Digital Design LU

IP-Cores

Jakob Lechner, Thomas Polzer
{lechner, tpolzer}@ecs.tuwien.ac.at
Department of Computer Engineering
University of Technology Vienna

Vienna, October 10, 2011

Contents

1 Mathematical Support Package

1.1 Description

The mathematical support package (*math_pkg*) adds support for mathematical functions which are not available in VHDL.

1.2 Dependencies

- None

1.3 Required VHDL files

- math_pkg.vhd

1.4 Supported Functions

- log2c:
  Calculates the logarithm dualis of the integer operand and rounds it up to the next integer. Its main usage is to calculate the minimum required memory address width to store a certain amount of data words.

- max:
  Determines the maximum of the integer operands. This function is available with two and three operands.

2 Output logic

2.1 Description

The output logic implements the basic functionality of the project. It coordinates the different components (keymatrix, serial port and vga output).

2.2 Dependencies

- Text-mode VGA controller

```vhdl
component output_logic is
  port
  (
    clk               : in  std_logic;
    res_n             : in  std_logic;

    -- Port to keypad
    ascii             : in  std_logic_vector(7 downto 0);
    ascii_commit      : in  std_logic;
    ascii_rd          : out std_logic;
    ascii_full        : in  std_logic;
    ascii_empty       : in  std_logic;
    color_change      : in  std_logic;
    key_decoder_error : in  std_logic;
    -- Port to RS232
    rs232_data        : in  std_logic_vector(7 downto 0);
    rs232_rd          : out std_logic;
    rs232_full        : in  std_logic;
    rs232_empty       : in  std_logic;

    -- Port to VGA controller
    vga_command       : out std_logic_vector(COMMAND_SIZE - 1
        downto 0);
    vga_command_data  : out std_logic_vector(3 * COLOR_SIZE +
        CHAR_SIZE - 1 downto 0);
    vga_free          : in  std_logic
  );
end component output_logic;
```

Listing 1: Output logic declaration.

2.3 Required VHDL Files

- output_logic_pkg.vhd
- output_logic.vhd
- output_logic_beh.vhd

2.4 Component Declaration

The declaration of the output logic can be found in Listing 1, while the functionality of each signal is described in Table 1.

Name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal (low active, not internally synchronized).
ascii	in	8	Keypad ASCII data (from FIFO)
ascii_commit	in	1	The previously received character is finalized, if this signal is 1.
ascii_rd	out	1	Requests the the next character from the keypad FIFO buffer.
ascii_full	in	1	1, if the keypad FIFO is full.
ascii_empty	in	1	1, if the keypad FIFO is empty.
color_change	in	1	A color change is requested by an rising edge on this signal.
key_decoder_error	in	1	1, if the keymatrix decoder is in an erroneous state and no longer functional.
rs232_data	in	8	RS232 ASCII data (from FIFO)
rs232_rd	out	1	Requests the the next character from the RS232 FIFO buffer.
rs232_full	in	1	1, if the RS232 FIFO is full.
rs232_empty	in	1	1, if the RS232 FIFO is empty.
vga_command	out	$COMMAND\_SIZE$	Command interface of the VGA controller. See Section 9 for details.
vga_command_data	out	$3 \cdot COLOR\_SIZE + CHAR\_SIZE$	Command interface of the VGA controller. See Section 9 for details.
vga_free	in	1	Status interface of the VGA controller. See Section 9 for details.

Table 1: Output logic signal description.

```
output_logic_inst : output_logic
  port map
  (
    clk => clk,
    res_n => res_n,
    ascii => ascii,
    ascii_commit => ascii_commit,
    ascii_rd => ascii_rd,
    ascii_full => ascii_full,
    ascii_empty => ascii_empty,
    color_change => color_change,
    rs232_data => rs232_data,
    rs232_rd => rs232_rd,
    rs232_full => rs232_full,
    rs232_empty => rs232_empty,
    key_decoder_error => key_decoder_error,
    vga_command => vga_command,
    vga_command_data => vga_command_data,
    vga_free => vga_free
  );
```

Listing 2: Output logic instantiation template.

2.5 Instantiation Template

To be able to instantiate the output logic, the package *output_logic_pkg* must be included within your VHDL source. An instantiation template can be found in Listing 2.

For details on the type and size of the used signals, see Section 2.4.

2.6 Interface Protocol

The timing of the port connected to the key matrix and the port connected to the serial port are based on the FIFO timing (see Section 3.6), the port to the text-mode VGA controller on the text-mode VGA controller timing (see Section 9.6). If on both input ports a character is received simultaneously, the RS232 port has priority.

An example how a character is handled on each of the two input ports can be found in Figure 1 and Figure 2, respectively. The commit handling for the keypad is shown in Figure 3.

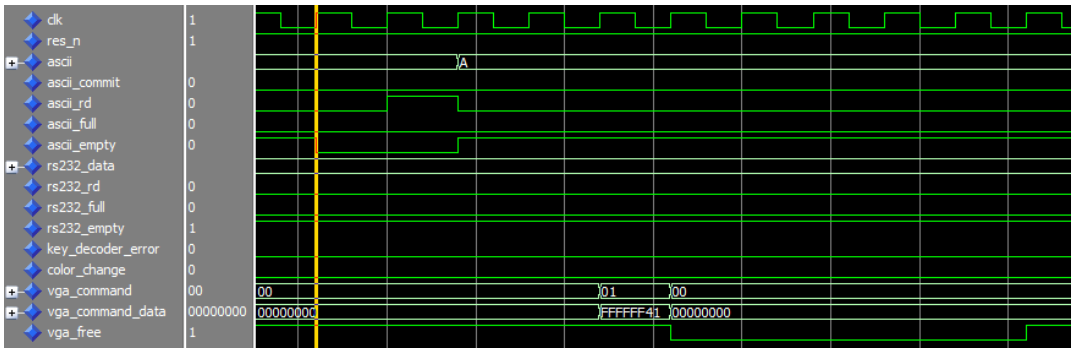

Figure 1: Output logic keypad timing.

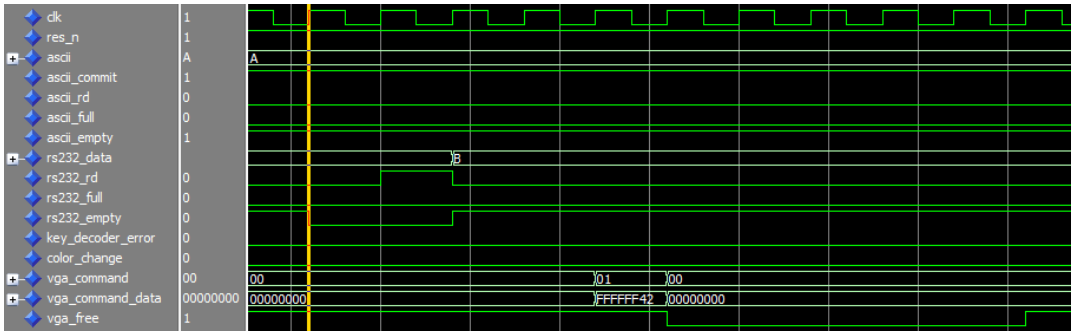

Figure 2: Output logic RS232 timing.

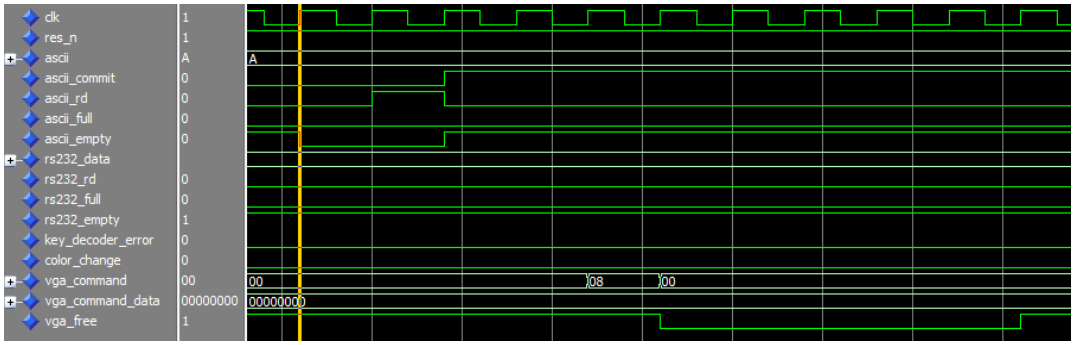

Figure 3: Output logic commit timing.

2.7 Statemachine Description

The output logic is controlled by a statemachine. The operation of the state machine is described by its state chart (see Figure 4), its state transition table (see Table 2 and Table 3) and its output behavior (see Table 4 and Table 5). The reset state is *RESET*.

State	Condition	Next state
$RESET$	$vga\_free = 1$	$INIT$
$INIT$	None	$WAIT\_NOT\_FREE$
$WAIT\_NOT\_FREE$	$vga\_free = 0$	$WAIT\_FREE$
$WAIT\_FREE$	$vga\_free = 1$	$IDLE$
$IDLE$	$ascii\_full = 1$ or $rs232\_full = 1$ or $key\_decoder\_error = 1$	$ERROR$
	$ascii\_full \neq 1$ and $rs232\_full \neq 1$ and $key\_decoder\_error \neq 1$ and $color\_change = 1$	$CHANGE\_COLOR$
	$ascii\_full \neq 1$ and $rs232\_full \neq 1$ and $key\_decoder\_error \neq 1$ and $color\_change \neq 1$ and $rs232\_empty = 0$	$READ\_NEXT\_RS232$
	$ascii\_full \neq 1$ and $rs232\_full \neq 1$ and $key\_decoder\_error \neq 1$ and $color\_change \neq 1$ and $rs232\_empty \neq 0$ and $ascii\_empty = 0$	$READ\_NEXT\_ASCII$
$READ\_NEXT\_ASCII$	None	$PROCESS\_NEXT\_ASCII$
$PROCESS\_NEXT\_ASCII$	$ascii\_commit = 1$	$ASCII\_COMMIT$
	$ascii\_commit \neq 1$	$ASCII\_NEW$
$ASCII\_NEW$	$ascii = LF$	$WAIT\_NOT\_FREE\_FOR\_CR$
	$ascii \neq LF$	$WAIT\_NOT\_FREE$

Table 2: Output logic FSM state transition table (part 1).

State	Condition	Next state
$READ\_NEXT\_RS232$	None	$PROCESS\_NEXT\_RS232$
$PROCESS\_NEXT\_RS232$	None	$RS232\_NEW$
$RS232\_NEW$	None	$RS232\_WAIT\_NOT\_FREE$
$RS232\_WAIT\_NOT\_FREE$	$vga\_free = 0$	$RS232\_WAIT\_FREE$
$RS232\_WAIT\_FREE$	$vga\_free = 1$ and ($rs232\_data = LF$ or $rs232\_data = CR$) $vga\_free = 1$ and $rs232\_data \neq LF$ and $rs232\_data \neq CR$	$IDLE$ $ASCII\_COMMIT$
$WAIT\_NOT\_FREE\_FOR\_CR$	$vga\_free = 0$	$WAIT\_FREE\_FOR\_CR$
$WAIT\_FREE\_FOR\_CR$	$vga\_free = 1$	CR
CR	None	$WAIT\_NOT\_FREE$
$ASCII\_COMMIT$	None	$WAIT\_NOT\_FREE$
$ERROR$	None	$ERROR\_SET\_BACKGROUND$
$ERROR\_SET\_BACKGROUND$	None	$ERROR\_WAIT\_NOT\_FREE$
$ERROR\_WAIT\_NOT\_FREE$	$vga\_free = 0$	$ERROR\_WAIT\_FREE$
$ERROR\_WAIT\_FREE$	$vga\_free = 1$	$ERROR\_IDLE$
$ERROR\_IDLE$		None
$CHANGE\_COLOR$	$color\_index =$ $MAX\_COLOR\_INDEX - 1$ $color\_index \neq$ $MAX\_COLOR\_INDEX - 1$	$FIRST\_COLOR$ $NEXT\_COLOR$
$FIRST\_COLOR$	None	$CHANGE\_COLOR\_SET\_CURSOR$
$NEXT\_COLOR$	None	$CHANGE\_COLOR\_SET\_CURSOR$
$COLOR\_SET\_CURSOR$	None	$CHG\_COL\_WAIT\_NOT\_FREE$
$CHG\_COL\_WAIT\_NOT\_FREE$	$vga\_free = 0$	$CHG\_COL\_WAIT\_FREE$
$CHG\_COL\_WAIT\_FREE$	$vga\_free = 1$	$WAIT\_BUTTON\_RELEASE$
$WAIT\_BUTTON\_RELEASE$	$color\_change = 0$	$IDLE$

Table 3: Output logic FSM state transition table (part 2).

State	Outputs
Default	$vga\_command\_next = COMMAND\_NOP$ $vga\_command\_data\_next = 0$ $ascii\_rd = 0$ $rs232\_rd = 0$ $color\_index\_next = color\_index$
$RESET$	
$INIT$	$vga\_command\_next =$ $COMMAND\_SET\_CURSOR\_AUTO$ $vga\_command\_data\_next = 0$
$WAIT\_NOT\_FREE$	
$WAIT\_FREE$	
$IDLE$	
$READ\_NEXT\_ASCII$	$ascii\_rd = 1$
$PROCESS\_NEXT\_ASCII$	
$ASCII\_NEW$	$vga\_command\_next = COMMAND\_SET\_CHAR$ $vga\_command\_data\_next = color\&ascii$

Table 4: Output logic FSM output table (part 1).

State	Outputs
$READ\_NEXT\_RS232$	$rs232\_rd = 1$
$PROCESS\_NEXT\_RS232$	
$RS232\_NEW$	$vga\_command\_next = COMMAND\_SET\_CHAR$ $vga\_command\_data\_next = color \& rs232_data$
$RS232\_WAIT\_NOT\_FREE$	
$RS232\_WAIT\_FREE$	
$WAIT\_NOT\_FREE\_FOR\_CR$	
$WAIT\_FREE\_FOR\_CR$	
CR	$vga\_command\_next = COMMAND\_SET\_CHAR$ $vga\_command\_data\_next = CR$
$ASCII\_COMMIT$	$vga\_command\_next =$ $COMMAND\_SET\_CURSOR\_NEXT$ $vga\_command\_data\_next = 0$
$ERROR$	
$ERROR\_SET\_BACKGROUND$	$vga\_command\_next =$ $COMMAND\_SET\_BACKGROUND$ $vga\_command\_data\_next = red$
$ERROR\_WAIT\_NOT\_FREE$	
$ERROR\_WAIT\_FREE$	
$ERROR\_IDLE$	
$CHANGE\_COLOR$	
$FIRST\_COLOR$	$color\_index\_next = 0$
$NEXT\_COLOR$	$color\_index\_next = color\_index + 1$
$CHANGE\_COLOR\_SET\_CURSOR$	$vga\_command\_next =$ $COMMAND\_SET\_CURSOR\_COLOR$ $vga\_command\_data\_next = color$
$CHG\_COL\_WAIT\_NOT\_FREE$	
$CHG\_COL\_WAIT\_FREE$	
$WAIT\_BUTTON\_RELEASE$	

Table 5: Output logic FSM output table (part 2).

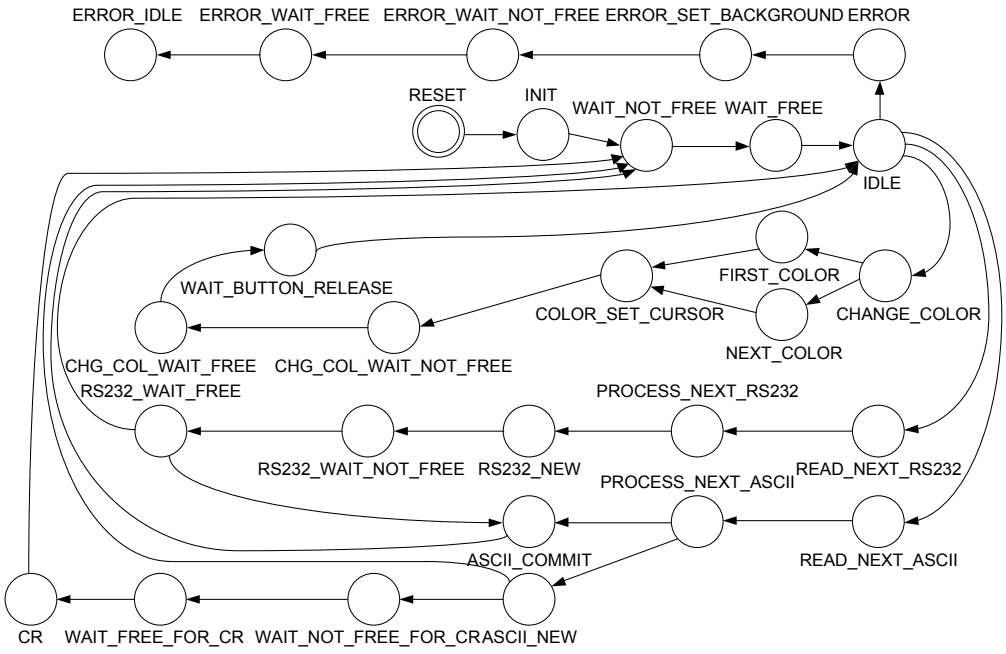

Figure 4: Output logic statemachine.

3 On-chip RAM

3.1 Description

Important components in nearly each integrated circuit are memories. If a memory with full access speed is required, only on-chip memories are an option. This package provides an easy way to instantiate on-chip RAMs.

Currently there are two RAM memories available, a single clock dual-port RAM with a read and a write port and a single clock dual port FIFO with a read and a write port.

3.2 Dependencies

- Mathematical support package (*math_pkg*)

3.3 Required VHDL Files

- ram_pkg

```
component dp_ram_1c1r1w is
  generic
  (
    -- Addresswidth
    ADDR_WIDTH : integer;
    -- Datawidth
    DATA_WIDTH : integer
  );
  port
  (
    clk    : in std_logic;

    -- Read port
    raddr1 : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
    rdata1 : out std_logic_vector(DATA_WIDTH - 1 downto 0);
    rd1    : in std_logic;

    -- Write port
    waddr2 : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
    wdata2 : in std_logic_vector(DATA_WIDTH - 1 downto 0);
    wr2    : in std_logic
  );
end component dp_ram_1c1r1w;
```

Listing 3: RAM declaration.

- dp_ram_1c1r1w
- dp_ram_1c1r1w_beh
- fifo_1c1r1w
- fifo_1c1r1w_mixed

3.4 Component Declaration

The declaration of the single clock dual-port RAM with a read and a write port can be found in Listing 3, while the functionality of each generic is described in Table 6 and of each signal in Table 7.

The declaration of the single clock dual port FIFO with a read and a write port can be found in Listing 4, while the functionality of each generic is described in Table 8 and of each signal in Table 9.

Name	Functionality
ADDR_WIDTH	The number of address bits.
DATA_WIDTH	The number of data bits.

Table 6: RAM generics description.

```
component fifo_1c1r1w is
  generic
  (
    -- Minimum depth, the actual depth is rounded up to
    -- the next power of 2
    MIN_DEPTH  : integer;
    -- Datawidth
    DATA_WIDTH : integer
  );
  port
  (
    clk       : in std_logic;
    res_n     : in std_logic;
    -- Read port
    data_out1 : out std_logic_vector(DATA_WIDTH - 1 downto 0)
       ;
    rd1       : in std_logic;
    -- Write port
    data_in2  : in std_logic_vector(DATA_WIDTH - 1 downto 0);
    wr2       : in std_logic;
    -- Status port
    empty     : out std_logic;
    full      : out std_logic
  );
end component fifo_1c1r1w;
```

Listing 4: FIFO declaration.

3.5 Instantiation Template

To be able to instantiate an on-chip RAM, the package *ram_pkg* must be included within your VHDL source. An instantiation template for the single clock dual-port RAM with a read and a write port can be found in Listing 5, for the single clock dual port FIFO with a read and a write port in Listing 6.

For details on the type and size of the used signals, see Section 3.4.

Name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
$raddr1$	in	$ADDR\_WIDTH$	Address signal of the read port.
$rdata1$	out	$DATA\_WIDTH$	Data signal of the read port.
$rd1$	in	1	If 1, a read operation is performed on the next rising edge of the clock signal.
$waddr2$	in	$ADDR\_WIDTH$	Address signal of the write port.
$wdata2$	in	$DATA\_WIDTH$	Data signal of the write port.
$wr2$	*input*	1	If 1, the data of $wdata2$ is written to address $waddr2$ of the memory.

Table 7: RAM signal description.

Name	Functionality
MIN_DEPTH	The minimum depth of the FIFO. The actual depth is rounded up to the next power of two.
DATA_WIDTH	The number of data bits.

Table 8: FIFO generics description.

```
ram_inst : dp_ram_1c1r1w
  generic map
  (
    ADDR_WIDTH => 5,
    DATA_WIDTH => 8
  )
  port map
  (
    clk => clk,
    raddr1 => raddr1,
    rdata1 => rdata1,
    rd1 => rd1,
    waddr2 => waddr2,
    wdata2 => wdata2,
    wr2 => wr2
  );
```

Listing 5: RAM instantiation template.

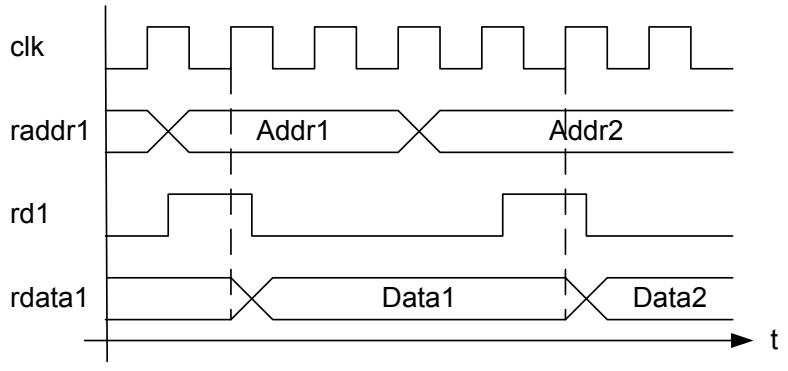

Figure 5: RAM read timing.

3.6 Interface Protocol

A standard synchronous memory access protocol is used for accessing the RAM. At any rising edge of the *clk* signal, when the *rd*1 signal is high, the data word stored at address *raddr*1 is written to the *data*1 port (see Figure 5).

Name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal, low active, not internally synchronized.
*data_out*1	out	*DATA_WIDTH*	Output data.
*rd*1	in	1	If 1, a read operation is performed at the next rising edge of the clock signal. If the FIFO is empty, the result is undefined.
*data_in*2	in	*DATA_WIDTH*	Data for the write operation
*wr*2	in	1	If 1, the data of *data_in*2 is written to the next free memory location. If the FIFO is full, the write request is ignored.
empty	out	1	1, if the memory is empty.
full	out	1	1, if the memory is full.

Table 9: FIFO signal description.

```
fifo_inst : fifo_1c1r1w
  generic map
  (
    MIN_DEPTH => 5,
    DATA_WIDTH => 16
  )
  port map
  (
    clk => clk,
    res_n => res_n,
    data_out1 => data_out1,
    rd1 => rd1,
    data_in2 => data_in2,
    wr2 => wr2,
    empty => empty,
    full => full
  );
```

Listing 6: FIFO instantiation template.

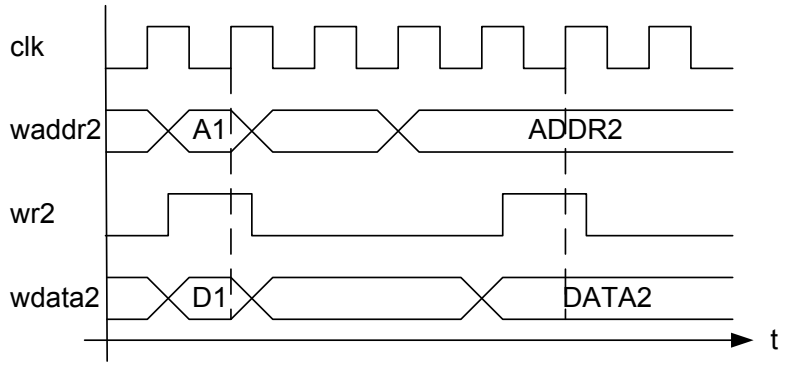

Figure 6: RAM write timing.

At any rising edge of the *clk* signal, when the *wr*2 signal is high, the data word at *wdata*2 is written to address *waddr*2 (see Figure 6).

The FIFO memory uses a similar interface but does not require the address inputs. The read operation is again initiated by a one on the *rd*1 signal. If the FIFO is not empty the next data word is assigned to port *data_out*1 (see Figure 7). If the FIFO is empty, the result of the read operation is undefined.

The write operation is started with a one on *wr*2. The data word from *data_in*2 is stored to the next free location in the memory array. (see Figure 8). If the FIFO is full, the write operation is ignored.

If the first item is written to the FIFO, the *empty* signal becomes zero

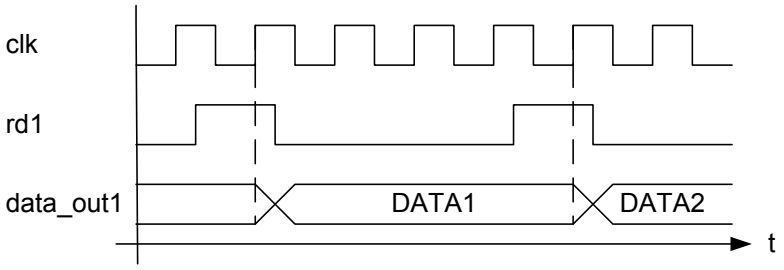

Figure 7: FIFO read timing.

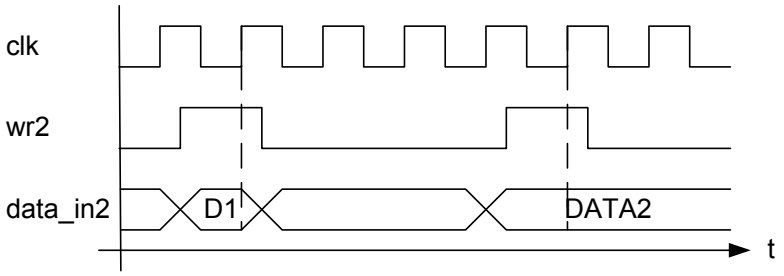

Figure 8: FIFO write timing.

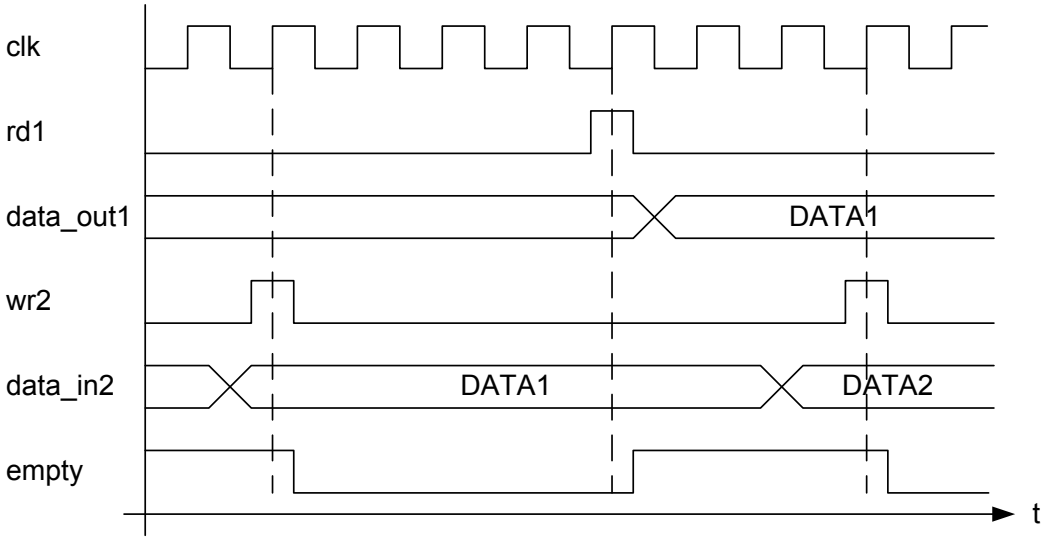

Figure 9: FIFO empty handling.

parallel to the storage operation. If the last item is read from the FIFO, the *empty* signal becomes one at the same time the output data is set (see Figure 9).

If the FIFO becomes full by a write operation, parallel to the storing process the $full$ signal becomes one. If afterwards a data word is read, the $full$ signals becomes zero again at the same time as the output port is set (see Figure 10).

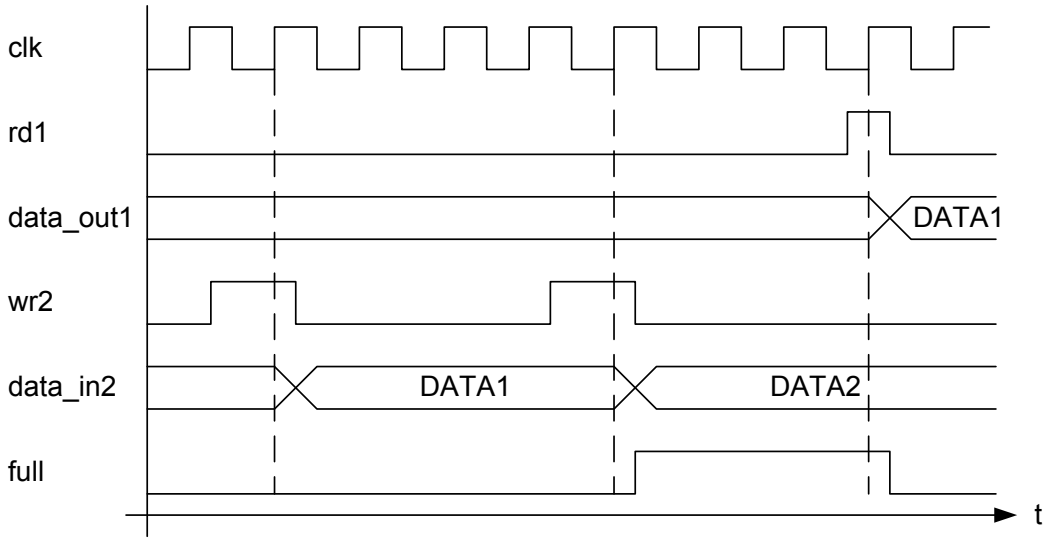

Figure 10: FIFO full handling.

4 On-chip ROM

4.1 Description

Important components in nearly each integrated circuit are memories. If a memory with full access speed is required, only on-chip memories are an option. This package provides an easy way to instantiate on-chip ROMs.

Currently there is one synchronous single port ROM available.

4.2 Dependencies

- None

4.3 Required VHDL Files

- rom_pkg
- rom_sync_1r
- rom_sync_1r_beh

```
component rom_sync_1r is
  generic
  (
    -- Address width
    ADDR_WIDTH   : integer;
    -- Data width
    DATA_WIDTH   : integer;
    -- Memory contents
    INIT_PATTERN : rom_array
  );
  port
  (
    clk  : in  std_logic;
    addr : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
    rd   : in  std_logic;
    data : out std_logic_vector(DATA_WIDTH - 1 downto 0)
  );
end component rom_sync_1r;
```

Listing 7: ROM declaration.

Name	Functionality
ADDR_WIDTH	The number of address bits.
DATA_WIDTH	The number of data bits.
INIT_PATTERN	This generic parameter is used to specify the contents of the ROM. The specification is written using an array of *std_logic_vectors*, a vector for each memory address.

Table 10: ROM generics description.

4.4 Component Declaration

The declaration of the ROM can be found in Listing 7, while the functionality of each generic is described in Table 10 and of each signal in Table 11.

4.5 Instantiation Template

To be able to instantiate an on-chip ROM, the package *rom_pkg* must be included within your VHDL source. An instantiation template for the ROM can be found in Listing 8.

For details on the type and size of the used signals, see Section 4.4.

Name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
addr	in	$ADDR\_WIDTH$	Address.
data	out	$DATA\_WIDTH$	Data.
rd	in	1	If 1, a read operation is performed on the next rising edge of the clock signal.

Table 11: ROM signal description.

```
data_rom_inst : rom_sync_1r
  generic map
  (
    ADDR_WIDTH => 5,
    DATA_WIDTH => 8,
    INIT_PATTERN => (x"31", x"21", x"2C", x"2E", x"2B",
                     x"2D", x"2A", x"2F", x"41", x"42",
                     x"43", x"61", x"62", x"63", x"32",
                     x"44", x"45", x"46", x"64", x"65",
                     x"66", x"33", x"47", x"48", x"49",
                     x"67", x"68", x"69", x"34", x"4A",
                     x"4B", x"4C")
  )
  port map
  (
    clk => clk,
    addr => addr,
    rd => rd,
    data => data
  );
```

Listing 8: ROM instantiation template.

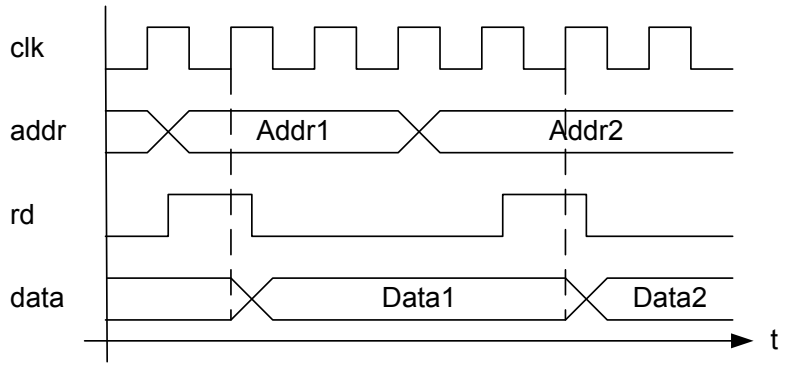

Figure 11: ROM read timing.

4.6 Interface Protocol

A standard synchronous memory protocol is used for accessing the ROM. At any rising edge of the *clk* signal, when the *rd* signal is high, the data word stored at address *raddr* is written to the *data* port (see Figure 11).

Name	Functionality
CLK_FREQ	Actual clock frequency of the *clk* signal given in Hz.
$DISPLAY\_COUNT$	Numbers of controlled seven segment digits.
$UPDATE\_FREQ$	Display update frequency. Used for multiplexing the displays.

Table 12: Seven-segment-display control generics description.

5 Seven-Segment-Display Control

5.1 Description

In many cases (e.g. debugging, simple user interaction) a simple, easy to control output device is needed. A seven segment display is such a device. This component converts a parallel signal vector to its seven segment representation and can control multiple displays concurrently. The displays share the same data line and are controlled using multiplexed select signals.

5.2 Dependencies

- None

5.3 Required VHDL Files

- seven_segment_display_pkg.vhd
- seven_segment_display_multiplexed.vhd
- seven_segment_display_multiplexed_beh.vhd

5.4 Component Declaration

The declaration of the seven segment control component can be found in Listing 9, while the functionality of each generic is described in Table 12 and of each signal in Table 13.

Name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal (low active, not internally synchronized).
data	in	$4 \cdot DISPLAY\_COUNT$	The data which should be displayed.
seg_sel	out	$DISPLAY\_COUNT$	Select the active display. At most one display is selected at a time
seg_data	out	8	Output data (should be directly connected to the seven segment display).

Table 13: Seven-segment-display control signal description.

```
component seven_segment_display_multiplexed is
  generic
  (
    -- Clock frequency [Hz]
    CLK_FREQ : integer;
    -- Number of digits
    DISPLAY_COUNT : integer;
    -- Display update frequency
    UPDATE_FREQ : integer range 1000 to 1000000
  );
  port
  (
    sys_clk : in std_logic;
    sys_res_n : in std_logic;

    -- Internal interface
    data : in std_logic_vector(4 * DISPLAY_COUNT - 1 downto
       0);

    -- External interface
    seg_sel : out std_logic_vector(DISPLAY_COUNT - 1 downto
       0);
    seg_data : out std_logic_vector(7 downto 0)
  );
end component seven_segment_display_multiplexed;
```

Listing 9: Seven-segment-display control declaration.

5.5 Instantiation Template

To be able to instantiate the seven-segment-display control component, the *seven_segment_display_pkg* package must be included within your VHDL source. An instantiation template can be found in Listing 10.

For details on the type and size of the used signals, see Section 5.4.

5.6 Interface Protocol

The interface is straight forward. The signal which should be displayed is assigned to the data port. Note that the data is not internally buffered. Based on the *clk* signal and the $UPDATE\_FREQ$ generic, an activation pattern for the displays is generated. At every time, at most a single digit is active (see Section 5.7 for details.

```
seven_segment_inst : seven_segment_display_multiplexed
  generic map
  (
    CLK_FREQ => 25000000,
    DISPLAY_COUNT => 2,
    UPDATE_FREQ => 1000000
  )
  port map
  (
    clk => clk,
    res_n => res_n,
    data => data,
    seg_sel => seg_sel,
    seg_data => seg_data
  );
```

Listing 10: Seven-segment-display control instantiation template.

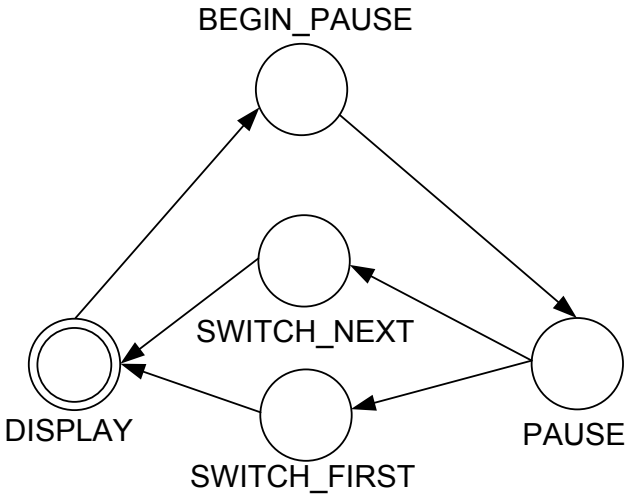

Figure 12: Seven-segment-display control statemachine.

5.7 Statemachine Description

This component is controlled by a single state machine. The operation of the state machine is described by its state chart (see Figure 12), its state transition table (see Table 14) and its output behavior (see Table 15). The reset state is $DISPLAY$.

State	Condition	Next state
$DISPLAY$	$timeout\_cnt = TIMEOUT\_CNT\_MAX - 1$	$BEGIN\_PAUSE$
$BEGIN\_PAUSE$	None	$PAUSE$
$PAUSE$	$timeout\_cnt = TIMEOUT\_CNT\_MAX - 1$ and $selected\_display < DISPLAY\_COUNT - 1$	$SWITCH\_NEXT$
	$timeout\_cnt = TIMEOUT\_CNT\_MAX - 1$ and $selected\_display = DISPLAY\_COUNT - 1$	$SWITCH\_FIRST$
$SWITCH\_FIRST$	None	$DISPLAY$
$SWITCH\_NEXT$	None	$DISPLAY$

Table 14: Seven-segment-display control FSM state transition table.

State	Outputs
Default	seg_sel = all inactive (1) $selected\_display\_next = selected\_display$ $seg\_data\_next = seg\_data\_int$
$DISPLAY$	$seg\_sel(DISPLAY\_COUNT - selected_disp lay - 1) = 0$ $timeout\_cnt\_next = timeout\_cnt + 1$
$BEGIN\_PAUSE$	seg_data_next = all inactive (1) $timeout\_cnt\_next = 0$
$PAUSE$	$timeout\_cnt\_next = timeout\_cnt + 1$
$SWITCH\_FIRST$	$selected\_display\_next = 0$ $timeout\_cnt\_next = 0$ seg_data_next is set
$SWITCH\_NEXT$	$selected\_display\_next = selected\_display + 1$ $timeout\_cnt\_next = 0$ seg_data_next is set

Table 15: Seven-segment-display control FSM output table.

Name	Functionality
$SYNC\_STAGES$	Number of flip flop stages used for synchronization.
$RESET\_VALUE$	The value, the output signal should have directly after reset.

Table 16: Synchronizer generics description.

6 Synchronizer

6.1 Description

The synchronizer component is used to connect external signals (e.g. from push buttons or serial ports) to a design. As these input devices generate signals which are out of synchronization with the global clock signal, these signals must be synchronized to the clock signal (*clk*), otherwise the behavior of the system may become unpredictable.

6.2 Dependencies

- None

6.3 Required VHDL Files

- sync_pkg.vhd
- sync.vhd
- sync_beh.vhd

6.4 Component Declaration

The declaration of the synchronizer can be found in Listing 11, while the functionality of each generic is described in Table 16 and of each signal in Table 17.

In the special case that the synchronizer is used for an external global reset signal, the *res_n* port is set to constant one and the reset signal is connected to *data_in*. The processed reset signal can be accessed on port *data_out*.

Name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal (low active, not internally synchronized).
data_in	in	1	The signal which should be synchronized.
data_out	out	1	The synchronized version of the input signal.

Table 17: Synchronizer signal description.

```
component sync is
  generic
  (
    -- Number of synchronizer stages
    SYNC_STAGES : integer range 2 to integer'high;
    -- Value of data_out directly after reset
    RESET_VALUE : std_logic
  );
  port
  (
    clk : in std_logic;
    res_n : in std_logic;
    -- External interface
    data_in : in std_logic;
    -- Internal interface
    data_out : out std_logic
  );
end component sync;
```

Listing 11: Synchronizer declaration.

```
sync_inst : sync
  generic map
  (
    SYNC_STAGES => 2,
    RESET_VALUE => '0'
  )
  port map
  (
    clk => clk,
    res_n => res_n,
    data_in => data_in,
    data_out => data_sync
  );
```

Listing 12: Synchronizer instantiation template.

6.5 Instantiation Template

To be able to instantiate the synchronizer, the package *sync_pkg* must be included within your VHDL source. An instantiation template can be found in Listing 12.

For details on the type and size of the used signals, see Section 6.4.

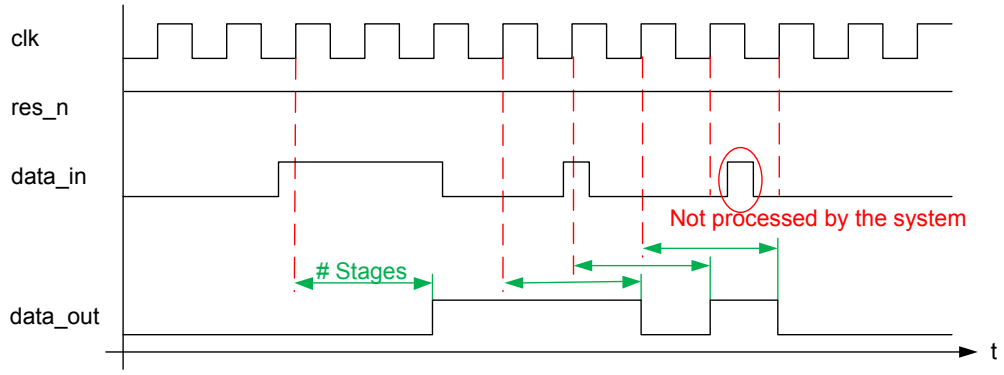

Figure 13: Synchronizer timing.

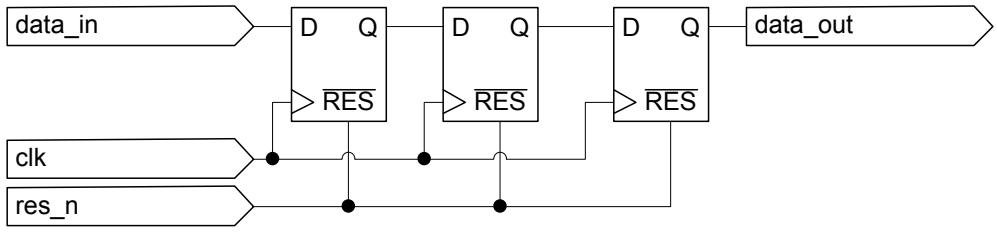

Figure 14: Synchronizer internal circuitry.

6.6 Interface Protocol

The synchronizer has no special interface protocol. The input signal is sampled with the global clock signal (clk). Therefore the output signal is align to the global clock (clk) and has a delay of n clock cycles, where n is the number of synchronizer stages. Spikes not overlapping a clock edge (see Figure 13 as example) will not show up at the synchronizer output.

6.7 Internal Structure

The synchronizer internally consists of a D-Flipflop chain. Figure 14 shows an example of a three state synchronizer.

7 PS/2 Keyboard Controller

7.1 Description

The PS/2 keyboard controller core can be used to easily interface with a standard PS/2 keyboard. The core is initialized automatically. On the reception of a new scan code (corresponding to set 2, see [SCAa, SCAb]) the controller transmits it to the user logic without any further processing. Therefore the handling of special keys (like Ctrl or Shift) as well as the key pressed and key release handling are not performed by the controller. Instead the corresponding scan codes are directly sent to the user logic.

7.2 Dependencies

The controller has no external dependencies.

7.3 Required VHDL Files

The PS/2 keyboard controller interface requires the following source files:

- ps2_keyboard_controller.vhd
- ps2_keyboard_controller_beh.vhd
- ps2_keyboard_controller_pkg.vhd
- ps2_transceiver.vhd
- ps2_transceiver_beh.vhd
- ps2_transceiver_pkg.vhd

7.4 Component Declaration

The declaration of the PS/2 keyboard controller can be found in Listing 13, while the functionality of each generic is described in Table 18 and of each signal in Table 19.

```
component ps2_keyboard_controller is
  generic
  (
    -- Frequency of the global clock in Hertz
    CLK_FREQ : integer;
    -- Number of sync stages
    SYNC_STAGES : integer
  );
  port
  (
    -- Global Signals
    clk, res_n : in std_logic;

    -- Keyboard interface
    ps2_clk, ps2_data : inout std_logic;

    -- User logic interface
    new_data : out std_logic;
    data : out std_logic_vector(7 downto 0)
  );
end component ps2_keyboard_controller;
```

Listing 13: PS/2 keyboard controller declaration.

Generic name	Functionality
CLK_FREQ	Frequency of the system clock given in Hz.
SYNC_STAGES	Number of synchronizer stages used for synchronizing external signals.

Table 18: PS/2 keyboard controller generics description.

7.5 Instantiation Template

To be able to instantiate the keyboard controller, the package *ps2_keyboard_controller_pkg* must be included within your VHDL source. An instantiation template can be found in Listing 14.

For details on the type and size of the used signals, see Section 7.4.

7.6 Interface Protocol

The keyboard controller utilizes a simple, straight forward interface. If a new scan code is available, it is assigned to the data port and the signal new data is

Signal name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal (low active, not internally synchronized).
new_data	out	1	Signalizes the availability of a new scan code.
data	out	8	Scan code output.
ps2_clk	bidirectional	1	Keyboard clock line.
ps2_data	bidirectional	1	Keyboard data line

Table 19: PS/2 keyboard controller signal description.

```
ps2_keyboard_controller_inst : ps2_keyboard_controller
  generic map
  (
    CLK_FREQ => CLK_FREQ,
    SYNC_STAGES => SYNC_STAGES
  )
  port map
  (
    clk => clk,
    res_n => res_n,
    ps2_clk => ps2_keyboard_clk,
    ps2_data => ps2_keyboard_data,
    new_data => new_scancode,
    data => scancode
  );
```

Listing 14: PS/2 keyboard controller instantiation template.

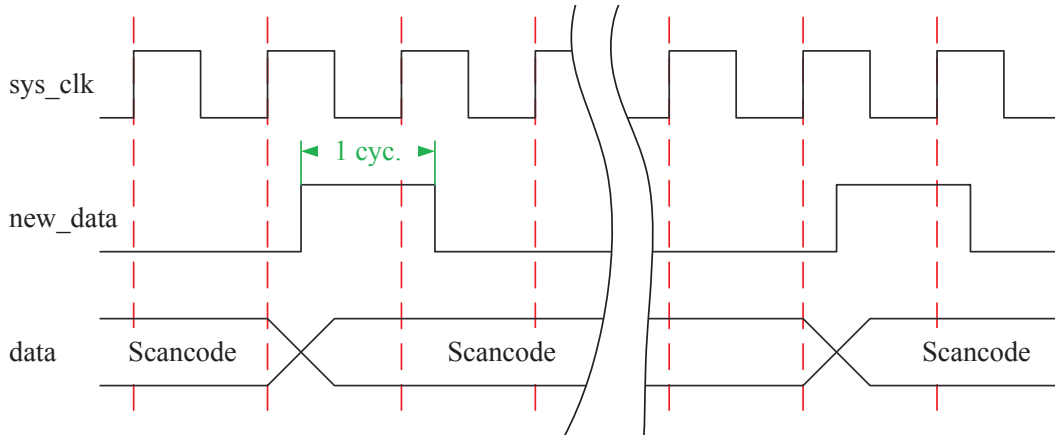

Figure 15: PS/2 keyboard controller timing.

set high for exactly one clock cycle. The data port stays unchanged until the next scan code is received from the keyboard. For details on the protocol see Figure 15.

8 PS/2 to ASCII Converter

8.1 Description

The PS/2 to ASCII converter is used to convert sequences of keyboard scan codes into valid ASCII characters. It can be directly interfaced with the PS/2 keyboard controller.

8.2 Dependencies

The converter has the following external dependencies:

- Mathematical support package (*math_pkg*)
- On-chip ROM package (*rom_pkg*)

8.3 Required VHDL Files

The PS/2 to ASCII controller requires the following source-files:

- ps2_ascii_pkg.vhd
- ps2_ascii.vhd
- ps2_ascii_struct.vhd

8.4 Component Declaration

The declaration of the PS/2 to ASCII converter can be found in Listing 15, the functionality of each signal in Table 20.

8.5 Instantiation Template

To be able to instantiate the PS/2 to ASCII converter, the package

- *ps2_ascii_pkg*

must be included within your VHDL source. An instantiation template can be found in Listing 16.

For details on the type and size of the used signals, see Section 8.4.

Signal name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal (low active, not internally synchronized).
scancode	in	8	Scan code generated by the keyboard controller.
new_scancode	in	1	A new scan code is available at the input. Must be 1 for exactly one clock cycle for each new scan code.
ascii	out	8	The generated ASCII character.
new_ascii	out	1	If 1, a new ASCII character is available. Stays one for exactly one clock cycle.

Table 20: PS/2 to ASCII converter signal description.

```
component ps2_ascii is
  port
  (
    -- Global signals
   clk, res_n : in std_logic;

   -- Interface to keyboard controller
   scancode : in std_logic_vector(7 downto 0);
   new_scancode : in std_logic;

   -- Interface to user logic
   ascii : out std_logic_vector(7 downto 0);
   new_ascii : out std_logic
  );
end component ps2_ascii;
```

Listing 15: PS/2 to ASCII converter declaration.

```
ps2_ascii_inst : ps2_ascii
  port map
  (
   clk => clk,
   res_n => res_n
   scancode => scancode,
   new_scancode => new_scancode,
   ascii => ascii,
   new_ascii => new_ascii
  );
```

Listing 16: PS/2 to ASCII converter instantiation template.

8.6 Interface Protocol

Each new scan code is indicated through a one on the *new_scancode* signal. The signal must stay one for exactly one clock cycle.

If the last scan code inputed to the converter generates a valid ASCII code, it is converted to ASCII and the result is assigned to the *ascii* output port. To indicate the arrival of a new ASCII code, the *new_ascii* signal becomes one for exactly one clock cycle.

9 Text Mode Video Controller

9.1 Description

The text mode video controller core establishes a simple character (ASCII) based interface to an external LCD. As video mode 800x480 using 16 colors is used.

By default, the controller organizes the screen into 100 columns by 30 lines. All 256 ASCII characters are supported and decoded using the Western European Code-page (CP850)[CP8].

The controller implements a simple cursor interface. Characters are always written to the current cursor position. The newline and carrige return characters can be used for cursor control. In normal operation mode the cursor is automatically set to the next position. It is possible to switch this behavior off. In this mode, the cursor is not moved after a new character is written. Therefore, if two successive writes occur, the first character is replaced by the second one.

The cursor can be manually forwarded to the next position or freely positioned on the screen using special instructions.

By writing the backspace character, the cursor is moved to the previous writing position and the characters displayed at the current and the previous position are erased.

The cursor supports three states (on, off or blinking) and can be displayed in any color.

Furthermore the background color is not fixed but can be freely chosen.

The integration of the controller into a custom design is simple, as the implemented controller interface is straight forward. Furthermore the user interface is independent of the target hardware (always 16 colors and 8-bit ASCII data). All necessary adjustments are done by the controller internally. In the following sections, the controller and its interfaces will be described in more detail.

9.2 Dependencies

The controller has the following external dependencies:

- Mathematical support package (*math_pkg*)

Generic name	Functionality
CLK_FREQ	Actual clock frequency of the *clk* signal given in Hz.
ROW_COUNT	Number of displayed rows (Default is 30)
$COLUMN\_COUNT$	Number of displayed columns (Default is 100).

Table 21: Text-mode video controller generics description.

9.3 Required VHDL Files

The text mode video controller interface requires the following source-files:

- cursor_controller.vhd
- cursor_controller_pkg.vhd
- font_rom.vhd
- font_rom_beh.vhd
- font_rom_pkg.vhd
- textmode_controller_1c.vhd
- textmode_controller_fsm.vhd
- textmode_controller_pkg.vhd
- video_ram.vhd
- video_ram_pkg.vhd
- display_controller.vhd
- display_controller_pkg.vhd

9.4 Component Declaration

The declaration of the textmode video controller can be found in Listing 17, while the functionality of each generic is described in Table 21 and of each signal in Table 22.

Signal name	Direction	Signal width	Functionality
clk	in	1	Global clock signal.
res_n	in	1	Global reset signal (low active, not internally synchronized).
wr	in	1	If 1, an instruction is written to the controller.
busy	out	1	If 1, the controller is busy and can not accept new instructions.
instr	in	8	Instruction which should be executed by the controller.
instr_data	in	16	The data field of the instruction.
vd	out	1	Low active vertical synchronization signal (VGA interface).
hd	out	1	Low active horizontal synchronization signal (VGA interface).
den	out	1	If 1, color data is transmitted to the display, otherwise padding pixels are written.
r	out	8	Red color output (VGA interface).
g	out	8	Green color output (VGA interface).
b	out	8	Blue color output (VGA interface).
grest	out	1	Reset signal for the LCD.

Table 22: Text-mode video controller signal description.

```
component textmode_controller_1c is
  generic
  (
    ROW_COUNT : integer := 30;
    COLUM_COUNT : integer := 100;
    CLK_FREQ : integer := 25000000
  );
  port
  (
    clk : in  std_logic;
    res_n : in  std_logic;

    wr : in std_logic;
    busy : out std_logic;

    instr : in std_logic_vector(7 downto 0);
    instr_data : in std_logic_vector(15 downto 0);

    hd : out std_logic; -- horizontal sync signal
    vd : out std_logic; -- vertical sync signal
    den : out std_logic; -- data enable
    r : out std_logic_vector(7 downto 0); -- pixel color
       value (red)
    g : out std_logic_vector(7 downto 0); -- pixel color
       value (green)
    b : out std_logic_vector(7 downto 0); -- pixel color
       value (blue)

    grest : out std_logic -- display reset
  );
end component textmode_controller_1c;
```

Listing 17: Text-mode video controller declaration.

9.5 Instantiation Template

To be able to instantiate the textmode video controller, the package

- *textmode_controller_pkg*

must be included within your VHDL source. An instantiation template can be found in Listing 18.

For details on the type and size of the used signals, see Section 9.4.

```
textmode_inst : textmode_controller_1c
    generic map
    (
      ROW_COUNT => 30,
      COLUM_COUNT => 100,
      CLK_FREQ => VGA_CLK_FREQ
    )
    port map
    (
     clk => sys_clk,
     res_n => sys_res_n_sync,
     wr => textmode_wr,
     busy => textmode_busy,
     instr => textmode_instruction,
     instr_data => textmode_instruction_data,
     hd => hsync_n,
     vd => vsync_n,
     den => den,
     r => r,
     g => g,
     b => b,
     grest => vga_res_n_out
    );
```

Listing 18: Text-mode video controller instantiation template.

9.6 Interface Protocol

The video controller is operated by a simple instruction interface. To execute an instruction, it must be assigned to the *instr* port for exactly one cycle. At the same time, the corresponding data field must be applied to the *instr_data* input and the *wr* signal must be set to 1. The controller buffers the request internally and acknowledges its reception by setting the *busy* signal to high. As long as it remains high (it could stay high for multiple *clk* cycles), no new instruction will be accepted by the controller and therefore the instruction $INSTR\_NOP$ should be applied during this time frame. When the execution of the instruction is finished, the *busy* signal is set to low again. At this point a new instruction may be asserted.

9.6.1 Instruction Description

In the previous section we have described how to execute a instruction on the textmode video controller. Table 23 summarizes the available instructions.

Instruction	Data	Description
INSTR_NOP	15-0: Don't Care	No operation
INSTR_SET_CHAR	15-8: Attributes 7-0: Character code	Writes one character with the specified attributes to the current cursor position on the screen.
INSTR_DELETE	11-8: Background color Others: Don't care	Moves the cursor back one symbol and clears this position. Clearing means writing the character code 0x00 with the specified background color to the video ram.
INSTR_CLEAR_SCREEN	15-8: Attributes 7-0: Character code	Initializes the video ram with the specified data.
INSTR_MOVE_CURSOR_NEXT	11-8: Background color Others: Don't care	Moves the cursor to the next position. This instruction also moves the cursor to the next line, if required. If the auto scroll option is activated a background color is required to clear the newly created line.
INSTR_NEW_LINE	11-8: Background color Others: Don't Care	If the auto scroll option is activated a background color is required.
INSTR_CFG	11-8: Cursor color 3: Auto increment cursor 2: Auto scroll 1-0: Cursor state Others: Don't care	Used to configure the controller. The auto scroll/increment features are active after startup and can be deactivated by writing '1' to the specified bits. The cursor state can be set to OFF, ON or BLINKING.
INSTR_SET_CURSOR_POSITION	15-8: Column (x) 7-0: Row (y)	Moves the cursor to the specified position.

Table 23: Text-mode video controller instructions.

References

[CP8] Codepage 850. (e.g. Wikipedia: `http://de.wikipedia.org/wiki/CP850`).

[SCAa] Description of scan codes including a list of scan codes for US keyboards (http://www.win.tue.nl/~aeb/linux/kbd/scancodes-10.html).

[SCAb] Wikipedia scan code description (http://en.wikipedia.org/wiki/Scancode).