Digital Design and Computer Architecture LU

IP Cores Manual

Florian Huemer, Florian Kriebel
{fhuemer,fkriebel}@ecs.tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, April 7, 2022

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

-
  ```
  function log2c(constant value : in integer) return integer;
  ```
  Calculates the logarithm dualis (base 2) 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 log10c(constant value : in integer) return integer;
  ```
  Calculates the logarithm base 10 of the integer operand and rounds it up to the next integer.

-
  ```
  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

VHDL Component Declaration:

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

Generics Description:

Name	Functionality
SYNC_STAGES	Number of flip flop stages used for synchronization
RESET_VALUE	The value, the output signal should have directly after reset

Port Signals Description:

Name	Dir.	Width/Type	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

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.

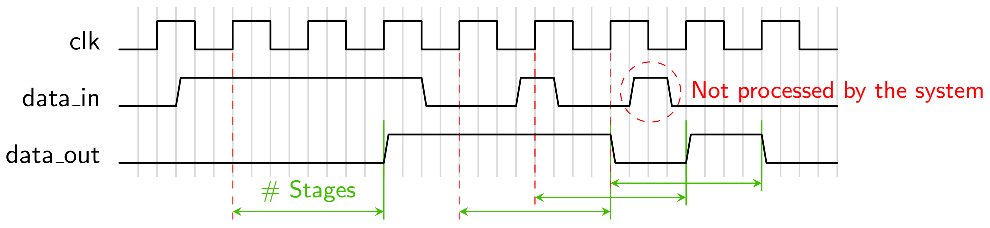


Figure 2.1: Synchronizer timing

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.

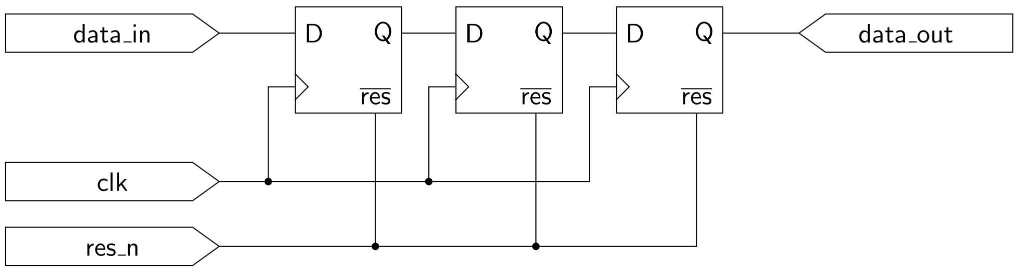


Figure 2.2: Synchronizer circuit

3 On-chip Memory

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 memory, with different access strategies.

Currently there are three memories types available, a single clock dual-port RAM with one read and one write port and two single clock FIFOs with one read and one write port. The FIFO differ in how their read port is operated. While one has a classic read port controlled by a read flag, where the data is then available in the next clock cycle, the other one exhibits first word fall through (FWFT) behavior. For more information see Section 3.5.

3.2 Dependencies

- Mathematical support package (math_pkg)

3.3 Required VHDL Files

- `ram_pkg.vhd`
- `dp_ram_1c1r1w.vhd`
- `fifo_1c1r1w.vhd`
- `fifo_1c1r1w_fwft.vhd`

3.4 Component Declarations

3.4.1 Single clock dual-port RAM

VHDL Component Declaration:

```
component dp_ram_1c1r1w is
  generic (
    ADDR_WIDTH : integer;
    DATA_WIDTH : integer
  );
  port (
    clk : in std_logic;
    -- read port
    rd1_addr : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
    rd1_data : out std_logic_vector(DATA_WIDTH - 1 downto 0);
    rd1 : in std_logic;
    --write port
    wr2_addr : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
    wr2_data : in std_logic_vector(DATA_WIDTH - 1 downto 0);
    wr2 : in std_logic
  );
end component;
```

Generics Description:

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

Port Signals Description:

Name	Dir.	Width/Type	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

3.4.2 Single clock FIFO

VHDL Component Declaration:

```
component fifo_1c1r1w is
  generic (
    MIN_DEPTH : integer;
    DATA_WIDTH : integer
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    --read port
    rd_data : out std_logic_vector(DATA_WIDTH - 1 downto 0);
    rd : in std_logic;
    --write port
    wr_data : in std_logic_vector(DATA_WIDTH - 1 downto 0);
    wr : in std_logic;
    --status signals
    empty : out std_logic;
    full : out std_logic;
    half_full : out std_logic
  );
end component;
```

Generics Description:

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

Port Signals Description:

Name	Dir.	Width/Type	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 wr_data 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 contrains data.

3.4.3 Single clock FIFO with FWFT behavior

VHDL Component Declaration:

```vhdl
component fifo_1c1r1w_fwft is
  generic (
    DEPTH : integer;
    DATA_WIDTH : integer
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    rd_data : out std_logic_vector(DATA_WIDTH - 1 downto 0);
    rd_ack : in std_logic;
    rd_valid : out std_logic;
    wr_data : in std_logic_vector(DATA_WIDTH - 1 downto 0);
    wr : in std_logic;
    full : out std_logic;
    half_full : out std_logic
  );
end component;
```

Generics Description:

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

Port Signals Description:

Name	Dir.	Width/Type	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_valid	out	1	If 1, the data at rd_data is valid and can be used.
rd_ack	in	1	Indicated to the FIFO that the data at rd_data has been consumed and new data can be applied to rd_data. If no new data is available, because the FIFO is empty, the rd_valid signal, goes low in the next cycle.
wr_data	in	DATA_WIDTH	Data for the write operation
wr	in	1	If 1, the data of wr_data is written to the next free memory location. If the FIFO is full, the write request is ignored
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.

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).

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).

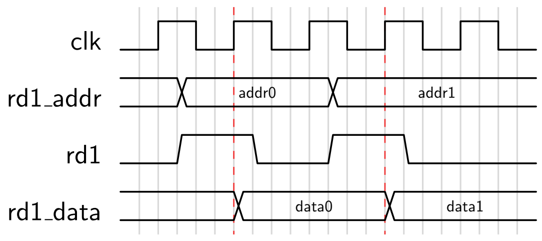


Figure 3.1: RAM read timing.

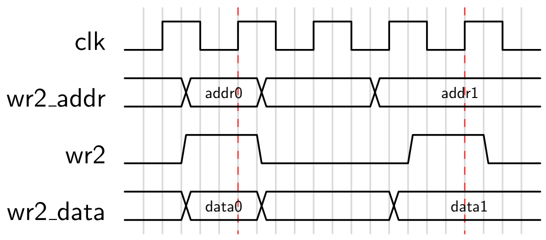


Figure 3.2: RAM write timing

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.

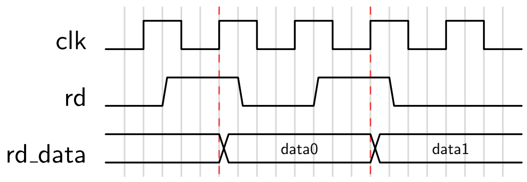


Figure 3.3: FIFO read timing

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).

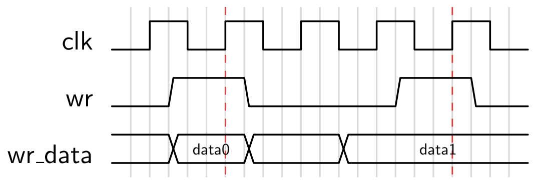


Figure 3.4: FIFO write timing

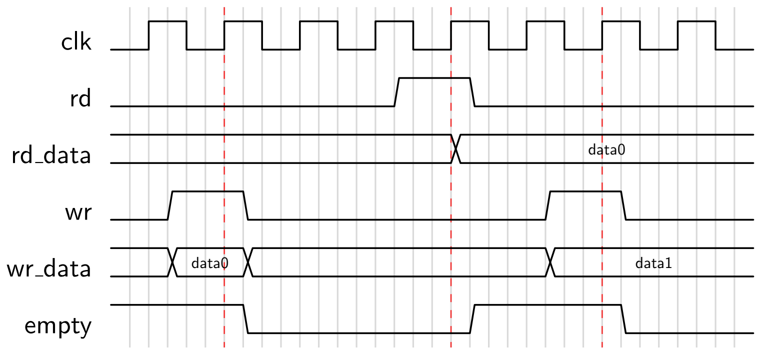


Figure 3.5: FIFO empty handling

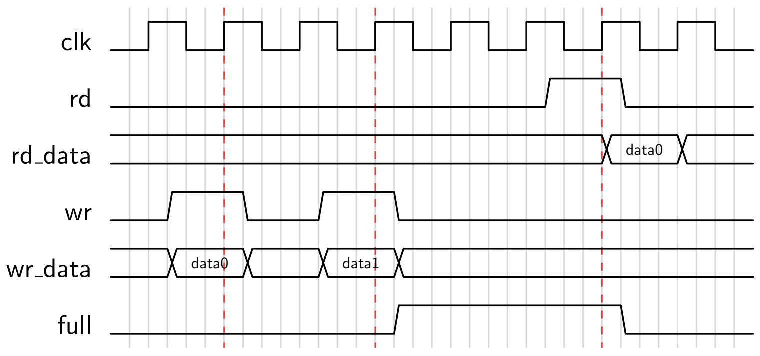


Figure 3.6: FIFO full handling

3.5.3 Single clock FIFO with FWFT behavior

The write port of the fifo_1c1r1w_fwft behaves exactly to same as for the other FIFO. Figure 3.7 demonstrates how the read port is operated. As soon as the FIFO contains data the rd_valid signal is asserted and the next data value can be retrieved at the rd_data output. This data value will be kept as long as it is not acknowledged by a high signal level at rd_ack. If rd_ack is asserted, the FIFO will either deassert rd_valid in the next clock cycle, indicating that the FIFO is now empty, or output the next data value at rd_data. Note that rd_ack must only be asserted if rd_valid is asserted. The role of the empty signal is now covered by the rd_valid signal.

The name for the read behavior of this FIFO comes from the fact, that no interaction is necessary to retrieve the first data value of the FIFO, i.e., it simply “falls through” the FIFO.

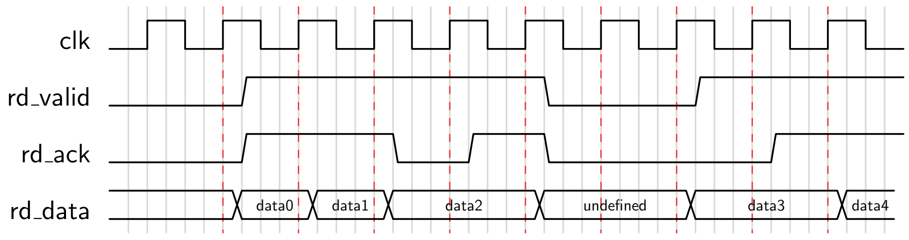


Figure 3.7: FIFO FWFT read timing

4 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 4.1 shows to the general structure of the audio controller.

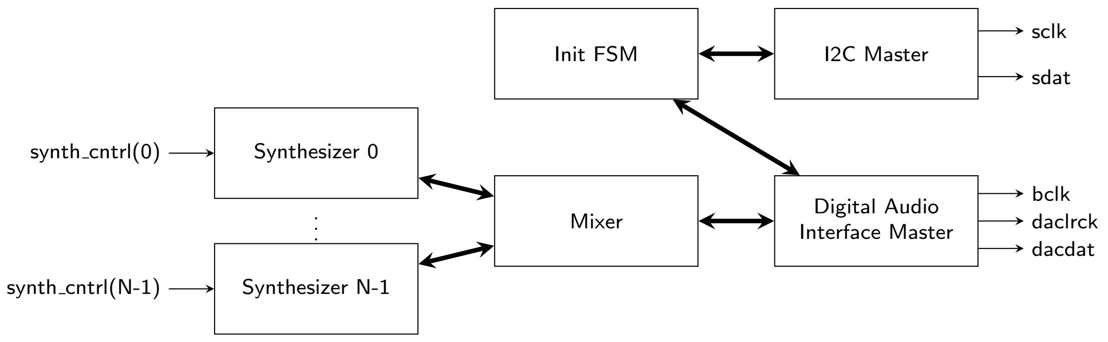


Figure 4.1: Audio controller internal structure

The audio controller must be clocked by a 12 MHz clock (which will internally be forwarded to the xck 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 4.4).

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

4.1 Dependencies

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

4.2 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.

4.3 Component Declaration

VHDL Component Declaration:

```
component audio_cntrl_2s is
  port (
    clk    : in std_logic; --12 MHz input clock
    res_n  : in std_logic;

    --clock output signal for the wm8731
    wm8731_xck      : out std_logic;

    --cfg interface to wm8731: i2c configuration interface
    wm8731_sdat : inout std_logic;
    wm8731_sclk : inout std_logic;

    --data interface to wm8731: digital audio interface
    wm8731_dacdat   : out std_logic;
    wm8731_daclrck  : out std_logic;
    wm8731_bclk     : out std_logic;

    --internal interface to the stynthesizers
    synth_cntrl : in synth_cntrl_vec_t(0 to 1)
  );
end component;
```

Port Signals Description:

Name	Dir.	Width/Type	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

4.4 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.

```
type synth_cntrl_t is record
  play : std_logic;
  high_time : std_logic_vector(7 downto 0);
  low_time : std_logic_vector(7 downto 0);
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.

5 SRAM Controller

The sram_cntrl interfaces with the external SRAM located on the DE2-115 FPGA board. This chip features a 16 bit wide interface for 2 MB of Static RAM. The interface exposed by the controller allows to access individual addresses and store or read 16 bit at a time.

Although the external SRAM is only a single port memory, i.e., read and write operations cannot take place simultaneously, the SRAM controller provides two (more or less) independent ports for reading (rd_* signals) and writing (wr_* signals). Write operations are internally buffered in a FIFO, and only executed when there is currently no pending read operation. This means that read operations have priority over writes. The size of the write buffer FIFO can be configured using a generic.

5.1 Dependencies

- Mathematical support package (see Section 1)
- RAM package (see Section 3)

5.2 Required VHDL Files

- `sram_pkg.vhd`
- `sram_cntrl.vhd`

5.3 Component Declaration

VHDL Component Declaration:

```vhdl
component sram_cntrl is
  generic (
    ADDR_WIDTH : integer := 20;
    DATA_WIDTH : integer := 16;
    WR_BUF_SIZE : integer := 8
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    --write port
    wr_addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    wr_data : in std_logic_vector(DATA_WIDTH-1 downto 0);
    wr : in std_logic;
    wr_full : out std_logic;
    wr_half_full : out std_logic;
    --read port
    rd_addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    rd : in std_logic;
    rd_busy : out std_logic;
    rd_data : out std_logic_vector(DATA_WIDTH-1 downto 0);
    rd_valid : out std_logic;
    --interface to external SRAM
    sram_dq : inout std_logic_vector(DATA_WIDTH-1 downto 0);
    sram_addr : out std_logic_vector(ADDR_WIDTH-1 downto 0);
    sram_ub_n : out std_logic;
    sram_lb_n : out std_logic;
    sram_we_n : out std_logic;
    sram_ce_n : out std_logic;
    sram_oe_n : out std_logic
  );
end component;
```

Generics Description:

Name	Functionality
ADDR_WIDTH	The number of address bits of the external SRAM chip
DATA_WIDTH	The number of data bits of the external SRAM chip. Only 16 bits are supported.
WR_BUF_SIZE	The size of the write buffer.

Port Signals Description:

Name	Dir.	Width/Type	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal, low active, not internally synchronized
rd_addr	in	ADDR_WIDTH	Address for the read operation
rd	in	1	If 1, a read operation on the address supplied at rd_addr is performed at the next rising edge of the clock signal.
rd_data	out	DATA_WIDTH	The data output for read operations.
rd_valid	out	1	Indicator signals if rd_data is valid. This signal is basiaclly the rd signal delayed by two clock cycles.
rd_busy	out	1	If 1, the controller is busy performing a write operation, hence rd must stay low.
wr_addr	in	ADDR_WIDTH	Address for the write operation
wr_data	in	DATA_WIDTH	Data for the write operation
wr	in	1	If 1, a write operation for the address at wr_addr and the data at wr_data is added to the write buffer.
wr_full	out	1	1, if the write buffer is full.
wr_half_full	out	1	1, if the write buffer is half full.
sram_dq	inout	DATA_WIDTH	SRAM data inputs/outputs
sram_addr	out	ADDR_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

5.4 Interface Protocol

Internally the SRAM controller uses a FIFO to buffer incoming write requests. Hence the write port of the controller is operated exactly like the write port of a FIFO as discussed in Section 3. For that purpose the SRAM controller exposes the full and half full flags of this FIFO at the ports wr_full and wr_half_full as well as the actul write flag wr itself. If wr_full is asserted wr must not be asserted.

The read port of the controller consists of the signals rd, rd_addr, rd_busy, rd_data and rd_valid. To perform a read operation rd must be asserted and the address of the desired memory location must be applied at rd_addr for one clock cycle. The actual read operation on the external SRAM takes two cycles, which means that the data can be retrieved at rd_data after two cycles. Valid data at rd_data is indicated by the rd_valid signal, which is essentially the rd signal delayed by two clock cycles.

If rd_busy is asserted the controller currently performs a write operation on the external SRAM. In this case rd must not be asserted.

Note, however, that read operations can also be pipelined as demonstrated by the timing diagram in Figure 5.1. Read bursts of arbitrary length are possible, which means that once asserted rd may stay high for an arbitrary number of cycles. The rd_busy signal only needs to be checked if rd was low in the previous cycle. Keep in mind that this behavior allows to starve the write port of the controller.

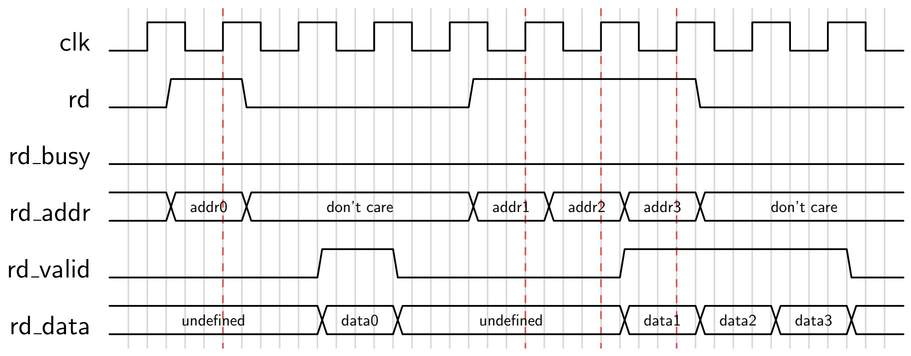


Figure 5.1: SRAM controller read timing

6 GFX Utility Package

6.1 Description

The gfx_util_pkg package contains two cores that implement simple graphical operations:

- GFX Line (gfx_line)
  Draws a line between two points unsing Bresenham's line algorithm.
- GFX Blitter (gfx_bb)
  Performs a bit blit operation using an internal (on-chip) ROM memory as the source memory. The contents of this memory can be set using a generic.

The 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 (using the output signals pixel_valid, pixel_x and pixel_y and in the case of gfx_bb 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. Note however that the cores don't perform any form of clipping. Hence, the *signed* output coordinates can be in the full range supported by their data type (i.e., `std_logic_vector(DATA_WIDTH-1 downto 0)`).

6.2 Dependencies

- Mathematical support package (math_pkg)

6.3 Required VHDL Files

- `gfx_util_pkg.vhd`
- `gfx_line.vhd`
- `gfx_bb.vhd`

6.4 Component Declarations

6.4.1 GFX Line

VHDL Component Declaration:

```vhdl
component gfx_line is
  generic (
    DATA_WIDTH : integer
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    start : in std_logic;
    stall : in std_logic;
    busy : out std_logic;
    x0 : in std_logic_vector(DATA_WIDTH-1 downto 0);
    y0 : in std_logic_vector(DATA_WIDTH-1 downto 0);
    x1 : in std_logic_vector(DATA_WIDTH-1 downto 0);
    y1 : in std_logic_vector(DATA_WIDTH-1 downto 0);
    pixel_valid : out std_logic;
    pixel_x : out std_logic_vector(DATA_WIDTH-1 downto 0);
    pixel_y : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );
end component;
```

Generics Description:

Name	Functionality
DATA_WIDTH	The width of the coordinates the core operates on.

Port Signals Description:

Name	Dir.	Width/Type	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 start is asserted and must remain valid (and unchanged) until the busy signal goes low.
stall	in	1	This signal 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, busy goes low again, which allows for a new drawing operation to be started.
x0	in	DATA_WIDTH	The x coordinate of the first point of the line (i.e., the start point of the line). [drawing parameter]
y0	in	DATA_WIDTH	The y coordinate of the first point of the line (i.e., the start point of the line). [drawing parameter]
x1	in	DATA_WIDTH	The x coordinate of the second point of the line (i.e., the end point of the line). [drawing parameter]
y1	in	DATA_WIDTH	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	DATA_WIDTH	The signed x coordinate of the output pixel.
pixel_y	out	DATA_WIDTH	The signed y coordinate of the output pixel.

6.4.2 GFX Blitter

VHDL Component Declaration:

```vhdl
component gfx_bb is
  generic (
    DATA_WIDTH : integer;
    BB_ROM : bb_rom_t
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    start : in std_logic;
    stall : in std_logic;
    busy : out std_logic;
    w : in std_logic_vector(DATA_WIDTH-1 downto 0);
    h : in std_logic_vector(DATA_WIDTH-1 downto 0);
    x_src : in std_logic_vector(DATA_WIDTH-1 downto 0);
    y_src : in std_logic_vector(DATA_WIDTH-1 downto 0);
    x_dest : in std_logic_vector(DATA_WIDTH-1 downto 0);
    y_dest : in std_logic_vector(DATA_WIDTH-1 downto 0);
    vflip : in std_logic;
    hflip : in std_logic;
    pixel_valid : out std_logic;
    pixel_color : out std_logic_vector(3 downto 0);
    pixel_x : out std_logic_vector(DATA_WIDTH-1 downto 0);
    pixel_y : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );
end component;
```

Generics Description:

Name	Functionality
DATA_WIDTH	The width of the coordinates the core operates on.
BB_ROM	This generic defines the contents of the Bit Blit (BB) ROM, i.e., the interal memory that serves as the source image for bit blit operations. It has the datatype `bb_rom_t`, which is declared as an array with `std_logic_vector` elements in the gfx_util_pkg package. The length of the `BB_ROM` array must be a even power of 2, i.e., 2^{2n}. Its contents are interpreted as a n times n bitmap, where `BB_MEM(x+y*n)` defines the 4-bit color value of the pixel with the coordinates x and y.

Port Signals Description:

Name	Dir.	Width/Type	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 start is asserted and must remain valid (and unchanged) until the busy signal goes low.
stall	in	1	This signal 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, busy goes low again, which allows for a new drawing operation to be started.
w	in	DATA_WIDTH	The width of the source image section in BB ROM. [drawing parameter]
h	in	DATA_WIDTH	The height of the source image section in BB ROM. [drawing parameter]
x_src	in	DATA_WIDTH	The unsigned x coordinate of the upper left corner of the source image section in BB ROM. [drawing parameter]
y_src	in	DATA_WIDTH	The unsigned y coordinate of the upper left corner of the source image section in BB ROM. [drawing parameter]
x_dest	in	DATA_WIDTH	The signed x coordinate of the destition for the blitting operation. [drawing parameter]
y_dest	in	DATA_WIDTH	The signed y coordinate of the destition for the blitting operation. [drawing parameter]
hflip	in	1	Flip the source image section horizontally during the blitting operation [drawing parameter]
vflip	in	1	Flip the source image section vertically during the blitting operation [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	DATA_WIDTH	The signed x coordinate of the output pixel.
pixel_y	out	DATA_WIDTH	The signed y coordinate of the output pixel.
pixel_color	out	4	

6.5 Interface Protocol

Because the interface protocols of the cores are essentially the same, only one example timing diagram is presented (Figure 6.1). 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 6.1 the core 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_bb core) must only be used when pixel_valid is asserted, otherwise their value is

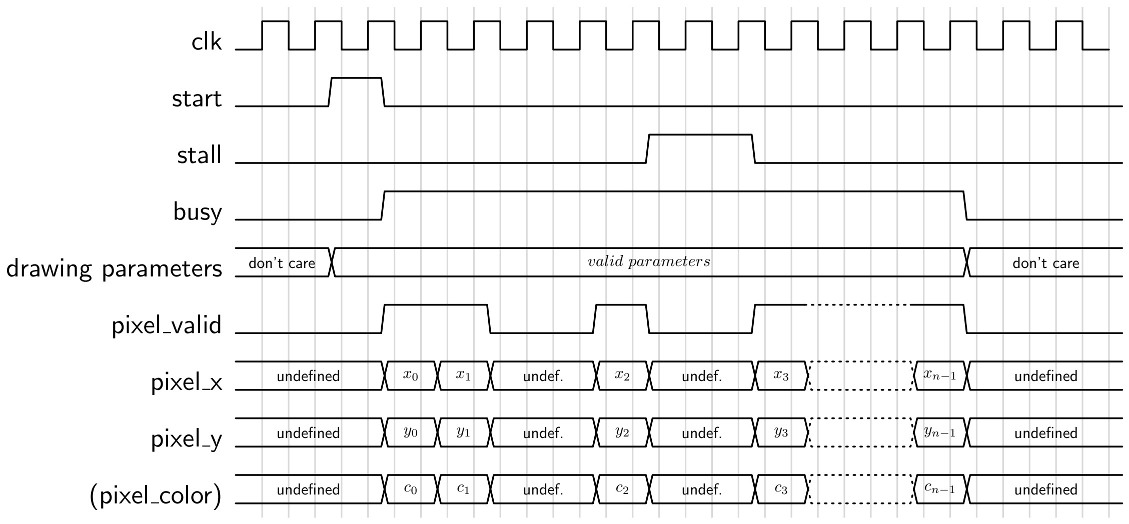


Figure 6.1: Example timing diagram for gfx_line and gfx_bb

undefined (i.e., it must not be used or interpreted). The stall signal can be asserted at any time during the drawing process, and causes the core to pause their operation and not output new coordinates. Hence, for every clock cycle where stall is asserted pixel_valid stays low. In the example timing diagram, the stall signal is asserted for two cycles after the thired output coordinate has been generated. After the deassertion of the stall signal the core resumes its operation. This feature can be used as back-pressure signal. Note that the stall and start signals may not be asserted at the same time.

Don't supply coordinates to the cores that cause them to draw "outside" of the dimensions supported by DATA_WIDTH. In such a case the behavior of the core becomes undefined. Moreover don't specify image sections outside of the bounds of BB_MEM

7 VGA Graphics Controller

The VGA Graphics Controller performs simple graphical operations using an instruction based interface. It uses external SRAM to store its frame buffers and interfaces with the ADV7123 video DAC to produce an RGB analog component video signal that can be output through a VGA connector. The VGA Graphics Controller supports one fixed resolution of 320x240 pixels and a color depth of 16 bit (RGB565).

7.1 Internal Structure

Figure 7.1 shows an overview of the internal structure of the core.

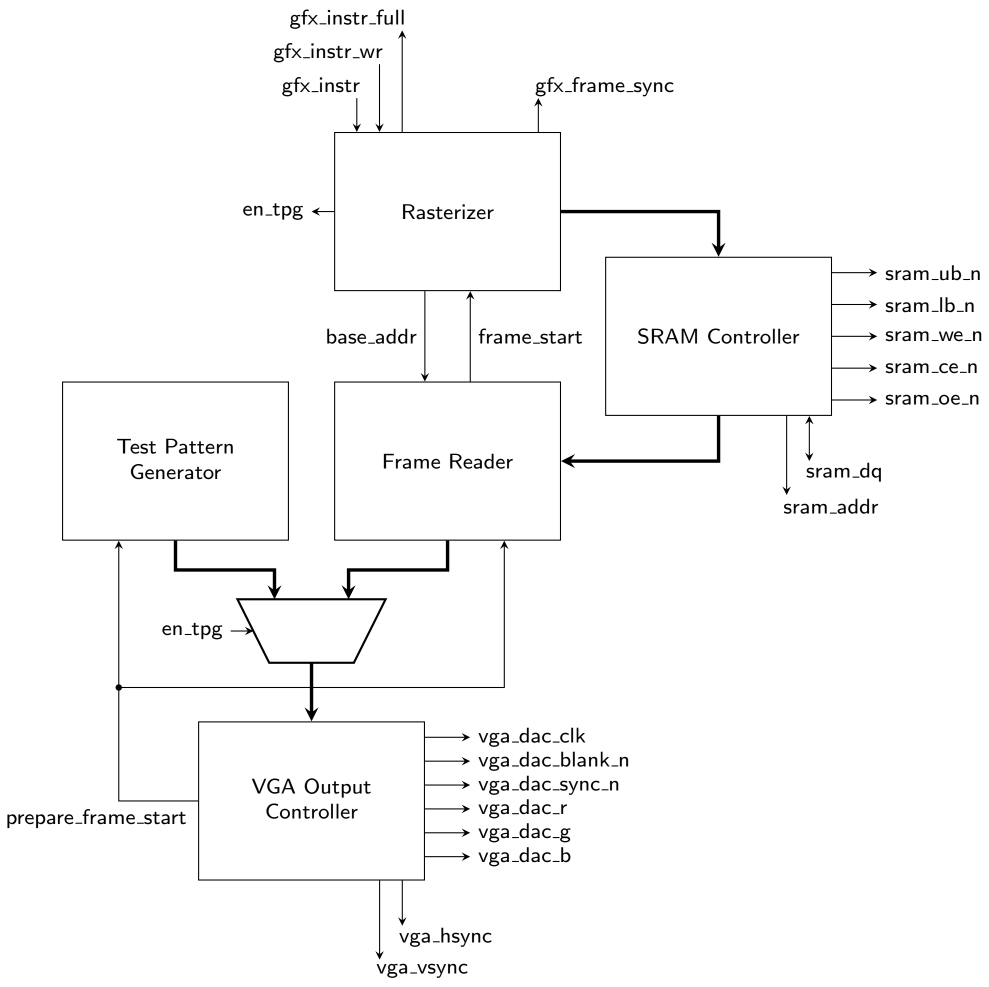


Figure 7.1: VGA Graphics Controller overview

The Rasterizer provides the internal interface to the core, i.e., the signals other IP cores have to connect to, in order to use the VGA Graphics Controller. Internally it uses the modules of the GFX utility package (see Section 6) to perform the graphical operations. Pixel data is written to the external SRAM using the sram_cntrl (see Section 5).

To generate the actual video signal the VGA Output Controller (VOG) creates the appropriate signals for the video DAC (vga_dac_*) and the synchronization signals (vga_hsync and vga_vsync). The actual pixel data required by the VOG either comes from the Test Pattern Generator (TPG) or the Frame Reader. The prepare_frame_start signal is used to synchronize these modules. Although the supported resolution of the VGA Graphics Controller is only 320x240 pixels, the VOG outputs a resolution of 640x480. This is done for the sake of compatibility, since not all monitors support resolutions lower than 640x480.

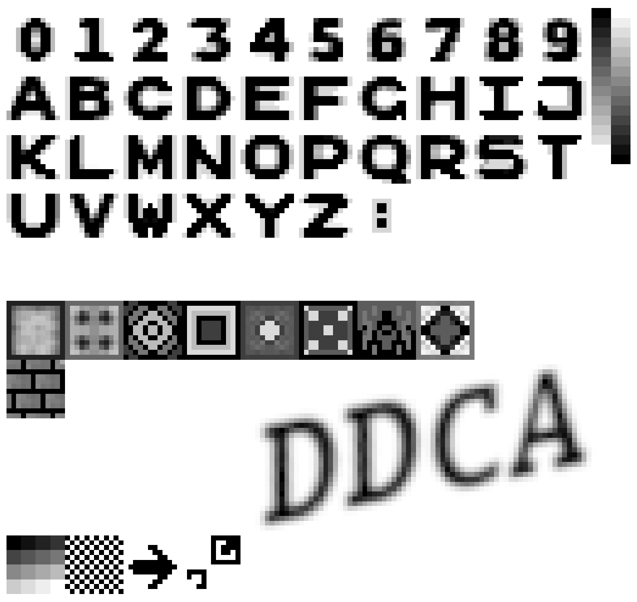


Figure 7.2: 128x128 BB ROM contents

Hence the Frame Reader and the TPG have to supply image data with the higher resolution. Since the TPG only outputs a static image, this is not really an issue. However, the Frame Reader must perform an upscaling operation on the data read from the SRAM. The Frame Reader uses the frame_start signal to synchronize the switching of the frame buffer with the Rasterizer, which is necessary in order to support double buffering.

7.2 Supported Operations

The VGA Graphics Controller basically features the following graphical operations:

- Clearing of the whole frame buffer at once
- Setting individual pixels in the frame buffer
- Drawing lines between two points in the frame buffer
- Copying and transforming (i.e., flipping) a section of an image stored in a special on-chip ROM to the frame buffer, an operation also referred to as blitting[1]

Figure 7.2 shows the image (128x128 pixel) stored in the on-chip ROM which we will refer to as the Bit Blit (BB) ROM. The numbers and letters are arranged in a grid with a cell size of 12x12 pixels. Hence, the coordinates of the digit “0” are (x, y)=(0,0), while the letter “A” can be found at (0,12). The image in BB ROM has a color depth of 4 bit. As explained in more detail in Section 7.7 this “color” will be used as an index to a color palette to determine the actual color that will be written to the frame buffer.

Figure 7.3 shows an example of how such a blitting operations can look like.

7.3 Dependencies

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

[1]`https://en.wikipedia.org/wiki/Bit_blit`

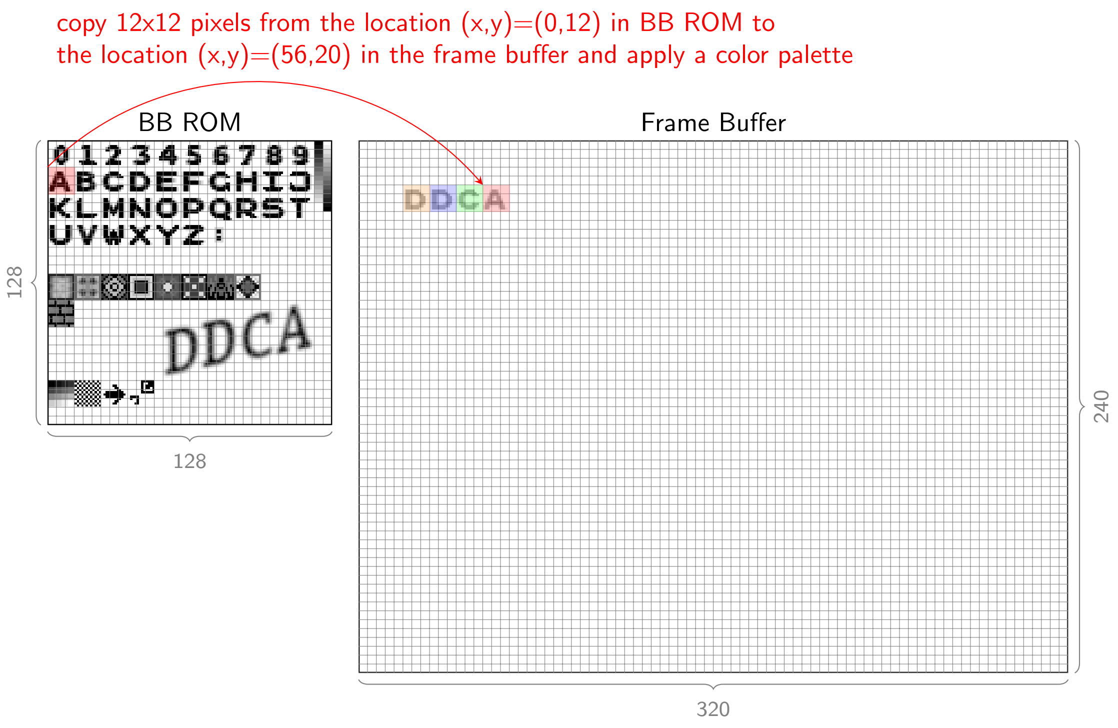

copy 12x12 pixels from the location (x,y)=(0,12) in BB ROM to
the location (x,y)=(56,20) in the frame buffer and apply a color palette
BB ROM
Frame Buffer
128
128
240
320

Figure 7.3: Blitting example (for better orientation the images have been overlayed with a 4x4 pixel grid)

7.4 Required VHDL Files

The VGA 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.

- `vga_gfx_cntrl_pkg.vhd`
- `precompiled_vga_gfx_cntrl.qxp`
- `precompiled_vga_gfx_cntrl.vho`

Hence, if you want to perform a simulation on the module in Questa/Modelsim, use the package `vga_gfx_cntrl_pkg.vhd` and the netlist file `precompiled_vga_gfx_cntrl.vho`. For synthesis in Quartus use `vga_gfx_cntrl_pkg.vhd` and the Exported Partition File `precompiled_vga_gfx_cntrl.qxp` file.

7.5 Component Declaration

VHDL Component Declaration:

```
component precompiled_vga_gfx_cntrl is
  port (
    clk : in std_logic;
    res_n : in std_logic;
    display_clk : in std_logic;
    display_res_n : in std_logic;
    -- instruction interface
    gfx_instr : in std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
    gfx_instr_wr : in std_logic;
    gfx_instr_full : out std_logic;
    gfx_frame_sync : out std_logic;
    -- external SRAM
    sram_dq : inout std_logic_vector(15 downto 0);
    sram_addr : out std_logic_vector(19 downto 0);
```

```
        sram_ub_n : out std_logic;
        sram_lb_n : out std_logic;
        sram_we_n : out std_logic;
        sram_ce_n : out std_logic;
        sram_oe_n : out std_logic;
        -- VGA
        vga_hsync : out std_logic;
        vga_vsync : out std_logic;
        vga_dac_clk : out std_logic;
        vga_dac_blank_n : out std_logic;
        vga_dac_sync_n : out std_logic;
        vga_dac_r : out std_logic_vector(7 downto 0);
        vga_dac_g : out std_logic_vector(7 downto 0);
        vga_dac_b : out std_logic_vector(7 downto 0)
    );
end component;
```

Port Signals Description:

Name	Dir.	Width/Type	Functionality
clk	in	1	Global clock signal (50 MHz)
res_n	in	1	Global reset signal, low active, not internally synchronized
display_clk	in	1	Clock signal for the actual video signal (25 MHz)
display_res_n	in	1	Reset signal for the display_clk clock domain, low active, not internally synchronized
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_instr	in	GFX_INSTR_WIDTH	The actual commands and operands
gfx_frame_sync	out	1	The synchronization signal that allows other cores to synchronize to the start of a new frame
sram_dq	inout	20	SRAM data inputs/outputs
sram_addr	out	16	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
vga_hsync	out	1	The horizontal synchronization signal going directly to the VGA connector.
vga_vsync	out	1	The vertical synchronization signal going directly to the VGA connector.
vga_dac_clk	out	1	The clock signal for the video DAC (25 MHz)
vga_dac_blank_n	out	1	DAC control signal to set the analog output signals for the red, green and blue channels to the blank (black) level
vga_dac_sync_n	out	1	DAC control signal used to embed synchronization information in the green channel, unused, constant 1
vga_dac_r	out	8	DAC input data for the red output channel
vga_dac_g	out	8	DAC input data for the green output channel
vga_dac_b	out	8	DAC input data for the blue output channel

7.6 Interface

Graphics instructions are fed into the core using the signals gfx_instr_wr and gfx_instr. Internally those signals directly feed a FIFO, referred to as the Instruction FIFO, whose full flag is in turn output at gfx_instr_full. Hence this port behaves exactly like the write port of a FIFO, discussed in Section 3.

The gfx_frame_sync signal is only activated for a single clock cycle in response to the execution of a FRAME_SYNC instruction and allows the synchronization of an interfacing IP core to the start of a new frame output by the VOG.

7.7 Graphics Instructions

A graphics instruction always consists of the actual 16 bit command and a number of 16 bit operands associated with the command. Depending on the instruction the number of operands can range from 0 to 16. This means that for most instructions multiple 16 bit words must be issued to the Instruction FIFO using the inputs gfx_instr_wr and gfx_instr. The execution of an instruction only begins when the command and all required operands are present.

Internally the VGA Graphics Controller maintains three registers, which are used by and modified through the execution of graphics instructions:

- Graphics Pointer (`gp`), 2x16 bit
- Alpha Color (`ac`), 4 bit
- Selected Palette (`sp`), 5 bit

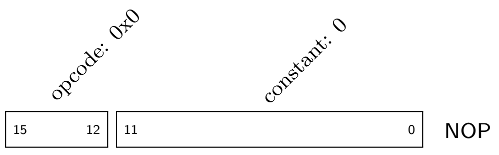
After reset all registers are initialized to 0.

The `gp` register represents a 2D coordinate on the frame buffer (i.e., the image in the external SRAM that the graphics instructions draw to) and is used by most drawing instructions as a parameter. We use `gp.x` and `gp.y` to refer to the x and y coordinate of the `gp`. The coordinate `gp.x` =0, `gp.y` =0 points to the pixel of the upper left corner of the frame buffer, while `gp.x` =319, `gp.y` =239 points to the lower right corner. However, since `gp.x` and `gp.y` are **signed 16 bit** values, it is possible that the `gp` points to a location outside of the bounds of the physical frame buffer. This is completely fine, as the Rasterizer ensures that an access to a pixel outside of bounds, will simply have no effect (i.e., it performs clipping). Hence, drawing a line from $(0, 0)$ to $(10, 0)$ will produce the exact same result as drawing a line from $(-10, 0)$ to $(10, 0)$.

Although the VGA Graphics Controller supports a color depth of 16-bit, the drawing instructions only have a 4-bit color information field, giving them access to 16 colors at a time. The exact RGB values of these colors are stored in a palette. The VGA Graphics Controller can hold 32 such palettes. Using the `sp` register, one of those palettes is selected for drawing. Hence before the Rasterizer writes a pixel to the frame buffer in memory, it looks up its actual color in the currently selected color palette.

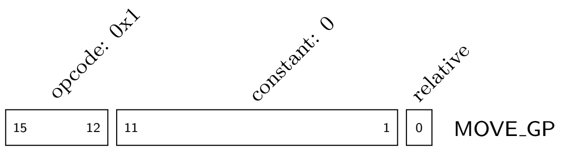
Finally the `ac` register specifies a color that should be interpreted as transparent by the bit blit instructions if the alpha mode is activated. This means that, if the Rasterizer encounters a pixel with this color, while executing a bit blit instruction, it will not write this pixel to the frame buffer.

Now a detailed bit-level specification of each instruction supported by the VGA Graphics Controller follows. Note that the core does not check or prevent overflows in the `gp` or the execution of bit blit operations with illegal source image section dimensions. In such a scenario the behavior of the core becomes undefined.



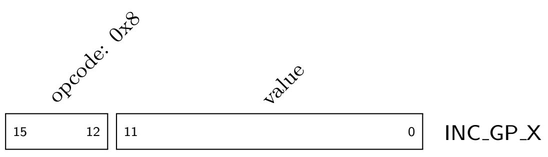
Operands: None

Description: Do nothing.



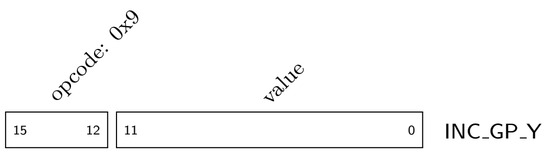
Operands: x, y

Description: Sets the `gp` to (`x`,`y`). If the `relative` bit is set, `x` and `y` will instead be added to the current `gp` (i.e., `gp.x` += `x`, `gp.y` += `y`).



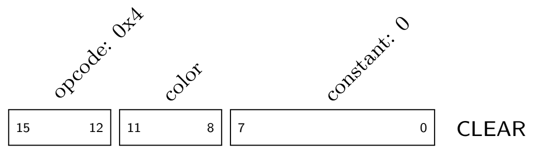
Operands: None

Description: Adds the signed 12 bit integer in `value` to `gp.x`. For that purpose `value` is sign-extended to 16 bit.



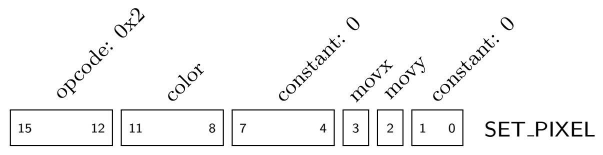
Operands: None

Description: Adds the signed 12 bit integer in `value` to `gp.y`. For that purpose `value` is sign-extended to 16 bit.



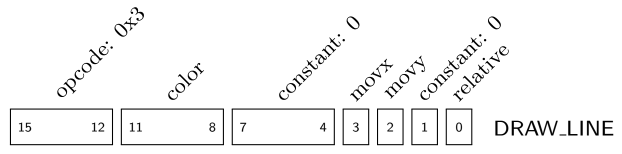
Operands: None

Description: Sets every pixel in the frame buffer to the specified `color`. Does not change the `gp`.



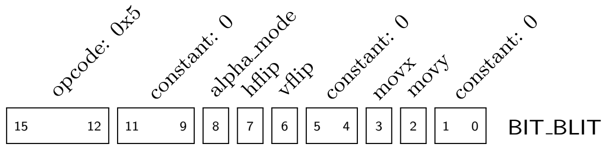
Operands: None

Description: Sets the pixel the `gp` currently points at to the specified `color`. After that `gp.x` (`gp.y`) is incremented by one if `movx` (`movy`) is set.



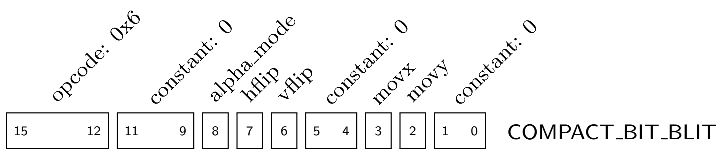
Operands: `x`, `y`

Description: Draws a line between the `gp` and the destination coordinate specified by `x` and `y` using the specified `color`. If the `relative` flag is set, `x` and `y` shall be interpreted as signed offsets to the `gp`. This means that in this case the actual x and y coordinates of the end point of the line are given by x=`gp.x` +`x` and y=`gp.y` +`y`. After the line has been drawn `gp.x` is set to the destination x coordinate of the line if `movx` is set. Likewise `gp.y` is changed if `movy` is set.



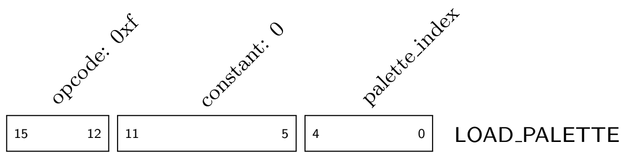
Operands: `x`, `y`, `w`, `h`

Description: Performs a bit blit operation from the BB ROM (see Figure 7.2) to the frame buffer. The unsigned operands `x`, `y`, `w`, `h` define the section of the image in BB ROM that will be copied to the location the `gp` points to in the frame buffer. The operands must not exceed the dimensions of the BB ROM (otherwise the behavior is undefined). If the `alpha_mode` flag is set, pixels in the source image that match the `ac` register are not copied to the frame buffer. If the `hflip` and/or `vflip` flags are set the image will be flipped horizontally and/or vertically, respectively. After the drawing operation is finished `gp.x` is incremented by `w` if `movx` is set. Likewise `gp.y` is incremented by `h` if `movy` is set. This feature is convenient for printing strings, as it removes the need to reset the `gp` after printing each individual character.



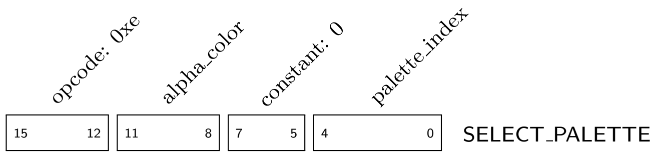
Operands: `c`

Description: Works the same as the `BIT_BLIT` instruction, but only takes one operand `c`, which is basically a concatenation of four unsigned 4-bit integers `x`, `y`, `w`, `h` (i.e., `c` = `x` & `y` & `w` & `h`) . Hence, using this instruction the width and height of the source image section in BB ROM is restricted to $2^4 - 1$. To obtain the x_s and y_s coordinates of the source image section the instruction uses x_s=`x`*`w` and y_s=`y`*`h`. This instruction is ideal to address a set of image sections in BB ROM that is arranged in a regular grid (e.g., the characters).



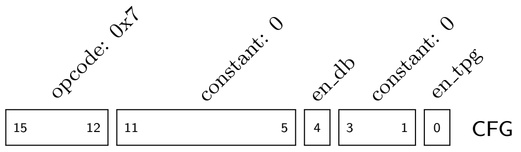
Operands: c_0,...,c_{15}

Description: Loads the actual colors for the color palette identified by `palette_index`. Note that this instruction has 16 operands. The individual 16 bit color values are specified using the RGB565 format. This means that given a color value `c`, the bits `c(15 downto 11)` are devoted to the blue channel, while the bits `c(10 downto 5)` and `c(4 downto 0)` constitute the green and the red channel respectively.



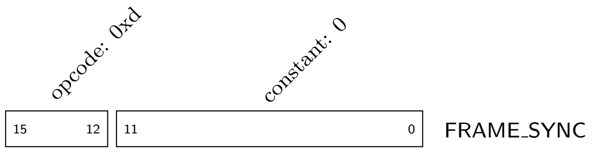
Operands: None

Description: Selects a particular color palette and alpha color (`ac`:= `alpha_color`, `sp`:= `palette_index`).



Operands: None

Description: This instruction enables or disables double buffering and can switch the output to the test pattern. If the `en_db` bit is set double buffering is enabled, which means that drawing operations will be performed on a different buffer than the one currently shown. Switching those frame buffers is then achieved using the `FRAME_SYNC` instruction. If the `en_tpg` bit is set to one the VOG is switched over to the Test Pattern Generator.



Operands: None

Description: This instruction blocks the execution of the following instructions until the Frame Reader starts to fetch the next frame to be output by the VOG (indicated by the frame_start signal of the Frame Reader). If double buffering is enabled the frame buffers will be switched. The gfx_frame_sync is asserted for exactly one clock cycle to indicate that the instruction has been executed.

---

8 Tetris Utility Package

8.1 Description

The tetris_util_pkg package contains some utility functions and modules that deal with drawing tetrominoes[2] and colliding them with a 2D map of blocks (and its borders). A tetromino is composed of a number of blocks arranged in a 4x4 grid. Figure 8.1 shows the 7 classic tetromino shapes, which are also referred to with the letters which they (approximately) resemble.

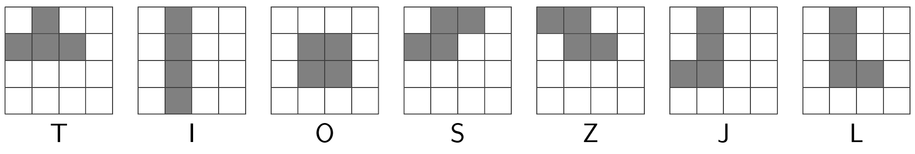


Figure 8.1: Classic tetrominoes

The package features a special datatype that is used to refer to these tetromino shapes and declares constants for each of them. Notice that the tetrominoes are numbered according to the sequence shown in Figure 8.1.

```
subtype tetromino_t is std_logic_vector(2 downto 0);
constant TET_T : tetromino_t := "000";
[...]
constant TET_L : tetromino_t := "110";
```

tetrominoes can also be rotated (clockwise) by 90, 180 and 270 degrees, which is illustrated in Figure 8.2.

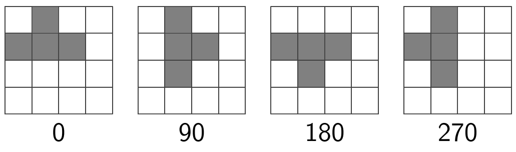


Figure 8.2: T tetromino and its 4 rotation states

Note that for some tetrominoes some rotation states are equivalent, e.g., an S tetromino gives the same shape when rotated by 0 or 180 degrees. To represent a rotation, the package uses the `rotation_t` datatype and defines constants for the four possible rotation states.

```
subtype rotation_t is std_logic_vector(1 downto 0);
constant ROT_0   : rotation_t := "00";
constant ROT_90  : rotation_t := "01";
constant ROT_180 : rotation_t := "10";
constant ROT_270 : rotation_t := "11";
```

Hence every possible tetromino shape that can appear in a Tetris game is identified by a tetromino_t and rotation_t value.

To test whether a certain block in the 4x4 grid belongs to a specific tetromino with a specific rotation state, the package provides the `is_tetromino_solid_at` function.

```
function is_tetromino_solid_at(tetromino : tetromino_t; rotation : rotation_t; x, y :
    ↪ std_logic_vector(1 downto 0)) return boolean;
```

This function returns `true` if the block identified by the `x` and `y` coordinates belongs to the specified tetromino. The coordinate system used for this purpose is shown in Figure 8.3.

The following code shows some examples on how to use this function.

```
assert is_tetromino_solid_at(TET_T, ROT_0, "00", "00") = false;
assert is_tetromino_solid_at(TET_T, ROT_0, "01", "00") = true;
assert is_tetromino_solid_at(TET_T, ROT_0, "10", "01") = true;
```

[2]`https://en.wikipedia.org/wiki/Tetromino`, in the context of the game Tetris also the term "tetriminos" is used.

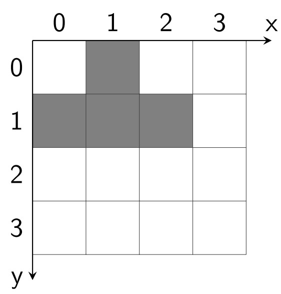


Figure 8.3: Tetromino coordinate system (demonstrated for the tetromino identified by `TET_T` and `ROT_0`)

8.2 Dependencies

- Mathematical support package (math_pkg)
- VGA Graphics Controller package (vga_gfx_cntrl_pkg)

8.3 Required VHDL Files

- `tetris_util_pkg.vhd`
- `tetromino_collider.vhd`
- `tetromino_drawer.vhd`

8.4 Component Declarations

8.4.1 Tetromino Drawer

VHDL Component Declaration:

```vhdl
component tetromino_drawer is
  generic (
    BLOCK_SIZE : integer;
    BLOCK_ROW : integer
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    start : in std_logic;
    busy : out std_logic;
    x : in std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
    y : in std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
    tetromino : in tetromino_t;
    rotation : in rotation_t;
    gfx_instr : out std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
    gfx_instr_wr : out std_logic;
    gfx_instr_full : in std_logic
  );
end component;
```

Generics Description:

Name	Functionality
BLOCK_SIZE	Defines the size in pixels of the individual square blocks the tetromino is composed of.
BLOCK_ROW	To draw the blocks that make up a tetromino, the core uses BIT_BLIT instructions. The value of BLOCK_ROW*BLOCK_SIZE defines the y coordinate in BB ROM that is used as the source of the blitting operations. To get the accompanying x coordinate the core multiplies the current (3-bit) value at the tetromino input with BLOCK_SIZE.

Port Signals Description:

Name	Dir.	Width/Type	Functionality
clk	in	1	Global clock signal
res_n	in	1	Low active reset signal
start	in	1	This signal is used to 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 input signals marked as *input parameters* must be valid when start is asserted and must remain valid (and unchanged) until the busy signal goes low.
busy	out	1	The core asserts this signal to indicate that it is currently performing a drawing operation. As soon as the it is done, busy goes low again, which allows for a new operation to be started using the start input.
tetromino	in	tetromino_t	The tetromino to draw. [input parameter]
rotation	in	rotation_t	The rotation to draw the tetromino with. [input parameter]
x	in	GFX_INSTR_WIDTH	The signed x coordinate of the destination position of the tetromino in the frame buffer (i.e., the x coordinate of the upper left corner of the 4x4 tetromino grid.) [input parameter]
y	in	GFX_INSTR_WIDTH	The signed y coordinate of the destination position of the tetromino in the frame buffer (i.e., the y coordinate of the upper left corner of the 4x4 tetromino grid.) [input parameter]
gfx_instr	out	GFX_INSTR_WIDTH	The actual graphics instructions.
gfx_instr_wr	out	1	The write flag for graphics instructions.
gfx_instr_full	in	1	The full flag for graphics instructions.

8.4.2 Tetromino Collider

VHDL Component Declaration:

```vhdl
component tetromino_collider is
  generic (
    BLOCKS_X : integer;
    BLOCKS_Y : integer
  );
  port (
    clk : in std_logic;
    res_n : in std_logic;
    start : in std_logic;
    busy : out std_logic;
    collision_detected : out std_logic;
    tetromino_x : in std_logic_vector(log2c(BLOCKS_X) downto 0);
    tetromino_y : in std_logic_vector(log2c(BLOCKS_Y) downto 0);
    tetromino : in tetromino_t;
    rotation : in rotation_t;
    block_map_x : out std_logic_vector(log2c(BLOCKS_X)-1 downto 0);
    block_map_y : out std_logic_vector(log2c(BLOCKS_Y)-1 downto 0);
    block_map_rd : out std_logic;
    block_map_solid : in std_logic
  );
end component;
```

Generics Description:

Name	Functionality
BLOCKS_X	The width (in blocks) of the 2D map processed by this core.
BLOCKS_Y	The height (in blocks) of the 2D map processed by this core

Port Signals Description:

Name	Dir.	Width/Type	Functionality
clk	in	1	Global clock signal
res_n	in	1	Low active reset signal
start	in	1	This signal is used to initiate the collision check run. It must be asserted for exactly one clock cycle. The core will react to this event by asserting the busy signal. The input signals marked as *input parameters* must be valid when start is asserted and must remain valid (and unchanged) until the busy signal goes low.
busy	out	1	The core asserts this signal to indicate that it is currently performing a collision check run. As soon as the it is done, busy goes low again, which allows for a new run to be started using the start input.
collision_detected	out	1	The signal indicating if a collision between the supplied tetromino (defined by the input parameters) and the map has been detected. This signal is only updated when the busy signal goes to low. A high signal level indicates a collision.
tetromino	in	tetromino_t	The tetromino to check. [input parameter]
rotation	in	rotation_t	The rotation of the tetromino. [input parameter]
tetromino_x	in	log2c(BLOCKS_X)+1	Signed x position of the tetromino on the map (i.e., the x coordinate of the upper left block of the 4x4 tetromino grid.). [input parameter]
tetromino_y	in	log2c(BLOCKS_Y)+1	Signed y position of the tetromino on the map (i.e., the y coordinate of the upper left block of the 4x4 tetromino grid.). [input parameter]
block_map_x	out	log2c(BLOCKS_X)	The x coordinate of a map position that needs to be sampled. Only valid when block_map_rd is asserted.
block_map_y	out	log2c(BLOCKS_Y)	The y coordinate of a map position that needs to be sampled. Only valid when block_map_rd is asserted.
block_map_rd	out	1	A flag indicating that the core wants to sample the position defined by the coordinates block_map_x and block_map_y. If this signal is asserted, the environment of the core needs to provide an appropriate answer at the input block_map_solid in the next clock cycle.
block_map_solid	in	1	A signal indicating whether the sampled map position is solid or not. Must be valid in the cycle after block_map_rd was asserted (a high signal level corresponds to solid map positions).

8.5 Interface Protocol

8.5.1 Tetromino Drawer

To draw a tetromio, the relevant input parameters (i.e., the inputs tetromino, rotation, x and y) must be applied to the core and the start signal must be asserted for exactly one clock cycle. The core will then assert the busy signal and keep it high as long as it is processing and/or generating graphics instructions. During this time the input parameters must not be changed and the start signal must not be asserted.

The tetromino_drawer is designed to communicate (directly) with the vga_gfx_cntrl. It hence obeys its interface protocol with respect to the gfx_instr* signals. Figure 8.4 shows an example timing diagram for the tetromino_drawer.

As soon as the busy signal is deasserted, a new drawing process can be started.

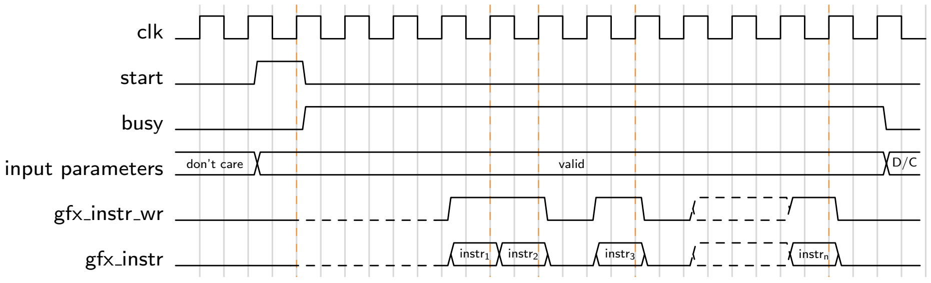


Figure 8.4: Tetromino Drawer example timing diagram

8.5.2 Tetromino Collider

The tetromino_collider can be used to check whether a given tetromino (defined by signals tetromino and rotation) collides with a 2D map of blocks or its borders when placed on this map at the coordinates tetromino_x and tetromino_y. Figure 8.5 illustrates the coordinate system that is used for this purpose. The size of the map is defined by the generics BLOCKS_X and BLOCKS_Y. The gray blocks in the figure indicate map positions that are occupied or solid. If a block of a tetromio would overlap with such a position, a collision would be reported. Hence for the shown scenario, there are no collisions.

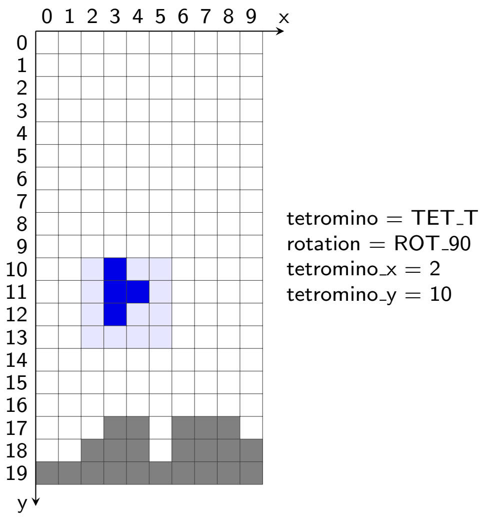


Figure 8.5: Block map coordinate system

Figure 8.6 shows an example timing diagram that demonstrates how to interface with the tetromino_collider. The inputs that define the shape and position of the tetromino to be checked are referred to as *input parameters*. To start a collision check run, the start signal must be asserted for exactly one clock cycle. The input parameters must be applied at the same time as the start signal and must not be changed during the whole run. The core asserts the busy signal to indicate that it is processing. A run is complete when the busy signal returns to low again. The result of the collision check can then be obtained at the collision_detected output (low meaning that no collision was detected).

During a run, the core samples various map positions to check whether they are solid or not. This is achieved via the signals block_map_x, block_map_y, block_map_rd and block_map_solid. Whenever block_map_rd is high, the core outputs the desired map coordinates at block_map_x and block_map_y and expects an answer on whether the respective position is solid or not on the next clock signal at the signal block_map_solid (low indicating that the position is not solid). At all other times the value of block_map_solid is ignored. Note that block_map_rd will never be high for more than one cycle at a time and there may be pauses of arbitrary lengths in between consecutive requests.

The basic idea with this interface is to use the signals block_map_x, block_map_y and block_map_rd to access an on-chip block RAM (see Section 3) that represents the map.

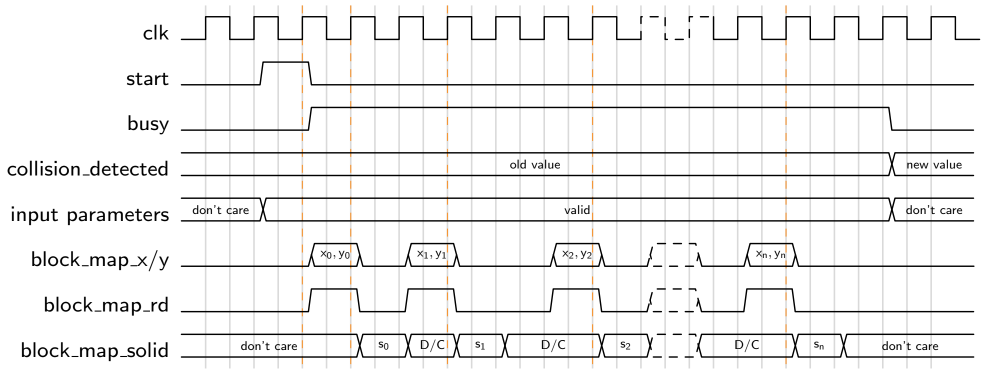


Figure 8.6: Tetromino Collider example timing diagram

Note, that the inputs tetromino_x and tetromino_y are **signed** numbers! This is necessary to be able to place certain tetrominoes everywhere on the map. Figure 8.7 shows some examples of tetrominoes that (i) don't produce collisions (blue) and (ii) that collide with either the borders of the map or some solid block (red). Note that the left-most tetromino on both maps has a negative x coordinate.

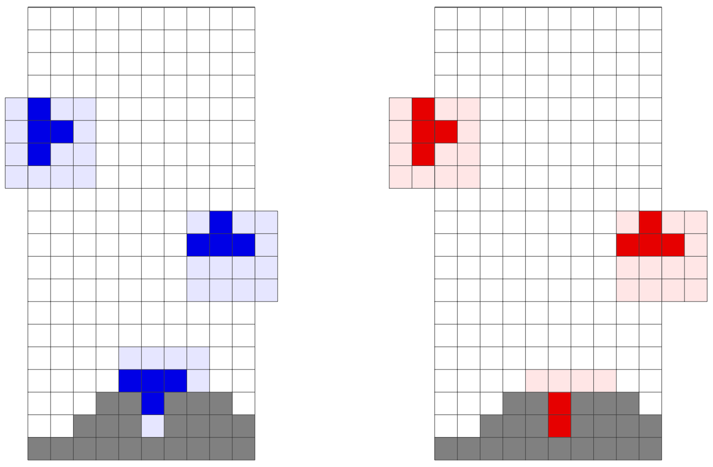


Figure 8.7: Tetromino placement and collision examples

Revision History

Revision	Date	Author(s)	Description
1.1	07.04.2022	FH	Added documentation for the tetris_util_pkg package
1.0	06.03.2022	FH	Initial version

Author Abbreviations:

FH	Florian Huemer
FK	Florian Kriebel