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, March 9, 2023

Contents

1 Mathematical Support Package **2**
1.1 Description . . . 2
1.2 Dependencies . . . 2
1.3 Required VHDL files . . . 2
1.4 Supported Functions . . . 2

2 Synchronizer **3**
2.1 Description . . . 3
2.2 Dependencies . . . 3
2.3 Required VHDL Files . . . 3
2.4 Component Declaration . . . 3
2.5 Interface Protocol . . . 4
2.6 Internal Structure . . . 4

3 On-chip Memory Package **5**
3.1 Description . . . 5
3.2 Dependencies . . . 5
3.3 Required VHDL Files . . . 5
3.4 Component Declarations . . . 5
3.4.1 Single clock dual-port RAM . . . 5
3.4.2 Single clock FIFO . . . 6
3.4.3 Single clock FIFO with FWFT behavior . . . 7
3.5 Interface Protocol . . . 7
3.5.1 Single clock dual-port RAM . . . 7
3.5.2 Single clock FIFO . . . 8
3.5.3 Single clock FIFO with FWFT behavior . . . 9

4 Audio Controller **11**
4.1 Dependencies . . . 11
4.2 Required Source Files . . . 11
4.3 Component Declaration . . . 12
4.4 Interface Protocol . . . 12

5 VGA Graphics Controller **14**
5.1 Internal Structure . . . 14
5.2 Dependencies . . . 15
5.3 Required VHDL Files . . . 15
5.4 Component Declaration . . . 15
5.5 Interface . . . 17
5.6 Graphics Commands . . . 17

Revision History **25**

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 (i.e
    ↪ ., flip flops)
    RESET_VALUE : std_logic -- reset value of the output signal
  );
  port (
    clk        : in std_logic;
    res_n      : in std_logic;
    data_in    : in std_logic;
    data_out   : out std_logic
  );
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.

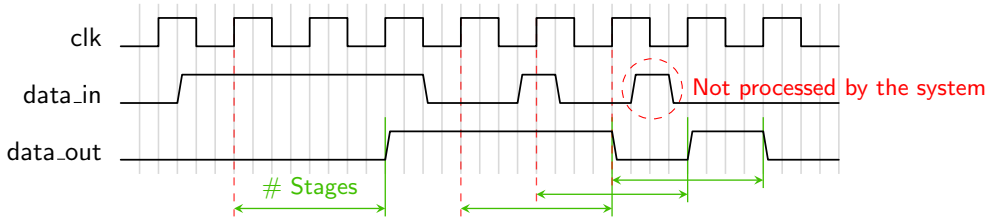


Figure 2.1: Synchronizer timing

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.

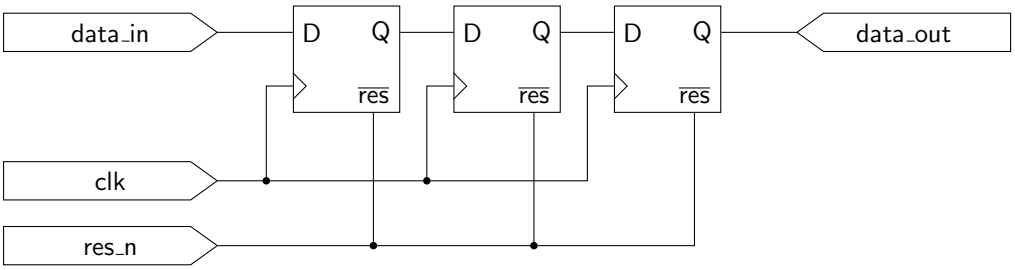


Figure 2.2: Synchronizer circuit

3 On-chip Memory Package

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 (
    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 flags
    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:

```
component fifo_1c1r1w_fwft is
  generic (
    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_ack : in std_logic;
    rd_valid : out std_logic;
    -- write port
    wr_data : in std_logic_vector(DATA_WIDTH - 1 downto 0);
    wr : in std_logic;
    -- status flags
    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).

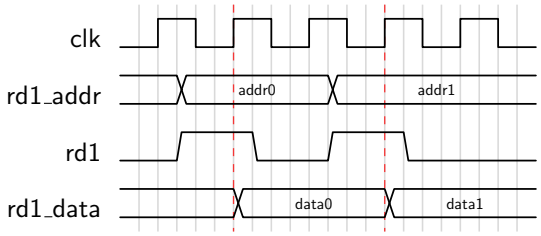


Figure 3.1: RAM read timing.

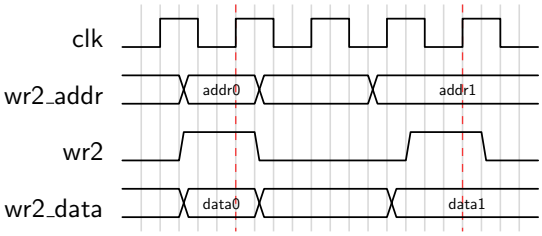


Figure 3.2: RAM write timing

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.

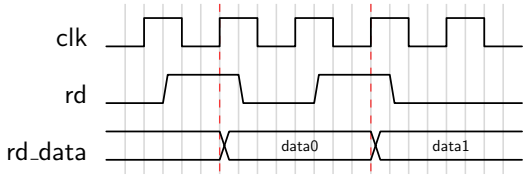


Figure 3.3: FIFO read timing

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).

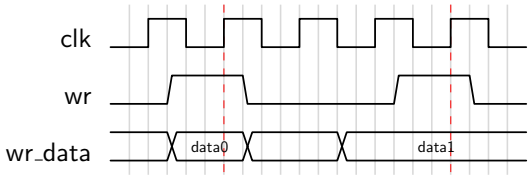


Figure 3.4: FIFO write timing

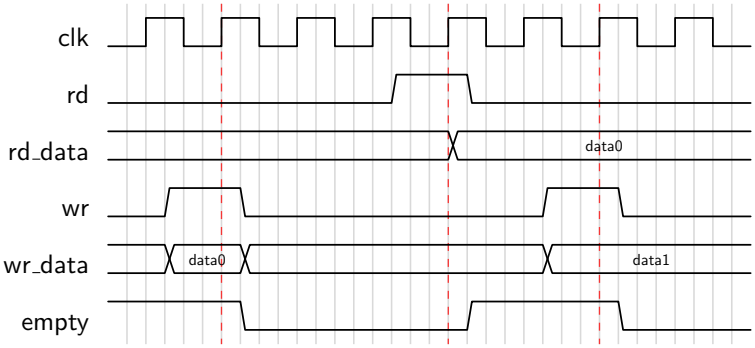


Figure 3.5: FIFO empty handling

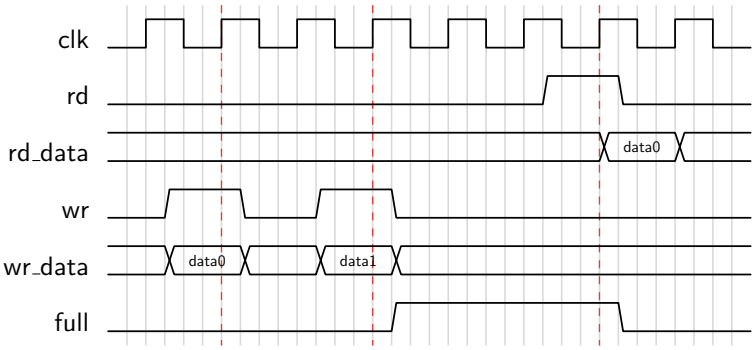


Figure 3.6: FIFO full handling

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.

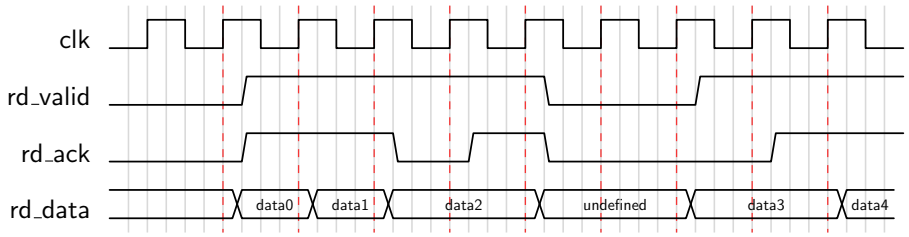


Figure 3.7: FIFO FWFT read timing

4 Audio Controller

The audio_ctrl 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.

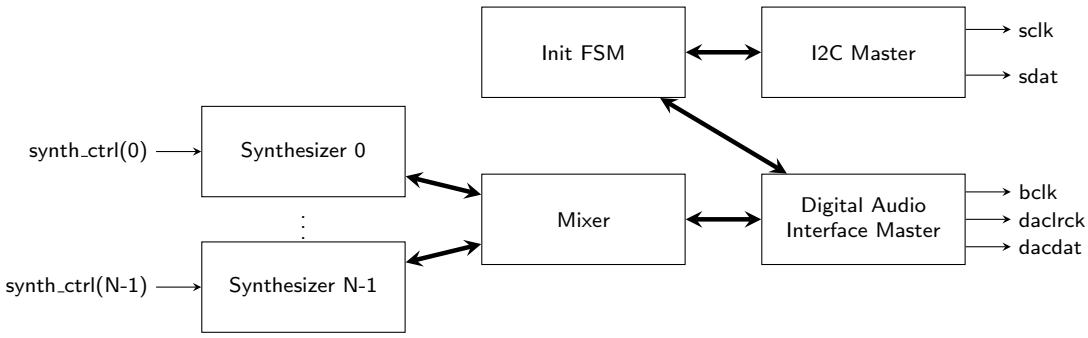


Figure 4.1: Audio controller internal structure

The audio controller must be clocked by a 12 MHz clock (which will internally be forwarded to the xck output). The synth_ctrl 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_ctrl_s2 is required. The audio_ctrl_pkg package provides the component declaration as well as the required type declaration for the synthesizer interface.

- `audio_ctrl_pkg.vhd`
- `audio_ctrl_s2.vhd`
- `audio_ctrl_top.vho`
- `audio_ctrl_top.qxp`

Hence, if you want to simulate your design in Questa/Modelsim, use the files `audio_ctrl_s2.vhd` and `audio_ctrl_pkg.vhd` as well as the netlist file `audio_ctrl_top.vho`. For synthesis in Quartus use `audio_ctrl_s2.vhd` and `audio_ctrl_pkg.vhd` and the Exported Partition File `audio_ctrl_top.qxp` file.

4.3 Component Declaration

VHDL Component Declaration:

```
component audio_ctrl_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_ctrl : in synth_ctrl_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_ctrl	in	synth_ctrl_vec_t (0 to 1)	The synthesizer control signals

4.4 Interface Protocol

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

```
type synth_ctrl_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 VGA Graphics Controller

The VGA Graphics Controller performs graphical operations on bitmaps using a command-based interface. These operations include simple geometric operations (e.g., setting individual pixels, drawing lines, etc) as well as the ability to copy and transform (i.e., rotating) a section of one bitmap onto another one, an operation also referred to as bit blitting[1] Bitmaps are stored in external SRAM, referred to as Video RAM (VRAM) throughout this document. The VGA Graphics Controller interfaces with the ADV7123 video DAC to produce an RGB analog component video signal that can be output through a VGA connector. It supports one fixed output resolution of 320x240 pixels and a color depth of 8 bit (RGB332).

5.1 Internal Structure

Figure 5.1 shows an overview of the internal structure of the core.

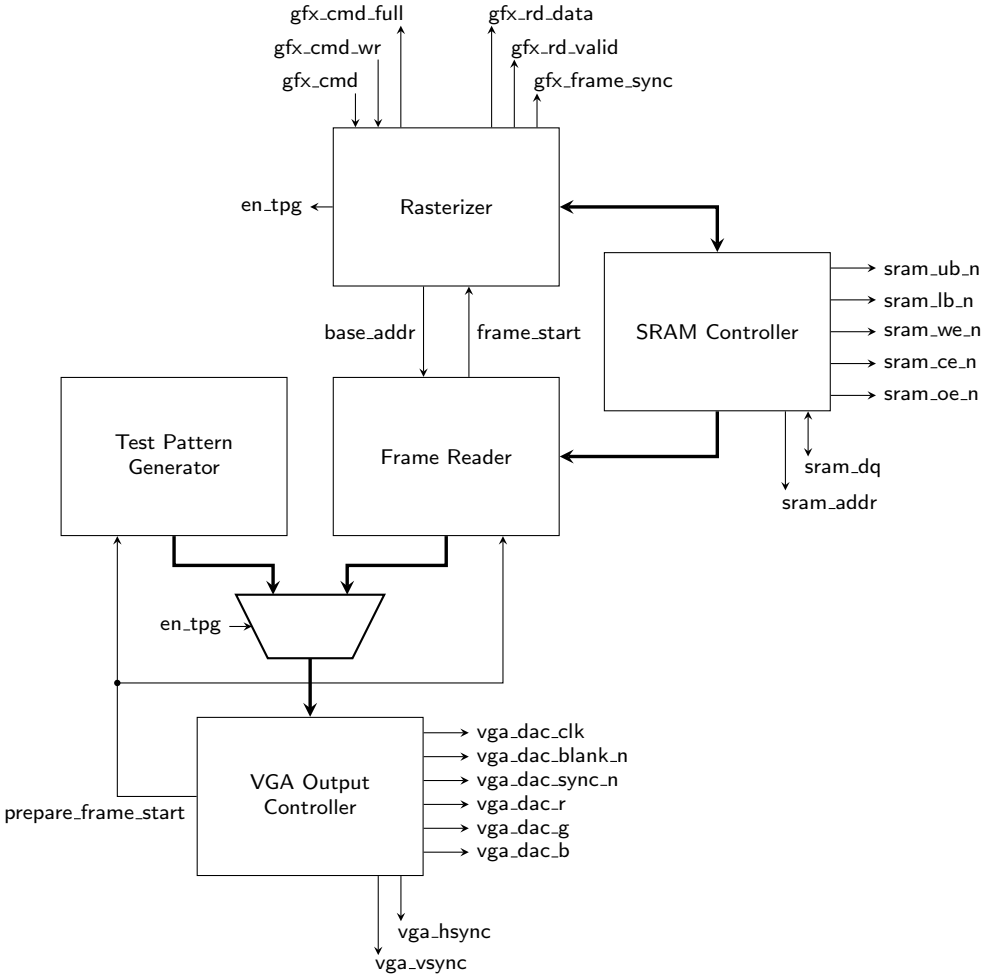


Figure 5.1: VGA Graphics Controller overview

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. It executes the actual graphics commands by reading and writing pixel data from and to bitmaps in VRAM. VRAM is byte-addressed, which means that every address corresponds to the 8-bit color value of a single pixel in the RGB332 format. Bitmaps in VRAM are identified by Bitmap Descriptors (BDs), which are stored internally in the Rasterizer. A BD contains 3 values:

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

- base address (unsigned 21 bit)
- width (unsigned 15 bit)
- height (unsigned 15 bit)

BDs are used by the Rasterizer to calculate the effective addresses of pixels in VRAM. Given the pixel coordinates x and y for some pixel, its effective address in VRAM is given by

$$[\text{effective address}] = [\text{base address}] + y * [\text{width}] + x$$

The coordinate $(x, y) = (0, 0)$ points to the pixel of the upper left corner of the bitmap, while $(x, y) = ([\text{width}] - 1, [\text{height}] - 1)$ points to the lower right corner. Note that bitmaps must always start at even addresses in VRAM (i.e., the LSB of the base address must be zero).

To generate the actual video signal the VGA Output Controller (VOC) 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 VOC 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 VOC outputs a resolution of 640x480. This is done for the sake of compatibility, since not all monitors support resolutions lower than 640x480.

Hence, the Frame Reader and the TPG have to supply image data with a 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 VRAM. 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.

5.2 Dependencies

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

5.3 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_ctrl_pkg.vhd`
- `vga_gfx_ctrl.qxp`
- `vga_gfx_ctrl.vho`

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

5.4 Component Declaration

VHDL Component Declaration:

```
component vga_gfx_ctrl is
  port (
    clk   : in std_logic;
    res_n : in std_logic;
    display_clk   : in std_logic;
    display_res_n : in std_logic;

    -- command interface
```

```
9       gfx_cmd        : in std_logic_vector(15 downto 0);
10      gfx_cmd_wr     : in std_logic;
11      gfx_cmd_full   : out std_logic;
12      gfx_rd_data      : out std_logic_vector(15 downto 0);
13      gfx_rd_valid     : out std_logic;
14      gfx_frame_sync   : out std_logic;
15
16      -- external interface to SRAM
17      sram_dq : inout std_logic_vector(15 downto 0);
18      sram_addr : out std_logic_vector(19 downto 0);
19      sram_ub_n : out std_logic;
20      sram_lb_n : out std_logic;
21      sram_we_n : out std_logic;
22      sram_ce_n : out std_logic;
23      sram_oe_n : out std_logic;
24
25      -- connection to VGA connector/DAC
26      vga_hsync        : out std_logic;
27      vga_vsync        : out std_logic;
28      vga_dac_clk      : out std_logic;
29      vga_dac_blank_n : out std_logic;
30      vga_dac_sync_n   : out std_logic;
31      vga_dac_r        : out std_logic_vector(7 downto 0);
32      vga_dac_g        : out std_logic_vector(7 downto 0);
33      vga_dac_b        : out std_logic_vector(7 downto 0)
34    );
35  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_cmd_wr	in	1	The write signal of the instruction FIFO
gfx_cmd_full	out	1	The full signal of the instruction FIFO
gfx_cmd	in	16	The actual commands and operands
gfx_rd_valid	out	1	Valid flag for the data returned by read commands. This signal is 1 of exactly one clock cycle.
gfx_rd_data	out	16	The actual data returned by a read command
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
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

5.5 Interface

Graphics commands are fed into the core using the signals gfx_cmd_wr and gfx_cmd. Internally those signals directly feed a FIFO, referred to as the Command FIFO, whose full flag is in turn output at gfx_cmd_full. Hence, this port (i.e., the signals gfx_cmd_wr, gfx_cmd and gfx_cmd_full) 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 `DISPLAY_BMP` command and allows the synchronization of an interfacing IP core to the start of a new frame output by the VOC.

The signals gfx_rd_data and gfx_rd_valid are used by commands that read data from VRAM. Whenever the core writes new data to gfx_rd_data, gfx_rd_valid is asserted for exactly one clock cycle. The data on gfx_rd_data remains valid until gfx_rd_valid goes high again.

5.6 Graphics Commands

A graphics command always consists of a 16 bit instruction and a number of 16 bit operands associated with the instruction. Depending on the instruction the number of operands can range from 0 to 4, with one command having a variable number of upto 2^{16+3} operands. This means that for most instructions multiple 16 bit words must be issued to the Command FIFO using the inputs gfx_cmd_wr and gfx_cmd.

Internally the VGA Graphics Controller maintains a number of registers, which define its internal state are used by and modified through the execution of graphics commands:

- Active Bitmap Descriptor `abd` (51 bit)
  - `abd.base` (unsigned 21 bit)
  - `abd.width` (unsigned 15 bit)
  - `abd.height` (unsigned 15 bit)
- Graphics Pointer `gp` (2x16 bit)
  - `gp.x` (signed 16 bit)
  - `gp.y` (signed 16 bit)
- Primary Color (8 bit)
- Secondary Color (8 bit)
- Bitmap Descriptor Table `bdt` (8 descriptors, i.e., 8x51 bit)
  - `bdt[0]`
    - `bdt[0].base` (unsigned 21 bit)
    - `bdt[0].width` (unsigned 15 bit)
    - `bdt[0].height` (unsigned 15 bit)
  - ...
  - `bdt[7]`
    - `bdt[7].base` (unsigned 21 bit)
    - `bdt[7].width` (unsigned 15 bit)
    - `bdt[7].height` (unsigned 15 bit)
- Bit Blit Effect (10 bit)
  - `maskop` (2 bit)
  - `mask` (8 bit)
- Current Output Address (21 bit)

After reset all registers are initialized to 0.

The `abd` defines the bitmap that is the target of all subsequent drawing operations. The `gp` register represents a 2D coordinate on this bitmap (i.e., the current image in VRAM that the graphics instructions draw to) and is used by most drawing commands (e.g., `SET_PIXEL`, `DRAW_HLINE`, `BB_*`, etc.). The coordinate `gp.x` =0, `gp.y` =0 points to the pixel of the upper left corner of the bitmap, while `gp.x` =`abd.width`-1, `gp.y` =`abd.height`-1 points to the lower right corner. 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 active bitmap. This is completely fine, as the Rasterizer ensures that a write access to a pixel outside of the 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, e.g., $(-10, 0)$ to $(10, 0)$. The `gp` is changed explicitly using the `MOVE_GP` and `INC_GP` instructions. However, all drawing commands offer the possibility to automatically move the `gp` upon command completion.

The `bdt` holds 8 bitmap descriptors that are used (referenced) by certain commands (using the 3 bit instruction field `bmpidx` as an index to this table). The `BB_*` commands always read this table to get the descriptors of the source bitmap of the blitting operation. Data is written to the `bdt` using the `DEFINE_BMP` command. Furthermore, the commands `DISPLAY_BMP` and `ACTIVATE_BMP` read the `bdt`.

The core also contains two 8 bit color registers, the primary and the secondary color. An instruction can refer to one of these colors using its `cs` (color selector) field, where 0 (1) refers to the primary (secondary) color. The colors are set using the `SET_COLOR` instruction.

A few commands (`VRAM_*`, `DEFINE_BMP`) have to specify a VRAM address in their operands. Since operands are only 16 bits wide and a VRAM address requires 21 bits, two operands (`addrlo` and `addrhi`) are used for this purpose. The resulting VRAM (byte) address is obtained by concatenating the lower 16 bits (`addrlo`) with the upper 5 bits (`addrhi`), which we denote by `addrhi & addrlo`. The `VRAM_*` commands allow to access individual bytes and words[2] in VRAM. Accessing words is only permitted on even addresses, i.e., the LSB of the address must be zero. The byte order used is little-endian. Assume that 0x12 is stored in address 0x0 in VRAM and that address 0x1 stores 0x34. Reading a word from address 0x0, thus, returns 0x3412.

The `BB_CLIP` command is the most generic of the bit blit commands, but needs 4 separate arguments. For this reason the core also supports the simpler `BB_FULL` and `BB_CHAR` commands. The following (Python) listing demonstrates how the bit blit operations calculate the read address for a pixel in the source bitmap and the write address for the respective pixel destination bitmap.

```
#!/bin/env python3

from collections import namedtuple
from enum import Enum

BD = namedtuple("BD", ["base_address", "width", "height"])
BMPSection = namedtuple("BMPSection", ["x", "y", "width", "height"])
Point = namedtuple("Point", ["x", "y"])
Rotation = Enum("Rotation", "R0 R90 R180 R270")

def BitBlit(dst, dst_position, src, src_section, rotation):
  for x in range(0, src_section.width):
    for y in range(0, src_section.height):
      dst_x_offset = src_section.width-1-x if rotation in [Rotation.R180, Rotation.R270]
    ↪  else x
      dst_y_offset = src_section.height-1-y if rotation in [Rotation.R90, Rotation.R180]
    ↪  else y
      if rotation in [Rotation.R90, Rotation.R270]:
        # swap coordinates
        dst_x_offset, dst_y_offset = dst_y_offset, dst_x_offset
      dst_x = dst_position.x + dst_x_offset
      dst_y = dst_position.y + dst_y_offset
      src_x = src_section.x + x
      src_y = src_section.y + y

      src_addr = src.base_address + src_y * src.width + src_x
      dst_addr = dst.base_address + dst_y * dst.width + dst_x
```

[2]In the context of this core a word is considered to be 16 bit (i.e., 2 byte) wide.

```
26        print(f"source pixel ({src_x},{src_y}) (addr={hex(src_addr)}) --> destination
      ↪ pixel ({dst_x},{dst_y}) (addr={hex(dst_addr)})")
27
28  BitBlit(
29    dst=BD(0,320,240),
30    dst_position=Point(10, 20),
31    src=BD(320*240,8,4),
32    src_section=BMPSection(0,0,4,4),
33    rotation=Rotation.R0
34  )
```

Now a detailed bit-level specification of each command 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.

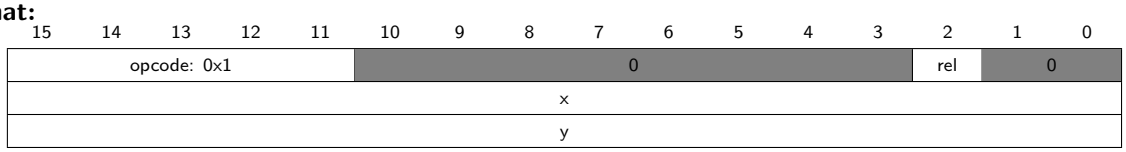
NOP

Format:

Description: Do nothing.

MOVE_GP

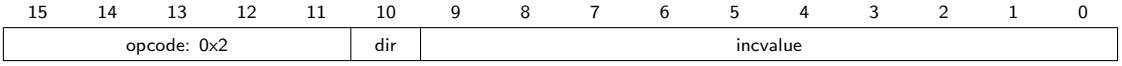
Format:



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`). The operands `x` and `y` are signed 16-bit values.

INC_GP

Format:



Description: Adds the signed 10 bit integer in `incvalue` to either `gp.y` (`dir`=1) or `gp.x` (`dir`=0). For that purpose `incvalue` is sign-extended.

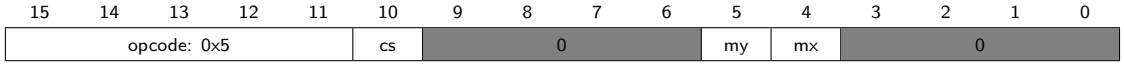
CLEAR

Format:

Description: Sets every pixel in active bitmap to the color specified by `cs`. Does not change the `gp`.

SET_PIXEL

Format:



Description: Sets the pixel in the active bitmap the `gp` currently points at to the color specified by `cs`. If the `gp` is outside of the bounds of the active bitmap no pixel is set. After that `gp.x` (`gp.y`) is incremented by one if `mx` (`my`) is set.

DRAW_HLINE

Format:

Description: Draws a horizontal line between the `gp` and the destination coordinate at (`gp.x` + `dx`, `gp.y`) using the color specified by `cs`. The operand `dx` is a signed 16-bit value. After the line has been drawn `gp.x` is set to the destination x coordinate of the line if `mx` is set. If `my` is set `gp.y` is incremented by one.

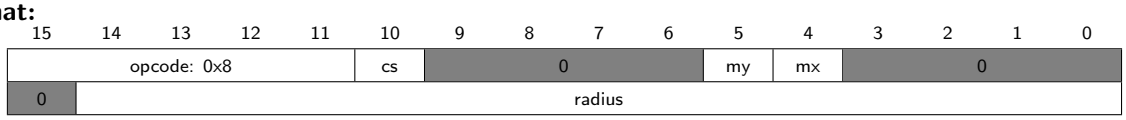
DRAW_VLINE

Format:

Description: Draws a vertical line between the `gp` and the destination coordinate at (`gp.x`, `gp.y` + `dy`) using the color specified by `cs`. The operand `dy` is a signed 16-bit value. After the line has been drawn `gp.y` is set to the destination y coordinate of the line if `my` is set. If `mx` is set `gp.x` is incremented by one.

DRAW_CIRCLE

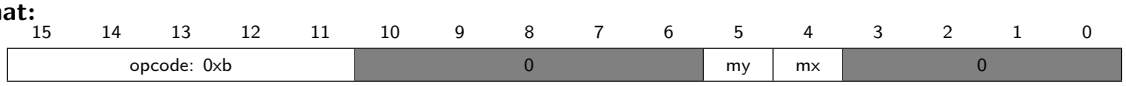
Format:



Description: Draws a circle with a radius specified by the operand `radius` and its center at the `gp` using the color specified by `cs`. The operand `radius` is an unsigned 15 bit value. After the circle has been drawn `gp.x` (`gp.y`) is incremented by `radius` if `mx` (`my`) is set.

GET_PIXEL

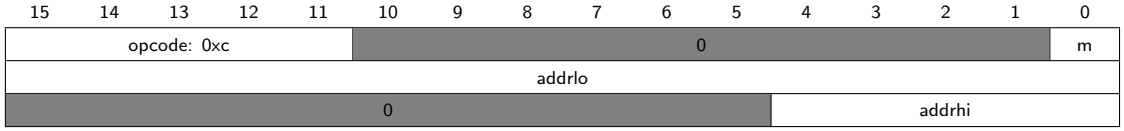
Format:



Description: Reads the color of the pixel in the active bitmap the `gp` currently points to and outputs it using `gfx_rd_data`/`gfx_rd_valid`. Since pixels are 8 bits wide only the lower 8 bits (i.e., 7 downto 0) of `gfx_rd_data` are used. The upper bits are set to zero. If the `gp` is outside of the bound of the active bitmap all bits in `gfx_rd_data` are set. After that `gp.x` (`gp.y`) is incremented by one if `mx` (`my`) is set.

VRAM_READ

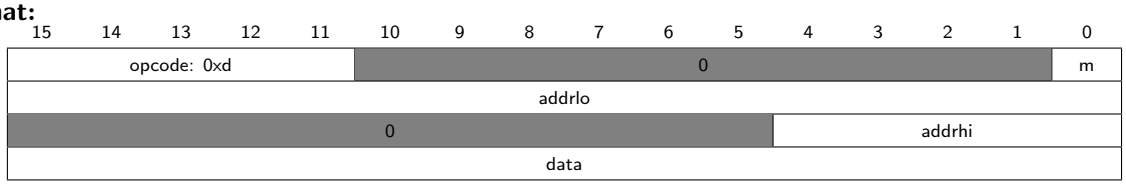
Format:



Description: Performs a read operation on the VRAM address `addrhi & addrlo` and outputs the result using gfx_rd_data/gfx_rd_valid. If `m` is 0, a byte access is performed. In this case the upper byte (15 downto 8) in gfx_rd_data is set to zero and the read byte is placed in the lower byte (7 downto 0). If `m` is 1, a word access is performed. For a word access the LSB of the address must be zero.

VRAM_WRITE

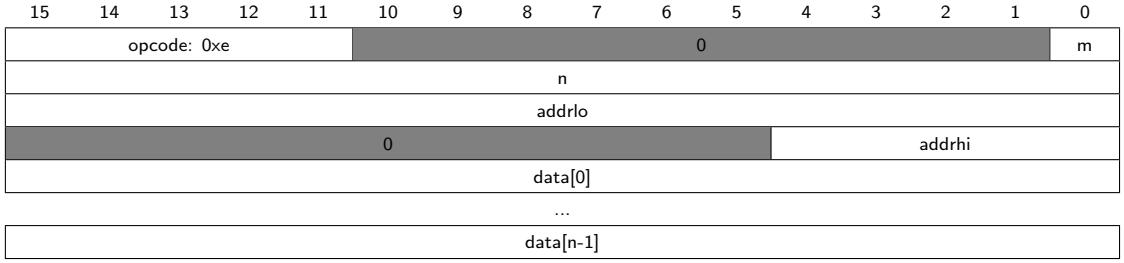
Format:



Description: Writes a single byte (`m`=0) or word (`m`=1) to VRAM. If a byte access is performed only the lower 8 bits of the `data` operand are used, the upper 8 bits are ignored. For a word access the LSB of the address must be zero.

VRAM_WRITE_SEQ

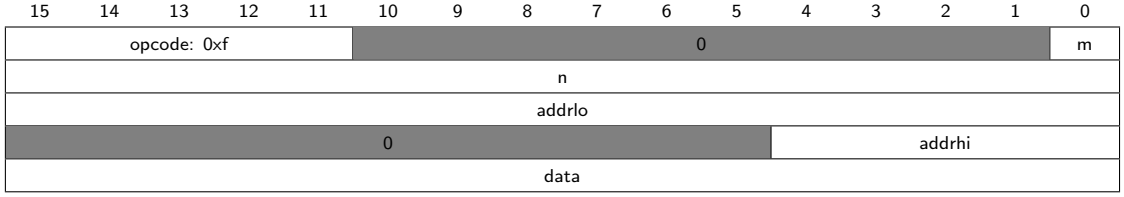
Format:



Description: Writes a sequence of bytes (`m`=0) or words (`m`=1) to VRAM starting at the address `addrhi & addrlo`. In byte mode (`m`=0) only the lower 8 bits of each of the operands `data[0]` to `data[n-1]` are used. The last address written is (`addrhi & addrlo`) + `n` - 1. In word mode (`m`=1) the last address written is (`addrhi & addrlo`) + 2*(`n` - 1).

VRAM_WRITE_INIT

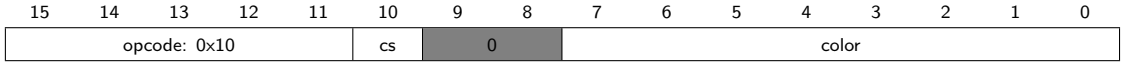
Format:



Description: Initializes a range of memory addresses starting at `addrhi & addrlo` to `data`. In byte mode (`m`=0) only the lower 8 bits of the operand `data` are used. The last address written is (`addrhi & addrlo`) + `n` - 1. In word mode (`m`=1) the last address written is (`addrhi & addrlo`) + 2*(`n` - 1).

SET_COLOR

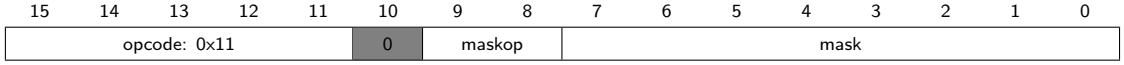
Format:



Description: Sets the primary (secondary) color to `color` if `cs` is 0 (1). For the actual color value in `color` the RGB332 format is used.

SET_BB_EFFECT

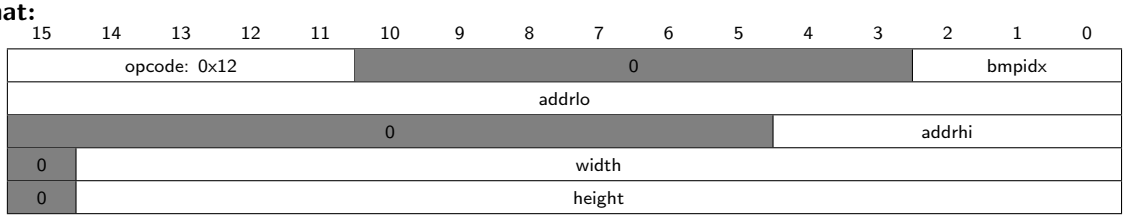
Format:



Description: Sets the current BB Effect (i.e., the registers `maskop` and `mask`) used by all subsequent `BB_*` commands. See the entry for `BB_CLIP` for the purpose of these registers.

DEFINE_BMP

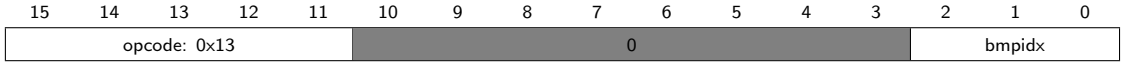
Format:



Description: Writes a Bitmap Descriptor to `bdt[bmpidx]`. Bitmaps always start at even addresses. This means that the LSB of the low address is assumend to be zero.

ACTIVATE_BMP

Format:



Description: Sets the Active Bitmap Descriptor by copying `bdt[bmpidx]` to `abd`. All subsequent drawing commands will use this bitmap as their target. Changing `bdt[bmpidx]` with a subsequent `DEFINE_BMP` command does not affect the `abd`.

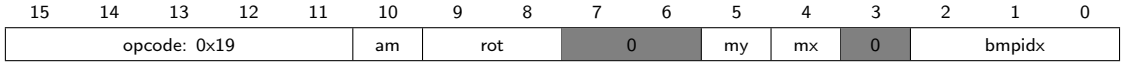
DISPLAY_BMP

Format:

Description: Sets the Current Output Address for the Frame Reader to `bdt[bmpidx].base`. If the frame synchronization flag `fs` is 1, the command blocks the execution of the following graphics commands until the Frame Reader starts to fetch a new frame. This feature makes it possible to implement double buffering. If `fs`=1 the gfx_frame_sync signal is asserted for exactly one clock cycle to indicate that the command has been executed. The dimension of the bitmap referenced by `bmpidx` must be 320x240 pixels, `bdt[bmpidx].width` and `bdt[bmpidx].height` are ignored.

BB_FULL

Format:



Description: This command is equivalent to calling `BB_CLIP` with the operands (0, 0, `bdt[bmpidx].width`, `bdt[bmpidx].height`).

BB_CHAR

Format:

Description: This command is equivalent to calling `BB_CLIP` with the operands (`xoffset`, 0, `charwidth`, `bdt[bmpidx].height`).

BB_CLIP

Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode: 0x18					am	rot		0		my	mx	0	bmpidx		
0	x														
0	y														
0	width														
0	height														

Description: Performs a bit blit operation by copying (and transforming) the bitmap section defined by `x`, `y`, `width` and `height` of the source bitmap identified by `bmpidx` to the Active Bitmap to the position of the `gp`. If the rectangle defined by unsigned 15-bit operands `x`, `y`, `width` and `height` lies (even only partially) outside of the bound of the source bitmap the behavior of the command is undefined. If the bounds of the drawn image section are outside of the bounds of the destination bitmaps (i.e., the Active Bitmap) clipping is performed. The `rot` field is used to control the rotation of the copied image section and can take the following values:

- 00: no rotation
- 01: 90°clockwise rotation
- 10: 180°clockwise rotation
- 11: 270°clockwise rotation

After the execution of the command `gp.x` (`gp.y`) is incremented by dx (dy) if `mx` (`my`) is 1:

$$dx = \begin{cases} \texttt{width} & \text{if } \texttt{rot}{=}00 \text{ or } \texttt{rot}{=}10 \\ \texttt{height} & \text{otherwise} \end{cases}, \quad dy = \begin{cases} \texttt{height} & \text{if } \texttt{rot}{=}00 \text{ or } \texttt{rot}{=}10 \\ \texttt{width} & \text{otherwise} \end{cases}$$

If the alpha mode (`am`) flag is 1, pixels in the source image that match the secondary color are not copied to the active bitmap. The behavior of this command is changed by the registers `mask` and `maskop`. Depending on the value of `maskop` the pixels read from the source bitmap are transformed by performing a bitwise Boolean operation with the `mask` register before writing them to the Active Bitmap. Let c be the pixel color read for some pixel of the source bitmap, then c' will be written to the Active Bitmap:

$$c' = \begin{cases} c & \text{if } \texttt{maskop} = 00 \\ c \text{ and } \texttt{mask} & \text{if } \texttt{maskop} = 01 \\ c \text{ or } \texttt{mask} & \text{if } \texttt{maskop} = 10 \\ c \text{ xor } \texttt{mask} & \text{if } \texttt{maskop} = 11 \end{cases}$$

When the alpha mode is active (i.e., if `am` is 1) the same transformation is also applied to the secondary color. This is done such that the “transparency” of pixels is preserved. However, the actual register where the secondary color is stored is not changed.

Revision History

Revision	Date	Author(s)	Description
1.0	09.03.2023	FH	Initial version

Author Abbreviations:

FH Florian Huemer
FK Florian Kriebel