

**Digital Design and  
Computer Architecture LU**

# **IP Cores Manual**

Florian Huemer, Jürgen Maier  
{fhuemer, jmaier}@ecs.tuwien.ac.at  
Department of Computer Engineering  
TU Wien

Vienna, April 5, 2021

## Contents

<b>1</b>	<b>Mathematical Support Package</b>	<b>3</b>
1.1	Description . . . . .	3
1.2	Dependencies . . . . .	3
1.3	Required VHDL files . . . . .	3
1.4	Supported Functions . . . . .	3
<b>2</b>	<b>Synchronizer</b>	<b>4</b>
2.1	Description . . . . .	4
2.2	Dependencies . . . . .	4
2.3	Required VHDL Files . . . . .	4
2.4	Component Declaration . . . . .	4
2.5	Interface Protocol . . . . .	5
2.6	Internal Structure . . . . .	5
<b>3</b>	<b>On-chip RAM</b>	<b>6</b>
3.1	Description . . . . .	6
3.2	Dependencies . . . . .	6
3.3	Required VHDL Files . . . . .	6
3.4	Component Declarations . . . . .	6
3.4.1	Single clock dual-port RAM . . . . .	6
3.4.2	Single clock FIFO . . . . .	7
3.5	Interface Protocol . . . . .	8
3.5.1	Single clock dual-port RAM . . . . .	8
3.5.2	Single clock FIFO . . . . .	8
<b>4</b>	<b>Pseudo Random Number Generator (PRNG)</b>	<b>10</b>
4.1	Dependencies . . . . .	10
4.2	Required VHDL Files . . . . .	10
4.3	Component Declaration . . . . .	10
4.4	Interface Protocol . . . . .	11
<b>5</b>	<b>LCD Graphics Controller</b>	<b>12</b>
5.1	Features . . . . .	12
5.2	Dependencies . . . . .	13
5.3	Required VHDL Files . . . . .	14
5.4	Component Declaration . . . . .	14
5.5	Interface Protocol . . . . .	15
5.6	Graphics Interface Package . . . . .	19
5.7	Graphics Controller Package . . . . .	20

<b>6</b>	<b>Audio Controller</b>	<b>21</b>
6.1	Dependencies . . . . .	21
6.2	Required Source Files . . . . .	21
6.3	Component Declaration . . . . .	21
6.4	Interface Protocol . . . . .	22
<b>7</b>	<b>GFX Utility Package</b>	<b>24</b>
7.1	Description . . . . .	24
7.2	Dependencies . . . . .	24
7.3	Required VHDL Files . . . . .	24
7.4	Component Declarations . . . . .	24
7.4.1	GFX Line . . . . .	24
7.4.2	GFX Circle . . . . .	25
7.4.3	GFX Rectangle . . . . .	26
7.5	Interface Protocol . . . . .	27
<b>8</b>	<b>Object Collider</b>	<b>29</b>
8.1	Description . . . . .	29
8.2	Dependencies . . . . .	29
8.3	Required VHDL Files . . . . .	29
8.4	Component Declarations . . . . .	29
8.5	Interface Protocol . . . . .	31
	<b>Revision History</b>	<b>34</b>

# 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

- `function log2c(constant value : in integer) return integer;`  
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.
- `function max(constant value1, value2 : in integer) return integer;`  
`function max(constant value1, value2, value3 : in integer) return integer;`  
Determines the maximum of the integer operands. This function is available with two and three operands.

## 2 Synchronizer

### 2.1 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 not synchronous to internal FPGA clocks, using them without proper synchronization can lead to upsets and hence malfunction of a design.

### 2.2 Dependencies

- None

### 2.3 Required VHDL Files

- sync\_pkg.vhd
- sync.vhd

### 2.4 Component Declaration

The declaration of the synchronizer can be found in Listing 1, while the functionality of each generic and port signal is described in Table 2.1 and 2.2, respectively.

```

1 component sync is
2   generic (
3     SYNC_STAGES : integer range 2 to integer'high; -- Number of synchronizer stages
4     RESET_VALUE : std_logic -- Value of data_out directly after reset
5   );
6   port (
7     clk : in std_logic;
8     res_n : in std_logic;
9     data_in : in std_logic; -- External interface
10    data_out : out std_logic -- Internal interface
11  );
12 end component;
```

Listing 1: Synchronizer declaration.

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 2.1: Synchronizer generics description

Name	Dir.	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 2.2: Synchronizer signal description

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`.

## 2.5 Interface Protocol

The synchronizer has no special interface protocol. The input signal is sampled with the clock signal `clk`. Therefore an output signal generated which is aligned to the `clk` and has a delay of  $n$  clock cycles, where  $n$  is the number of synchronizer stages (i.e., `SYNC_STAGES`). Spikes or glitches not overlapping a rising clock edge (see example trace in Figure 2.1) will not show up at the synchronizer output.

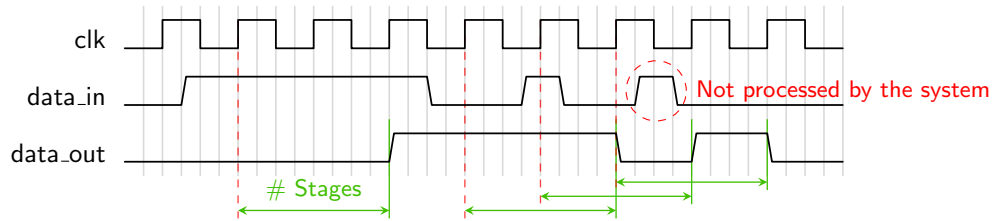


Figure 2.1: Synchronizer timing

## 2.6 Internal Structure

The synchronizer internally consists of a D flip-flop chain. Figure 2.2 shows an example of a three stage synchronizer.

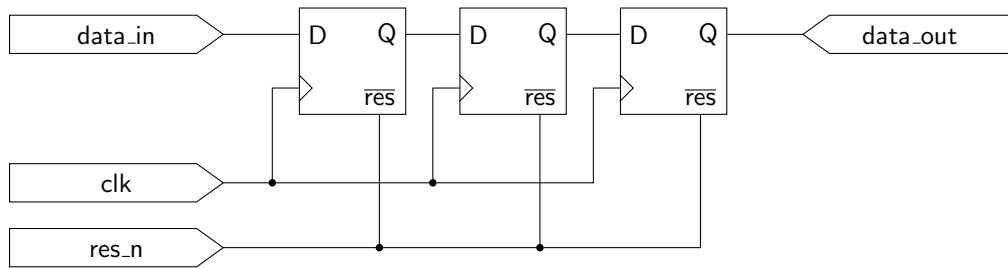


Figure 2.2: Synchronizer circuit

## 3 On-chip RAM

### 3.1 Description

Important components in nearly every integrated circuit are memories. If storage with full access speed is required, only on-chip memories are viable options. 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 one read and one write port and a single clock dual port FIFO with one read and one write port.

### 3.2 Dependencies

- Mathematical support package (math\_pkg)

### 3.3 Required VHDL Files

- ram\_pkg.vhd
- dp\_ram\_1c1r1w.vhd
- fifo\_1c1r1w.vhd

### 3.4 Component Declarations

#### 3.4.1 Single clock dual-port RAM

The declaration of the single clock dual-port RAM with one read and one write port can be found in Listing 2, while the functionality of each generic and port signal is described in Table 3.1 and 3.2.

```

1 component dp_ram_1c1r1w is
2   generic (
3     ADDR_WIDTH : integer;
4     DATA_WIDTH : integer
5   );
6   port (
7     clk : in std_logic;
8     -- read port
9     rd1_addr : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
10    rd1_data : out std_logic_vector(DATA_WIDTH - 1 downto 0);
11    rd1 : in std_logic;
12    --write port
13    wr2_addr : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
14    wr2_data : in std_logic_vector(DATA_WIDTH - 1 downto 0);
15    wr2 : in std_logic
16  );
17 end component;
```

Listing 2: RAM Component declaration

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

Table 3.1: RAM generics description

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
rd1_addr	in	ADDR.WIDTH	Address signal of the read port
rd1_data	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
wr2_addr	in	ADDR.WIDTH	Address signal of the write port
wr2_data	in	DATA.WIDTH	Data signal of the write port
wr2	in	1	If 1, the data of wr2_data is written to address wr2_addr of the memory

Table 3.2: RAM signal description

### 3.4.2 Single clock FIFO

The declaration of the single clock FIFO with one read and one write port can be found in Listing 3, while the functionality of each generic is described in Table 3.3 and of each signal in Table 3.4.

```

1 component fifo_1c1rw is
2   generic (
3     MIN_DEPTH : integer;
4     DATA_WIDTH : integer
5   );
6   port (
7     clk : in std_logic;
8     res_n : in std_logic;
9     --read port
10    rd_data : out std_logic_vector(DATA_WIDTH - 1 downto 0);
11    rd : in std_logic;
12    --write port
13    wr_data : in std_logic_vector(DATA_WIDTH - 1 downto 0);
14    wr : in std_logic;
15    --status signals
16    empty : out std_logic;
17    full : out std_logic;
18    half_full : out std_logic
19  );
20 end component;
```

Listing 3: FIFO declaration.

Name	Functionality
DEPTH	The depth of the FIFO. This generic must be set to a power of two.
DATA.WIDTH	The number of data bits

Table 3.3: FIFO generics description

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal, low active, not internally synchronized
rd_data	out	DATA.WIDTH	Output data
rd	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
wr_data	in	DATA.WIDTH	Data for the write operation
wr	in	1	If 1, the data of data.in2 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
half_full	out	1	1, if at least half of the memory of the FIFO contains data.

Table 3.4: FIFO signal description



### 3.5 Interface Protocol

#### 3.5.1 Single clock dual-port RAM

A standard synchronous memory access protocol is used for accessing the RAM. At any rising edge of the `clk` signal, when the `rd1` signal is high, the data word stored at address `rd1_addr` is written to the `rd1_data` port (see Figure 3.1).

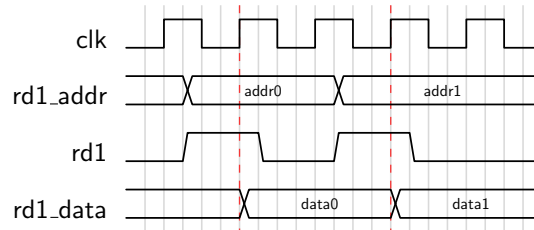


Figure 3.1: RAM read timing.

At any rising edge of the `clk` signal, when the `wr2` signal is high, the data word at `wr2_data` is written to address `wr2_addr` (see Figure 3.2).

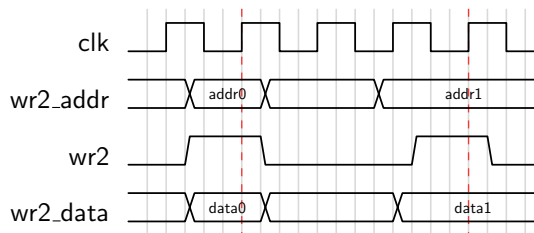


Figure 3.2: RAM write timing

#### 3.5.2 Single clock FIFO

The FIFO memory uses a similar interface but does not require the address inputs. The read operation is again initiated by asserting the `rd` signal. If the FIFO is not empty the next data word is assigned to the output `rd_data` (see Figure 3.3). If the FIFO is empty, the result of the read operation is undefined.

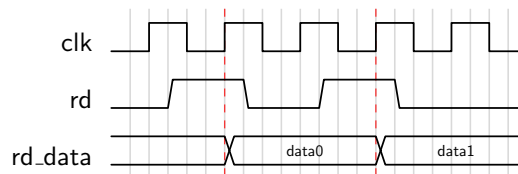


Figure 3.3: FIFO read timing

Asserting the input `wr` performs a write operation on the FIFO. The data word at `wr_data` is stored to the next free location of the internal memory (see Figure 3.4). While the FIFO is full, write operations are ignored.

If the first item is written to the FIFO, the `empty` signal becomes zero in 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 3.5).

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 3.6).

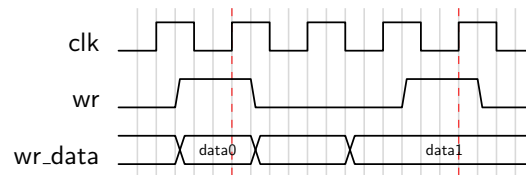


Figure 3.4: FIFO write timing

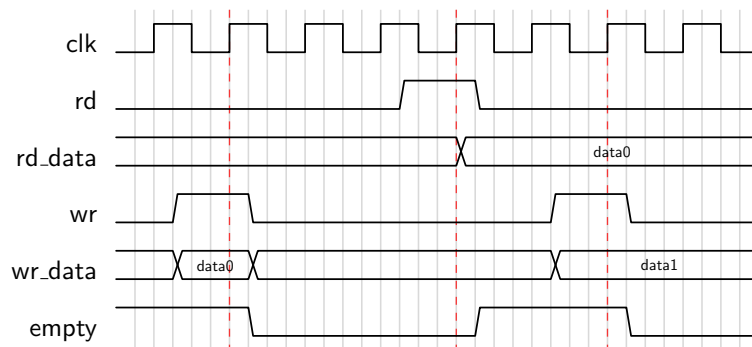


Figure 3.5: FIFO empty handling

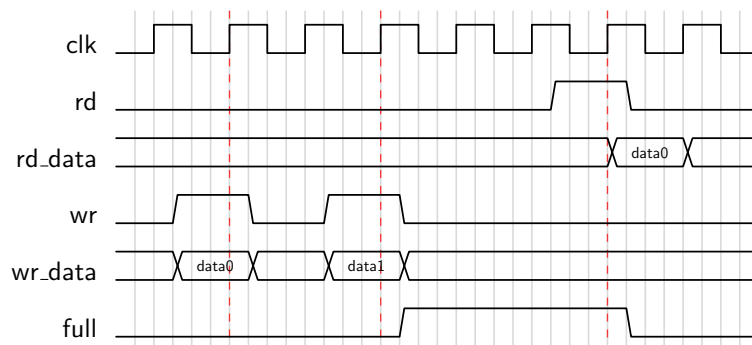


Figure 3.6: FIFO full handling

## 4 Pseudo Random Number Generator (PRNG)

The `prng` module implements a simple pseudo random number generator using a 16-bit linear feedback shift register (LFSR).

### 4.1 Dependencies

- None

### 4.2 Required VHDL Files

The `prng` is supplied as a precompiled module that comes in the form of a Quartus II Exported Partition File (.qxp) for synthesis and a netlist file (.vho) for simulation.

- `prng.vhd`
- `prng.vho`
- `prng.qxp`

Hence, if you want to simulate your design in Questa/Modelsim, use the package file `prng_pkg.vhd` and the netlist file `prng.vho`. For synthesis in Quartus use the package `prng_pkg.vhd` and the Exported Partition File `prng.qxp` file.

### 4.3 Component Declaration

The declaration of the `prng` can be found in Listing 4, the functionality of each signal is described in Table 4.1.

```

1 component prng is
2   port (
3     clk : in std_logic;
4     res_n : in std_logic;
5     load_seed : in std_logic;
6     seed : in std_logic_vector(7 downto 0);
7     en : in std_logic;
8     prdata : out std_logic
9   );
10 end component;
```

Listing 4: PRNG declaration

Name	Dir.	Width	Functionality
<code>clk</code>	in	1	Global clock signal
<code>res_n</code>	in	1	Global reset signal (low active not internally synchronized)
<code>en</code>	in	1	The enable signal for the PRNG. If <code>en</code> is high, the core outputs a new random bit at <code>prdata</code> in the next cycle.
<code>prdata</code>	out	1	The output holding the random data
<code>load_seed</code>	in	1	If <code>load_seed</code> is high, the core loads uses the value at the <code>seed</code> input to initialize its internal shift register and to configure the LFSR's polynomial. As long as <code>load_seed</code> is asserted, <code>en</code> has no effect.
<code>seed</code>	in	8	The seed value to (re-)initialize the core.

Table 4.1: PRNG signal description.

#### 4.4 Interface Protocol

Figure 4.1 shows an example timing diagram of the PRNG. During a reset the core is internally initialized with the seed value 0x00. The seed affects the initial value of the shift register as well as the polynomial used. This means that depending on the seed the period of the PRNG can be vastly different. Asserting `en` yields a new (pseudo) random bit at `prdata` at the next rising clock edge. In the example timing diagram shown in Figure 4.1 the core outputs the stream  $(r_0, r_1, r_2, r_3)$ . The timing diagram also shows how the PRNG is reinitialized with a (new) seed value. Since in this example again 0x00 is used to initialize the PRNG, the same sequence of random bits appears at the output.

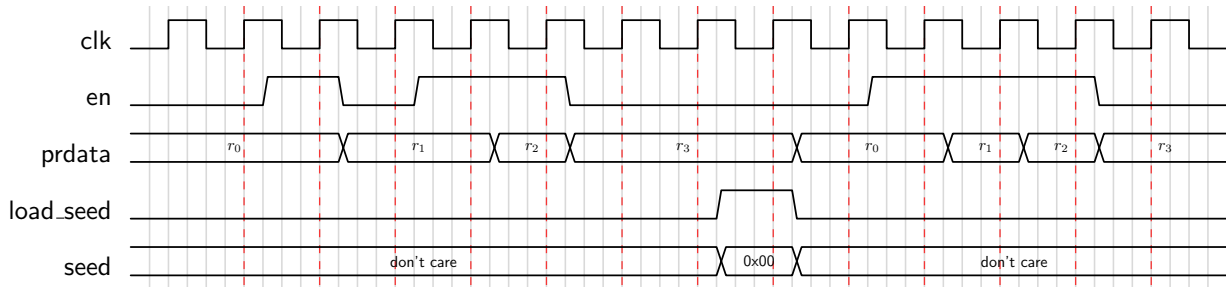


Figure 4.1: PRNG example timing diagram

## 5 LCD Graphics Controller

The LCD Graphics Controller is used to draw simple geometric shapes (lines, rectangles and circles) onto a display. It supports a color depth of 16 bits per pixel and a resolution of 400x240 pixels. For this purpose it utilizes an external SRAM to store the frame buffer (or buffers if double buffering is enabled), i.e., the video RAM. Figure 5.1 shows an overview of the architecture of this core. The FIFO-like instruction/data interface of the *rasterizer* sub component is used to issue commands to the core. The instructions are then executed over the course of multiple clock cycles using the cores in the *gfx\_util\_pkg* package, which will color the required pixels in the frame buffer in the external SRAM. The *framereader* constantly reads the SRAM and forwards the data to the *display controller*, which implements the display interface protocol.

For accessing the SRAM the LCD Graphics Controller uses an internal bus system with 16 bit data width (which is also the data width of the external SRAM). The access to the bus is controlled by the *bus arbiter* with priority scheduling. The *frame reader* has higher priority, since a certain minimum data rate (for reading the frame buffer) has to be guaranteed for the display controller to work correctly. Using the internal signals *base\_addr* and *frame\_start* the *frame reader* communicates with the *rasterizer*, to enable the synchronization of drawing operations to the start of a frame and to organize a frame buffer switch (*base\_addr* pointing to the video RAM address form where the *frame reader* fetches the data to be displayed).

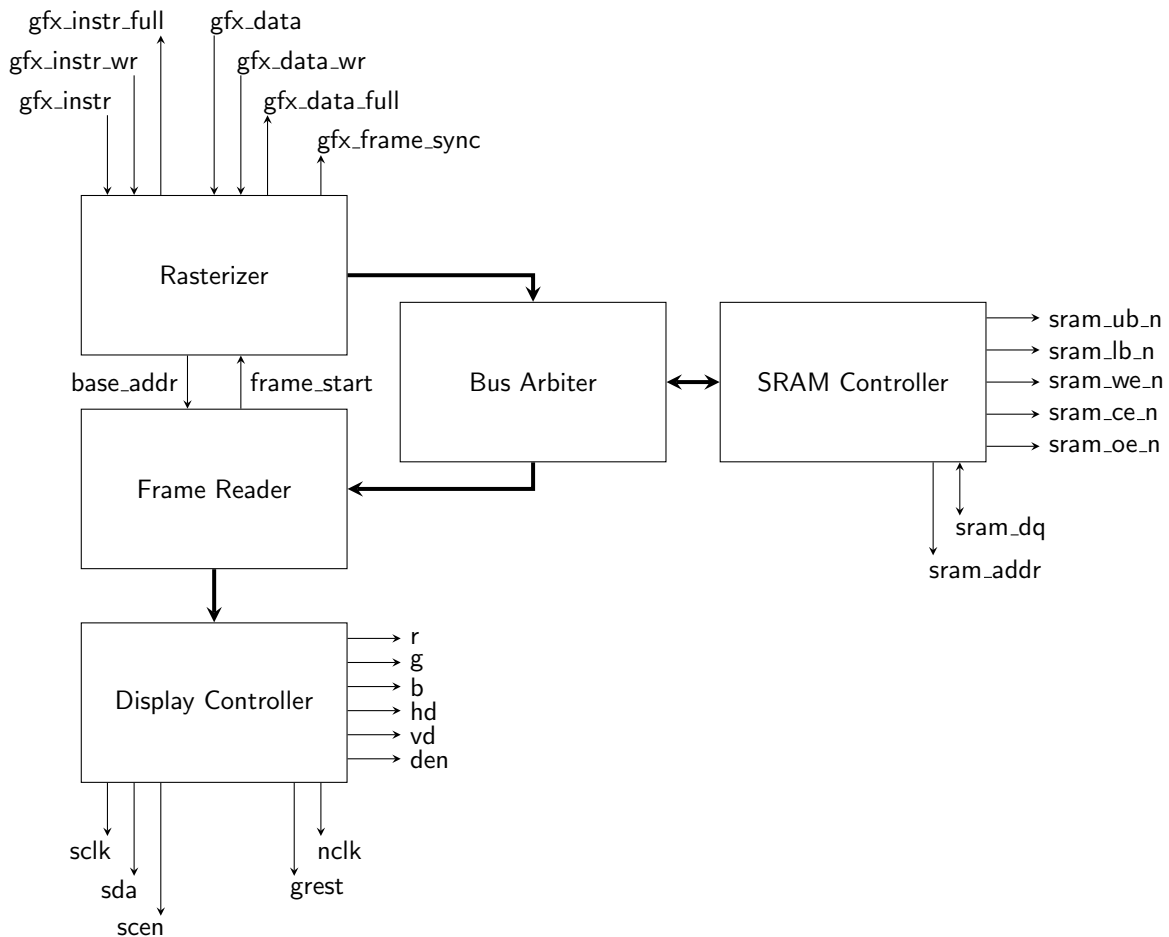


Figure 5.1: LCD Graphics Controller architecture overview

### 5.1 Features

These colors are user configurable via the `SET.COLOR` instruction<sup>1</sup>. The LCD Graphics Controller has two internal color registers referred to as the primary and secondary drawing color. Clearing the screen using

<sup>1</sup>Specific information about all supported instructions and their bit-level representations can be found in Section 5.5

the `CLEAR` instruction will use the secondary color. The graphical operations performed by the `SET_PIXEL`, `DRAW_LINE` and `DRAW_CIRCLE` instructions use the primary drawing color. The `DRAW_RECT` instruction also uses the primary color, but has certain variants where the secondary color is used as well.

Drawing a circle (i.e., `DRAW_CIRCLE`) only draws the outer perimeter. Rectangles can also be filled using simple patterns. A pattern is specified by the 5-tuple  $(bw, bh, dx, dy, ls)$ , consisting of the block width and block height parameters  $bw$  and  $bh$ , the x and y distances  $dx$  and  $dy$  as well as the line shift parameter  $ls$  (see Figure 5.2). Pixels that belong to a block (i.e., the black pixels in the figure), are drawn using the primary color. The others pixels are either drawn with the secondary color or not drawn at all (alpha mode), which means that the background “shines through” in these areas.

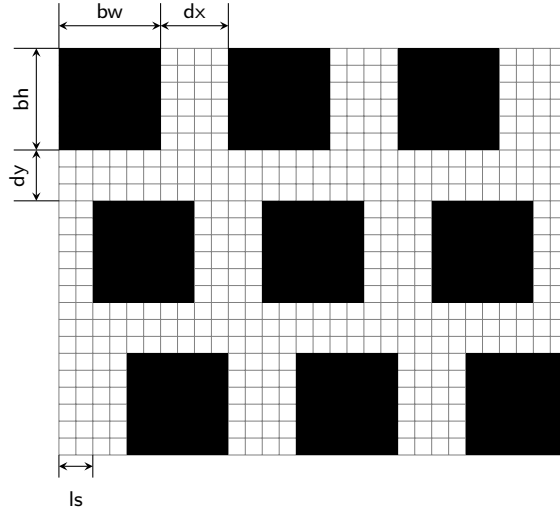


Figure 5.2: LCD Graphics Controller rectangle pattern specification

Figure 5.3 shows how various different pattern styles can be generated using this technique.

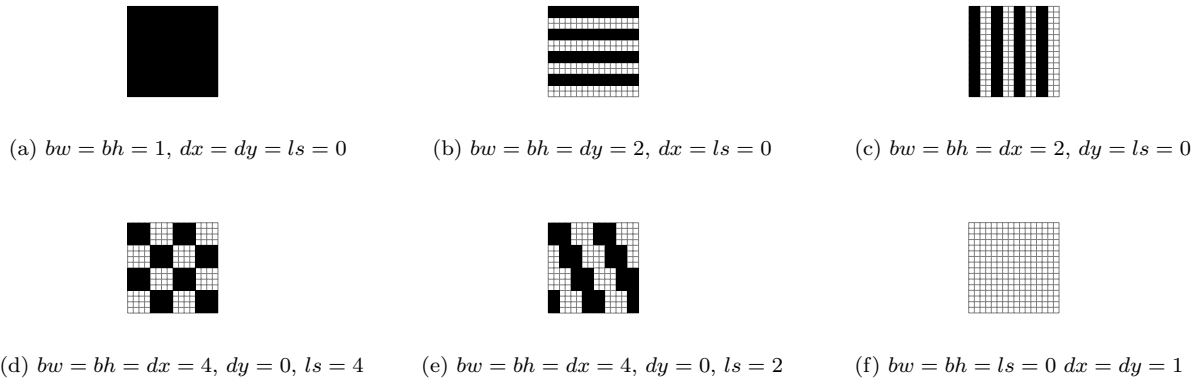


Figure 5.3: Rectangle fill patterns examples

The LCD Graphics Controller supports two predefined patterns and six user patterns which can be configured using the `SET_PATTERN` instruction. The predefined patterns correspond to patterns shown in Figures 5.3a and 5.3f. These patterns are meant to be used when a solid rectangle filled with the primary or secondary drawing color should be drawn.

Special instructions are used to enable double buffering and switch frame buffers.

## 5.2 Dependencies

Since the LCD Graphics Controller is provided as a precompiled module, there are no external dependencies.

### 5.3 Required VHDL Files

The LCD Graphics Controller is supplied as a precompiled module in the form of a Quartus II Exported Partition File (.qxp) for synthesis and a netlist file (.vho) for simulation. Additionally two packages (lcd\_graphics\_controller\_pkg and gfx\_if\_pkg) containing the component declaration and some utility functions and constants are provided.

- gfx\_if\_pkg.vhd
- lcd\_graphics\_controller\_pkg.vhd
- lcd\_graphics\_controller.qxp
- lcd\_graphics\_controller.vho

Hence, if you want to simulate your design in Questa/Modelsim, use the files gfx\_if\_pkg.vhd and lcd\_graphics\_controller\_pkg.vhd as well as the netlist file lcd\_graphics\_controller.vho. For synthesis in Quartus use gfx\_if\_pkg.vhd, lcd\_graphics\_controller\_pkg.vhd and the Exported Partition File lcd\_graphics\_controller.qxp file.

### 5.4 Component Declaration

The declaration of the LCD Graphics Controller can be found in Listing 5, the functionality of each signal is listed in Table 5.1. Figure 5.1 shows to which sub components of the LCD Graphics Controller these port signals are connected to (the clock and reset signals have been omitted from the figure).

```

1 component lcd_graphics_controller is
2   port (
3     clk      : in std_logic;
4     res_n    : in std_logic;
5     display_clk : in std_logic;
6     display_res_n : in std_logic;
7     --instruction interface
8     gfx_instr : in std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
9     gfx_instr_wr : in std_logic;
10    gfx_instr_full : out std_logic;
11    gfx_data      : in std_logic_vector(GFX_DATA_WIDTH-1 downto 0);
12    gfx_data_wr   : in std_logic;
13    gfx_data_full : out std_logic;
14    gfx_frame_sync : out std_logic;
15    --external interface to the SRAM
16    sram_dq : inout std_logic_vector(SRAM_DATA_WIDTH-1 downto 0);
17    sram_addr : out std_logic_vector(SRAM_ADDRESS_WIDTH-1 downto 0);
18    sram_ub_n : out std_logic;
19    sram_lb_n : out std_logic;
20    sram_we_n : out std_logic;
21    sram_ce_n : out std_logic;
22    sram_oe_n : out std_logic;
23    --external interface to the LCD
24    nclk : out std_logic;
25    hd   : out std_logic;
26    vd   : out std_logic;
27    den  : out std_logic;
28    r    : out std_logic_vector(7 downto 0);
29    g    : out std_logic_vector(7 downto 0);
30    b    : out std_logic_vector(7 downto 0);
31    grest : out std_logic;
32    --serial interface to LCD driver IC
33    sda : out std_logic;
34    scl : out std_logic;
35    scen : out std_logic;
36  );
37 end component;
```

Listing 5: LCD Graphics Controller declaration

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal (low active, not internally synchronized)
display_clk	in	1	The clock signal used to interface with the display (max. 8.3 MHz)
display_res_n	in	1	Display clock reset signal (low active, not internally synchronized)
gfx_frame_sync	out	1	The frame synchronization signal
gfx_instr	in	GFX_INSTR_WIDTH	The actual instruction
gfx_instr_wr	in	1	The write signal of the instruction FIFO
gfx_instr_full	out	1	The full signal of the instruction FIFO
gfx_data	in	GFX_DATA_WIDTH	The data associated with the instruction (coordinates, colors, etc.)
gfx_data_wr	in	1	The write signal of the data FIFO
gfx_data_full	out	1	The full signal of the data FIFO
sram_dq	inout	SRAM_DATA_WIDTH	SRAM data inputs/outputs
sram_addr	out	SRAM_ADDRESS_WIDTH	SRAM address input
sram_lb_n	out	1	SRAM lower-byte control
sram_ub_n	out	1	SRAM upper-byte control
sram_we_n	out	1	SRAM write enable
sram_ce_n	out	1	SRAM chip enable
sram_oe_n	out	1	SRAM output enable
nclk	out	1	LCD clock signal
grest	out	1	LDC Global reset, low active
hd	out	1	LCD Horizontal sync input
vd	out	1	LCD Vertical sync input
den	out	1	LCD RGB data enable
r	out	8	LCD red color data bus
g	out	8	LCD green color data bus
b	out	8	LCD blue color data bus
sda	out	1	Data signal of the serial interface to LCD Driver IC
scen	out	1	Chip select signal of the serial interface to LCD Driver IC
sclk	out	1	Clock signal of the serial interface to LCD Driver IC

Table 5.1: LCD Graphics Controller signal description

## 5.5 Interface Protocol

Internally the LCD Graphics Controller uses two FIFOs to buffer instructions and data issued to it. We refer to these FIFOs as the *instruction* and the *data* FIFO. Instructions are 8 bit wide, while data items have a width of 16 bit. The write ports of these FIFOs are exposed at the signals `gfx_instr`, `gfx_instr_wr` and `gfx_instr_full` as well as `gfx_data`, `gfx_data_wr` and `gfx_data_full`. Figure 5.4 shows how to operate these interfaces. The figure only shows the interface to the instruction FIFO, the data FIFO interface behaves identical. To issue a new instruction, it must be applied to the `gfx_instr` port and the `gfx_instr_wr` must be high for one clock cycle. Note that if `gfx_instr_full` is one, `gfx_instr_wr` must not be set to one.

Most instructions need additional data (coordinates, color information etc.) which has to be supplied using the data FIFO. Instructions can have 0 to 4 data FIFO entries associated with them. The sequence in which the data items and the actual instruction are pushed into the FIFOs does not matter. The data items can be written first and then the instruction or the other way around. An instruction is executed when all required data items are present. The LCD Graphics Controller uses three instruction formats shown in Figure 5.5, referred to as Formats A, B and C.

In the following all individual instructions supported by the LCD Graphics Controller are listed. Recall that the display resolution supported by the core is 400x240 pixels. Hence, for coordinate entries in the data



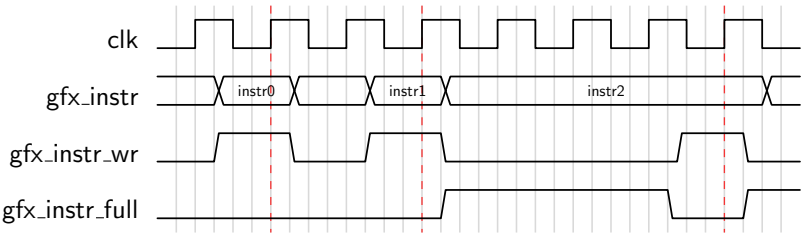


Figure 5.4: Instruction interface timing

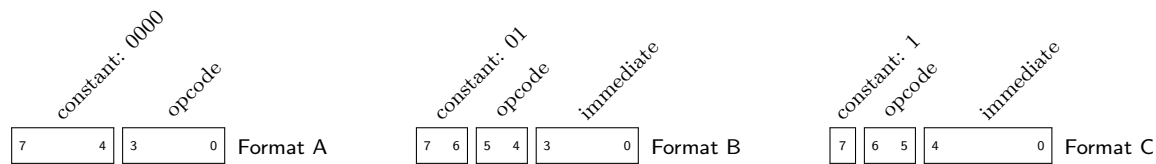
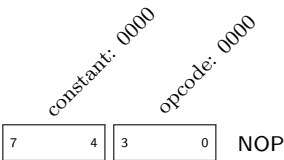
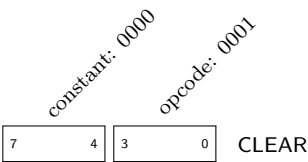


Figure 5.5: LCD Graphics Controller instruction formats

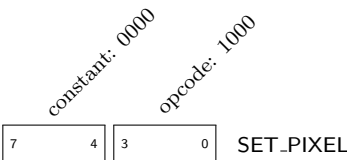
FIFO only the lower 9 (or 8) bits contain actual data, the unused bits should be set to zero. The origin of the display’s coordinate system (i.e., the point (0,0)) is the upper left corner.

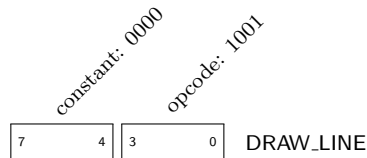
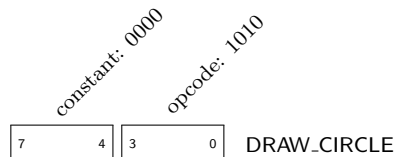
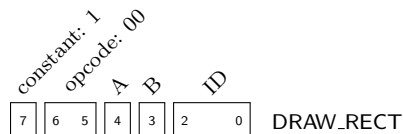


**Format A**  
**Data Operands** None  
**Description** Do nothing.



**Format A**  
**Data Operands** None  
**Description** Sets every pixel in the frame buffer to the secondary color.



**Format A****Data Operands**  $x, y$ **Description** Sets the pixel at the coordinates  $(x, y)$  to the primary drawing color.**Format A****Data Operands**  $x_0, y_0, x_1, y_1$ **Description** Draws a line from  $(x_0, y_0)$  to  $(x_1, y_1)$  using the primary drawing color.**Format A****Data Operands**  $x, y, r$ **Description** Draws a circle with the center at  $(x, y)$  and a radius of  $r$  using the primary drawing color.**Format C****Data Operands**  $x, y, w, h$ 

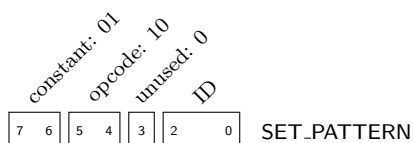
**Description** Draws a rectangle with the upper left corner at  $(x, y)$ , a width and height defined by the operands  $w$  and  $h$  and uses the pattern specified by ID to fill it. If B is set to one the border of the rectangle will be drawn (see Figure 5.6a). Figure 5.6b shows the same pattern without a border. Notice that the border does not affect the position of the blocks in the pattern. In both cases the first block of the pattern is drawn at position  $(x_0, y_0)$ , the border is then simply drawn “over” the pattern. The blocks of the pattern as well as the border are drawn using the primary color. Depending on the value of A, the other pixels will either be drawn with the secondary color (A=0), or not drawn at all (A=1, alpha mode).



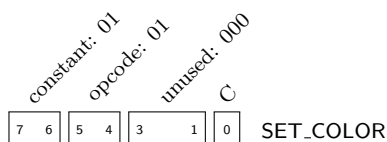
(a) Pattern with border



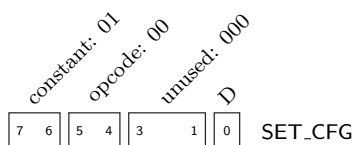
(b) Pattern without border

Figure 5.6: Rectangle pattern ( $bw = bh = 2$ ,  $dx = 4$ ,  $dy = 1$ ,  $ls = 3$ )**Format B****Data Operands**  $p_0$ ,  $p_1$ 

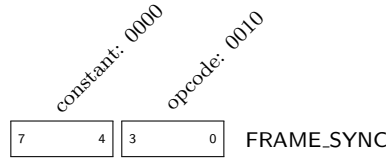
**Description** Sets the user pattern with the specified ID. Valid IDs are numbers from 1 to 6, the IDs 0 and 7 refer to the predefined patterns that cannot be changed. If such an ID is encountered, the instruction and the associated data items are simply ignored. The data item  $p_0$  contains the concatenated 4 bit pattern parameters  $dx$ ,  $dy$ ,  $bw$  and  $bh$  in that order from MSB to LSB. This means that the bits 15 to 12 correspond to  $dx$ , bits 11 to 8 to  $dy$  and so one. The data item  $p_1$  only contains the 5 bit parameter  $ls$  in its lower bits (4 to 0).

**Format B****Data Operands**  $color$ 

**Description** Depending on the value of C, this instruction either sets the primary (C=1) or the secondary (C=0) drawing color to  $color$ . Initially the primary color is set to 0xffff (i.e., white) and the secondary color is set to 0x0 (i.e., black).

**Format B****Data Operands** None

**Description** This instruction enables (D=1) or disables (D=0) double buffering. When double buffering is enabled the graphics controller uses two separate frame buffers. The contents of one buffer are output on the LCD while the other one is used for performing the drawing operations. Executing a `FRAME_SYNC` instructions switches those two buffers.



### Format A

**Data Operands** None

**Description** This instruction simply waits for the start of a new frame and then assert the `gfx.frame_sync` for exactly one clock cycle. It thus blocks the execution of the following instructions until the framereader starts to fetch a new frame. If double buffering is enabled, this instruction also switches the frame buffers after the new frame started.

## 5.6 Graphics Interface Package

The package `gfx_if_pkg` contains constants that define the opcodes for the various instructions as well as some helper functions to create instructions that can be used in e.g., testbenches.

The following listings shows the opcodes of all supported instructions.

```

1 --Format A opcodes
2 constant OP_CODE_NOP : std_logic_vector(3 downto 0) := x"0";
3 constant OP_CODE_CLEAR : std_logic_vector(3 downto 0) := x"1";
4 constant OP_CODE_FRAME_SYNC : std_logic_vector(3 downto 0) := x"2";
5 constant OP_CODE_SET_PIXEL : std_logic_vector(3 downto 0) := x"8";
6 constant OP_CODE_DRAW_LINE : std_logic_vector(3 downto 0) := x"9";
7 constant OP_CODE_DRAW_CIRCLE : std_logic_vector(3 downto 0) := x"a";
8 --Format B opcodes
9 constant OP_CODE_SET_CFG : std_logic_vector(1 downto 0) := "00";
10 constant OP_CODE_SET_COLOR : std_logic_vector(1 downto 0) := "01";
11 constant OP_CODE_SET_PATTERN : std_logic_vector(1 downto 0) := "10";
12 --Format C opcodes
13 constant OP_CODE_DRAW_RECT : std_logic_vector(1 downto 0) := "00";

```

For A type instructions the package also contains constant declarations for the complete 8-bit instruction:

```

1 constant GFX_INSTR_NOP : std_logic_vector(7 downto 0) := x"0" & OP_CODE_NOP;
2 constant GFX_INSTR_CLEAR : std_logic_vector(7 downto 0) := x"0" & OP_CODE_CLEAR;
3 constant GFX_INSTR_FRAME_SYNC : std_logic_vector(7 downto 0) := x"0" & OP_CODE_FRAME_SYNC;
4 constant GFX_INSTR_SET_PIXEL : std_logic_vector(7 downto 0) := x"0" & OP_CODE_SET_PIXEL;
5 constant GFX_INSTR_DRAW_LINE : std_logic_vector(7 downto 0) := x"0" & OP_CODE_DRAW_LINE;
6 constant GFX_INSTR_DRAW_CIRCLE : std_logic_vector(7 downto 0) := x"0" & OP_CODE_DRAW_CIRCLE;

```

B and C type instructions cannot be declared as constants since they contain immediate values. Hence here functions are used:

```

1 function GFX_INSTR_SET_CFG(enable_double_buffering : boolean) return std_logic_vector;
2 function GFX_INSTR_SET_COLOR(color_selector : boolean) return std_logic_vector;
3 function GFX_INSTR_SET_PATTERN(pattern_id : integer) return std_logic_vector;
4 function GFX_INSTR_DRAW_RECT(enable_alpha : boolean; draw_border : boolean; pattern_id :
   integer) return std_logic_vector;

```

Additionally there are functions to test whether a given (8-bit) vector is an A, B or C type instruction.

```
1 function is_instr_format_A(instr : std_logic_vector) return boolean;  
2 function is_instr_format_B(instr : std_logic_vector) return boolean;  
3 function is_instr_format_C(instr : std_logic_vector) return boolean;
```

## 5.7 Graphics Controller Package

The package `lcd_graphics_controller_pkg` contains the component declaration of the core as well as constants and functions to process color values. The following constants define the width of the individual color channels (red, green and blue) in the 16-bit color values.

```
1 constant GCNTL_COLOR_RED_WIDTH : integer := 5;  
2 constant GCNTL_COLOR_GREEN_WIDTH : integer := 6;  
3 constant GCNTL_COLOR_BLUE_WIDTH : integer := 5;
```

To extract a particular channel the following functions can be used.

```
1 function get_blue(color : std_logic_vector) return std_logic_vector;  
2 function get_green(color : std_logic_vector) return std_logic_vector;  
3 function get_red(color : std_logic_vector) return std_logic_vector;
```

## 6 Audio Controller

The `audio_cntrl` module implements a simple synthetic sound generator that interfaces with the board's audio DAC (digital to analog converter) WM8731. This chip has two separate (serial) interfaces, one for configuration purposes (control interface) and another one to receive the actual audio samples (digital audio interface). The control interface is only required during start-up to configure the sampling rate and set up the digital audio interface. Figure 6.1 shows to the general structure of the audio controller.

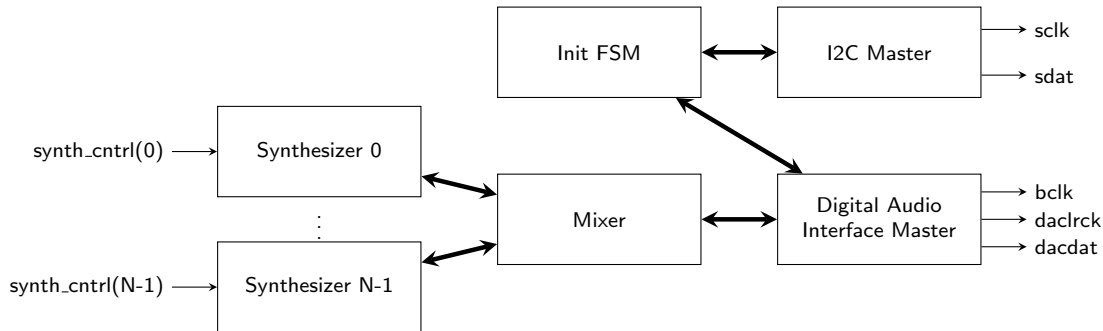


Figure 6.1: Audio controller internal structure

The audio controller must be clocked by a 12 MHz clock (which will internally be forwarded to the `xclk` output). The `synth_cntrl` signals can be written from any clock domain since the core uses synchronizers to bring the required signals into its (12 MHz) clock domain (see Section 6.4).

Note that the audio controller is provided as a precompiled module with two synthesizers (`SYNTH_COUNT = 2`).

### 6.1 Dependencies

Since the audio controller is provided as a precompiled module, there are no external dependencies.

### 6.2 Required Source Files

The audio controller is supplied as a precompiled module in the form of a Quartus II Exported Partition File (.qxp) for synthesis and a netlist file (.vho) for simulation. Additionally a wrapper module `audio_cntrl_s2` is required. The `audio_cntrl_pkg` package provides the component declaration as well as the required type declaration for the synthesizer interface.

- `audio_cntrl_pkg.vhd`
- `audio_cntrl_s2.vhd`
- `audio_cntrl_top.vho`
- `audio_cntrl_top.qxp`

Hence, if you want to simulate your design in Questa/Modelsim, use the files `audio_cntrl_s2.vhd` and `audio_cntrl_pkg.vhd` as well as the netlist file `audio_cntrl_top.vho`. For synthesis in Quartus use `audio_cntrl_s2.vhd` and `audio_cntrl_pkg.vhd` and the Exported Partition File `audio_cntrl_top.qxp` file.

### 6.3 Component Declaration

The declaration of the audio controller can be found in Listing 6, the functionality of each signal in Table 6.1.

```

1 component audio_cntrl_2s is
2   port (
3     clk      : in std_logic; --12 MHz input clock
4     res_n    : in std_logic;
5
6     --clock output signal for the wm8731
7     wm8731_xck      : out std_logic;
8
9     --cfg interface to wm8731: i2c configuration interface
10    wm8731_sdat : inout std_logic;
11    wm8731_sclk : inout std_logic;
12
13    --data interface to wm8731: digital audio interface
14    wm8731_dacdat : out std_logic;
15    wm8731_daclrck : out std_logic;
16    wm8731_bclk  : out std_logic;
17
18    --internal interface to the stynthesizers
19    synth_cntrl : in synth_cntrl_vec_t(0 to 1)
20  );
21 end component;

```

Listing 6: Audio controller declaration

Name	Dir.	Width	Functionality
clk	in	1	12 MHz clock signal
res_n	in	1	Reset signal (low active, not internally synchronized)
wm8731_xck	out	1	The 12 MHz clock signal from the clk input.
wm8731_sdat	inout	1	The data signal of the I2C bus of the WM8731's control interface
wm8731_sclk	inout	1	The clock signal of the I2C bus of the WM8731's control interface
wm8731_dacdat	out	1	DAC Digital Audio Data Input of the WM8731's digital audio interface
wm8731_daclrck	out	1	DAC Sample Rate Left/Right Clock of the WM8731's digital audio interface
wm8731_bclk	out	1	Digital Audio Bit Clock of the WM8731's digital audio interface
synth_cntrl	in	synth_cntrl_vec.t (0 to 1)	The synthesizer control signals

Table 6.1: audio\_cntrl signal description

## 6.4 Interface Protocol

To interface with the audio controller, the `synth_cntrl` input is used, which allows to control the individual synthesizers. This signal is a 2-element vector of the record type `synth_cntrl_t` shown below.

```

1 type synth_cntrl_t is record
2   play : std_logic;
3   high_time : std_logic_vector(7 downto 0);
4   low_time : std_logic_vector(7 downto 0);
5 end record;

```

Every synthesizer produces a PWM signal which can be configured via the `high_time` and `low_time` entries of this record. These values have to be interpreted with respect to the sampling frequency of the DAC (in this case 8 KHz). If both values are 1, the maximum frequency output signal is generated. This means that in this case the actual samples that are sent to the DAC switch between the maximum and minimum value at every sampling period.

The high-active `play` signal controls the sound play-back, i.e., as long as this signal is high, the respective is played. When the `play` signal switches from low to high, the synthesizer reads the current values of `high_time` and `low_time` and uses those values to generate the PWM signal until `play` returns to zero again (this means that changing those values while `play` is high has no effect). Hence, to change the PWM signal, the `play` signal must be low for at least one clock cycle (of the 12MHz input clock of the audio controller).

Since the audio controller can be controlled from any clock domain, care must be taken, to correctly handle the clock domain crossing. For that purpose, the core uses 3-stage synchronizers on the **play** signals. The **high\_time** and **low\_time** are not synchronized! This means that whenever these values are changed, it must be made sure that they are stable long enough such that the audio controller can sample them, without errors. Hence one has to take the synchronization delay into account.



## 7 GFX Utility Package

### 7.1 Description

The `gfx_util_pkg` package contains three cores that implement simple graphical operations:

- **GFX Line (`gfx_line`)**  
Draws a line between two points.
- **GFX Circle (`gfx_circle`)**  
Draws a circle given a center point and a radius.
- **GFX Rectangle (`gfx_rect`)**  
Draws rectangles and (optionally) fills them with simple user definable patterns.

All three cores in this package have very similar interfaces. The meaning of the input/output signals `start`, `stall`, `busy`, `pixel_valid`, `pixel_x` and `pixel_y` is the same for all cores. After a drawing operation has been started (using the `start` input) the cores output a sequence of coordinates that describe the respective geometric shape (using the output signals `pixel_valid`, `pixel_x` and `pixel_y` and in the case of `gfx_rect` also `pixel_color`). These coordinates can then be used to e.g., calculate and access a memory location that stores the color information for the pixel.

### 7.2 Dependencies

- Mathematical support package (`math_pkg`)

### 7.3 Required VHDL Files

- `gfx_util_pkg.vhd`
- `gfx_line.vhd`
- `gfx_circle.vhd`
- `gfx_rect.vhd`

### 7.4 Component Declarations

Note that all three cores have the same generics (Table 7.1).

#### 7.4.1 GFX Line

The declaration of the `gfx_line` module can be found in Listing 7, while the functionality of each generic and port signal is described in Table 7.1 and 7.2.

```

1 component gfx_line is
2   generic (
3     WIDTH : integer;
4     HEIGHT : integer
5   );
6   port (
7     clk : in std_logic;
8     res_n : in std_logic;
9     start : in std_logic;
10    stall : in std_logic;
11    busy : out std_logic;
12    x0 : in std_logic_vector(log2c(WIDTH)-1 downto 0);
13    x1 : in std_logic_vector(log2c(WIDTH)-1 downto 0);
14    y0 : in std_logic_vector(log2c(HEIGHT)-1 downto 0);
15    y1 : in std_logic_vector(log2c(HEIGHT)-1 downto 0);

```

```

16 pixel_valid : out std_logic;
17 pixel_x : out std_logic_vector(log2c(WIDTH)-1 downto 0);
18 pixel_y : out std_logic_vector(log2c(HEIGHT)-1 downto 0);
19 );
20 end component;

```

Listing 7: gfx\_line Component declaration

Name	Functionality
WIDTH	The width of the target image. Determines the vector length of input/output signals that carry $x$ coordinates.
HEIGHT	The height of the target image. Determines the vector length of input/output signals that carry $y$ coordinates.

Table 7.1: Generics description for gfx\_line, gfx\_circle and gfx\_rect

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Low active reset signal
start	in	1	This signal is used initiate the drawing operation. It must be asserted for exactly one clock cycle. The core will react to this event by asserting the busy signal. The drawing parameters must be valid when this start is asserted and must remain valid (and unchanged) until the busy signal goes low.
stall	in	1	This signals can be used to pause the drawing operation. Asserting it will cause the pixel_valid to remain low, until the signal is deasserted again. If this functionality is not required, this input can be driven with constant '0'.
busy	out	1	The core asserts this signal to indicate that it is currently performing a drawing operation. As soon as the drawing operation is complete, the busy goes low again, which allows for a new drawing operation to be started.
x0	in	WIDTH	The $x$ coordinate of the first point of the line (i.e., the start point of the line). [drawing parameter]
y0	in	HEIGHT	The $y$ coordinate of the first point of the line (i.e., the start point of the line). [drawing parameter]
x1	in	WIDTH	The $x$ coordinate of the second point of the line (i.e., the end point of the line). [drawing parameter]
y1	in	HEIGHT	The $y$ coordinate of the second point of the line (i.e., the end point of the line). [drawing parameter]
pixel_valid	out	1	This signal indicates that the current data at the pixel.* is valid. It will only go high during a drawing operation, i.e., when busy is asserted.
pixel_x	out	WIDTH	The $x$ coordinate of the output pixel.
pixel_y	out	HEIGHT	The $y$ coordinate of the output pixel.

Table 7.2: gfx\_line signal description

### 7.4.2 GFX Circle

The declaration of the gfx\_circle module can be found in Listing 8, while the functionality of each generic and port signal is described in Table 7.1 and 7.3.

```

1 component gfx_circle is
2   generic (
3     WIDTH : integer;
4     HEIGHT : integer
5   );
6   port (
7     clk : in std_logic;
8     res_n : in std_logic;
9     start : in std_logic;
10    stall : in std_logic;
11    busy : out std_logic;
12    x_center : in std_logic_vector(log2c(WIDTH)-1 downto 0);
13    y_center : in std_logic_vector(log2c(HEIGHT)-1 downto 0);

```

```

14 radius : in std_logic_vector(log2c(WIDTH)-1 downto 0);
15 pixel_valid : out std_logic;
16 pixel_x : out std_logic_vector(log2c(WIDTH)-1 downto 0);
17 pixel_y : out std_logic_vector(log2c(HEIGHT)-1 downto 0)
18 );
19 end component;

```

Listing 8: gfx\_line Component declaration

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Low active reset signal
start	in	1	This signal is used initiate the drawing operation. It must be asserted for exactly one clock cycle. The core will react to this event by asserting the busy signal. The drawing parameters must be valid when this start is asserted and must remain valid (and unchanged) until the busy signal goes low.
stall	in	1	This signals can be used to pause the drawing operation. Asserting it will cause the pixel_valid to remain low, until the signal is deasserted again. If this functionality is not required, this input can be driven with constant '0'.
busy	out	1	The core asserts this signal to indicate that it is currently performing a drawing operation. As soon as the drawing operation is complete, the busy goes low again, which allows for a new drawing operation to be started.
x	in	WIDTH	The $x$ coordinate of the center point of the circle. [drawing parameter]
y	in	HEIGHT	The $y$ coordinate of the center point of the circle. [drawing parameter]
radius	in	WIDTH	The radius $r$ of the circle. Note that the resulting circle has a diameter of $2 * r + 1$ pixels. [drawing parameter]
pixel_valid	out	1	This signal indicates that the current data at the pixel.* is valid. It will only go high during a drawing operation, i.e., when busy is asserted.
pixel_x	out	WIDTH	The $x$ coordinate of the output pixel.
pixel_y	out	HEIGHT	The $y$ coordinate of the output pixel.

Table 7.3: gfx\_circle signal description

### 7.4.3 GFX Rectangle

The declaration of the gfx\_line module can be found in Listing 9, while the functionality of each generic and port signal is described in Table 7.1 and 7.4. For more information on the input signals bw, bh, dx, dy and ls please refer to the Section 5.1.

```

1 component gfx_rect is
2   generic (
3     WIDTH : integer;
4     HEIGHT : integer
5   );
6   port (
7     clk : in std_logic;
8     res_n : in std_logic;
9     start : in std_logic;
10    stall : in std_logic;
11    busy : out std_logic;
12    x : in std_logic_vector(log2c(WIDTH)-1 downto 0);
13    y : in std_logic_vector(log2c(HEIGHT)-1 downto 0);
14    w : in std_logic_vector(log2c(WIDTH)-1 downto 0);
15    h : in std_logic_vector(log2c(HEIGHT)-1 downto 0);
16    bw : in std_logic_vector(3 downto 0);
17    bh : in std_logic_vector(3 downto 0);
18    dx : in std_logic_vector(3 downto 0);
19    dy : in std_logic_vector(3 downto 0);
20    ls : in std_logic_vector(4 downto 0);
21    fill : in std_logic;
22    draw : in std_logic;
23    pixel_valid : out std_logic;
24    pixel_x : out std_logic_vector(log2c(WIDTH)-1 downto 0);
25    pixel_y : out std_logic_vector(log2c(HEIGHT)-1 downto 0);

```

```

26 pixel_color : out std_logic
27 );
28 end component;

```

Listing 9: gfx\_line Component declaration

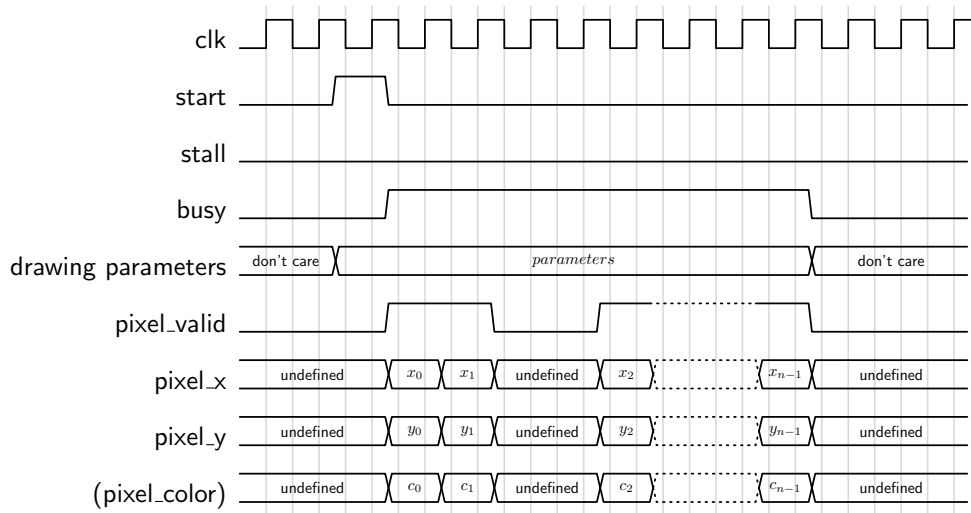
Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Low active reset signal
start	in	1	This signal is used initiate the drawing operation. It must be asserted for exactly one clock cycle. The core will react to this event by asserting the busy signal. The drawing parameters must be valid when this start is asserted and must remain valid (and unchanged) until the busy signal goes low.
stall	in	1	This signals can be used to pause the drawing operation. Asserting it will cause the pixel_valid to remain low, until the signal is deasserted again. If this functionality is not required, this input can be driven with constant '0'.
busy	out	1	The core asserts this signal to indicate that it is currently performing a drawing operation. As soon as the drawing operation is complete, the busy goes low again, which allows for a new drawing operation to be started.
x	in	WIDTH	The $x$ coordinate of the upper left corner of the rectangle. [drawing parameter]
y	in	HEIGHT	The $y$ coordinate of the upper left corner of the rectangle. [drawing parameter]
w	in	WIDTH	The width of the rectangle. [drawing parameter]
h	in	HEIGHT	The height of the rectangle. [drawing parameter]
bw	in	4	The block width used for the fill pattern (only relevant when fill is one). [drawing parameter]
bh	in	4	The block height used for the fill pattern (only relevant when fill is one). [drawing parameter]
dx	in	4	The $x$ distance used for the fill pattern (only relevant when fill is one). [drawing parameter]
dy	in	4	The $y$ distance used for the fill pattern (only relevant when fill is one). [drawing parameter]
ls	in	5	The line shift used for the fill pattern (only relevant when fill is one). [drawing parameter]
draw	in	1	Specifies whether the core should draw the rectangle's border. [drawing parameter]
fill	in	1	Specifies whether the core should fill the rectangle with the pattern specified by the inputs bw, bh, dx, dy and ls. [drawing parameter]
pixel_valid	out	1	This signal indicates that the current data at the pixel_* is valid. It will only go high during a drawing operation, i.e., when busy is asserted.
pixel_x	out	WIDTH	The $x$ coordinate of the output pixel.
pixel_y	out	HEIGHT	The $y$ coordinate of the output pixel.
pixel_color	out	1	This signals indicates whether the pixel at the current coordinates (pixel_x, pixel_y) is part of a pattern block (pixel_color='1') or the border (pixel_color='0'). A value of '0' indicates that the pixel neither belongs to a block/border.

Table 7.4: gfx\_rect signal description

## 7.5 Interface Protocol

Because the interface protocols of the cores are essentially the same, only one example timing diagram is presented (Figure 7.1). The trace labeled *drawing parameters* refers to the set of signals marked as drawing parameters in the signal description tables. These signals must be kept stable throughout the whole drawing process (i.e., when start is asserted until busy is deasserted).

When the drawing process has been started the core outputs  $n$  pixel coordinates, where  $n$ , of course, depends on the size of the geometric shape currently drawn. Whenever a new set of  $(x, y)$  coordinates has been calculated by the core the pixel\_valid signal is asserted for exactly one clock cycle. Note, however, that the core may not output new coordinate data on every cycle. In the example shown in Figure 7.1 the core

Figure 7.1: Example timing diagram for `gfx.line`, `gfx.circle` and `gfx.rect`

introduces a 2 cycle long “break” between the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . The outputs `pixel_x`, `pixel_y` and `pixel_color` (for the `gfx.rect` core) must only be used when `pixel_valid` is asserted, otherwise their value is undefined.

The behavior of the core with respect to `stall` is not shown, because you won’t need it for the exercise. Don’t supply coordinates to the cores that cause them to draw “outside” of the image dimensions specified by the generics `WIDTH` and `HEIGHT`.

## 8 Object Collider

### 8.1 Description

The `object_collider` is intended to be used by simple games to move the player object across a 2D game board (i.e., the screen), where it can collide with other game objects. For that purpose the record type `game_object.t` shown in the listing below (defined in the `object_collider_pkg` package) is used. It keeps track of the relevant information associated with a game object.

```

1  type game_object_t is
2  record
3      id : std_logic_vector(GAME_OBJECT_ID_WIDTH-1 downto 0);
4      x  : std_logic_vector(COORDINATE_WIDTH-1 downto 0);
5      y  : std_logic_vector(COORDINATE_WIDTH-1 downto 0);
6      w  : std_logic_vector(COORDINATE_WIDTH-1 downto 0);
7      h  : std_logic_vector(COORDINATE_WIDTH-1 downto 0);
8  end record;
```

As the definition of the `game_object.t` record implies, all game object are treated as rectangles for the collision detection. The `id` field of the record is not used by the core, i.e., its value has no effect on its operation. Its intended use is to attach other information to a game object that is not directly relevant for the movement/collision detection. To use the core it has to be supplied with a game object representing the player (input `player`), as well as information about the intended movement (inputs `player_speed`, `player_dir`, `gravity`, `apply_movement` and `apply_gravity`). The movement operation is then initiated using the `start` input. During a movement operation, which we also refer to as a *run*, the `object_collider` goes through the “list” of all game object placed on the screen multiple times. To access this list the signals `object_req`, `object_valid`, `object.is_blocking`, `object.eol` and `object` are used. Collisions with other game objects are indicated using the `collision_detected` signal. Finally the core uses the `done` output to signal that the current run is complete. The final position of the player is then reported using the `player_x` and `player_y` outputs.

The `object_collider` also performs collision detection between the player and the left and right border of the screen. This means that it will not let the player move to an `x` position less than 0 or to values larger than or equal to `DISPLAY_WIDTH`. It will however, not check for collisions between to top or bottom edge of the screen.

### 8.2 Dependencies

- None

### 8.3 Required VHDL Files

- `object_collider_pkg.vhd`
- `object_collider.vhd`

### 8.4 Component Declarations

The declaration of the `object_collider` module can be found in Listing 10, while the functionality of each generic and port signal is described in Table 8.1 and 8.2.

```

1  component object_collider is
2      generic (
3          DISPLAY_WIDTH : integer := 400;
4          DISPLAY_HEIGHT : integer := 240
5      );
6      port (
7          clk : in std_logic;
8          res_n : in std_logic;
9          start : in std_logic;
10         done : out std_logic;
11         apply_movement : in std_logic;
```

```

12  apply_gravity : in std_logic;
13  player_x : out std_logic_vector(COORDINATE_WIDTH-1 downto 0);
14  player_y : out std_logic_vector(COORDINATE_WIDTH-1 downto 0);
15  player : in game_object_t;
16  player_speed : in std_logic_vector(GAME_OBJECT_SPEED_WIDTH-1 downto 0);
17  player_dir : in std_logic;
18  gravity : in std_logic_vector(GAME_OBJECT_SPEED_WIDTH-1 downto 0);
19  object : in game_object_t;
20  object_req : out std_logic;
21  object_valid : in std_logic;
22  object_is_blocking : in std_logic;
23  object_eol : in std_logic;
24  collision_detected : out std_logic
25 );
26 end component;

```

Listing 10: object.collider Component declaration

Name	Functionality
DISPLAY_WIDTH	The width of the game board (screen)
DISPLAY_HEIGHT	The height of the game board (screen)

Table 8.1: object.collider generics description

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Low active reset signal
start	in	1	Asserting this signal for one clock cycle starts the player movement process (i.e., the run).
done	out	1	This signal indicates when the current run is complete. It will be asserted for exactly one clock cycle.
player	in	game_object.t	The game object representing the player. Must be valid exactly when <b>start</b> is asserted.
player_speed	in	GAME.OBJECT_SPEED.WIDTH	The horizontal speed of the player, i.e., the maximal horizontal distance it is allowed to be moved by the object collider during one run. Must be valid exactly when <b>start</b> is asserted.
player_dir	in	1	The direction the player should move in ('0' → Left, '1' → Right). Must be valid exactly when <b>start</b> is asserted.
gravity	in	GAME.OBJECT_SPEED.WIDTH	The vertical speed of the player, i.e., the maximal vertical distance it is allowed to be moved by the object collider during one run. Must be valid exactly when <b>start</b> is asserted.
apply_movement	in	1	A flag indicating if the player should be moved horizontally during the run. Must be valid exactly when <b>start</b> is asserted.
apply_gravity	in	1	A flag indicating if the player should be moved vertically during the run. Must be valid exactly when <b>start</b> is asserted.
object_req	out	1	This signal is used by the core to request the next game object from the game object list to check for a collision with the player. This signal stays asserted until either <b>object.valid</b> or <b>object.eol</b> go high.
object	in	game_object.t	The actual game object requested by the core. This signal must only be changed when the <b>object.valid</b> is being asserted, otherwise it has to keep its value.
object.valid	in	1	This signal indicates that the data at the <b>object</b> and <b>object.is_blocking</b> inputs is valid. It must only be asserted for exactly one clock cycle and only as a response to an object request ( <b>object_req</b> going high).
object.is_blocking	in	1	This flag indicates that the game object supplied at the <b>object</b> input is blocking. Blocking object stop the movement of the player and collisions with them are not reported using the <b>collision.detected</b> output signal (i.e., for a blocking object <b>collision.detected</b> will always be deasserted). This input must only be changed when the <b>object.valid</b> is being asserted, otherwise it has to keep its value.
object.eol	in	1	The object end-of-list signal is asserted instead of <b>object.valid</b> to notify the core that all objects from the game object list have been processed. It must only be asserted for exactly one clock cycle and only as a response to an object request ( <b>object_req</b> going high).
collision.detected	out	1	This output indicates whether the player collided with the game object currently applied at the <b>object</b> input. It is only valid for exactly one clock cycle, one clock cycles after <b>object.valid</b> has been asserted (i.e., after asserting <b>object.valid</b> there is one wait cycle before <b>collision.detected</b> can be evaluated).
player_x	out	COORDINATE_WIDTH	The final horizontal position of the player. Valid as soon as the <b>done</b> output is asserted until a new run is started using the <b>start</b> input.
player_y	out	COORDINATE_WIDTH	The final vertical position of the player. Valid when <b>player_x</b> is valid.

Table 8.2: object\_collider signal description

## 8.5 Interface Protocol

Figure 8.1 shows an example timing diagram, demonstrating its interface protocol. Trace sections marked with D/C, indicate “don’t care” values. In the beginning the **start** signal is asserted to initiate the run. To perform its operation the core also needs a game object at the **player** input as well as values for all the other movement parameters (i.e., **player\_speed**, **player\_dir**, **gravity**, **apply\_movement** and **apply\_gravity**). For the sake of clarity these signals have been combined into a single trace and their combined values are denoted with  $P_n$ . Notice that until the **start** signal is asserted the previous output values for **player\_x** and **player\_y** are still valid.

After 2 clock cycles the core starts to request game objects from the game object list to check them for collisions with the player. In this example there are two game objects, where  $obj_0$  is blocking and  $obj_1$  is non blocking. Note that there may be an arbitrary number of clock cycles between the assertion of the **object\_req**



signal and the point in the time when the actual game object is supplied to the core. After two objects have been processed, the core request another object, but is notified with the `object_eol` signal that there are no objects left to check. It then goes over the whole list again. The second time it detects a collision with *obj<sub>1</sub>*, which is indicated using the `collision_detected` signal. Recall that the `collision_detected` signal must only be read one clock cycle after `object_valid` has been asserted, otherwise it is undefined. Collisions are only reported for non-blocking game objects.

Notice that the number of game objects does not need to stay constant during one run. It may be the case that, depending on the actual game (logic), a game object is removed because of a collision with the player.

Finally the core asserts the `done` signal to indicate that the run is complete. The final position of the player can then be obtained at the `player_x` and `player_y` outputs.

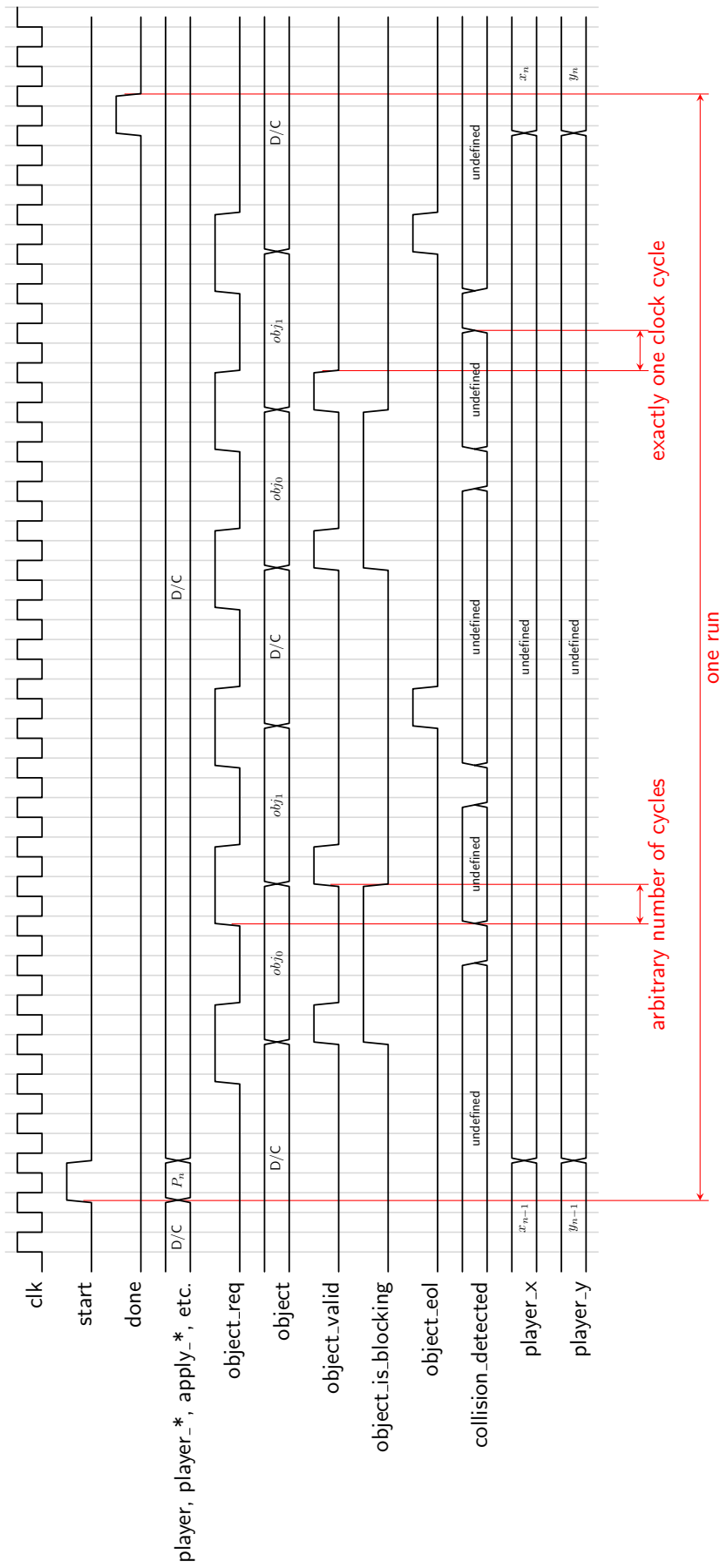


Figure 8.1: object\_collider example timing diagram

## Revision History

Revision	Date	Author(s)	Description
2.0	05.04.2021	FH	Added GFX Utility Package and Object Collider.
1.2	23.03.2021	FH	Fixed instruction format figure and opcodes for the SET_CFG and SET_PATTERN instructions.
1.1	14.03.2021	FH	Fixed the operand description and instruction format figure of the DRAW_RECT instruction.
1.0	11.03.2021	FH	Initial version

### Author Abbreviations:

FH Florian Huemer  
JM Jürgen Maier