The advantages of the usage of the record data type in the synthesizable HDL code

Jakub Šťastný
jakub.stastny@asicentrum.com
ASICentrum spol. s r.o.

1 Introduction

Well decomposed digital design often possesses a hierarchical structure composed of a lot of smaller, hierarchically organized, blocks. Such a design structure is natural and very practical. Why? The initial specification of the product views the designed system as one compact entity (a black box) and describes its behavior and its functions from the external perspective. This type of product description is useful for the initial definition of the design goals and facilitates understanding of the proposed system by its future user. However, such a specification is not practical as a design specification since it is not possible to deal with the system described in a "monolithical" way. Parameters of the future product as well as of the design project cannot be estimated without decomposition of the product into smaller parts; the same holds true also for the assignments of the design tasks to the design team members. Because of this we decompose the product (here digital design) during the system level into subblocks connected with suitable interfaces. A result of the decomposition is the described hierarchical structure of the system with internal blocks connected by many interfaces passing across the design hierarchies.

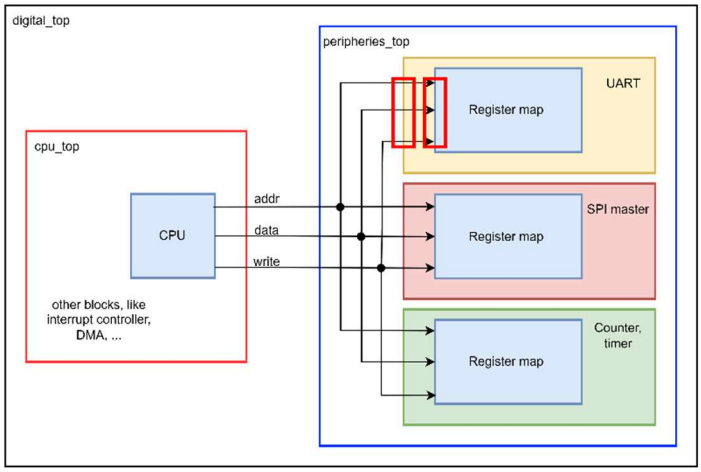

Figure 1: Block diagram of the demonstration system.

An example of such a structure is the design of a microcomputer system depicted in the Figure 1. It includes a Central Processing Unit (CPU), instantiated in its own component called *cpu_top* along with all its supporting

circuits (DMA, interrupt controller, and others). Further, on the right side of the Figure 1 we can see a few peripherals grouped together in the block *peripheries_top*. Peripherals are controlled via their own local register maps, as is common with microcomputers. To allow the application running on the CPU to control the peripherals, each of the register maps is connected to the CPU via a bus allowing the CPU to read/write data from/to the register map by instructions executed by the microprocessor. For the sake of clarity the figure outlines only the write part of the bus, description of its signals can be found in the Table 1.

Signal	**Width**	**Purpose**
addr	32	Address to write the data to or read from.
data	32	Data to be written.
write	1	Write control signal; register map is updated with the clock rising edge. If *write* = '1', word on the *data* bus will be written to the address currently present at the *addr* bus.

Table 1: Description of the write interface to the register maps.

There are plenty of interfaces connecting blocks in a typical hierarchically structured system and this brings one small disadvantage. To see it have a look at Listing 1 with an excerpt of the VHDL RTL code for the *peripheries_top* block. The listing contains only the parts of the code describing the write register bus connecting the CPU and the peripherals. Note in the listing that the RTL code of the block *peripheries_top* contains the write bus signals repeated seven times. All these occurrences you can for sure quickly pinpoint in the code (examine Listing 1 along with the explanations):

- one occurrence is in the *peripheries_top* block,
- three occurences are in the components of the peripherals,
- finally, three occurences are in the instances of the peripheral blocks.

Looking further at the RTL code of the peripherals (UART, SPI master, counter and timer – see Figure 1), we can see that in each of them the write bus is present at the peripheral interface, register map interface, in the component of the register map, in the architecture block of the peripheral, and in the connecting RTL code (see UART block in the Figure 1 in red) – there are four occurrences per one peripheral, twelve for all three of them. Then on the CPU side (again see Figure 1, the left part) is this interface present at the CPU entity, *cpu_top* block entity, at the CPU component in the *cpu_top* block architecture, and in the RTL code responsible for the connections. Finally, the reader can now very likely accept without the need of an elaborate proof that we can find the interface in the

source codes five more times on the *dig_top* level (try to find all of the occurrences and do not forget that one of them is again in the RTL implementing the connections of the bus signals).

What does it mean for us? In case we want to extend the CPU write interface by one more signal (e.g, *wait_s* indicating the bus wait state), we have to manually change interface at 7+12+4+5 = 28 places in the code; to be precise:

- in one place we need to implement the logic driving the new signal – at its "source", which is here in the processor unit (CPU block in the Figure 1). This is a creative design work – implementation of the functionality.
- in three places we need to implement the code to handle the new signal as needed – in the peripheral register maps (Register map blocks in the Figure 1). We are solving a creative task here as well, the newly added signal must be properly used in all the peripheral blocks.
- finally, we have to manage correct feed-through of the new signal through the block interfaces **in 24 distinctive places** in the code. In this case we are doing more or less repetitive and quite dull work – we have to fix one interface after the other until we add the new signal everywhere and all is "clean".

Let us have a closer look at the fact that we have to add new signal to 24 places in the RTL code, everywhere the same, and we need to avoid injecting a new bug during this. This is not as trivial as it seems even though that some modern HDL code editors can slightly help here with the refactorization function "Add New Port to Module". The easiest bug you can do here is to forget either to update one of the interfaces or the whole peripheral block. Discovering such a bug by verification is not an easy task. In addition to this, the poor designer can have a feeling that s/he does not deserve such a repetitive work and that it should be all doable in some simpler way; we have to admit that the designer is right in this point.

```
ENTITY peripheries_top IS
  PORT (
       … block signals …
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
  );
END ENTITY
ARCHITECTURE rtl OF peripheries_top is

  COMPONENT uart IS
    PORT (
       … block signals …
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
    )
  END COMPONENT uart;
```

```
  COMPONENT spi_master IS
    PORT (
       … block signals …
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
    )
  END COMPONENT spi_master;

  COMPONENT counter_timer IS
    PORT (
       … block signals …
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
    )
  END COMPONENT counter_timer;

BEGIN
  i_uart : uart
  PORT MAP (
    … other signals as needed …
    addr  => addr,
    data  => data,
    write => write,
    …
  );

  i_spi_master : spi_master
  PORT MAP (
    … other signals as needed …
    addr  => addr,
    data  => data,
    write => write,
    …
  );

  i_counter_timer : counter_timer
  PORT MAP (
    … other signals as needed …
    addr  => addr,
    data  => data,
    write => write,
    …
  );
END ARCHITECTURE rtl;
```

Listing 1: Excerpts of the source code of the system depicted in the Figure 2.

2 Record data type

Before we will have a look at how to solve the problem of our poor designer, let us speak a bit about the record data type. Data type *record* (and its equivalents, e.g., *struct* in C language or SystemVerilog) is a composite data type commonly available in all the modern programming languages. Its basic purpose is to gather together data items which are mutually related to each other by their purpose or their meaning (see also e.g., [1, page 36], [2]), each of the items can be of a different data type. We can easily declare signals, variables, or constants of the record type in VHDL, arrays with items of record type and records can be further hierarchically composed. For some examples see Listing 2. Signals of the record type can be also used at the input/output interface of any block, which will be of an interest for us.

```
TYPE T_REG_WRITE IS RECORD
  addr  : std_logic_vector (31 DOWNTO 0);
  data  : std_logic_vector (31 DOWNTO 0);
  write : std_logic;
END RECORD T_REG_WRITE;

CONSTANT C_REG_WR_RESET : T_REG_WRITE := (
   addr  => (OTHERS => '0'),
   data  => (OTHERS => '0'),
   write => '0'
 );


TYPE T_WRITE_ARR IS ARRAY (0 TO 3) OF T_REG_WRITE;

SIGNAL t_wr_bus : T_REG_WRITE;
```

Listing 2: Examples of the record type declaration, signal of the record type, constant, and type with elements of the record type.

3 Record data type on a block interface

Let us now have a look on how to facilitate designer's work. Actually, we do not need a lot. We need a construct which will encapsulate the repeating part of the interface and allow to quickly reuse it as needed. The reader is for sure now already expecting that the record data type will help us. Let us go directly to the point: using it we will rework the design to get the schematic as in Figure 2. Equivalent RTL code is then in the Listing 3; compare its length with what was presented in the Listing 2. Let us now summarize how much work it is to add signal *wait_s* to the write bus and compare with the previous case. The designer must now instead of changing 28 places in the code:

- first, extend the declaration of the *record* data type by adding the new signal in one place. Despite the fact that there are still 7 places in the code where is the write bus used, the declaration is only one – placed in our case in the package in the file *reg_bus_pkg.vhd* (to place the record type definition into the VHDL package is useful as it allows to share it among the design entities). Adding the *wait_s* signal thus means to modify only one place in the code, now.
- second, we have to design the control logic to drive the new signal in one place – „at its source", here in the CPU unit (CPU block in the Figure 2),
- third, we have to design the processing of the newly added signal in three places – in the peripheral register maps (blue blocks „Register map" in the Figure 2). In this as well as in the previous cases the designer does a creative work.

And that is all. Prior to the change we had to ensure the right handling of the new signal in 24 distinct locations, where the signal was crossing the boundaries of the blocks. This is now handled automatically by the update of

the data type declaration in the first step; the change is spread everywhere automatically as the compiler plainly uses the updated definition of the data type.

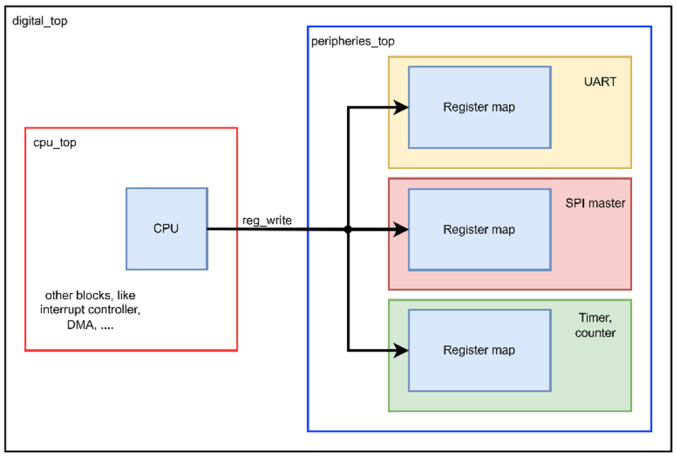

Figure 2: A block diagram of the demonstrational system refactored to encapsulate the write bus into the record data type.

reg_bus_pkg.vhd:

```
TYPE T_REG_WRITE IS RECORD
  addr  : std_logic_vector (31 DOWNTO 0);
  data  : std_logic_vector (31 DOWNTO 0);
  write : std_logic;
  wait_s: std_logic;
END RECORD T_REG_WRITE;
```

peripheries_top.vhd:

```
ENTITY peripheries_top IS
  PORT (
       … block ports …
       reg_write : IN T_REG_WRITE;
       … block ports …
  );
END ENTITY
ARCHITECTURE rtl OF peripheries_top is

  COMPONENT uart IS
    PORT (
       … block ports …
       reg_write : IN T_REG_WRITE;
       … block ports …
    )
  END COMPONENT uart;

  COMPONENT spi_master IS
    PORT (
       … block ports …
       reg_write : IN T_REG_WRITE;
       … block ports …
    )
  END COMPONENT spi_master;

  COMPONENT counter_timer IS
    PORT (
       … block ports …
       reg_write : IN T_REG_WRITE;
       … block ports …
    )
  END COMPONENT counter_timer;

BEGIN
  i_uart : uart
  PORT MAP (
    … block ports …
```

```
    reg_write => reg_write,
    … block ports …
  );

  i_spi_master : spi_master
  PORT MAP (
    … block ports …
    reg_write => reg_write,
    … block ports …
  );

  i_counter_timer : counter_timer
  PORT MAP (
    … block ports …
    reg_write => reg_write,
    … block ports …
  );
END ARCHITECTURE rtl;
```

Listing 3: Refactored code of the peripheries_top entity using the signals of the record data type.

4 Implementation of registers

Usage of the signals of the *record* data type facilitates designer's work also under other circumstances. It is quite practical when we implement a register. If we need to register the original write bus (as in the Listing 1) with the clock rising edge (flip-flop), a naïve approach leads to the VHDL code in the Listing 4.

```
SIGNAL addr_int : std_logic_vector (31 DOWNTO 0);
SIGNAL data_int : std_logic_vector (31 DOWNTO 0);
SIGNAL write_int: std_logic;

output_reg:PROCESS (res_n, clk)
BEGIN
  IF res_n = '0' THEN
    addr  <= (OTHERS => '0');
    data  <= (OTHERS => '0');
    write <= '0';
  ELSIF rising_edge(clk) THEN
    addr  <= addr_int;
    data  <= data_int;
    write <= write_int;
  END IF;
END PROCESS output_reg;
```

Lising 4: Register on the write bus, the naive RTL design approach.

We can simplify the code significantly if we refactor the code and merge three distinctive signals into one record, see Listing 5.

```
SIGNAL reg_wr_int : T_REG_WRITE;

output_reg_2: PROCESS (res_n, clk)
BEGIN
  IF res_n = '0' THEN --(#)
    reg_wr.addr  <= (OTHERS => '0');
    reg_wr.data  <= (OTHERS => '0');
    reg_wr.write <= '0';
  ELSIF rising_edge(clk) THEN
    reg_wr <= reg_wr_int; --(*)
  END IF;
END PROCESS output_reg_2;
```

Listing 5: The first version of the RTL code refactored to benefit from the record data type.

This already brings a positive contribution: if we change the record datatype declaration, the change will be automatically reflected on line **(*)** and the registers will be implemented according to the current declaration. However, we still will have to hand-modify the process. Why? Have a look at the code handling the asynchronous reset marked with (#): if we add the following code to the record the new signal *wait_s*, we will have to add to the process under the line marked with (#)

```
reg_wr.wait_s <= '0';
```

to handle the proper reset of the signal to the logic 0. If we forget the initialization, synthesis tool will drive the signal *wait_s* using a register without asynchronous reset and this can make problems in the design. In such a case only meticulous verification can save us – testcases can (not necessarily) catch the consequence of the fact that the signal *wait_s* will be in the 'U' state after the start of the RTL simulation or not reset together with other signals. We avoid this in an elegant way by a declaration of a constant of the record type. The constant will define the reset values (inactive state) at the register output. The whole solution is in the Listing 6.

```
CONSTANT C_REG_WR_RES : T_REG_WRITE := (
  addr => (OTHERS => '0'),
  data => (OTHERS => '0'),
  write => '0'
);

SIGNAL reg_wr : T_REG_WRITE;

output_reg_3:PROCESS (res_n, clk)
BEGIN
  IF res_n = '0' THEN
    reg_wr <= C_REG_WR_RES;
  ELSIF rising_edge(clk) THEN
    reg_wr <= reg_wr_int;
  END IF;
END PROCESS output_reg_3;
```

Listing 6: Usage of a constant of the record type to simplify the implementation of the reset.

Now if we need to change the record type we just need to change the corresponding constant defining the reset state and the register will be "updated automatically" with the next compilation or synthesis. The beauty of this approach lies also in the fact that the compiler watches us: if the designer changes the type declaration and forgets to change the constant, compilation will end up with an error and the designer will have to fix the design. As an example, the Vivado compilator will report

```
ERROR: [VRFC 10-3717] some record elements are missing in this aggregate of
't_reg_write' [record_logic_demo.vhd:49]
```

At this moment we are automatically notified of all the places where we need to do the RTL change, which significantly reduces a risk of a bug injection as a result of a design change.

5 Combinatorial functions

The reader will not be surprised if we write that there are other language constructs which are normally dealing with simpe signals behaving in such a “friendly” way. Signals of the record type we can easily multiplex, see Listing 7, process *comb_func*. Similarly we can implement gating of the signal to its inactive values using the pre-declared constant, see also listing 7, line marked with (#). Last, but not least, using a simple construction sketched in the process *overwrite_write* you can change the value of only one item in the signal of the record data type.

```
SIGNAL reg_wr_0 : T_REG_WRITE;
SIGNAL reg_wr_1 : T_REG_WRITE;
SIGNAL sel : std_logic;

SIGNAL write_2  : std_logic;
SIGNAL reg_wr_2 : T_REG_WRITE


comb_func: PROCESS (sel, reg_rw_0, reg_rw_1)
BEGIN
  IF sel = '1' THEN
    out_wr <= reg_rw_0;
  ELSE
    out_wr <= reg_rw_1;
  END IF;
END PROCESS comb_func;

out_wr <= reg_rw_0 WHEN disable = '0' ELSE C_REG_WR_RES; --(#)

overwrite_write : PROCESS (reg_wr_0, write_2)
BEGIN
  reg_wr_2       <= reg_wr_0;
  reg_wr_2.write <= write_2;
END PROCESS overwrite_write;
```

Listing 7: Some other ways how to deal with signals of the record type.

6 Caveats

Advantages of the usage of the record data type were thouroughly described in the previous text. It looks like that the designer here only gains a lot and does not lose anything. We can ask then – does the usage of records bring also some drawbacks?

First, some VHDL language constructs (mainly from the newer versions of the VHDL language standard) are sometimes not fully supported in the current synthesizers and simulation tools. The good news here is that the record datatype is fully supported by all the up-to-date tools we know about and routinely using. Thus, there is no drawback, here.

Second, it is better to avoid usage of the signals of the record type at the top level interface of the design hierarchy, see Figure 3, block *DUT* (Design Under Test), all interfaces and signals marked in blue color. If you do this and use such a signal here, you will have to maintain two versions of the instance of the DUT entity in the *tb_top* entity

(*testbench top*). This is caused by the need to do simulation (design verification) on the RTL level (the first version of the instance in the *tb_top*) as well as on the gate level (the second version of the DUT instace in the *tb_top* block) where we need to instantiate netlist of the DUT annotated by the SDF file instead of the DUT RTL implementation (see also [3] and link on the page [4]). Netlist is generated from the RTL code by the physical implementation (synthesis and place and route) and describes the real physical schematic of the design. The signal of the record data type is replaced in such a schematic representation by its real physical implementation – i.e., every item of the record datatype is replaced by a standalone signal. This does not bring any problem if the signal of a record type is used inside the DUT; however, if we use such a signal at the DUT top interface, then the interface implementations will differ between the RTL design and netlist. This will cause some small (easily solvable) complications at the moment when we want to instantiate the DUT block in the testbench top entity *tb_top*. The issue can be solved using the VHDL construction in the Listing 8.

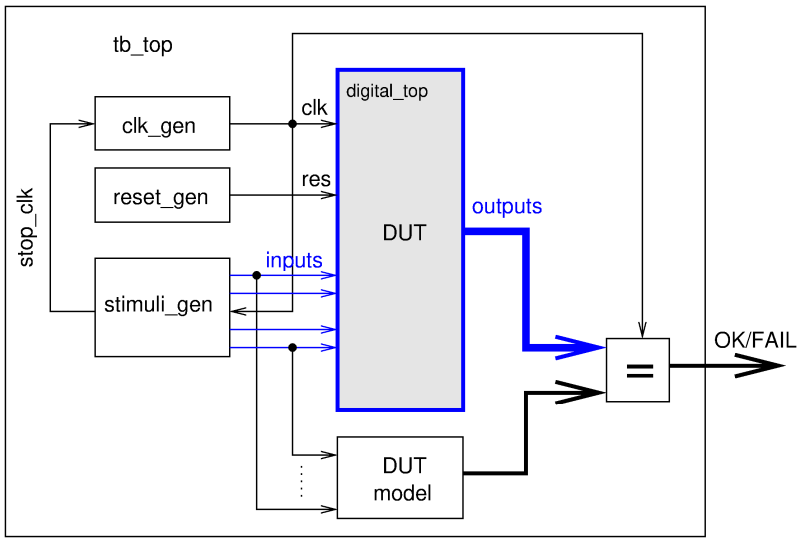

Figure 3: An example of a structure of the verification environment of a digital design. Signals and interfaces where there is not recommended to use records are marked in blue.

Third, some small complications also arise if the signal of the record type is used together with the hierarchical names construction [5, 6] (and some proprietary tools like *SignalSpy* in the Siemens simulators or *nc_mirror* in the simulators from Cadence). Such a language construct allows the testcases and modules in the verification environment to look into the internal implementation of the block. We need this in many cases during the block verification, typically when some block internal signal carries an important information on the state of the design. And we have here the same problem with the structure of the block differing between the RTL and the gate level implementations – again we will have to maintain two ver sions of the testbench and testcases code, one for the

RTL and the other for the gate level simulations. Also here we can easily solve the problem by a small modification of the already presented IF-GENERATE construct generating both mappings, see the Listing 8.

Fourth, advisable is to group into one signal of the record type signals which are related to each other by similar meaning – e.g., in our case we grouped together signals which form the write bus of the CPU perihperal bus. Creating of a "superrecord" which contains plenty of signals which are not mutually related and have in common only the fact that they occur at the same block interface would make the code hard to understand.

```
ENTITY dut IS
  GENERIC (
    G_GATE_SIMS : natural := 0; -- 0 - RTL, 1 - hradlová úroveň
  )
  PORT ( … )
END ENTITY dut;

ARCHITECTURE rtl OF dut IS
…
BEGIN

-- connections for the RTL level
gen_rtl : IF G_GATE_SIMS = 0 GENERATE
  i_dut : dut_rtl
    PORT MAP (
…
    )
END GENERATE gen_rtl;

-- connections for the gate level
gen_gate : IF G_GATE_SIMS = 1 GENERATE
  i_dut : dut_gate
    PORT MAP (
…
    )
END GENERATE gen_gate;

END ARCHITECTURE rtl;
```

Listing 8: IF – GENERATE VHDL construct.

7 Conclusions

This paper summarizes the contributions as well as potential drawbacks of the usage of the record data type in the HDL languages. Designer can save a lot of dull and monotonous work by using the record datatype, write better and more readable code, and prevent injection of some rather unpleasant bugs. In addition to this, usage of the record datatype does not bring any extra implementation effort and is supported by all the tools known to the author without limitations. To conclude, we can only recommend to the reader to use the record data type wherever appropriate in his/her future designs.

8 Acknowledgement

It is my pleasant duty to thank to my wonderful wife Julia and colleagues Robert Kvaček, Tomáš Novák, and Luboš Hradecký for their language, stylistic, and content review and great comments to the text.

Used abbreviations

ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DMA	Direct Memory Access
DUT	Design Under Test
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IP	Intellectual Property
RTL	Register Transfer Level
tb_top	TestBench TOP

References for further study

[1] PINKER Jiří, POUPA Martin. *Číslicové systémy a jazyk VHD*L (Digital systems and the VHDL Language, in Czech). BEN Praha 2006

[2] NAND Land [online]. Records – VHDL Example [cit 25.3.2024]. Available at https://nandland.com/record/

[3] ŠŤASTNÝ, Jakub. Simulace číslicových obvodů na hradlové úrovni (Gate level simulations of the digital circuits, in Czech). *DPS Elektronika od A do Z,* květen/červen 2015, s 8-11.

[4] ŠŤASTNÝ, Jakub. Minimized Logic. [online; cit 25.3.2024]. Available at www.minimizedlogic.com

[5] DOULOS [online]. VHDL-2008: Easier to use, Hierarchical names [cit 25.3.2024]. Available at https://www.doulos.com/knowhow/vhdl/vhdl-2008-easier-to-use/#hierarchicalnames

[6] CHIPVERIFY [online]. Verilog Hierarchical Reference Scope [cit 25.3.2024]. Available at https://www.chipverify.com/verilog/verilog-hierarchical-reference-scope