 SMBus controller design

2.1 Architecture

The I^2C controller consists of an I^2C master core [4] that provides a byte level interface to the physical I^2C interface and a SMBus [6] protocol processor that allows access based on a set of opcodes. A sequence of opcodes is called a protocol and defines for example an I^2C write byte access. The SMBus protocol processor can run a protocol and report its result. The CTRL block provides a memory access interface to the protocol list and protocol result registers.

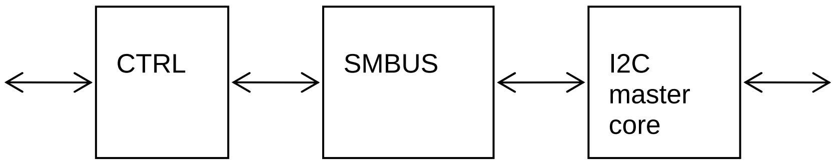

Figure 1: I^2C module block diagram

Appendix 4.1 lists the VHDL source code for the I^2C module and some more useful files for test benches and I^2C slave models. Appendix 4.2 reports issues that were encountered with the I2C master core.

2.1.1 Opcodes

An I^2C access is defined by a sequence of basic operations called opcodes. Table 1 lists the I^2C access opcodes that are sufficient to define any standard I^2C accesses.

OPCODE	**Description**
OP_LD_ADR	Load address register
OP_LD_CNT	Load count register
OP_WR_CNT	I^2C write count register
OP_WR_ADR_WR	I^2C start and write address for write
OP_WR_ADR_RD	I^2C start and write address for read
OP_WR_DAT	I^2C write byte of data
OP_WR_BLOCK	I^2C write block of count data bytes
OP_RD_ACK	I^2C read byte data and acknowledge
OP_RD_NACK	I^2C read byte data and do not acknowledge
OP_RD_BLOCK	I^2C read block of count data bytes
OP_STOP	I^2C stop

Table 1: Opcodes for an I^2C access

The opcodes work like instructions for a dedicated microprocessor. Given such a microprocessor it is possible to add some more opcodes for other functions. This is useful to control the flow of I^2C accesses. Table 2 lists the control opcodes.

OPCODE	Description
OP_IDLE	Idle
OP_END	Signal end of list of protocols
OP_LD_TIMEOUT	Load timeout value
OP_WAIT	Wait for timeout time units
OP_RD_SDA	Sample SDA line

Table 2: Opcodes for flow control and monitoring

2.1.2 Protocols

The network layer of the SMBus standard [6] specifies the protocols for several common I^2C accesses. For example the write quick protocol can be implemented by applying opcodes: OP_LD_ADR, OP_WR_ADR_WR, OP_STOP. In addition some custom protocols are supported. Each protocol returns a status and the read data (in case of an I^2C read access). For the complete list of protocols that is supported by the SMBus protocol processor see section 2.2.

2.2 Interface

The protocol list registers contain the protocols that need to be executed by the SMBus protocol processor in order to do the appropriate I^2C accesses on an I^2C bus. The I^2C access results can be read back from the protocol result register.

BIT / O/S	7	6	5	4	3	2	1	0
0	PROTOCOL_0[7:0]							
1	PROTOCOL_1[7:0]							
2								
…								

Table 3: Protocol list register

BIT / O/S	7	6	5	4	3	2	1	0
0	RESULT_0[7:0]							
1	RESULT_1[7:0]							
2								
…								

Table 4: Protocol results register

2.2.1 Protocol request message

To execute a protocol the dedicated protocol processor needs to get a request message containing the protocol ID followed by the appropriate parameters, see Table 5.

Protocol name	Message fields (bytes)	
	ID	Parameters
PROTOCOL_WRITE_QUICK	02	ADDR
PROTOCOL_READ_QUICK	03	ADDR
PROTOCOL_SEND_BYTE	04	ADDR, DATA
PROTOCOL_RECEIVE_BYTE	05	ADDR
PROTOCOL_WRITE_BYTE	06	ADDR, CMD, DATA
PROTOCOL_READ_BYTE	07	ADDR, CMD
PROTOCOL_WRITE_WORD	08	ADDR, CMD, DATA, DATA
PROTOCOL_READ_WORD	09	ADDR, CMD
PROTOCOL_WRITE_BLOCK	0A	ADDR, CMD, CNT, DATA[1..CNT]
PROTOCOL_READ_BLOCK	0B	ADDR, CMD, CNT
PROTOCOL_PROCESS_CALL	0C	ADDR, CMD, DATA, DATA, ADDR, CMD
PROTOCOL_C_WRITE_BLOCK_NO_CNT	0D	ADDR, CMD, CNT, DATA[1..CNT]
PROTOCOL_C_READ_BLOCK_NO_CNT	0E	ADDR, CMD, CNT
PROTOCOL_C_SEND_BLOCK	0F	ADDR, CNT, DATA[1..CNT]
PROTOCOL_C_RECEIVE_BLOCK	10	ADDR, CNT
PROTOCOL_C_NOP	11	-
PROTOCOL_C_WAIT	12	TIMEOUT[0..3]
PROTOCOL_C_END	13	-
PROTOCOL_C_SAMPLE_SDA	14	TIMEOUT[0..3]

Table 5: Protocol request message format.

Where:

- ID	= Protocol ID byte.
- ADDR	= Slave I^{2}C address byte.
- CMD	= Command byte.
- CNT	= Count of number of data bytes to write or to read.
- DATA	= Data byte.
- TIMEOUT	= Timeout value, 4 bytes, LSByte first. For example TIMEOUT[0:3] = h'00 01 00 00 = 256 system clock cycles. Only values < 2^{28} are supported.

Protocols 0x02 to 0x0C follow the SMBus standard [6]. The SMBus protocols are not sufficient to access all kinds of I^{2}C slaves. Furthermore it is useful to define some protocols for protocol flow control. Therefore some custom protocols are added. For example to access the LOFAR Receiver Unit (RCU) the protocols PROTOCOL_C_SEND_BLOCK and PROTOCOL_C_RECEIVE_BLOCK need to be used.

Brief description of the protocols: The protocols for flow control do not access the I^{2}C interface, the others do. The SEND/RECEIVE protocols define a plain data access while the WRITE/READ protocols first write a command byte and then perform the data access. For the READ protocols first having to write the command byte causes that the required I^{2}C restart is performed to subsequently receive the actual data. The WRITE/READ_BLOCK_NO_CNT protocols are similar to the SMBus WRITE/READ_BLOCK protocols except that the count value is not written to the I^{2}C interface. For CNT=1 the WRITE/READ_BLOCK_NO_CNT is equivalent to SMBus WRITE/READ_BYTE and for CNT=2 the WRITE/READ_BLOCK_NO_CNT is equivalent to SMBus WRITE/READ_WORD. The SEND/RECEIVE_BLOCK protocols are similar to the SMBus SEND/RECEIVE_BYTE/WORD protocols, but allow an arbitrary number of bytes given by count. The SEND/RECEIVE_BLOCK protocols are also similar to the WRITE/READ_BLOCK_NO_CNT protocols, except that they do not write a command. The PROTOCOL_C_WAIT protocol allows postponing subsequent protocol handling. The timeout value is given by the 4 timeout bytes (LSByte first) and counts system clock cycles. An unknown protocol is detected and executed as PROTOCOL_C_END.

The PROTOCOL_C_SAMPLE_SDA samples the SDA line after a timeout. The timeout must be sufficient to ensure that the SDA has been pulled up, so typically use a timeout of at least half an SCL period. The result byte reports NOT(SDA), so for the correct pull up the result byte indicates NOT(‘H’) = 0 is OK.

2.2.2 Protocol response message

The protocol processor returns a protocol result byte for each protocol. In case the protocol concerned an I^{2}C read access, then the result byte is preceded by the read data bytes. The number of read data bytes depends on the protocol type. If the read access failed somehow, then the read data bytes are undefined. Often undefined read bytes show a 0xFF due to the SDA pull up.

The result byte == 0 when the protocol executed OK and <> 0 otherwise. The result byte defaults to OK except for an I^{2}C write. For an I^{2}C write the result becomes not OK in case a written byte was not acknowledged by the slave, this may occur for the write of the slave address or of the data.

2.2.3 Sequence of protocol messages

The protocol processor can play one or more protocols from a list of request messages. The corresponding response messages are returned in a response list. The PROTOCOL_C_END protocol is useful to signal that the end of a list was reached, so that all protocols in the list have been executed.

Note that by means of the PROTOCOL_C_WAIT timer it is possible to delay (parts of) the I^{2}C access execution of the protocol list. This can be useful to fine tune the synchronization between access to the TTD on ECB and the BF chips on the HEX boards. The time unit of the PROTOCOL_C_WAIT timer is one period of the system clock.

Appendix 4.3 gives an example on how to access an I^{2}C slave using the module interface.

3 Applications

3.1 Fixed protocol list in ROM

Figure 2 shows the block diagram of mms_unb_sens. The mms_unb_sens provides a MM register and MM bus interface to the unb_sens.

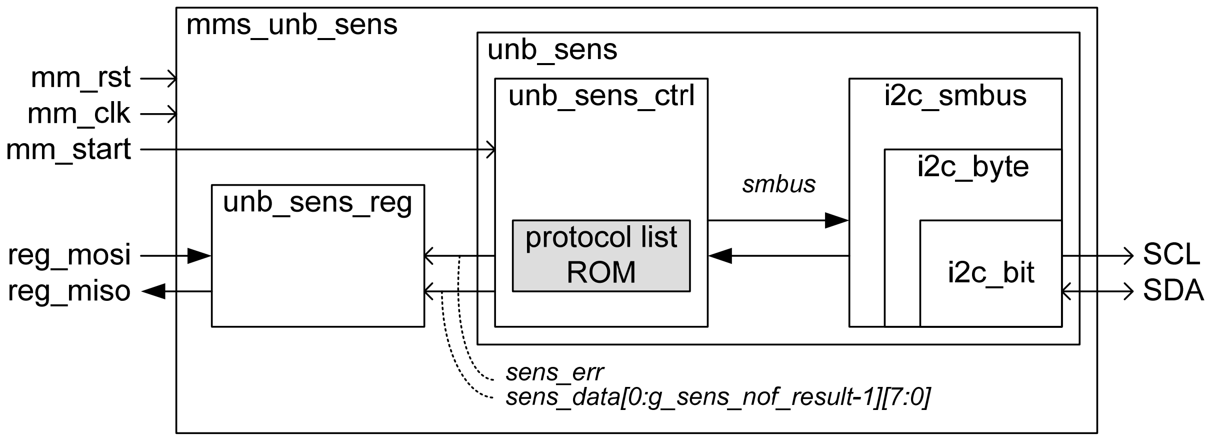

Figure 2: Block diagram of mms_unb_sens

The application functionality of mms_unb_sens is determined by the constant protocol list in unb_sens_ctrl. At a mm_start pulse the protocol list is executed by the i2c_smbus to access the I2C slave peripherals via the SCL and SDA lines. The read I2C bytes are collected in sens_data[] array of bytes. The I2C acknowledge signalling result is collected into sens_err. The sens_data bytes and the sens_err value can be read via the reg_mosi and reg_miso MM bus control signals as shown in Table 6.

```
   31             24 23             16 15              8 7              0  wi
  |-----------------|-----------------|-----------------|-----------------|
  |                xxx                      fpga_temp   = sens_data[0][7:0]|  0
  |-----------------------------------------------------------------------|
  |                xxx                      eth_temp    = sens_data[1][7:0]|  1
  |-----------------------------------------------------------------------|
  |                xxx                hot_swap_v_sense  = sens_data[2][7:0]|  2
  |-----------------------------------------------------------------------|
  |                xxx                hot_swap_v_source = sens_data[3][7:0]|  3
  |-----------------------------------------------------------------------|
  |                xxx                                         sens_err[0]|  4
  |-----------------------------------------------------------------------|
```

Table 6: MM register map defined in unb_sens_reg

The fpga_temp and eth_temp are in degrees (two's complement) and are measured by MAX1618 devices. The hot swap voltages are measured by an LTC4260 hot swap control device. The FPGA temperature can be measured via all UniBoard FPGAs. The eth_temp and the voltages can only be obtained via back node 3 (BN3). The protocol list in ROM is fixed and suits BN3, therefore the sens_err indication is only applicable to BN3. For the other nodes on UniBoard only the fpga_temp value is valid and the rest of the MM register sens_data and sens_err must be ignored.

The mms_unb_sens.vhd can be connected outside to an SOPC Builder system by means of the general purpose MM slave port avs_common_mm.vhd, which provides the MM bus signals as conduit wires..

3.2 Programmable protocol list

Figure 3 shows the block diagram of avs_i2c_master.vhd. The avs_i2c_master.vhd provides a MM slave peripheral for general I2C protocol list control that can be used within an SOPC builder system using the corresponding avs_i2c_master_hw.tcl hardware description file for SOPC Builder.

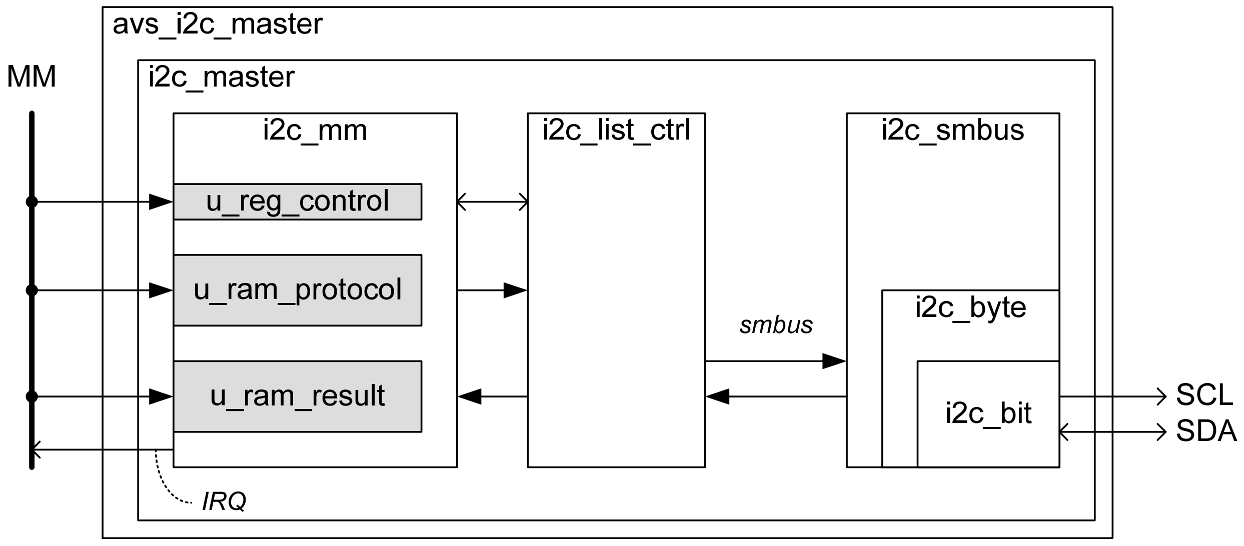

Figure 3: Block diagram of avs_i2c_master.vhd

The avs_i2c_master.vhd can be controlled in software by means of the functions provided in avs_i2c_master.h and unb_sensors.h as listed in Table 14.

Instead of using avs_i2c_master.vhd the i2c_master.vhd could also be connected outside to an SOPC Builder system by means of the general purpose MM slave port avs_common_mm.vhd. The test bench tb_i2c_master.vhd shows how the i2c_master.vhd is used.

The UniBoard ctrl_unb_common.vhd now contains mms_unb_sens, so the avs_i2c_master_hw.tcl within SOPC Builder is no longer used.

3.3 Programmable commander using a set of protocol lists

To not burden the user with the protocol list contents it is useful to be able to issue a protocol list as a command that select one out of a set of protocol lists. This is what the i2c_commander.vhd provides. The I2C commander can activate one out of a list of protocols. A difference between the i2c_commander and mms_unb_sens is that the i2c_commander offers multiple protocol lists that can be issued as commands. Optionally the protocol lists and result buffer can still accessible to the user via the MM bus, so then the i2c_commander.vhd still offers the same freedom of control as the avs_i2c_master.vhd and the i2c_master.vhd. Figure 4 shows the block diagram of i2c_commander.vhd.

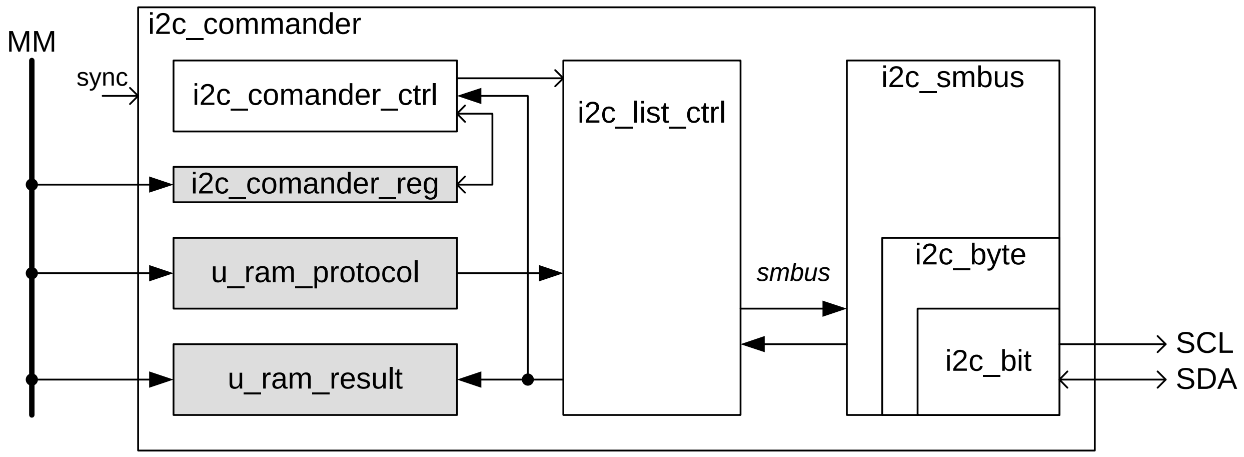

Figure 4: Block diagram of the i2c_commander.vhd

With Uniboard the I2C commander is used to control the I2C bus to ADU. The commander register settings for that are defined in i2c_commander_adu_pkg.vhd. There is also an i2c_commander_unb_pkg.vhd that provides similar functionality as mms_unb_sens, but that is not used. The I2C commander register map with up to 16 protocol lists for I2C commander control of ADU is shown in Table 7. The contents of the protocol RAM and the protocol offsets and the expected results are defined in the i2c_dev_adu_pkg.vhd. For more information see appendix 4.1.8 and 4.1.9.

The ADU I2C commander can execute one out of g_i2c_cmdr.nof_protocols = 16 protocol lists. Each protocol list can contain multiple I2C accesses. The protocol word index in Table 7 indicates the protocol list that will become active. The protocol lists are stored sequentially in u_protocol_ram. The protocol_offset array in Table 7 contains the offsets to the start of each protocol list. The protocol result is kept in u_result_ram, but it is also processed by the I2C commander. For each protocol result byte the I2C commander uses the active word from the result_expected array in Table 7 to know whether the result byte should be 0 (as for protocol control and write SMBUS protocol identifiers) or whether it contains read data (as read SMBUS protocol identifiers). For the bits in the result_expected word that are '0' the result byte must be 0 and for bits that are '1' the result byte gets stored in the result_data array in Table 7. If the result control byte is not 0 then the result_error_cnt in Table 7 gets incremented. The protocol_status in Table 7 reports when the protocol list has finished. The protocol_status can be idle (= 0), pending (= 1), busy (= 2) or done (= 3). The protocol list gets activated by the write access to the corresponding index address in the I2C commander register. Actual execution of the protocol list may be postponed until an external sync occurs. Default the sync input shown in Figure 4 is ‘1’ so the pending state is left immediately. Using the sync is useful if the I2C accesses must be synchronized among multiple nodes. Each protocol can have multiple I2C accesses. The maximum number of I2C read data bytes per protocol list is determined by g_i2c_cmdr.nof_result_data_max = 2 in in Table 7. In case a protocol list has no I2C read accesses then the all expected result bytes are 0, so then the size of the protocol list is only limited by the size of u_protocol_ram.

```
   31             24 23             16 15              8 7               0  wi
  |-----------------|-----------------|-----------------|-----------------|
  |         xxx                             write access issues protocol 0|  0
  |-----------------------------------------------------------------------|
  |         xxx                             write access issues protocol 1|  1
  |-----------------------------------------------------------------------|
  |         xxx                             write access issues protocol 2|  2
  |-----------------------------------------------------------------------|
  |                                  ...                                  | ..
  |-----------------------------------------------------------------------|
  |         xxx                            write access issues protocol 15| 15
  |-----------------------------------------------------------------------|
  |         xxx                      protocol_offset[protocol_adr_w-1:0] 0| 16
  |-----------------------------------------------------------------------|
  |         xxx                      protocol_offset[protocol_adr_w-1:0] 1| 17
  |-----------------------------------------------------------------------|
  |         xxx                      protocol_offset[protocol_adr_w-1:0] 2| 18
  |-----------------------------------------------------------------------|
  |                                  ...                                  | ..
  |-----------------------------------------------------------------------|
  |         xxx                     protocol_offset[protocol_adr_w-1:0] 15| 31
  |-----------------------------------------------------------------------|
  |         xxx                                    result_expected[31:0] 0| 32
  |-----------------------------------------------------------------------|
  |         xxx                                    result_expected[31:0] 1| 33
  |-----------------------------------------------------------------------|
  |                                  ...                                  | ..
  |-----------------------------------------------------------------------|
  |         xxx                                   result_expected[31:0] 15| 47
  |-----------------------------------------------------------------------|
  |         xxx                                      protocol_status[31:0]| 48
  |-----------------------------------------------------------------------|
  |         xxx                                     result_error_cnt[31:0]| 49
  |-----------------------------------------------------------------------|
  |         xxx                                         result_data[7:0] 0| 50
  |-----------------------------------------------------------------------|
  |         xxx                                         result_data[7:0] 0| 51
  |-----------------------------------------------------------------------|
```

Table 7: MM register map for i2c_commander_reg.vhd defined in i2c_commander_adu_pkg.vhd

To allow for dynamically programming of the protocol lists connect the u_ram_protocol MOSI/MISO to the MM bus. In this way it is possible to try many different protocols e.g. with different write data bytes. For normal operation typically a few fixed protocol lists are sufficient, so then u_ram_protocol MOSI/MISO can be left unconnected, which effectively makes u_protocol_ram a ROM.

Default g_use_result_ram = FALSE is fine because most result status info and data are accessible via the commander result_error_cnt and result_data array registers. Using TRUE allows for detailed debugging in case something goes wrong on the I2C bus. The result_expected array contains one mask word per protocol list. If the protocol list is long then this word is reused periodically. Hence if the protocol list contains multiple reads then these must occur within one mask word or they have to occur periodically. This can be achieved by inserting SMBUS_C_NOP in the protocol list.

4 Appendix

4.1 Firmware

To speed up simulation the I2C test benches and components use a generic g_sim or input cs_sim to support setting a faster I2C bit rate for simulation than the actual 50 kbps I2C bit rate defined in the i2c_pkg.vhd for the target hardware.

4.1.1 Project files

File	Description
unb_common/build/synth/quartus/ sopc_unb_common_sim/unb_common.mpf	Modelsim project file to compile unb_common_lib
unb_common/build/synth/quartus/unb_common.qip	Quartus list of IP source files
i2c/build/sim/modelsim/i2c.mpf	Modelsim project file to compile i2c_lib
i2c/build/synth/quartus/i2c.qip	Quartus list of IP source files

Table 8: Project files

4.1.2 I2C master core

The I2C master core is described in i2c/doc/I2C_specs.doc (pdf) from [4] and contains the VHDL files that are listed in Table 9.

File	Description
i2c/src/vhdl/i2c_bit.vhd	I2C master core from [4]
i2c/src/vhdl/i2c_byte.vhd	I2C master core from [4]

Table 9: I2C master core

4.1.3 I2C SMBus controller

File	Description
i2c/src/vhdl/i2c_list_ctrl.vhd	General protocol list controller
i2c/src/vhdl/i2c_smbus(pkg).vhd	Opcodes and protocols
i2c/src/vhdl/i2c_smbus.vhd	Opcodes processor

Table 10: I2C SMBus controller

4.1.4 I2C slave definitions

File	Description
i2c/src/vhdl/i2c_dev_max1617_pkg.vhd	Temperature sensor [8]
i2c/src/vhdl/i2c_dev_max6652_pkg.vhd	Temperature and voltages sensor [8]
i2c/src/vhdl/i2c_dev_ltc4260_pkg.vhd	Hot swap controller and votages sensor
i2c/src/vhdl/i2c_dev_unb_pkg.vhd	I2C slaves on UniBoard
i2c/src/vhdl/i2c_dev_adu_pkg.vhd	I2C slaves on ADU

Table 11: I2C slave definitions

4.1.5 I2C slave behavoral models

File	Description
i2c/tb/vhdl/i2c_slv_device.vhd	I2C slave RTL model, based on LOFAR RCU I2C slave VHDL code
i2c/tb/vhdl/dev_max1618.vhd	Temperature sensor
i2c/tb/vhdl/dev_max6652.vhd	Temperature and voltages sensor
i2c/tb/vhdl/dev_pca9555.vhd	16 bit IO expander
i2c/tb/vhdl/dev_ltc4260_pkg.vhd	Hot swap controller and voltages sensor
i2c/tb/vhdl/dev_unb_pkg.vhd	I2C slaves on UniBoard
i2c/tb/vhdl/dev_adu_pkg.vhd	I2C slaves on the ADU board

Table 12: I2C slave behavioural models

The $UNB/Firmware/modules/fmf code from [5] also contains a model for an I2C master and a slave, but this is not used.

4.1.6 Application mms_unb_sens

File	Description
unb_common/src/vhdl/unb_sens.vhd	
unb_common/src/vhdl/unb_sens_ctrl.vhd	
unb_common/src/vhdl/unb_sens_reg.vhd	
unb_common/src/vhdl/mms_unb_sens.vhd	See Figure 2
unb_common/tb/vhdl/tb_mms_unb_sens.vhd	Self checking test bench for mms_unb_sens.vhd. Usage: > as 10 > run -all
modules/src/reg_unb_sens.c (h)	

Table 13: Application mms_unb_sens VHDL and C code for on the NIOS embedded processor

The $UNB/Firmware/modules/Lofar/sens code contains a similar ROM based protocol list module that was ported from LOFAR RSP, but is not used within UniBoard.

4.1.7 Application avs_i2c_master

File	Decription
i2c/src/vhdl/i2c_pkg.vhd	I2C constants, e.g. bit rate = 50 kpbs
i2c/src/vhdl/i2c_mm.vhd	
i2c/src/vhdl/i2c_master.vhd	See Figure 3. General I2C master with protocol list, result buffer and control register
i2c/src/vhdl/avs_i2c_master.vhd	See Figure 3. Wrapper for i2c_master.vhd to fit Avalon bus naming
i2c/src/vhdl/avs_i2c_master_hw.tcl	SOPC Builder HW description file for avs_i2c_master.vhd
i2c/tb/vhdl/tb_i2c_master.vhd	Self checking test bench for i2c_master.vhd. Usage: > do wave_i2c_master.do > run -all
modules/src/i2c_max1617.h	
modules/src/i2c_ltc4260.h	
modules/src/i2c_smbus.c (h)	
modules/src/avs_i2c_master.c (h)	General I2C master control
modules/src/avs_i2c_master_regs.h	
modules/src/unb_sensors.c (h)	Hides avs_i2c_master.h, provides UniBoard sensor read out and fan control via functions. The NIOS /apps/unb_sens_main/main.c application gives an example of how unb_sensors.h is used

Table 14: Application I2C master VHDL and C code for on the NIOS embedded processor

4.1.8 Application mms_i2c_commander

File	Decription
i2c/src/vhdl/i2c_commander_pkg.vhd	
i2c/src/vhdl/i2c_commander_unb_pkg.vhd	I2C commander register settings for ADU control
i2c/src/vhdl/i2c_dev_unb_pkg.vhd	Protocol lists and expected results for UniBoard I2C slaves
i2c/src/vhdl/i2c_commander_aduh_pkg.vhd	I2C commander register settings for ADU control
i2c/src/vhdl/i2c_dev_adu_pkg.vhd	Protocol lists and expected results for ADU board I2C slaves
i2c/src/vhdl/i2c_commander_ctrl.vhd	
i2c/src/vhdl/i2c_commander_reg.vhd	
i2c/src/vhdl/i2c_commander.vhd	See Figure 4
i2c/tb/vhdl/tb_i2c_commander.vhd	Self checking test bench for i2c_commander.vhd. Usage: > do wave_i2c_commander.do > run -all
adu_protocol_ram_init.hex (txt, mif)	u_ram_protocol init file derived from i2c_commander_aduh_pkg.vhd (see section 4.1.9)
unb_protocol_ram_init.hex (txt, mif)	u_ram_protocol init file derived from i2c_commander_unb_pkg.vhd (see section 4.1.9)
pi_adu_i2c_commander.py	Python peripheral class for ADU I2C commander
tc_pi_adu_i2c_commander.py	Demo test case for pi_adu_i2c_commander.py
util_adu_i2c_commander.py	Utility script for using pi_adu_i2c_commander.py

Table 15: Application I2C commander VHDL code, RAM initialization files and Python code

4.1.9 How to create the I2C commander *_protocol_ram_init.hex file for the u_protocol_ram:

Text copied from Lofar/i2c/tb/data/ how_to_create_memory_init_hex_file.txt:

```
1) Run the tb_i2c_commander simulation to create the protocol_ram_init.txt file

2) Manual conversion steps
 a) In an editor determine number of bytes in protocol_ram_init, e.g. 5120 (the
    tb_i2c_commander uses a size that is a factor of 1024, i.e. in steps of 1
    M9K = 1 kByte)
 b) In Quartus use File-open of an existing *.mif file. Now the edit menu offers
    fields to change the *.mif file (the default edit menu does not show the
    fields for memory file manipulation).
      - in the Memory Size Wizard set width to 8 bits and size to 5120
      - via Custom Fill Cells fill address 0 to 13FF (=5119) with incrementing
        values
      - save as: protocol_ram_init.mif
 c) Open protocol_ram_init.mif in an editor and use column edit to replace the
    incrementing values with the values from protocol_ram_init.txt. Then save:
      ../../../tb/data/protocol_ram_init.mif
 d) Open protocol_ram_init.mif in Quartus and save it as:
      ../../../tb/data/protocol_ram_init.hex

3) Commit the files:
   . protocol_ram_init.txt
   . protocol_ram_init.mif
   . protocol_ram_init.hex
```

4.2 Known issues

The I^2C master module on LOFAR RSP and EMBRACE ECB and the I^2C slave on the LOFAR RCUs are working reliably in practise. However there have been several issues.

4.2.1 Clk rate and bit rate

LOFAR bug 575. Both the master and slave sample the SDA at the rising edge of SCL. The SDA setup time is (1+clk_cnt)*clk-period and the hold time is 4*(1+clk_cnt)*clk-period, where clk-period = 5 ns and clk_cnt = 399 for 200 MHz to get SCL clock rate of 1/(5*(1+clk_cnt)*clk-period) = 100 kbps. The setup and hold times are OK, so the I^2C master is robust. The master also allows clock stretching by the slave (provided parameter clk_cnt >= 2).

4.2.2 Bus arbitration

LOFAR bug 575. The bus arbitration logic in i2c_bit.vhd has been commented. This quick fix targets the problem of ocillating SDA line (looks like start every 4 I^2C clk en cycles). Probably the issue is more severe if the difference between clk rate and bit rate is high, causing the SDA line to be sampled to soon by the I^2C master, thereby mistaking its own activity for a collision. During an access SDA is sometimes not driven by neither the master nor the slave, causing the SDA to go briefly high while SCL = '0'.

4.2.3 Ringing on SCL and SDA edges

LOFAR bug 1111. On LOFAR RSP the SCL showed undershoot for larger FPGA drive strengths. Once it was observed that an SDA ack from the LOFAR RCU slave went high some 450 ns after the rising edge of SCL. The slave should release SDA after the falling edge. Hence apparently the RCU slave logic had recognized a falling edge in during the rising edge of SDA. There is indeed some ringing on the (slow) rising edge of SDA.. Sometimes the slave gives its ack one cylce too early, this can be seen e.g. when the last write bit is a 1, while on the line it is a 0 (due to the ack). This can be explained by an extra false SCL edge. The RCU slave release SDA on the falling edge of SCL. If timing is awkward, then SDA going high due to SCL going low could perhaps still be recognized as an I^2C stop by the slave. Similar for I^2C start. Using a stronger pull up 2k7 on the RCU instead of 10k gave a big improvement. In addition the RCU firmware has been adapted. The SCL and SDA are now first clocked in and filtered by the system clock. This avoids the potential false edge detections and it gives some more hold time of SDA after SCL falling edge.

4.2.4 Comma logic

LOFAR bug 995. The LOFAR HBA client I^2C handler on the LOFAR RCU is implemented in software. It requires that an idle comma is inserted between octets. A comma of 1 msec is sufficient. Probably also after start a comma should be inserted, since I^2C start (and restart) can also cause a slave interrupt. The I^2C master already supports clock stretching, but this is not sufficient for the HBA client I^2C slave, because it only can stretch the clk low for a read bytes access. Therefore comma logic was implemented in i2c_smbus(rtl).vhd.

4.2.5 Sometimes access to LOFAR TDS fails

LOFAR bug 1242. Programming the LOFAR TDS PLL requires several 100 I^2C accesses to the IO expander on TDS, because it uses bit-banging to program the PLL via SPI. About once every 5 runs the test fails, the protocol result buffer then shows that an I^2C write access did not get an ACK. This is still an open issue, but with low priority.

4.3 I^{2}C access example

This example shows how to configure and read the MAX6652 [8] I^{2}C sensor. To ensure that the I^{2}C access results are fresh it is useful to first overwrite the protocol results register with some arbitrary data. To access the I^{2}C sensor write the sequence from Table 16 to the protocol list register.

Octet	Value	Description
0	0x06	PROTOCOL_WRITE_BYTE
1	0x14	ADDR = I^{2}C address of MAX6652 connected to ground
2	0x40	CMD = Access MAX6652 config register
3	0x11	DATA = Enable config_start and config_line_freq_sel
4	0x07	PROTOCOL_READ_BYTE
5	0x14	ADDR
6	0x20	CMD = Read voltage measured at the MAX6652 2v5 input
7	0x07	PROTOCOL_READ_BYTE
8	0x14	ADDR
9	0x22	CMD = Read voltage measured at the MAX6652 3v3 input
10	0x07	PROTOCOL_READ_BYTE
11	0x14	ADDR
12	0x21	CMD = Read voltage measured at the MAX6652 12v input
13	0x07	PROTOCOL_READ_BYTE
14	0x14	ADDR
15	0x23	CMD = Read voltage measured at the MAX6652 vcc input
16	0x07	PROTOCOL_READ_BYTE
17	0x14	ADDR
18	0x27	CMD = Read the measured temperature
19	0x13	PROTOCOL_C_END

Table 16 Protocol list example

The protocol list execution starts after an activation pulse. A few ms later the result will be available in the protocol result register. The actual latency depends on the I^{2}C bit rate, which is set e.g. to 50 kbps. The expected protocol result for the protocol list of Table 16 is shown in Table 17.

Octet	Value	Description
0	0	0 = PROTOCOL_WRITE_BYTE went OK
1	#	Value of 2v5
2	0	0 = PROTOCOL_READ_BYTE done
3	#	Value of 3v3
4	0	0 = PROTOCOL_READ_BYTE done
5	#	Value of 12v
6	0	0 = PROTOCOL_READ_BYTE done
7	#	Value of vcc
8	0	0 = PROTOCOL_READ_BYTE done
9	#	Value of temperature
10	0	0 = PROTOCOL_READ_BYTE done
11	0	0 = PROTOCOL_C_END done

Table 17 Expected protocol result example

Octet 0 in Table 17 will read 1 if the I^{2}C slave did not give an ACK (pull up), same for octets 2, 4, 6, 8 and 10. Similar octets 1, 3, 5, 7 and 9 will read 0xFF (pull up) when the I^{2}C slave is not connected.