

Common Library Memory and Register Component Descriptions

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Eric Kooistra	ASTRON	
Controle / Checked: Andre Gunst	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: Handtekening / Signature	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 Daniel van der Schuur Rajan Raj Thilak Jonathan Hargreaves (JIVE) Salvatore Pirruccio (JIVE)	

Document history:

Revision	Date	Author	Modification / Change
0.1	2010-09-9	Eric Kooistra	Draft.
0.2	2010-11-03	Eric Kooistra	Updated.

Table of contents:

1	Introduction.....	5
2	Memory	6
2.1	IP	6
2.2	Wrapper component.....	6
3	Register	9
4	Memory mapped interface	10
5	Clock domain crossing	11
6	Dual page	14
7	Conclusion.....	15

Terminology:

FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IO	Input Output
IP	Intellectual Property
MISO	Master In Slave Out
MM	Memory-Mapped
MOSI	Master Out Slave In
Nof	Number of
PIO	Parallel IO
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SOPC	System On a Programmable Chip (Altera)
ST	Streaming
UNB	Path to UniBoard Firmware directory (https://svn.astron.nl/UniBoard_FP7/UniBoard/trunk)

References:

1. www.altera.com
2. www.xilinx.com
3. \$UNB/modules/common/doc/common.txt
4. \$UNB/doc/howto/How_to_use_MegaWizard.txt
5. "Avalon Interface Specifications", mnl_avalon_spec.pdf, www.altera.com
6. "Specification for module interfaces using VHDL records", ASTRON-RP-380, Eric Kooistra
7. \$UNB/doc/howto/How_to_use_SOPC_Component_Editor.txt

1 Introduction

Any somewhat more elaborate digital logic design needs some memory and some registers. The difference between memory and registers is that for a register the whole content is available in parallel, whereas for a memory only one word is available at a time. This document describes the generic memory and register components that are available in the common library and some usage examples. The common library is located in the UniBoard RadioNET FP7 SVN repository at:

https://svn.astron.nl/UniBoard_FP7/UniBoard/trunk/Firmware/modules/common

and the generated memory IP is located at:

https://svn.astron.nl/UniBoard_FP7/UniBoard/trunk/Firmware/modules/MegaWizard/mem

The common library contains many more low level VHDL components, see [3] for a brief overview.

The purpose of the common memory and register components is to hide vendor specific code or to provide a generic solution that can be reused.

2 Memory

2.1 IP

The memory can be RAM or ROM. In an FPGA larger memories (> 1 kBit) are typically implemented in RAM blocks, because otherwise they take too much of the logic resources. Memory can be inferred by writing VHDL in a form that the synthesizer recognizes as a memory or it can be generated using the Altera MegaWizard [1] or Xilinx CORE Generator [2]. For a generic VHDL implementation it is proper to wrap such vendor specific IP by a self defined component, because the wrapper provides a uniform interface for using the vendor specific IP.

For the memory components available in the UniBoard common library the Altera MegaWizard was used to create an IP memory component called `ram_crw_crw`. The generated file `ram_crw_crw.vhd` (stored at `modules/MegaWizard/mem`) was then edited manually (this is allowed, see [4]) to provide it with generics so that the memory size can be selected in VHDL when the component is used:

```
ENTITY ram_crw_crw IS
GENERIC (
    g_adr_w      : NATURAL := 5;
    g_dat_w      : NATURAL := 8;
    g_nof_words  : NATURAL := 2**5;
    g_init_file  : STRING  := "UNUSED"
);
PORT (
    address_a : IN STD_LOGIC_VECTOR (g_adr_w-1 DOWNTO 0);
    address_b : IN STD_LOGIC_VECTOR (g_adr_w-1 DOWNTO 0);
    clock_a   : IN STD_LOGIC := '1';
    clock_b   : IN STD_LOGIC ;
    data_a    : IN STD_LOGIC_VECTOR (g_dat_w-1 DOWNTO 0);
    data_b    : IN STD_LOGIC_VECTOR (g_dat_w-1 DOWNTO 0);
    enable_a  : IN STD_LOGIC := '1';
    enable_b  : IN STD_LOGIC := '1';
    wren_a    : IN STD_LOGIC := '0';
    wren_b    : IN STD_LOGIC := '0';
    q_a       : OUT STD_LOGIC_VECTOR (g_dat_w-1 DOWNTO 0);
    q_b       : OUT STD_LOGIC_VECTOR (g_dat_w-1 DOWNTO 0)
);
END ram_crw_crw;
```

2.2 Wrapper component

The common library provides a memory package and the following generic MM memory components:

```
common_mem(pkg).vhd
common_ram_crw_crw.vhd
common_ram_rw_rw.vhd
common_ram_r_w.vhd
common_rom.vhd
```

The used memory naming convention is as follows:

- r = single port read only (ROM)
- rw = single port ram, shared address for both write read address
- r_w = dual port ram, with separate write address and read address
- rw_rw = dual port ram with separate address per port
- crw_crw = dual port ram with separate clock and address per port

Note:

- Using a 'r_w' RAM with only the read port and an init file effectively makes it a 'r' RAM, i.e. a ROM
- Using a 'r_w' RAM with the same address effectively makes it a 'rw' RAM
- Using a 'rw_rw' RAM with only a write port and a read port effectively makes it a 'r_w' RAM
- Using a 'crw_crw' RAM with the same clock effectively makes it a 'rw_rw' RAM

Therefore the 'crw_crw' RAM, and therefore the ram_crw_crw.vhd IP with generics from section 2.1, covers all other memory variants either by:

- using an init file to set the initial contents
- not using some input (set to constant value)
- not using some output ports (left open)
- connecting some input ports together (clk or address).

Figure 1 shows the entity of common_ram_crw_crw.vhd that wraps the IP RAM from ram_crw_crw.vhd. The other memory components in common library use common_ram_crw_crw.

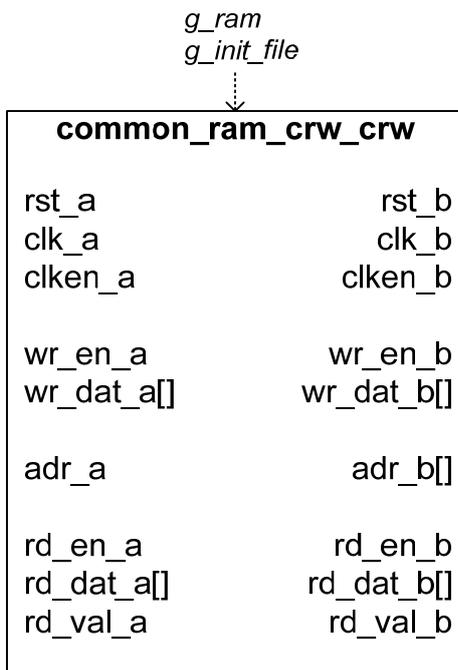


Figure 1: The general IP RAM wrapper component (common_ram_crw_crw.vhd)

The common_ram_crw_crw has two generic parameters. An optional *g_init_file* that provides the location of a memory initialisation file and a record defined in common_mem(pkg).vhd that defines the memory size and the read latency. The record definition is:

```

TYPE t_c_mem IS RECORD
  latency    : NATURAL;    -- read latency
  adr_w      : NATURAL;
  dat_w      : NATURAL;
  nof_dat    : NATURAL;    -- optional, nof dat words <= 2**adr_w
  init_sl    : STD_LOGIC;  -- optional, init all to std_logic '0', '1' or 'X'
  --init_file : STRING;    -- "UNUSED", unconstrained length can not be in
                                -- record
END RECORD;

```

The minimal read latency of the ram_crw_crw memory IP is 2, because it has been generated with registered read data. The common_ram_crw_crw internally uses common_pipeline (also from the common library) to increase this if a larger read latency is needed. The common_ram_crw_crw has the following control signals:

```

rst        = asynchronous reset
clken      = clock enable per clock domain
wr_en      = write enable per port
rd_en      = read enable per port
rd_val     = read valid per port

```

Typically the rd_en is not needed (i.e. always '1'), the read data then always shows the data for the read address. The rd_en is then only used to drive rd_val to account for the memory read latency. The RAM contents can not be reset, only the I/O registers. Resetting the data I/O registers is of little use. Therefore use RAM without reset, except for letting apply the rst input to rd_val.

The common_ram_crw_crw memory has the same data width for both read and write and also for both ports. Different aspect ratio memories (e.g. write byte, read word) are less common, so they can be added to the common library when needed. Typically different aspect ratio memories can not be inferred from RTL code, so an IP prototype will then need to be generated with the MegaWizard.

3 Register

Figure 2 shows the schematic of the `common_reg_r_w` register written in RTL VHDL. The `common_reg_r_w` is a MM register with separate read and write addresses. The register can have multiple words ($g_reg.nof_dat \geq 1$). The address selects a word in the register. The size of the register is set via a generic `t_c_mem` record (section 2.2) and it can be initialized via the `g_init_reg` generic. The minimal read latency of the `common_reg_r_w` is 1. The `common_reg_r_w` can be used in various ways:

- Connect `out_reg` to `in_reg` for write and readback register
- Do not connect `out_reg` to `in_reg` for separate write only register and read only register at the same address
- Leave `out_reg` OPEN for read only register
- Connect `wr_adr` and `rd_adr` to have a shared address bus register
- Pipeline to support MM read data latency ≥ 1

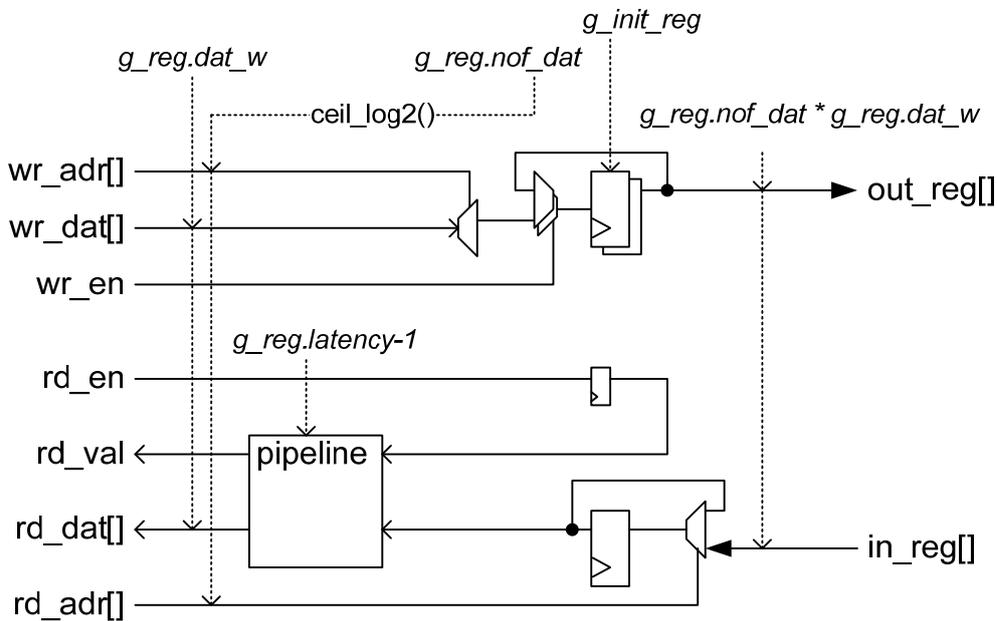


Figure 2: Memory mapped multi word read-write register (`common_reg_r_w.vhd`)

4 Memory mapped interface

Both `common_ram_crw_crw` and `common_reg_r_w` have a memory-mapped (MM) interface that fits the MOSI and MISO signal definitions [5, 6]. Therefore it is possible to make these components known within Altera SOPC Builder. For the `common_reg_r_w` this has been done using the SOPC Component Editor [6]:

- `avs_common_reg_r_w.vhd` = Avalon wrapper entity
- `avs_common_reg_r_w_hw.tcl` = Avalon hardware description file

The `avs_common_reg_r_w` can typically be used as a generic PIO, instead of using the Altera specific PIO.

For `common_ram_crw_crw` it is less useful to define a SOPC builder component, because for RAM it is easier to use the Altera specific `onchip_memory` component. When RAM is used inside a peripheral component then it the `common_ram_crw_crw` should be used. The ETH module shows an example of using `common_ram_crw_crw` and accessing it per word of 4 bytes. The I2C module shows an example of using `common_ram_crw_crw` and accessing it per byte.

5 Clock domain crossing

Reliably crossing a clock domain requires that the data is stable in one clock domain when it is captured in the other clock domain. If stable data is not guaranteed, then the logic that captures this data may get into an undefined state which then may cause the application to fail.

The base variant of `common_ram_crw_crw` provides a dual port, dual clock RAM which can be used to cross a clock domain.

For the `common_reg_r_w` some additional logic is needed to reliably let the register data cross a clock domain. Figure 3 shows `eth_mm_registers.vhd` from `$UNB/modules/tse` where the register data is crossed from the MM clock domain to the ST clock domain.

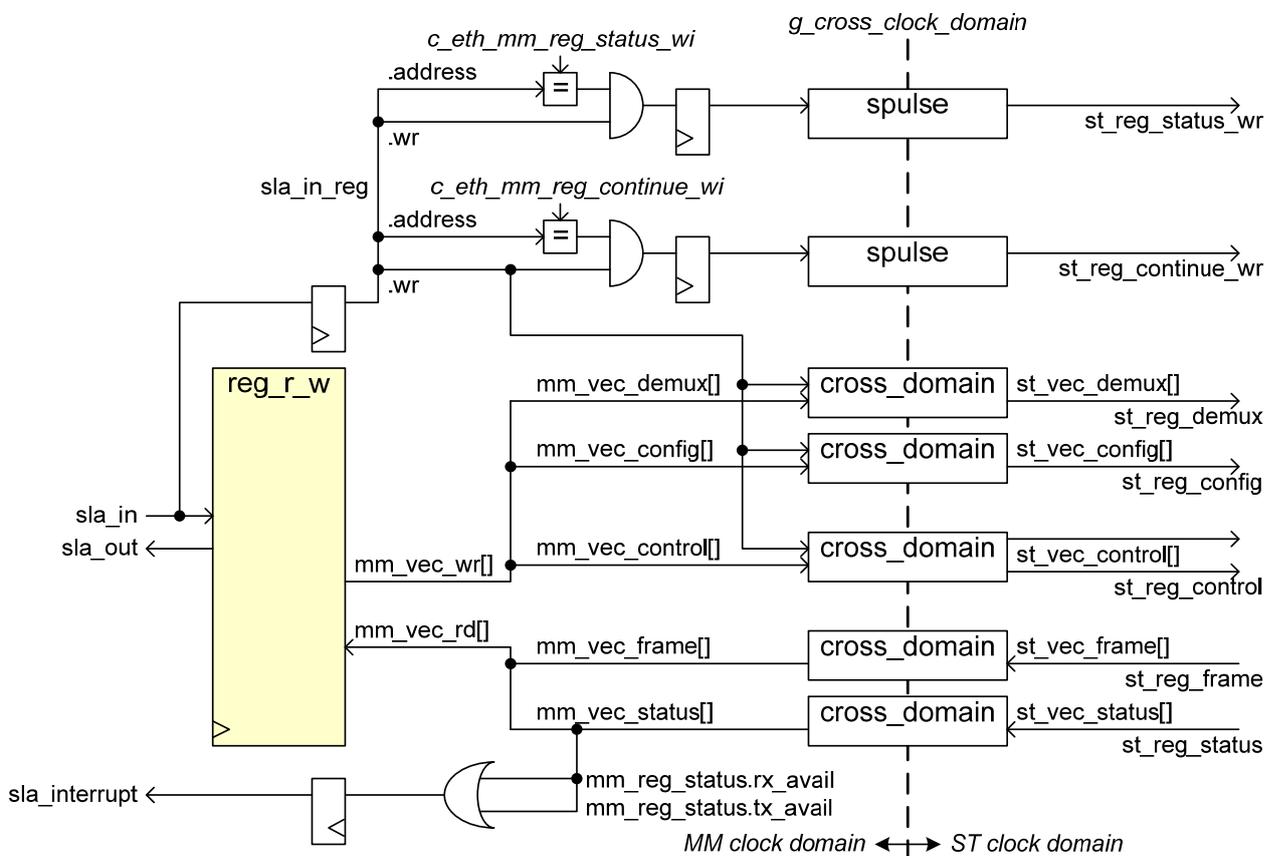


Figure 3: Clock domain crossing for register data (example `eth_mm_registers.vhd`)

The `eth_mm_registers` example of Figure 3 instantiates a multi word `common_reg_r_w`. Part of this register is read/write accessible and part is read only. The write part (demux, config, control) needs to be crossed from MM to ST clock domain and the read only part (frame, status) needs to be crossed from the ST to MM clock domain. If the MM clock and ST clock are in fact the same, then the `g_cross_clock_domain` generic can be set to `FALSE` to skip the clock domain crossing logic.

The `eth_mm_registers` uses separate `common_cross_domain` instances for the write and the read part of the register but also for the functional parts of the register. For demux, config and control this is a matter of convenience, because they could also be transferred across with a single `common_cross_domain` instance. The continue register is empty, only its address is defined. A write access to the continue register is used in

the ST clock domain as a special event and transferred by means of common_spulse. Similar a write access to the status register signals a special event in the ST clock domain.

The common_reg_r_w I/O is a multi word vector. First this vector is mapped on the separate register parts by indexing the appropriate section. Second when the register contents needs to be used it is mapped on to record structures by means of functions defined eth(pkg).vhd. Both these mappings are wires in the implementation, so they do not take resources, but they do make it easier to define and use the register contents.

Figure 4 shows the schematic diagram of common_spulse that is used to get a pulse across to the other clock domain. There is also a test bench tb_common_spulse.vhd.

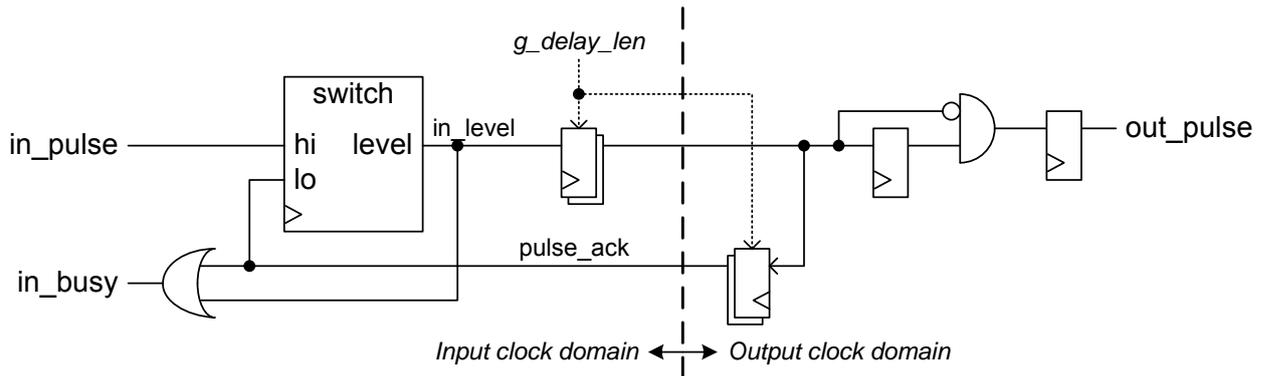


Figure 4: Schematic diagram of common_spulse

Figure 5 shows the schematic diagram of common_cross_domain that is used to get the register data across to the other clock domain. There is also a test bench tb_common_cross_domain.vhd.

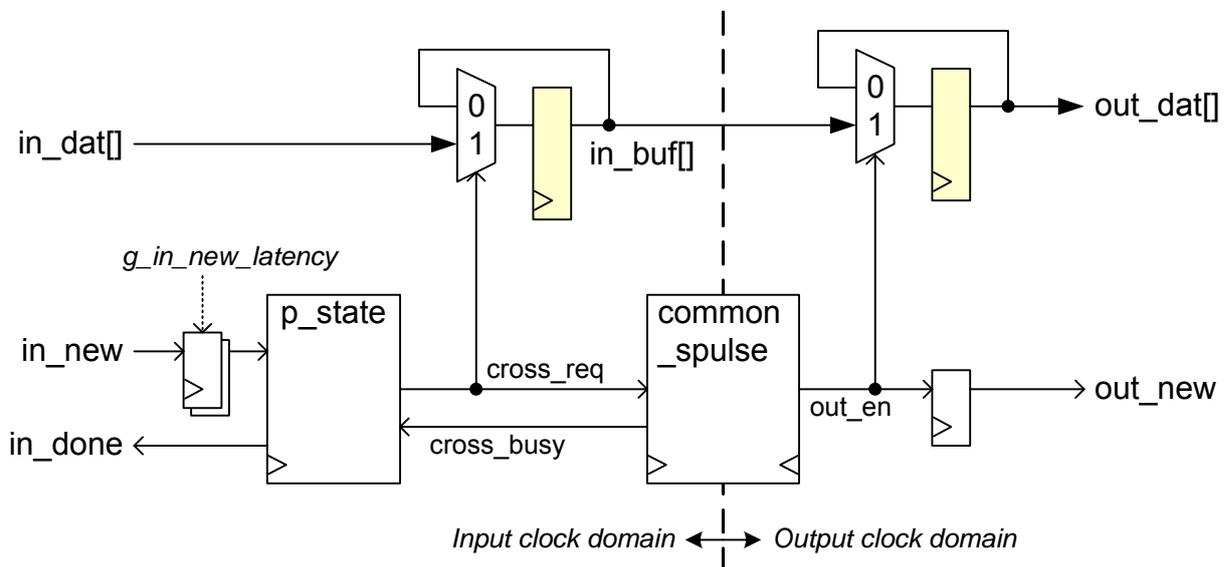


Figure 5: Schematic diagram of common_cross_domain

The crossing of in_dat[] starts when in_new is asserted. If in_new is a pulse, then the new in_dat is available after g_in_new_latency. The in_dat[] is captured in in_buf[]. The in_new pulse becomes out_en in the output clock domain and this is used to capture the in_buf[] in the out_dat[]. When the in_clk equals the out_clk then the latency between in_new and out_new is 9 clock cycles. It is also allowed to hold in_new high, then out_new will pulse once every 17 clock cycles. Use in_done to be sure that in_dat due to in_new has crossed the clock domain, in case of multiple in_new pulses in a row the in_done will only pulse when the

clock domain crossing has completely finished before the next in_new pulse arrives. If in_clk equals out_clock, then in_done will only pulse if the in_new pulses are ≥ 20 clock cycles apart. Therefore in_done will not pulse if in_new is kept high.

6 Dual page

A dual page memory or register is needed when the new contents needs to be available at clock cycle accuracy. One page is accessible to the MM interface and the other page is accessible to the peripheral application in the ST clock domain. A sync pulse in the ST clock domain defines when the two pages swap. The sync pulse interval must be long enough for the MM interface access to have started and finished.

7 Conclusion

The common library provides generic components for MM memory and MM registers.

The common library MM memory and MM register components can also be made available to SOPC Builder.

The common library provides generic components to get MM register data across to and from the ST clock domain. For the MM memory clock domain crossing is inherently already provided by using the dual port, dual clock memory component.