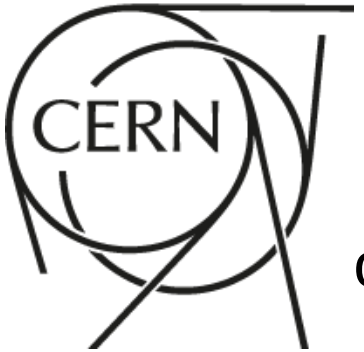
Automatic code generation for managing the firmware and software for configuration/status registers and memories in the ATLAS Level-1 Central Trigger



Anna Kulińska

on behalf of the ATLAS Level-1 Central Trigger (L1CT) Group

Overview:

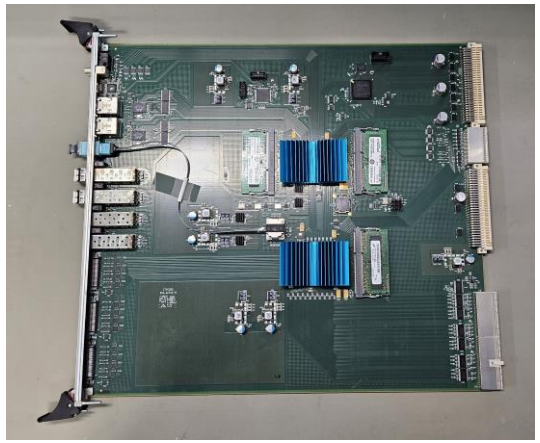
1. HardwareCompiler
2. Firmware & software development flow
3. XML register map
4. Firmware development
   a. VHDL package
   b. Register translator
5. Software development
   a. C++ code
   b. Python hardware test scripts
6. Summary

HardwareCompiler

- Tool developed by the ATLAS Level-1 Central Trigger (L1CT) group
- **VHDL and C++ source code generator**
- Code generation is based on a **XML register map** containing definitions of registers & memories
- Greatly simplifies firmware and software development and hardware tests for board with **complex FPGA designs**
- The hardware compiler was developed in **2013** for generating C++ software. Since then, it was constantly improved, notably with writing VHDL code for firmware registers, and recently with writing VHDL address decoders.
- It is currently being used for the development of all L1CT modules for the Run-4 TDAQ Upgrade, which are based on **AMD Xilinx FPGAs and SoCs**

ALTI board with one Artix 7-Series XC7A200T FPGA

MUCTPI board with two Virtex UltraScale+ XCVU9P FPGAs and one Kintex UltraScale XCKU095 FPGA



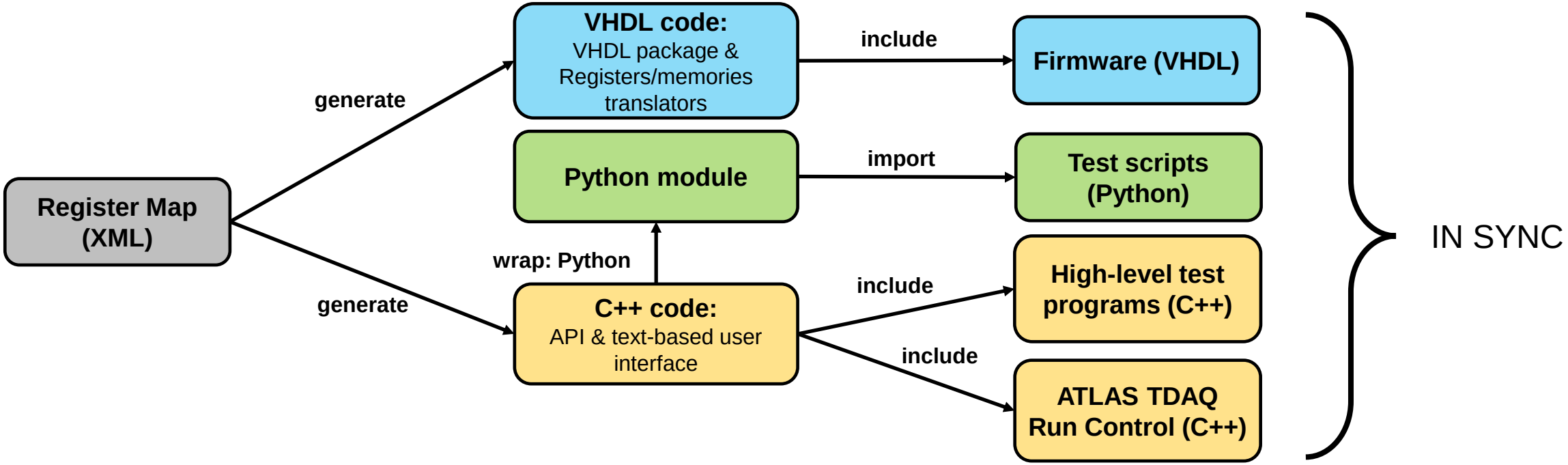
CTPCORE+ board with two Virtex 7-Series XC7VX485T FPGAs

Firmware & software development flow

1. Hardware compiler generates **VHDL code for running in FPGAs**, and **software for running on the Xilinx MPSoC** System-on-Chip

2. C++ software is wrapped in Python for simpler interactive testing and development of complex test sequences

- Calling the HardwareCompiler & Python Wrapper is fully integrated into the software **Continuous Integration** framework
- This scheme allows the software and firmware to **be kept consistent**



XML register map

```
<module name="MuonSectorProcessor" size="0x10000000" type="D32" addr="0x00000000">
  ...
  <block name="TTC"                    addr="0x00100000" decoder="SYS">

    <register name="CounterL1a"        addr="0x00000000" modf="R"/>
    <register name="CounterSync"       addr="0x00000004" mask="0xFFF" modf="R"/>
    <register name="Control"           addr="0x00000008" modf="RW">
      <field name="ResetCounter"       mask="0x00000001" multiple="2" offset="4"/>
    </register>
    ...
  </block>  <!-- TTC -->

  <block name="SLI"                    addr="0x00000000" decoder="SLR1">

    <register name="SpyPlayControl"    addr="0x00000008" modf="RW">
      <field name="SpyEnable"          mask="0x00000004" form="BOOLEAN"/>
      <field name="PlaybackEnable"     mask="0x00000008" form="BOOLEAN"/>
      <field name="PlaybackLastAddress" mask="0x0000FFF0" form="NUMBER"/>
      <field name="Mode"               mask="0x00000003">
        <value name="OFF"              data="0x00000000"/>
        <value name="SL"               data="0x00000001"/>
        <value name="PG"               data="0x00000002"/>
        <value name="SPY"              data="0x00000003"/>
      </field>
    </register>

    ...
    <memory name="SectorMemory"        addr="0x00010000" size="0x10000"/>

  </block>  <!-- SLI -->
  ...
</module>
```

REGISTERS ATTRIBUTES:

- address
- access type (read-only, read&write…)
- data mask

Register data can also be described using a bit string:

- bit fields with mask
- symbolic names (human readable)

MEMORIES ATTRIBUTES:

- Defined in the same way as registers
- Additional **size** attribute

HIERARCHY:

- Registers and memories can be arranged in blocks
- Blocks can contain other blocks
- Registers, memories, fields and blocks can have a “**multiple**“ attribute to generate arrays of elements
- Blocks can have a “**decoder**” attribute to generate register translator for registers and memories

Firmware development

VHDL code

- Automatically generated by HardwareCompiler
- Contains declarations for registers and memories described in XML register map
  - address, width, size constants
  - data type declarations
  - register field records
  - conversion functions
- Ensures consistency between XML register map and firmware implementation
- Follows XML register map block structure

```
use work.MuonSectorProcessor_pkg.all;

snaplay_i : entity work.sl_snaplay
  generic map(
    Mode       => SLI_SpyPlayControl_Mode_PG,
    ADDR_WIDTH => SLI_SECTORMEMORY_ADDR_WIDTH,
  )
  port map(
    rstn_i          => not (rst_i),
    clk_i           => clk40_i,
    mode_i          => ctrl_i.SpyPlayControl.Mode,
    --
    pg_enable_i     => ctrl_i.SpyPlayControl.PlaybackEnable,
    pg_last_addr_i  => ctrl_i.SpyPlayControl.PlaybackLastAddress,
    spy_enable_i    => ctrl_i.SpyPlayControl.SpyEnable,
    --
    sli_mem_mosi_i  => sli_mem_mosi_i.SectorMemory,
    sli_mem_miso_o  => sli_mem_miso_o.SectorMemory,
    pg_data_o       => sl_pg_data,
    sl_i            => sl_data,
);
```

CONTROL REGISTER FIELD'S VALUE

MEMORY ADDRESS WIDTH CONSTANT

CONTROL REGISTER RECORD'S FIELDS

MEMORY INTERFACE

```
<block name="SLI"                          addr="0x00000000" decoder="SLR1">

  <register name="SpyPlayControl"          addr="0x00000008" modf="RW">
    <field name="SpyEnable"                mask="0x00000004" form="BOOLEAN"/>
    <field name="PlaybackEnable"           mask="0x00000008" form="BOOLEAN"/>
    <field name="PlaybackLastAddress"      mask="0x0000FFF0" form="NUMBER"/>
    <field name="Mode"                     mask="0x00000003">
      <value name="OFF"                    data="0x00000000"/>
      <value name="SL"                     data="0x00000001"/>
      <value name="PG"                     data="0x00000002"/>
      <value name="SPY"                    data="0x00000003"/>
    </field>
  </register>
  <memory name="SectorMemory"              addr="0x00010000" size="0x10000"/>

</block>  <!-- SLI -->
```

Register/Memory Translator

Translator implements control/status registers and read/write access to external memories with an AXI4 full bus interface

Registers and memories are grouped into translator's blocks (based on the **"decoder"** block attribute):

- Registers are instantiated inside the translator
- Multiple **decoder** definitions can be used in a single XML register map
- **"decoder"** attribute can be specified at **any level** of the block hierarchy

Multiple translators may be needed in a design:

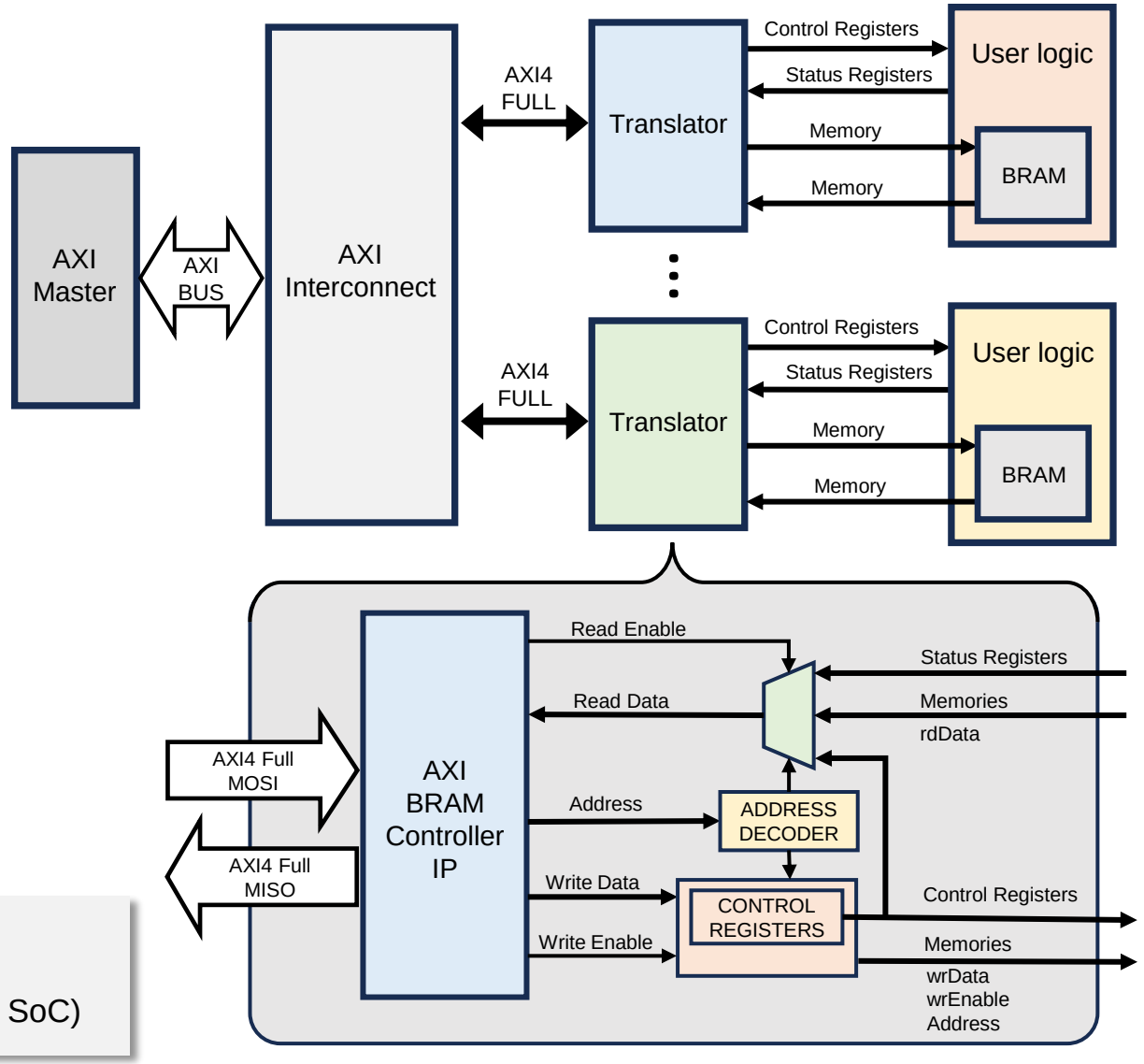
- Split across different **clock domains** or different **SLRs**
- Overcome address range limitation (1 Mbyte per translator)

Translator uses **Xilinx AXI BRAM Controller IP** core:

- Translates between AXI4 full bus and BRAM access signals (address, read enable, write enable, read data, write data)
- 20 bits of address width limitation

Recent L1CT boards use **Xilinx MPSoCs** to control the hardware

=> AXI is the natural choice for SoC FPGAs and processing FPGAs (without SoC)



Translator instance example

```
<block name="SLI"                    addr="0x00000000"  decoder="SLR1">

  <register name="SpyPlayControl"       addr="0x00000008" modf="RW">
    <field name="SpyEnable"             mask="0x00000004" form="BOOLEAN"/>
    <field name="PlaybackEnable"        mask="0x00000008" form="BOOLEAN"/>
    <field name="PlaybackLastAddress"   mask="0x0000FFF0" form="NUMBER"/>
    <field name="Mode"                  mask="0x00000003">
      <value name="OFF"                 data="0x00000000"/>
      <value name="SL"                  data="0x00000001"/>
      <value name="PG"                  data="0x00000002"/>
      <value name="SPY"                 data="0x00000003"/>
    </field>
  </register>

  <register name="BcidOffset"           addr="0x0000000C" modf="RW"/>

  <register name="SpyPlayStatus"        addr="0x00000000" modf="R">
    <field name="PlaybackBusy"          mask="0x00000008" form="BOOLEAN"/>
    <field name="SpyBusy"               mask="0x00000004" form="BOOLEAN"/>
    <field name="SpyAddress"            mask="0x0000FFF0" form="NUMBER"/>
  </register>

  <register name="BcidMonitor"          addr="0x00000004" modf="R"/>

  <memory name="SectorMemory"           addr="0x00010000" size="0x10000"/>

</block>  <!-- SLI -->
```

SLR1 DECODER

```
slr1_i : entity work.SLR1_reg_translator
  generic map(
    ADDR_BITS => SLR1_ADDR_BITS
  )
  port map(
    rst_i           =>  rst_i,
    clk_i           =>  clk_i,
    axi_full_mosi_i =>  axi_full_mosi_i(0),
    axi_full_miso_o =>  axi_full_miso_o(0),
    reg_mosi_o      =>  slr1_reg_mosi,
    reg_miso_i      =>  slr1_reg_miso
    mem_mosi_o      =>  slr1_mem_mosi,
    mem_miso_i      =>  slr1_mem_miso
  );
```

AXI4 FULL BUS INTERFACE CONNECTIONS

CONTROL REGISTERS OUTPUT RECORD

STATUS REGISTERS INPUT RECORD

MEMORY INTERFACE IN/OUT

Software development

&

Hardware testing

C++ methods

- HardwareCompiler generates C++ class based on XML register map
- Generates API methods to perform **read and write** operations on registers and memories
- Generates a **text-based user interface** for interactive use**:**
  - Exposes read and write operations for registers and memories
  - Menu entry for each method defined in the class
  - Adheres to **XML register map** hierarchy
  - Very effective for basic firmware checks
- Low-level software functions are used in higher-level user written C++ code (e.g. module configuration, control and monitoring, interface to control room framework)

```
“SLI”/“SLI_SpyPlayStatus" bit string = "MSP_SLI_SPYPLAYSTATUS_BITSTRING": 0x00000000
    "PlaybackBusy"              :              "false"
    "SpyBusy"                   :              “true"
    "SpyLastAddress"            :                 136

  "read  "SLI_SpyPlayStatus" [bit string]" returns 0
```

```
>>> MuonSectorProcessor main menu/SLI menu <<<

     0  quit
     1  read  "SLI_SpyPlayStatus" [bit string]
     2  read  "SLI_SpyPlayStatus" [number]
     …
     8  read  "SLI_SectorMemory" [number]
     9  write "SLI_SectorMemory" [number]
    10  read  "SLI_SectorMemory" [vector]
    11  write "SLI_SectorMemory" [vector]
    12  read  "SLI_SectorMemory" [block]
    13  write "SLI_SectorMemory" [block]
    14  read  "SLI_SectorMemory" [file]
    15  write "SLI_SectorMemory" [file]

  Enter number [0..15]:
```

```
>>> MuonSectorProcessor main menu <<<

     0  quit
     1  CFG menu
     2  SYS menu
     3  GT menu
     4  SLI menu
     5  SLM menu
     6  TTC menu

  Enter number [0..6]:
```

```
int MuctpiModule::SLISpyEnable() {

  MUONSECTORPROCESSOR_SLI_SPYPLAYCONTROL_BITSTRING bs;

  if((m_status = m_msp->SLI_SpyPlayControl_Read(bs)) != SUCCESS) {
    return(m_status);
  }
  bs.SpyEnable(true);
  bs.Mode("SPY");
  if((m_status = m_msp->SLI_SpyPlayControl_Write(bs)) != SUCCESS) {
    return(m_status);
  }
  return(m_status);
}
```

Python hardware test scripts

- **Python** is used for **testing** by firmware designers:
  - simpler interactive testing of new firmware versions
  - development of complex test sequences
  - iterative development process
- **C++ software** can be called from **Python** using **SWIG** (Simplified Wrapper and Interface Generator) tool
- SWIG automatically generates **python wrappers** from the C++ header files and a simple interface file created by the user:
  - Automatically generated C++ classes with registers/memories read/write methods, register bit strings
  - User written C++ code

```
<block name="SLI"                      addr="0x00000000"  decoder="SLR1">
  <register name="SpyPlayControl"      addr="0x00000008" modf="RW">
    <field name="SpyEnable"            mask="0x00000004" form="BOOLEAN"/>
    <field name="PlaybackEnable"       mask="0x00000008" form="BOOLEAN"/>
    <field name="PlaybackLastAddress"  mask="0x0000FFF0" form="NUMBER"/>
    <field name="Mode"                 mask="0x00000003">
      <value name="OFF"                data="0x00000000"/>
      <value name="SL"                 data="0x00000001"/>
      <value name="PG"                 data="0x00000002"/>
      <value name="SPY"                data="0x00000003"/>
    </field>
  </register>
  <register name="BcidMonitor"         addr="0x00000004" modf="R"/>
```

```python
# import swig wrapper module
import MuctpiModule

# create Muctpi object
m = MuctpiModule.MuctpiModule(0)

# low-level functions from XML

# read a register from multiple block as an integer
stat, data = m.msp[0].SLI_BcidMonitor_Read()

# create a bitstring object
bs = MuctpiModule.SLI_SPYPLAYCONTROL_BITSTRING()

# read a register as a bitstring
m.SLI_SpyPlayControl_Read(bs)

# set bit fields
bs.PlaybackEnable(True)
bs.PlaybackLastAddress(22)
bs.Mode("PG")

# write a register as a bitstring
m.SLI_SpyPlayControl_Write(bs)

# C++ function in user written code

# reset the board
m.MuctpiModuleReset()
```

Summary

- **HardwareCompiler** generates VHDL code and low-level C++ hardware access functions from a common XML description
- Maintains **consistency** between firmware and software development
- **Speeds up development and testing**, low-level software is **automatically** adjusted to changes in the firmware
- Generated C++ code used in higher-level software to interface with the ATLAS TDAQ run control system
- **HardwareCompiler used** for all ATLAS L1CT modules running in the experiment and currently being developed

Interested in further information on the L1CT HardwareCompiler? Don't hesitate to reach out to us!

email: anna.kulinska@cern.ch, ralf.spiwoks@cern.ch

Call for discussion!

Managing large number of registers and memories is a common challenge for complex FPGA designs. We would also like to hear others' experiences in this field

Thank you for your attention!

Questions?

VHDL package for registers includes:

CONSTANTS

NAME	DATA TYPE
<REG_NAME>_ADDR	std_logic_vector for “multiple” its array
<REG_NAME>_WIDTH	integer
<REG_NAME>_<FIELD_NAME>_WIDTH	integer
<REG_NAME>_<FIELD_NAME>_SHIFT	integer
<Reg_Name>_<Field_Name>_<VALUE>	std_logic_vector

SUBTYPES AND TYPES

NAME	DATA TYPE
<Reg_Name>_slv_t	std_logic_vector(DATA_WIDTH - 1 downto 0)
<Reg_Name>_reg_t	record of bitfields

FUNCTIONS

NAME	DATA TYPE CONVERSION
reg2slv	from record of bitfields to std_logic_vector
slv2reg	from std_logic_vector to record of bitfields

```
constant TTC_COUNTERSYNC_ADDR   : addr_slv_t := 28x"00100008";
constant TTC_COUNTERSYNC_WIDTH  : integer := 12;
```

```
subtype TTC_CounterSync_slv_t is std_logic_vector(TTC_COUNTERSYNC_WIDTH - 1 downto 0);
subtype TTC_CounterSync_reg_t is TTC_CounterSync_slv_t;
```

```
constant SLI_SPYPLAYCONTROL_ADDR   : addr_slv_t := 28x"00000008";
constant SLI_SPYPLAYCONTROL_WIDTH  : integer := 16;

-- Register "SpyPlayControl", bitfield "Mode"
constant SLI_SPYPLAYCONTROL_MODE_WIDTH  : integer := 2;
constant SLI_SPYPLAYCONTROL_MODE_SHIFT  : integer := 0;

-- Register "SpyPlayControl", bitfield "Mode", value constants
subtype SLI_SpyPlayControl_Mode_t is std_logic_vector(SLI_SPYPLAYCONTROL_MODE_WIDTH - 1 downto 0);
constant SLI_SpyPlayControl_Mode_OFF : SLI_SpyPlayControl_Mode_t := "00";
constant SLI_SpyPlayControl_Mode_SL : SLI_SpyPlayControl_Mode_t := "01";
constant SLI_SpyPlayControl_Mode_PG : SLI_SpyPlayControl_Mode_t := "10";
constant SLI_SpyPlayControl_Mode_SPY : SLI_SpyPlayControl_Mode_t := "11";

-- Register "SpyPlayControl", bitfield "SpyEnable"
constant SLI_SPYPLAYCONTROL_SPYENABLE_SHIFT  : integer := 2;

-- Register "SpyPlayControl", bitfield "PlaybackLastAddress"
constant SLI_SPYPLAYCONTROL_PLAYBACKLASTADDRESS_WIDTH  : integer := 12;
constant SLI_SPYPLAYCONTROL_PLAYBACKLASTADDRESS_SHIFT  : integer := 4;
```

```
subtype SLI_SpyPlayControl_slv_t is std_logic_vector(SLI_SPYPLAYCONTROL_WIDTH - 1 downto 0);

type SLI_SpyPlayControl_reg_t is record
   Mode : SLI_SpyPlayControl_Mode_t;
   SpyEnable : std_logic;
   PlaybackEnable : std_logic;
   PlaybackLastAddress : std_logic_vector(SLI_SPYPLAYCONTROL_PLAYBACKLASTADDRESS_WIDTH - 1 downto 0);
end record SLI_SpyPlayControl_reg_t;
```

```
function reg2slv (reg : SLI_SpyPlayControl_reg_t) return SLI_SpyPlayControl_slv_t;
function slv2reg (slv : SLI_SpyPlayControl_slv_t) return SLI_SpyPlayControl_reg_t;
```

VHDL package for memories includes:

```
constant SLI_SECTORMEMORY_SIZE    : addr_t     := 16#00004000#;
constant SLI_SECTORMEMORY_ADDR    : addr_slv_t := 28x"00010000";

constant SLI_SECTORMEMORY_ADDR_MASK  : addr_slv_t := 28x"0fff0000";
constant SLI_SECTORMEMORY_ADDR_WIDTH : integer := 16;
constant SLI_SECTORMEMORY_DATA_WIDTH : integer := 32;
```

CONSTANTS

NAME	DATA TYPE
<MEM_NAME>_SIZE	integer
<MEM_NAME>_ADDR	std_logic_vector for “multiple” its array
<MEM_NAME>_ADDR_MASK	std_logic_vector
<MEM_NAME>_ADDR_WIDTH	integer
<MEM_NAME>_DATA_WIDTH	integer

```
subtype SLI_SECTORMEMORY_addr_t is std_logic_vector(SLI_SECTORMEMORY_ADDR_WIDTH - 1 downto 2);
subtype SLI_SECTORMEMORY_data_t is std_logic_vector(SLI_SECTORMEMORY_DATA_WIDTH - 1 downto 0);

type SLI_SectorMemory_mem_t is array (0 to SLI_SECTORMEMORY_SIZE - 1) of SLI_SECTORMEMORY_data_t;
```

SUBTYPES AND TYPES

```
type SLI_SectorMemory_mosi_t is record
  addr  : SLI_SECTORMEMORY_addr_t;
  wdata : SLI_SECTORMEMORY_data_t;
  wren  : std_logic;
end record;

type SLI_SectorMemory_miso_t is record
  rdata : SLI_SECTORMEMORY_data_t;
end record;
```

NAME	DATA TYPE
<Mem_Name>_addr_t	std_logic_vector(ADDRESS_WIDTH - 1 downto 0)
<Mem_Name>_data_t	std_logic_vector(DATA_WIDTH - 1 downto 0)
<Mem_Name>_mem_array_t	array of data_t std_logic_vectors
<Mem_Name>_mosi_t	memory write record
<Mem_Name>_miso_t	memory read record

VHDL package for blocks and decoders:

BLOCK'S TYPES

NAME	DATA TYPE
<BLOCK_NAME>_miso_blk_t	record with all block's read-only registers
<BLOCK_NAME>_mosi_blk_t	record with all block's writable registers
<BLOCK_NAME>_mem_miso_blk_t	record with all block's memories read records
<BLOCK_NAME>_mem_mosi_blk_t	record with all block's memories write records

DECODER'S TYPES

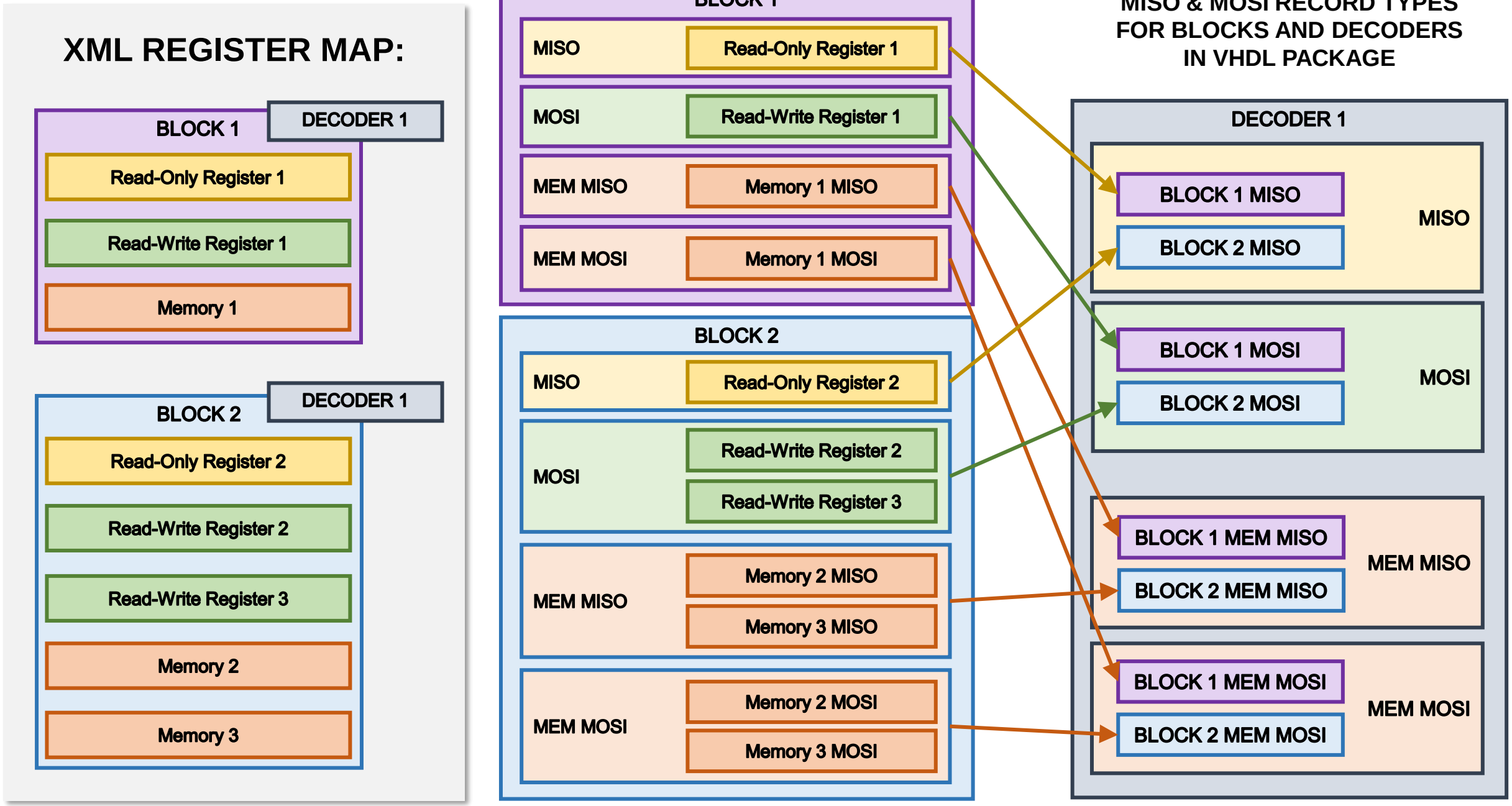
NAME	DATA TYPE
<DECODER_NAME>_miso_blk_t	record with all decoder's blocks with read-only registers
<DECODER_NAME>_mosi_blk_t	record with all decoder's blocks with writable registers
<DECODER_NAME>_mem_miso_blk_t	record with all decoder's blocks with memories miso records
<DECODER_NAME>_mem_mosi_blk_t	record with all decoder's blocks with memories mosi records

```
-----------------------------------------
-- Block "TTC" type(s)
-----------------------------------------
type TTC_miso_blk_t is record
  CounterL1a : TTC_CounterL1a_reg_t;
  CounterSync : TTC_CounterSync_reg_t;
end record TTC_miso_blk_t;

type TTC_mosi_blk_t is record
  Control : TTC_Control_reg_t;
end record TTC_mosi_blk_t;
```

```
-----------------------------------
-- Decoder "SYS" type(s)
-----------------------------------
type SYS_miso_blk_t is record
  TTC : TTC_miso_blk_t;
end record SYS_miso_blk_t;

type SYS_mosi_blk_t is record
  TTC : TTC_mosi_blk_t;
end record SYS_mosi_blk_t;
```

```
-----------------------------------------
-- Block "SLI" type(s)
-----------------------------------------
type SLI_miso_blk_t is record
  SpyPlayStatus : SLI_SpyPlayStatus_reg_t;
  CdcStatus : SLI_CdcStatus_reg_t;
end record SLI_miso_blk_t;

type SLI_mosi_blk_t is record
  SpyPlayControl : SLI_SpyPlayControl_reg_t;
  CdcControl : SLI_CdcControl_reg_t;
end record SLI_mosi_blk_t;

type SLI_mem_miso_blk_t is record
  SectorMemory : SLI_SectorMemory_miso_t;
end record SLI_mem_miso_blk_t;

type SLI_mem_mosi_blk_t is record
  SectorMemory : SLI_SectorMemory_mosi_t;
end record SLI_mem_mosi_blk_t;
```

```
-----------------------------------
-- Decoder "SLR1" type(s)
-----------------------------------
type SLR1_miso_blk_t is record
  SLI : SLI_miso_blk_t;
end record SLR1_miso_blk_t;

type SLR1_mosi_blk_t is record
  SLI : SLI_mosi_blk_t;
end record SLR1_mosi_blk_t;


type SLR1_mem_miso_blk_t is record
  SLI : SLI_mem_miso_blk_t;
end record SLR1_mem_miso_blk_t;

type SLR1_mem_mosi_blk_t is record
  SLI : SLI_mem_mosi_blk_t;
end record SLR1_mem_mosi_blk_t;
```


XML REGISTER MAP:
BLOCK 1
DECODER 1
Read-Only Register 1
Read-Write Register 1
Memory 1
BLOCK 2
DECODER 1
Read-Only Register 2
Read-Write Register 2
Read-Write Register 3
Memory 2
Memory 3
BLOCK 1
MISO
Read-Only Register 1
MOSI
Read-Write Register 1
MEM MISO
Memory 1 MISO
MEM MOSI
Memory 1 MOSI
BLOCK 2
MISO
Read-Only Register 2
MOSI
Read-Write Register 2
Read-Write Register 3
MEM MISO
Memory 2 MISO
Memory 3 MISO
MEM MOSI
Memory 2 MOSI
Memory 3 MOSI
MISO & MOSI RECORD TYPES
FOR BLOCKS AND DECODERS
IN VHDL PACKAGE
DECODER 1
BLOCK 1 MISO
BLOCK 2 MISO
MISO
BLOCK 1 MOSI
BLOCK 2 MOSI
MOSI
BLOCK 1 MEM MISO
BLOCK 2 MEM MISO
MEM MISO
BLOCK 1 MEM MOSI
BLOCK 2 MEM MOSI
MEM MOSI

Translator example

```
read_p : process(clk_i) is
   variable v_rd_data : std_logic_vector(31 downto 0);
   variable v_mem_miso_rdata : SLR1_mem_miso_blk_t;
 begin
   if rising_edge(clk_i) then

     -- default read data assignment
     v_rd_data := (others => '0');

     -- address pipelined to synchronise registers and memory reads
     addr_reg <= m_addr(addr_reg'range);

     -- SLI_SpyPlayStatus
     SLI_SpyPlayStatus_rd_sel <= addr_reg ?= SLI_SPYPLAYSTATUS_ADDR(addr_reg'range);
     reg_miso_rdata.SLI.SpyPlayStatus <= slv2reg(reg2slv(reg_miso_i.SLI.SpyPlayStatus) and SLI_SpyPlayStatus_rd_sel);
     v_rd_data(SLI_SpyPlayStatus_slv_t'range) := v_rd_data(SLI_SpyPlayStatus_slv_t'range) or reg2slv(reg_miso_rdata.SLI.SpyPlayStatus);

     -- SLI_SpyPlayControl
     SLI_SpyPlayControl_rd_sel <= addr_reg ?= SLI_SPYPLAYCONTROL_ADDR(addr_reg'range);
     reg_mosi_rdata.SLI.SpyPlayControl <= slv2reg(reg2slv(reg_mosi.SLI.SpyPlayControl) and SLI_SpyPlayControl_rd_sel);
     v_rd_data(SLI_SpyPlayControl_slv_t'range) := v_rd_data(SLI_SpyPlayControl_slv_t'range) or reg2slv(reg_mosi_rdata.SLI.SpyPlayControl);

     -- SLI_SectorMemory
     SLI_SectorMemory_rd_sel <= m_addr(ADDR_BITS - 1 downto SLI_SECTORMEMORY_ADDR_WIDTH) ?= SLI_SECTORMEMORY_ADDR(ADDR_BITS - 1 downto SLI_SECTORMEMORY_ADDR_WIDTH);
     v_mem_miso_rdata.SLI.SectorMemory.rdata := mem_miso_i.SLI.SectorMemory.rdata and SLI_SectorMemory_rd_sel;
     v_rd_data(SLI_SECTORMEMORY_data_t'range) := v_rd_data(SLI_SECTORMEMORY_data_t'range) or v_mem_miso_rdata.SLI.SectorMemory.rdata;

     -- rd_data assignment
     m_rddata <= v_rd_data;

   end if;
 end process read_p;
```

SINGLE READ-ONLY REGISTER WITHOUT FIELDS

SINGLE READ&WRITE REGISTER WITHOUT FIELDS

SINGLE MEMORY