Key Design Features

- Technology independent soft IP Core for FPGA, ASIC and SoC devices
- Supplied as human-readable VHDL (or Verilog) source code
- Fully pipelined architecture with input/output flow-control and AXI4-compatible data streaming interfaces
- Supports both text and graphics (bitmap) overlays over real-time video
- Support for RGB pixels in/out (YCbCr formats on request)
- Support for all video resolutions up to 4096x4096 pixels
- No external memory or frame buffer required
- Bitmaps organized into tiles with a choice of four possible tile sizes: 8x16, 16x32, 32x64 or 64x128
- Tiles organized into 3 bit-planes offering 3-bits/pixel
- Programmable text and graphics map directly to the display
- Programmable graphics-window position and size - offers functionality like a 2D 'blitter' but without the need for an external memory
- Programmable window clipping region
- Independent horizontal and vertical scrolling
- Choice of 8 x 24-bit colours from a user defined palette or per-pixel alpha blending with 8 levels of transparency
- Per-pixel alpha-blending removes jagged edges to give a smooth anti-aliased result
- User-defined 8-bit alpha transparency
- No complex programming required
- Cascade any number of cores in series for more complex text and graphical displays
- Optional I2C, SPI or UART interfaces for simple programming

Applications

- Clear and functional video overlays (on screen displays) featuring text and graphics
- Digital TV and home-media solutions
- Interactive guides, menus, tables, lists
- Animated graphics including hardware 'sprites', pointers, cursors, scrolling text, moving banners
- Instrumentation and monitoring applications including animated gauges, charts, dials, meters, counters
- Informational displays and simple HUDs for commercial, military and automotive applications

Block Diagram

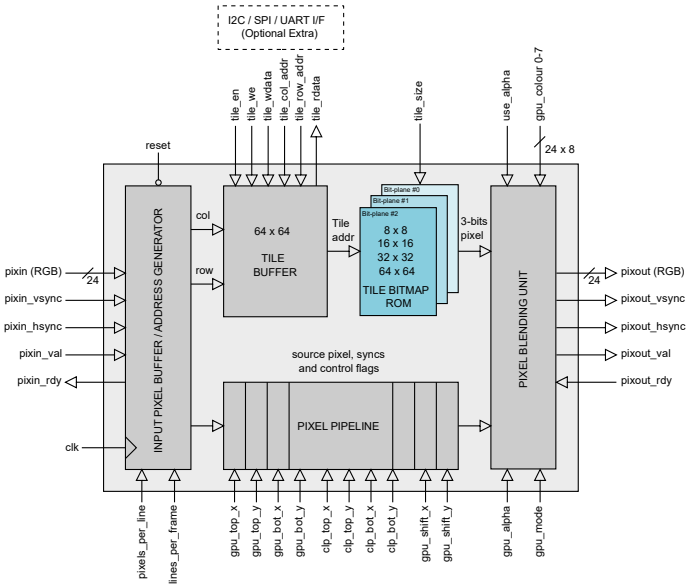

Figure 1: Simplified GPU overlay module architecture

General Description

The GPU_OVERLAY IP Core (Figure 1) is a highly versatile on-screen display processor that allows high-quality anti-aliased bitmap graphics and text to be inserted over RGB video. The module supports a wide range of graphics effects and the programming interface is very simple to use. The bitmap overlay is partitioned into an array of tiles which are stored in a local tile buffer memory. There are four tile sizes available which are either: 8x16, 16x32, 32x64 or 64x128.

The tiles in the buffer are displayed in a graphics window which may be positioned anywhere within the display area. Bitmaps for each tile are defined in a ROM which can contain up to 128 different bitmaps stored over three bit-planes. Depending on the chosen graphics mode, the 3-bits per pixel may be used to select one colour from a palette of eight, eight levels of alpha transparency or seven colours on a transparent background. There is also a feature to highlight a tile with a background colour. This is especially useful for interactive text-based menus and lists where the user has to select an option.

Pixels flow in and out of the overlay module in accordance with a simple streaming (valid/ready) protocol[1]. Pixels and syncs are sampled at the module inputs on a rising clock-edge when *pixin_val* is high and *pixin_rdy* is high. Likewise, pixels and syncs are transferred out of the module on a rising clock-edge when *pixout_val* and *pixout_rdy* are asserted high. The streaming protocol allows both input and output interfaces to be stalled independently.

The streaming protocol is very versatile and permits any number of GPU overlay modules to be cascaded in series. By placing more than one module together, the user is able to achieve more complex text and graphics displays with different fonts and colours.

1 See Zipcores application note: app_note_zc001.pdf for examples of how to use the valid-ready pipeline/streaming protocol

Generic Parameters

Generic name	Description	Type	Valid Range
tile_size	Tile size selection	integer	0: small (8x16) 1: medium (16x32) 2: large (32x64) 3: xlarge (64x128)
use_alpha	Enable/disable alpha-blend support	boolean	True / False
gpu_mode	Graphics mode selection	integer	0: unique-colour with per-pixel alpha blending 1: 8-colour palette with 8-bit user defined alpha 2: 7-colour palette with 8-bit user defined alpha + transparent background
pixels_per_line	No. of pixels in each input video line	integer	≤ 4096
lines_per_frame	No. of lines in each input video frame	integer	≤ 4096

Pin-out Description

Pin name	I/O	Description	Active state
clk	in	Synchronous clock	rising edge
reset	in	Asynchronous reset	low
gpu_alpha [7:0]	in	Alpha transparency of graphics window region	data
gpu_colour0-7[23:0]	in	User defined palette of 8x24-bit colours (RGB888)	data
gpu_top_x [11:0]	in	Top-left x position of graphics-window	data
gpu_top_y [11:0]	in	Top-left y-position of graphics-window	data
gpu_bot_x [11:0]	in	Bottom-right x position of graphics-window	data
gpu_bot_y [11:0]	in	Bottom-right y position of graphics-window	data
gpu_shift_x [11:0]	in	Horizontal shift in pixels	data
gpu_shift_y [11:0]	in	Vertical shift in pixels	data
clp_top_x [11:0]	in	Top-left x position of clip-box	data
clp_top_y [11:0]	in	Top-left y-position of clip-box	data
clp_bot_x [11:0]	in	Bottom-right x position of clip-box	data
clp_bot_y [11:0]	in	Bottom-right y position of clip-box	data

Pin-out Description cont...

Pin name	I/O	Description	Active state
tile_en	in	Tile buffer enable	high
tile_we	in	Tile buffer write enable	high
tile_wdata [7:0]	in	Tile buffer write data	data
tile_col_addr [5:0]	in	Tile buffer column address	data
tile_row_addr [4:0]	in	Tile buffer row address	data
tile_rdata [7:0]	out	Tile buffer read data	data
pixin [23:0]	in	24-bit RGB source pixel in	data
pixin_vsync	in	Vertical sync in (signifies start of frame)	high
pixin_hsync	in	Horizontal sync in (signifies start of line)	high
pixin_val	in	Input pixel valid	high
pixin_rdy	out	Ready to accept input pixel (handshake signal)	high
pixout [23:0]	out	24-bit pixel out	data
pixout_vsync	out	Vertical sync out	high
pixout_hsync	out	Horizontal sync out	high
pixout_val	out	Output pixel valid	high
pixout_rdy	in	Ready to accept output pixel (handshake signal)	high

Input pixel buffer / Address generator

Source video pixels are sampled at the input pixel buffer. The generic parameters *pixels_per_line* and *lines_per_frame* must be set correctly to match the exact number of pixels in x and y of the input video.

The main function of the input pixel buffer is to handle the flow control and to to generate the column and row addresses into the tile buffer RAM. The circuit also detects whether the current pixel lies within the graphics window defined by the parameters *gpu_top_x*, *gpu_top_y*, *gpu_bot_x* and *gpu_bot_y*. If the current pixel lies outside the graphics window, the input pixel passes though unchanged. If the pixel lies inside the graphics window, then the pixel is processed in the graphics overlay pipeline.

As well as the graphics window, the user may also specify a clip-box region. The clip-box is defined by the generic parameters *clp_top_x*, *clp_top_y*, *clp_bot_x* and *clp_bot_y*. Only the areas of the graphics window that lie within the clip-box boundaries will be displayed. Use of the clip-box gives an extra level of control, permitting the user to dynamically bring various areas of the graphics window into and out of view.

One final feature of the address generator is the implementation of a vertical or horizontal shift of the graphics in the graphics window region. The desired shift in pixels is specified in the *gpu_shift_x* and *gpu_shift_y* parameters. Applying a shift is useful for scrolling graphics and moving banner displays.

All address generator parameters may be updated in real time. If these parameters are not static, then it is desirable that they be updated simultaneously and once per frame in order to avoid interruption in the output video. Figure 2 shows the relationship between the input video display area, the graphics window region and the clip-box.

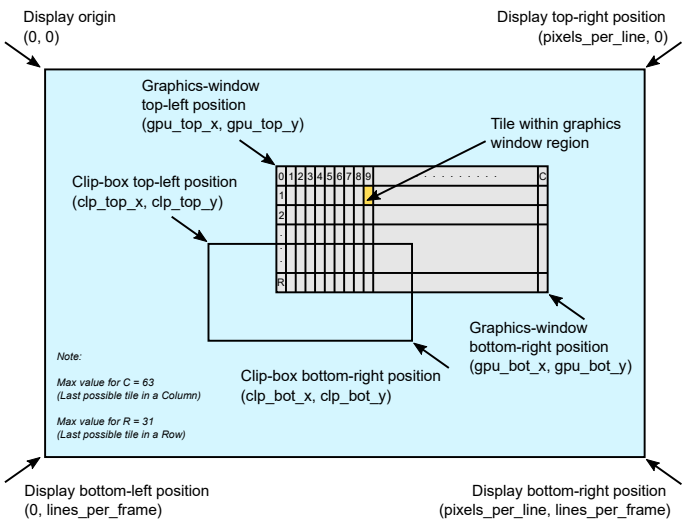

Figure 2: Relationship between display, graphics window and clip-box region

By modifying the graphics-window position, a similar effect to a 2D bit-blt operation is achievable. This is useful for the dynamic movement and simple animation such as 'sprites', pointers and moving banners. Numerous graphics overlay modules may be cascaded together in series for more complex animated effects.

Tile buffer

The tile buffer is a dual-port RAM organized as 64 columns x 32 rows of 8-bit values (Figure 3). Each 8-bit value in the buffer is a tile number that uniquely addresses a tile in the tile bitmap ROM. Note that the MSB of the tile value is a 'highlight' bit that causes the background colour of the tile to be swapped to a highlight colour.

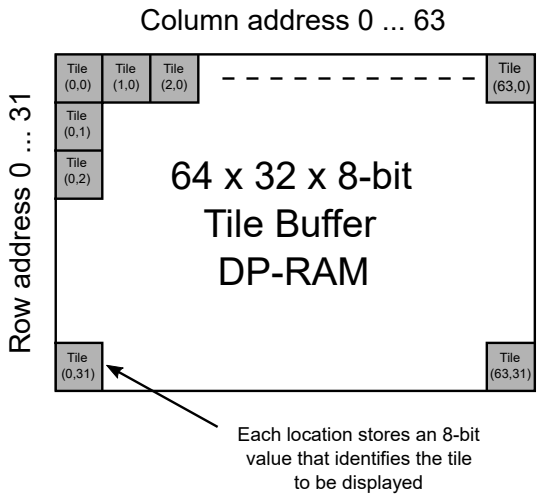

Figure 3: Tile buffer layout

The tile numbers in the tile buffer are uniquely programmable and may be updated as and when required. A value is written to the buffer on the rising-edge of *clk* when *tile_en* and *tile_we* are both high. Likewise, a value is read from the buffer on the rising-edge of *clk* when *tile_en* is high and *tile_we* is low. A write has a latency of 2 clock cycles and a read 3 cycles. By updating the buffer dynamically on a frame-by-frame basis with different tile values, it is possible to achieve 2D animated effects.

Each tile in the buffer maps to a bitmap stored in the ROM. Depending on the parameter *tile_size*, the tile number in the buffer will map to a different size tile: either 8x16, 16x32, 32x64 or 64x128 pixels. Referring to Figure 3, the tiles are logically arranged with tile(0,0) positioned in the top-left corner of the graphics window.

Note that if the graphics window is smaller than than the space required to display all the tiles in the buffer, then then resulting graphics overlay will be clipped to fit within the window region. Conversely, if the graphics window is larger than the tiles in the tile buffer then tiles may wrap-around in the display area.

Tile bitmap ROM

The bitmap ROM contains the bitmap images for the tiles to be displayed. The bitmaps are defined in four separate ROM files corresponding to the four available tile sizes: 8x16, 16x32, 32x64 and 64x128. The source files for these images are called: *gpu_rom_8x16.vhd*, *gpu_rom_16x32.vhd, gpu_rom_32x64.vhd* and *gpu_rom_64x128.vhd*.

By default, the ROM is left undefined and it is left to the user to define the contents of each tile. Up to 128 unique tiles may be defined with each tile being split over three bit-planes. The individual bitmaps for each tile may be coded manually but the simplest method is to use a third-party illustration or vector-drawing application with the ability to export drawings as bitmaps[2].

Figure 4 gives an example bitmap image of a series of rectangles in an 8x16 tile. By combining the bits over three bit-planes, a value between 0 and 7 is generated for each pixel. Depending on the graphics mode, the pixel will be encoded as either an alpha value or one of 8 possible colours from the palette.

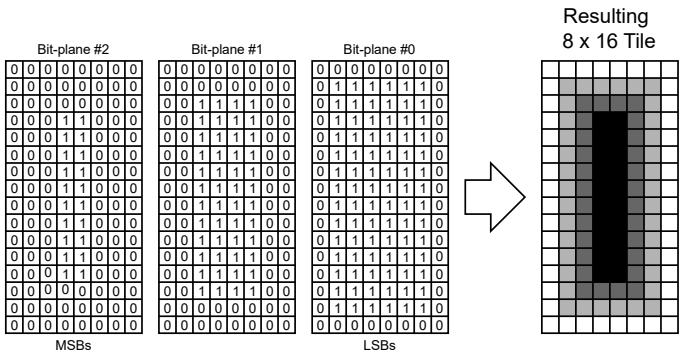

Figure 4: 8x16 tile bitmap as it´s stored in ROM

In the example, the central rectangle is encoded as “111”, the next rectangle as “011” and the outer rectangle as “001”. This is represented by the three different shades of grey in the resulting image.

2 Zipcores can supply scripts to parse bitmap images and generate a VHDL file suitable for the ROM. Please contact us for further details.

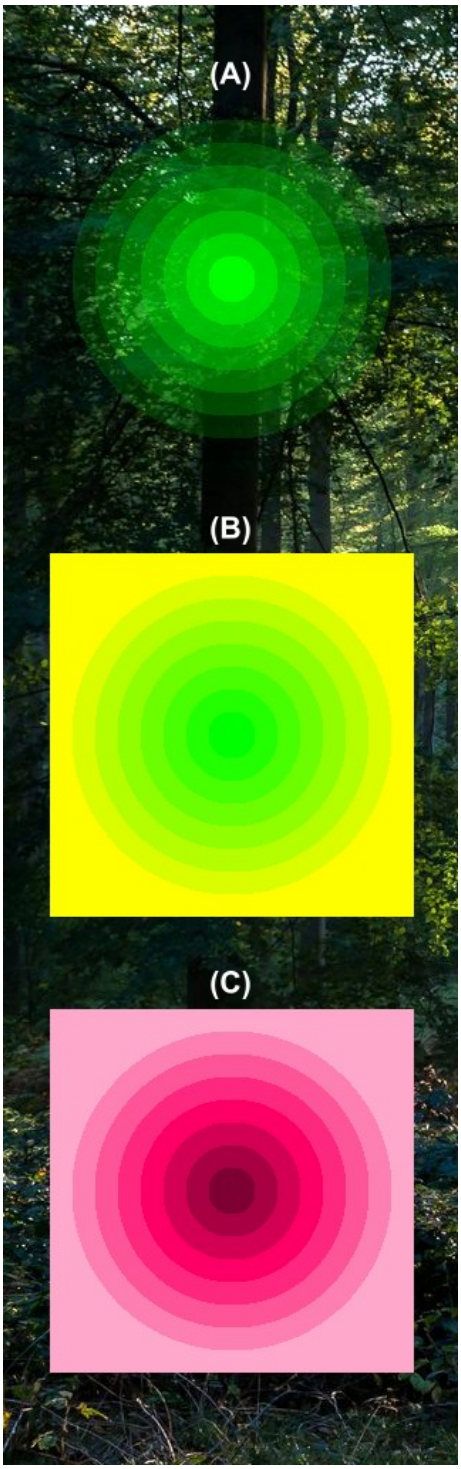

Figure 5: Pixel blending examples: A, B, C

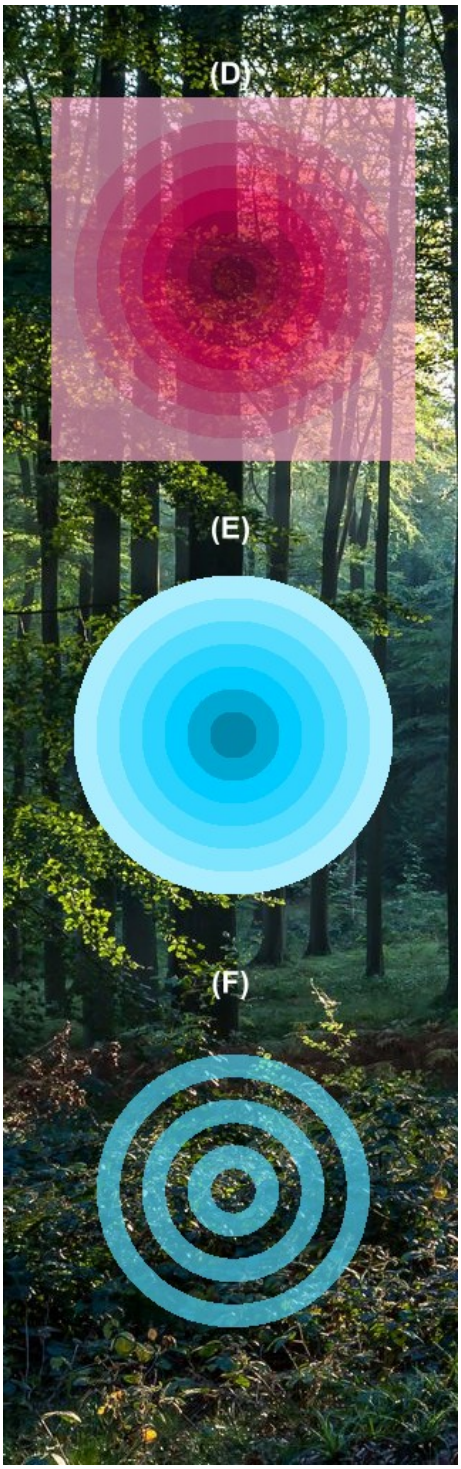

Figure 6: Pixel blending examples: D, E, F

Pixel blending unit

The pixel blending unit generates the output pixel to be displayed. The appearance of the pixel depends on whether the pixel lies inside the graphics-window and the chosen graphics mode. Figures 5 and 6 demonstrate some of the pixel blending effects that may be achieved.

Example A: gpu_mode = 0

When the parameter *gpu_mode* is set to '0' then the 3-bits/pixel from the bitmap ROM is decoded as an alpha value with the pixel colour defined in the parameter *gpu_colour0*. If the pixel value is “000” then this denotes a fully transparent pixel. If the pixel value is “111” then this denotes a fully opaque pixel. Values between “000” and “111” represent varying degrees transparency between the two extremes. Choosing this graphics mode allows mono-colour graphics to be displayed with smooth 'anti-aliased' edges. In this particular case *gpu_colour0* has been set to green or 0x00FF00 in hex.

Example B: gpu_mode = 0 (with highlighting)

This example is the same as example A with the exception that the top bit of the graphics tile value has been set to ‘1’. This causes the background highlight colour to be set as defined in *gpu_colour1*. In this example *gpu_colour0* is set to 0x00FF00 (green) and *gpu_colour1* is set to 0xFFFF00 (yellow).

Example C: gpu_mode = 1

When the parameter *gpu_mode* is set to '1' then the 3-bits/pixel from the bitmap ROM is decoded as a unique colour from the palette of 8 possible colours. The colour palette is defined in the parameters *gpu_colour0* to *gpu_colour7*. For example, if the pixel value is “000” then the pixel will be displayed as *gpu_colour0*, if the pixel has the value “001” then the pixel will be displayed as *gpu_colour1* etc. In this case, the colour palette has been set to varying degrees of pink.

Example D: gpu_mode = 1, gpu_alpha = 0xAA

This example is the same as example C but with the transparency (alpha channel) set to 66%. This is done by setting the parameter *gpu_alpha* to 0xAA in hex. Note that *gpu_alpha* defines an 8-bit alpha channel for the whole bitmap overlay.

Example E: gpu_mode = 2

When *gpu_mode* is set to ‘2’ then the blending operation is the same as *gpu_mode* = ‘1’ with the exception that *gpu_colour0* is decoded as totally transparent. This mode is useful if the user wants to 'key' the background video in the middle of a bitmap object. In this example, the square that encloses the concentric circles has been encoded as “000” in the bitmap causing it to be transparent.

Example F: gpu_mode = 2, gpu_alpha = 0xAA

In this final example, *gpu_mode* is set to ‘2’ as per example E. However, the concentric circles in the bitmap have been set to “000”, “111”, “000”, 111”, … etc. This results in the consecutive transparent rings. In addition, the overall alpha channel has been set to 66%.

Functional Timing

The internal tile buffer has an independent read/write interface. Tile numbers are written to the buffer on a rising clock edge when *tile_en* and *tile_we* are both high. A read occurs when *tile_en* is high and *tile_we* is low. Each tile number in the 64 x 32 array is uniquely addressable with separate column and row addresses.

Figure 7 demonstrates a sequence of write and read operations to the tile buffer. A write has a latency of two clock cycles before the buffer is updated. A read has a latency of three clocks.

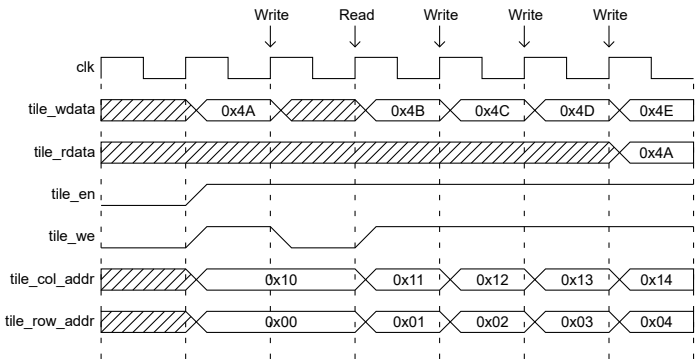

Figure 7: Writing and reading the tile buffer

RGB pixels are sampled according to the valid-ready streaming protocol[3]. Figure 8 shows the signalling at the input of the GPU overlay module at the start of a new frame. The first line of a new frame begins with *pixin_vsync* and *pixin_hsync* asserted high together with the first pixel.

It is important to note that input pixels and syncs are only sampled on a rising clock-edge when *pixin_val* and *pixin_rdy* are both high. If this protocol is not observed, then pixels and syncs will be lost and the resulting output video will be corrupted. As an example, the diagram shows what happens when *pixin_rdy* is de-asserted. In this case, the pipeline is stalled and the upstream interface must hold-off before further pixels are sampled.

3 See application note: app_note_zc001.pdf on the Zipcores website for more examples of the valid-ready streaming protocol

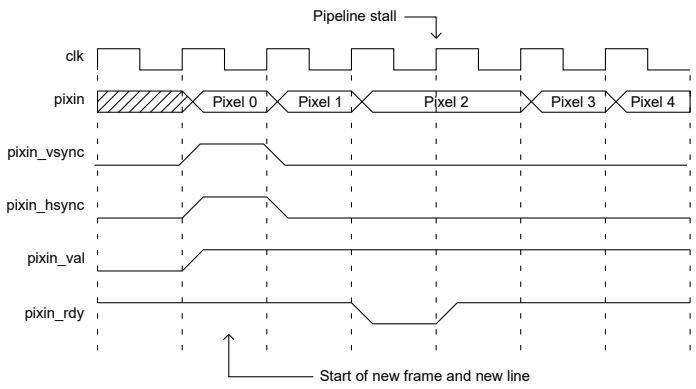

Figure 8: First line of a new frame

Figure 9 shows the signalling at the output of the GPU overlay module. The output uses exactly the same protocol as the input. Each new output line begins with *pixout_hsync* and *pixout_val* asserted high. In this particular example, it shows *pixout_val* de-asserted for 1 clock-cycle, in which case, the output pixel should be ignored. Transfers at the valid-ready interface are only permitted when valid and ready are both high.

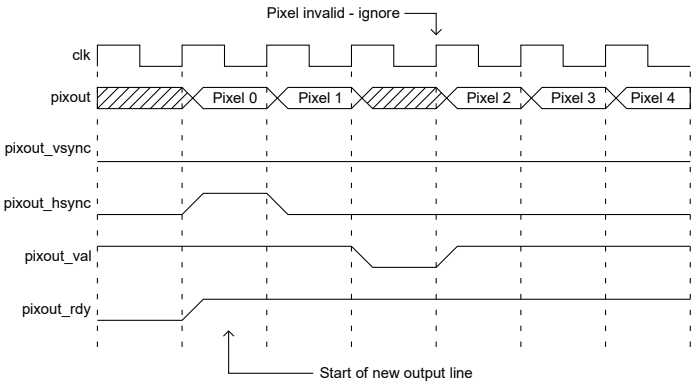

Figure 9: Graphics overlay output showing invalid pixel

The graphics-window coordinates and the user-defined alpha channel are sampled on the rising edge of the system clock. In the following clock cycle, these values will be active and be ready for use in the GPU overlay module. It is recommended these dynamic parameters are updated once per frame (e.g. during vertical blanking) in order to avoid flickering or corrupted video.

As a final point, it's worth noting that if the downstream interface can always accept pixels then the *pixout_rdy* signal may be tied high. In this situation the valid-ready flow control may be ignored. With *pixout_rdy* tied high, the GPU overlay module has a fixed latency from input to output and the *pixin_val* and *pixout_val* signals may be used as a simple clock-enables.

Source File Description

All source files are provided as text files coded in VHDL. The following table gives a brief description of each file.

Source file	*Description*
video_in.txt	Source video text file
tile0_in.txt	Tile buffer text file (GPU #0)
tile1_in.txt	Tile buffer text file (GPU #1)
video_file_reader.vhd	Source video file reader
tile_file_reader.vhd	Tile buffer input file reader
pipeline_reg.vhd	Pipeline register component
fifo_sync.vhd	Synchronous FIFO component
gpu_tile_buffer.vhd	Tile buffer DP-RAM
gpu_input_buffer.vhd	Input pixel buffer
gpu_blend.vhd	Pixel blending unit
gpu_rom_8x16.vhd	Small tile-size bitmap ROM image
gpu_rom_16x32.vhd	Medium tile-size bitmap ROM image
gpu_rom_32x64.vhd	Large tile-size bitmap ROM image
gpu_rom_64x128.vhd	Xlarge tile-size bitmap ROM image
gpu_overlay.vhd	Graphics overlay top-level component
gpu_overlay_bench.vhd	Top-level test bench

Functional Testing

An example testbench is provided for use in a suitable VHDL simulator. The compilation order of the source code is as follows:

1. video_file_reader.vhd
2. tile_file_reader.vhd
3. pipeline_reg.vhd
4. fifo_sync.vhd
5. gpu_tile_buffer.vhd
6. gpu_input_buffer.vhd
7. gpu_blend.vhd
8. gpu_rom_8x16.vhd
9. gpu_rom_16x32.vhd
10. gpu_rom_32x64.vhd
11. gpu_rom_64x128.vhd
12. gpu_overlay.vhd
13. gpu_overlay_bench.vhd

The testbench instantiates two GPU overlay IP Cores in series to generate a multi-coloured text and graphics display. In the example provided, a custom 1360x1028 image is used as the source video with a *tile_size* set to '1' (medium) in the first GPU overlay and a *tile_size* of '3' (xlarge) in the second. The *gpu_mode* has been set to '0' in both instances in order to generate smooth anti-aliased text and graphics.

Figure 10, on the following page shows the resulting output from the testbench example.

Figure 10: Output video frame from the testbench example

The tile numbers to be written to the tile buffer are stored in the files *tile0_in.txt* and *tile1_in.txt*. These files should be placed in the top-level simulation directory. Each line of this text file defines the state of the *tile_en*, *tile_we*, *tile_wdata*, *tile_col_addr* and *tile_col_row* signals on a clock-by-clock basis.

For example the line: '1 1 2B 02 04' will write tile number 0x2B to column 2 row 4 of the buffer.

The source video for the simulation is generated by the video file-reader component. As with the tile buffer, this component requires a text file to placed in the top-level simulation directory. The file is called *video_in.txt* and it contains the source pixels and syncs for the test.

The file *video_in.txt* follows a simple format which defines the state of signals: *pixin_val*, *pixin_vsync*, *pixin_hsync* and *pixin* on a clock-by-clock basis. An example file might be the following:

```
1  1  1  00  11  22  # pixel 0, line 0 (start of frame)
1  0  0  33  44  27  # pixel 1
1  0  0  66  77  88  # pixel 2
1  0  0  14  5D  6F # pixel 3
.
.

1  0  1  00 11 22 # pixel 0, line 1 etc..
```

In this example, the first line of of the *video_in.txt* file asserts the input signals pixin_val = 1, pixin_vsync = 1, pixin_hsync = 1 and pixin = 0x001122.

The simulation must be run for at least 20 ms during which time an output text file called *video_out.txt* will be generated. This file contains a sequential list of 24-bit output pixels in the same format as *video_in.txt*[4].

4 Scripts are provided as part of the IP Core package to help with the generation and processing of input and output files. Please contact Zipcores for more information.

Synthesis and Implementation

The files required for synthesis and the design hierarchy is shown below:

- gpu_overlay.vhd
  - gpu_input_buffer.vhd
  - gpu_tile_buffer.vhd
  - gpu_rom_8x16.vhd
  - gpu_rom_16x32.vhd
  - gpu_rom_32x64.vhd
  - gpu_rom_64x128.vhd
  - gpu_blend.vhd
  - fifo_sync.vhd
    - pipeline_reg.vhd

The IP Core is designed to be technology independent. However, as a benchmark, synthesis results have been provided for the Xilinx® 7- series FPGAs. Synthesis results for other FPGAs and technologies can be provided on request.

Note that the generic parameter *tile_size* will effect the number of embedded Block RAM components in the design. The largest tile size of 64x128 pixels will result in the greatest utilization of RAM and may not be suitable for very small FPGAs or SoCs.

If the application does not require alpha blending support, then the parameter *use_alpha* may be set to false. The result will be a saving on embedded multiplier components. If full data-streaming flow control is not needed, then the signal *pixout_rdy* may be tied to logic '1'. In this case, the output pixels from the IP Core must always be accepted by the downstream interface. Fixing *pixout_rdy* to logic '1' will result in a slightly speed-optimized design.

Trial synthesis results are shown in the following table. The design was synthesized with the generic parameters set as follows: tile_size = 1 (medium), use_alpha = true, gpu_mode = 0, pixels_per_line = 1920, lines_per_frame = 1080. Resource usage is specified after Place and Route.

XILINX® 7-SERIES FPGAS

Resource type	***Artix-7***	***Kintex-7***	***Virtex-7***
Slice Register	385	385	385
Slice LUTs	527	529	527
Block RAM	7	7	7
DSP48	0	0	0
Occupied Slices	228	214	218
Clock freq. (approx)	250 MHz	300 MHz	350 MHz

Revision History

Revision	***Change description***	***Date***
1.0	Initial revision	21/01/2019
1.1	Added tile highlighting functionality	29/09/2021