

## Specification for module interfaces using VHDL records

	Organisatie / Organization	Datum / Date
<b>Auteur(s) / Author(s):</b>  Eric Kooistra	ASTRON	
<b>Controle / Checked:</b>  Andre Gunst	ASTRON	
<b>Goedkeuring / Approval:</b>  Andre Gunst	ASTRON	
<b>Autorisatie / Authorisation:</b>  <b>Handtekening / Signature</b>	ASTRON	

© ASTRON 2010

All rights are reserved. Reproduction in whole or in part is prohibited without written consent of the copyright owner.

## Distribution list:

---

Group:	Others:
Andre Gunst (AG, ASTRON) Daniel van der Schuur (DS, ASTRON) Rajan Raj Thilak (RT, ASTRON) Arpad Szomoru (AS, JIVE) Jonathan Hargreaves (JH, JIVE) Salvatore Pirrucci (SP, JIVE)	Gijs Schoonderbeek (GS, ASTRON) Sjouke Zwier (SZ, ASTRON)

## Document history:

---

Revision	Date	Author	Modification / Change
0.1	2010-04-7	Eric Kooistra	Draft, ready for review.
0.2	2010-11-04	Eric Kooistra	Updated record functions. Removed appendix with VHDL code examples (too big).
0.3	2011-01-05	Eric Kooistra	Major rework. Now defined general record types for the MM interface and the ST interface, instead of defining module specific record types.
1.0	2011-02-03	Eric Kooistra	Updated after review on Jan 20, 2011 with EK, DS, JH. Clarified the definition of the terms: design, SOPC system, module and component as used for in this document. Clarified more why it is beneficial to use records and wrappers.

## Table of contents:

---

1	Introduction.....	5
1.1	Purpose .....	5
1.2	Design, SOPC system, module, component.....	5
1.3	Interfaces.....	5
2	Using VHDL records.....	7
2.1	Interface definition .....	7
2.1.1	MM interface records .....	7
2.1.2	ST interface records .....	8
2.2	Record instances.....	8
2.3	Functions .....	9
2.4	Packages.....	9
2.5	Simulation and synthesis.....	9
2.6	Test bench facilities .....	9
3	Using VHDL wrappers.....	11
3.1	Using a MegaWizard IP wrapper.....	11
3.2	Using an Avalon component wrapper .....	11
3.2.1	The hardware description script.....	12
4	Conclusion.....	13

## Terminology:

---

AVS	Avalon interface Slave (Avalon slave wrapper of a MM slave peripheral)
DP	Data Path
DSP	Digital Signal Processing
DUT	Device Under Test
eof	End of Frame
eop	End of Packet
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GbE	Giga Bit Ethernet
GUI	Graphical User Interface
HDL	Hardware Description Language
ID	Identification
IO	Input Output
IP	Intellectual Property
IRQ	Interrupt Request
MAC	Media Access Control (in communications) or Multiply and Accumulate (in DSP)
MISO	Master In Slave Out
MM	Memory-Mapped
MMS	MM Slave
MOSI	Master Out Slave In
Nof	Number of
PHY	Physical layer
PPS	Pulse Per Second
RX	Receive
SISO	Source In Sink Out
sof	Start of Frame
sop	Start of Packet
SOPC	System On a Programmable Chip (Altera)
SOSI	Source Out Sink In
ST	Streaming
TSE	Triple Speed Ethernet (Altera 10/100/1000MbE MAC)
TX	Transmit
UNB	<a href="https://svn.astron.nl/UniBoard_FP7/UniBoard/trunk">https://svn.astron.nl/UniBoard_FP7/UniBoard/trunk</a> , the UniBoard FP7 SVN repository

## References:

---

1. "Avalon Interface Specifications", mnl\_avalon\_spec.pdf, [www.altera.com](http://www.altera.com)
2. "Quartus II Handbook", quartusii\_handbook.pdf, [www.altera.com](http://www.altera.com)
3. \$UNB/Firmware/doc/howto/How\_to\_use\_SOPC\_Component\_Editor.txt
4. "Firmware development for the UniBoard", ASTRON-RP-315, Eric Kooistra
5. "RadioNet FP7: Modular design and module reuse: the ETH module as an example", UniBoard face to face meeting in Bordeaux 12-13 Oct 2010, Eric Kooistra
6. "UNB\_Common Module Description", ASTRON-RP-426, Eric Kooistra

## 1 Introduction

### 1.1 Purpose

This document defines how VHDL records are used to compactly represent module interfaces. The document also describes how VHDL wrappers are used to ease the (re)use of modules.

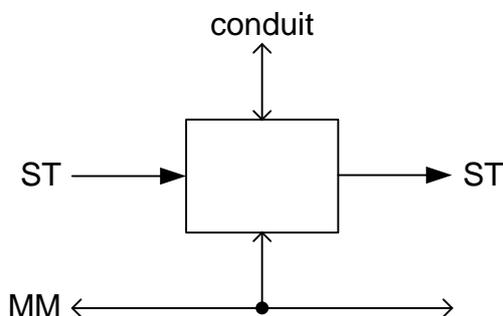
### 1.2 Design, SOPC system, module, component

The assumption is that we use a modular development approach [4, 5]. In this approach a firmware design is build up out of firmware modules. In the UniBoard SVN firmware project the firmware designs are kept in the designs/ directory and the modules are kept in the modules/ directory. In this document the terms design, module and component are used. Their definition is not strict, but typically a design consists of modules and a module consists of components. All three are defined by entity and architecture pairs in VHDL. The design entity is the top level entity that defines the firmware image that will run on the FPGA. The component entity can be a low level function like a FIFO or it may also be an elaborate IP module like e.g. the TSE MAC. The module architecture contains several components to define a more elaborate function, e.g. the UniBoard Ethernet ETH module that not only contains a TSE MAC, but also a received frames FIFO and a MM control message interface to a microprocessor. Instead of consisting of a single more elaborate function a module may also consist of a group of related low level functions, e.g. like the UniBoard COMMON module in \$UNB/Firmware/modules/common.

In addition to the term firmware design as defined above, it is also appropriate for UniBoard to define the term SOPC system. An SOPC system is a set of modules that are interconnected in Altera's SOPC Builder GUI and then generated into a VHDL file [2]. With some glue components an SOPC system can be made into a firmware design that can run on a FPGA [6].

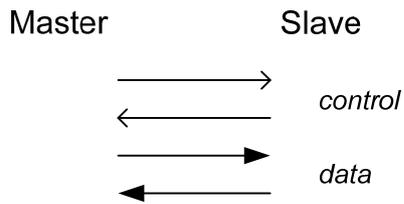
### 1.3 Interfaces

Besides a clock and reset interface the Altera Avalon interface [1] defines a memory-mapped (MM) interface and a streaming (ST) interface. Other component specific interfaces are called 'conduit' interfaces. The MM interface is a master-slave interface and typically used for control and monitoring. The streaming interface is a source-sink interface and used for data transport. The Avalon MM and ST interface are in fact quite generic and it appears that the MM and ST interfaces are sufficient for all modules. The conduit interface is typically only needed for off-FPGA PHY interfaces. Figure 1 shows the module interfaces. Note that a module does not have to have all interfaces.



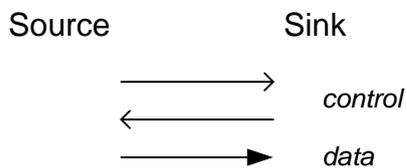
**Figure 1: Module interfaces: MM, ST and conduit**

Figure 2 shows a high level representation of the memory-mapped (MM) interface. Two key aspects of the MM interface are that it uses addressing to select a data location and that it allows data access in both directions.

**Figure 2: MM interface**

The Altera Avalon interface defines interrupts via a separate interrupt interface. However interrupts could also be considered as part of the MM interface, because the IRQ signal typically corresponds to a MM slave and can be regarded as a control signal from a slave to a master.

Figure 3 shows a high level representation of the streaming (ST) interface. The key aspect of the ST interface is that it streams data in only one direction from source to sink. Furthermore there is only one (control) signal from sink back to the source. This signal is used for backpressure flow control.

**Figure 3: ST interface**

## 2 Using VHDL records

### 2.1 Interface definition

There are three issues to handle:

- An instance of a record can only be IN or OUT not both
- Not every MM or ST interface requires the full set of available interface signals
- Not every MM or ST interface has the same signal vector widths

It appears best to define a separate record for the inputs and another record for the outputs, rather than to group all interface signals into one record. For the UniBoard modules it appears fine to define a general record type for the MM interface and for the ST interface, because most instances do use all fields and it is no problem if fields remain unused ('X'). For the vector widths it appears fine to define them such that they can fit the largest expected vector width. For interfaces that need smaller vector width it is fine to leave some bits unused ('X'). The advantage of this approach is that we only need a few record types that suit all UniBoard module MM and ST interfaces.

#### 2.1.1 MM interface records

The `common_mem_pkg` package in `$UNB/Firmware/modules/common/src/vhdl/` defines the MOSI and MISO records for the MM interface:

```
TYPE t_mem_miso IS RECORD -- Master In Slave Out
  rddata      : STD_LOGIC_VECTOR(c_mem_address_w-1 DOWNT0 0);
  waitrequest : STD_LOGIC;
END RECORD;

TYPE t_mem_mosi IS RECORD -- Master Out Slave In
  address     : STD_LOGIC_VECTOR(c_mem_address_w-1 DOWNT0 0);
  wrdata      : STD_LOGIC_VECTOR(c_mem_data_w-1 DOWNT0 0);
  wr          : STD_LOGIC; -- write strobe
  rd          : STD_LOGIC; -- read strobe
END RECORD;
```

Where:

`c_mem_address_w = 32`, is sufficient for a 32-bit microprocessor (like the Altera Nios II)  
`c_mem_data_w = 72`, is sufficient for up to 8 bytes, that can also be 9-bit bytes

The IRQ signal could be made available as a field in the `t_mem_miso` record. Alternatively interrupt signals can be treated as a separate interface like with the Avalon interface [1].

The '`t_mem_miso`' and '`t_mem_mosi`' records define a subset of the Avalon MM interface. The actual address range and data width of a register or memory that can be accessed via the MM interface are defined via a constants record that is also defined in the `common_mem_pkg` package:

```
TYPE t_c_mem IS RECORD
  latency     : NATURAL; -- read latency
  adr_w       : NATURAL;
  dat_w       : NATURAL;
  nof_dat     : NATURAL; -- optional, nof words <= 2**adr_w
  init_sl     : STD_LOGIC; -- optional, init all words to '0', '1' or 'X'
END RECORD;
```

### 2.1.2 ST interface records

The ST interface is called DP for data path. The `dp_stream_pkg` package in `$UNB/Firmware/modules/dp/src/vhdl/` defines the SOSI and SISO records for the ST interface:

```

TYPE t_dp_asiso IS RECORD -- Source In Sink Out
  ready    : STD_LOGIC;
  nc       : STD_LOGIC; -- not connect
END RECORD;

TYPE t_dp_sosi IS RECORD -- Source Out Sink In
  data     : STD_LOGIC_VECTOR(c_dp_stream_data_w-1 DOWNTO 0);
  valid    : STD_LOGIC;
  sop      : STD_LOGIC;
  eop      : STD_LOGIC;
  empty    : STD_LOGIC_VECTOR(c_dp_stream_empty_w-1 DOWNTO 0);
  channel  : STD_LOGIC_VECTOR(c_dp_stream_channel_w-1 DOWNTO 0);
  err      : STD_LOGIC_VECTOR(c_dp_stream_error_w-1 DOWNTO 0);
END RECORD;

```

Where:

`c_dp_stream_data_w = 72`, sufficient for maximum word 8 \* 9-bit  
`c_dp_stream_empty_w = 8`, is sufficient for maximum 256 symbols per data word  
`c_dp_stream_channel_w = 8`, sufficient for maximum 256 channels  
`c_dp_stream_error_w = 8`, sufficient for maximum 255 error numbers, 0 = OK

The `'t_dp_asiso'` and `'t_dp_sosi'` records define the whole set of the Avalon ST interface. The `'t_dp_asiso'` record only contains the back pressure ready signal, but is defined as a record to show its relation to the `'t_dp_sosi'` signals. The `'nc'` field is needed because it is not possible to initialize a record instance if the record has only one field.

In DSP systems it is useful to also have a synchronisation signal like a PPS along with the data. This `'sync'` signal can be defined as an extra field in the `'t_dp_sosi'` record. This `'sync'` field is not supported by the Avalon interface, but it could be provided via an extra `'data'` field bit or via the `'sop'` field if that is not used. In such a way the DP interface with `'sync'` field can then still be connected within the SOPC Builder GUI.

The `dp_stream_pkg` package also defines arrays of streaming records. These are e.g. useful to define multiple ST ports or to define a ST shift register:

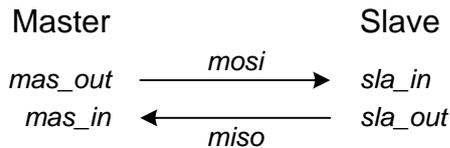
```

TYPE t_dp_asiso_arr IS ARRAY (INTEGER RANGE <>) OF t_dp_asiso;
TYPE t_dp_sosi_arr IS ARRAY (INTEGER RANGE <>) OF t_dp_sosi;

```

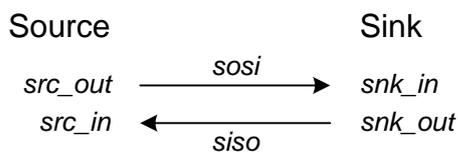
## 2.2 Record instances

Figure 4 defines the MM interface signal names. They are all of the same `'t_mem_miso'` and `'t_mem_mosi'` record types. On the master component port they have postfixes `'_mas_out'` and `'_mas_in'` and on the slave component port they respectively have postfixes `'_sla_in'` and `'_sla_out'`. Between the components they respectively have postfixes `'_mosi'` for Master Out Slave In and `'_miso'` for Master In Slave Out. In fact only using postfixes `'_mosi'` and `'_miso'` for all would be sufficient too, but using `'_mas_out'`, `'_mas_in'` and `'_sla_in'` and `'_sla_out'` seems clearer.



**Figure 4: MM interface signal names**

Figure 5 defines the ST interface signal names. They are all of the same 't\_dp\_isiko' and 't\_dp\_sosi' record types. On the source component port they have postfixes '\_src\_out' and '\_src\_in' and on the slave component port they respectively have postfixes '\_snk\_in' and '\_snk\_out'. Between the components they respectively have postfixes '\_sosi' for Source Out Sink In and '\_isiko' for Source In Sink Out. In fact only using postfixes '\_sosi' and '\_isiko' for all would be sufficient too, but using '\_src\_out', '\_src\_in' and '\_snk\_in' and '\_snk\_out' seems clearer.



**Figure 5: ST interface signal names**

Note that the naming conventions of Figure 4 and Figure 5 avoid the confusion that can arise when a signal is only called '\_in' or '\_out', because whether a signal is input or output is a matter of perspective.

## 2.3 Functions

For handling record signals it is convenient to define functions in a package, for example functions to manipulate fields in a record. The MM interface `common_mem_pkg` package and the ST interface `dp_stream_pkg` package contain many useful functions.

## 2.4 Packages

The constants, records and functions that are available for modules users should be defined in a module package. This package can also contain the record for a conduit interface if appropriate. See the `eth_pkg` and `tse_pkg` packages in `$UNB/Firmware/modules/tse/src/vhdl` for an example. Note that all package items should have the module name as an (abbreviated) <module name> prefix in their name to indicate where they originate from.

## 2.5 Simulation and synthesis

The record types and array of record types defined for the MM interface and ST interface in sections 2.1.1 and 2.1.2 can all be simulated with Modelsim and synthesized with Quartus II.

Unused record fields and unused vector bits show as 'X' in Modelsim and they get optimized away properly by Quartus II.

## 2.6 Test bench facilities

The `tb_common_mem_pkg` in `$UNB/Firmware/modules/common/tb/vhdl` defines useful procedures for accessing the MM interface. This allows simulating the behaviour of a master to verify a MM slave DUT in a VHDL test bench.

The `tb_dp_stream_pkg` and `tb_dp_pkg` in `$UNB/Firmware/modules/dp/tb/vhdl` defines useful procedures for accessing the ST interface. This allows simulating the behaviour of a source or a sync to verify a ST DUT in a VHDL test bench.

### 3 Using VHDL wrappers

Important reasons for using VHDL wrappers are [5]:

- Ensures that all instances use the vendor IP in the same way
- Clearly isolates vendor IP from our own generic VHDL
- Eases porting to tool upgrades, other FPGA types or vendor IP should this be necessary
- Allows redefining an interface using records
- The wrapper may also contain some extra (glue) logic
- Corrections in the wrapper automatically effect all instances

It appears that VHDL wrappers are beneficial to use at two levels in the UniBoard modules:

1. Isolate vendor specific IP (e.g. Altera MegaWizard IP) from the generic HDL, see section 3.1.
2. Make a module entity available to Altera's SOPC Builder, see section 3.2

#### 3.1 Using a MegaWizard IP wrapper

The purpose of a VHDL wrapper is to have a central file via which the IP is instantiated in our designs [5]. Vendor specific components like those created with the Altera MegaWizard or with Xilinx Coregen should not be used directly in the project firmware VHDL. By defining a wrapper entity and a vendor specific architecture name like 'stratix4', 'stratix5' or 'virtex6' it becomes easier to port between different FPGA types or vendors, should this be necessary. Using a wrapper entity also opens the possibility to use records, which can make the understanding and reuse of the component much easier. For example:

- In the ETH module the `tse(stratix4).vhd` wraps the `tse_sgmii_lvds.vhd` MegaWizard IP
- In the COMMON module the `common_ram_crw_crw(stratix4).vhd` wraps the `ram_crw_crw.vhd` MegaWizard IP

Note that such an IP component wrapper can also be used to wrap existing modules that do not (yet) use records. In this way the internal coding style of the module can be left as it is, while for the user it appears with the appropriate record type interfaces.

#### 3.2 Using an Avalon component wrapper

A module can be made available within the Altera SOPC Builder tool [2]. Within SOPC Builder a complete system of MM and ST components can be constructed via a GUI. The SOPC Builder then automatically generates the HDL code for this system ready to use in a design. To make a component or a module (which may contain several components) available to SOPC Builder requires a wrapper entity. This wrapper entity maps the generic interface signals to the Avalon interface signals. Using a wrapper entity is necessary because:

- The Avalon interface definition [1] defines as set of signal prefixes and postfixes for various interfaces like clock, memory-mapped, interrupt, streaming.
- The Avalon interface uses the default VHDL signal types `std_logic` and `std_logic_vector`, so it does not use records.

The component specific interface signals are not known to the Avalon interface. These signals need to be treated within a module or they can be made available outside the SOPC system by defining them as 'conduit' interface. The name of the Avalon interface wrapper file and entity has prefix 'avs\_' for a slave followed by the original component name, so `avs_<module name>` for the entity and `avs_<module name>.vhd` for the file. The file also contains the architecture. The architecture is called 'wrap'. Typically an Avalon wrapper only needs to be made for modules and reusable components [3]. See for an example

avs\_eth.vhd in \$UNB/Firmware/modules/tse/src/vhdl that wraps eth.vhd or avs\_tr\_nonbonded.vhd for the UniBoard non-bonded transceivers module in \$UNB/Firmware/modules/tr\_nonbonded/src/vhdl that wraps mms\_tr\_nonbonded.vhd.

A module can have multiple AVS wrapper components. For example one wrapper could present the eth.vhd module to SOPC Builder without the UDP off-load streaming ports (like avs\_eth.vhd) and another wrapper could present the eth.vhdl module to SOPC Builder with those ST ports.

### 3.2.1 The hardware description script

The Component Editor can be used to create a hardware description TCL script file of our module [3]. When the wrapper entity of the module uses the signal naming conventions of the Avalon interface [1], then the Component Editor automatically recognizes these signals. After filling in some more parameters and e.g. a module name the Component Editor creates the avs\_<module name>\_hw.tcl script file. This hardware description TCL file is used by SOPC Builder to be able to use the module in a SOPC firmware system.

## 4 Conclusion

1. A module can represent a single more elaborate component like e.g. the ETH module and the TR\_NONBONDED module, or a group of related more low level components like e.g. the COMMON module and the DP module.
2. Modules that represent more elaborate components or low level MM or ST components can use the same standard MM and ST interfaces defined in `common_mem(pkg).vhd` and `dp_stream(pkg).vhd`.
3. General record types have been defined for the MM and ST interfaces:
  - `'t_mem_miso'` and `'t_mem_mosi'` for the MM interface in `common_mem(pkg).vhd`
  - `'t_dp_asiso'` and `'t_dp_asisi'` for the ST interface in `dp_stream(pkg).vhd`
4. A scheme for input and output signal naming has been defined using postfixes:
  - `'_mas_in'`, `'_miso'`, `'_sla_out'` for MM Master In Slave Out
  - `'_mas_out'`, `'_mosi'`, `'_sla_in'` for MM Master Out Slave In
  - `'_src_in'`, `'_asiso'`, `'_snk_out'`, for ST Source In Slave Out
  - `'_src_out'`, `'_asisi'`, `'_snk_in'` for ST Source Out Slave In
5. For handling record signals it is convenient to define functions in a module package.
6. Modules can be interconnected using the SOPC Builder to create an SOPC system. By means of a wrapper entity the generic module or component can be wrapped to the Avalon bus interface and made available to SOPC Builder.
7. By means of a wrapper entity for MegaWizard components the vendor specific firmware is clearly separated from the generic firmware.

By adhering to these schemes for using records and wrappers it becomes easier to comprehend and reuse components and modules that have been made by different firmware designers and/or development groups.