Key Design Features

- Synthesizable, technology independent VHDL Core
- 2D bitmap overlays over real-time video
- 24-RGB pixels supported as standard - other formats (e.g. YCbCr 4:2:2) supported on request
- Supports 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: 8x8, 16x16, 32x32 or 64x64
- Tiles organized into 3 bit-planes offering 3-bits/pixel
- Tile numbers written to a buffer that map directly to the display
- Programmable graphics-window position and size
- Programmable clip-box (clip-plane) 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 overlay cores in series for more complex graphical displays
- Optional I2C, SPI or UART interfaces for simple micro-processor programming

Applications

- Professional and functional 2D graphic displays and video overlays
- Digital TV and home-media solutions
- Interactive guides, menus, tables, lists etc.
- Animated 2D graphics including hardware sprites, mouse pointers, cursors , parallax scrolling, moving banners etc.
- Window movement in a similar manner to a 2D 'BitBlt'
- Instrumentation and monitoring applications including animated gauges, charts, dials, meters, counters etc.
- Informational displays and simple HUDs for commercial, military and automotive applications

Block Diagram

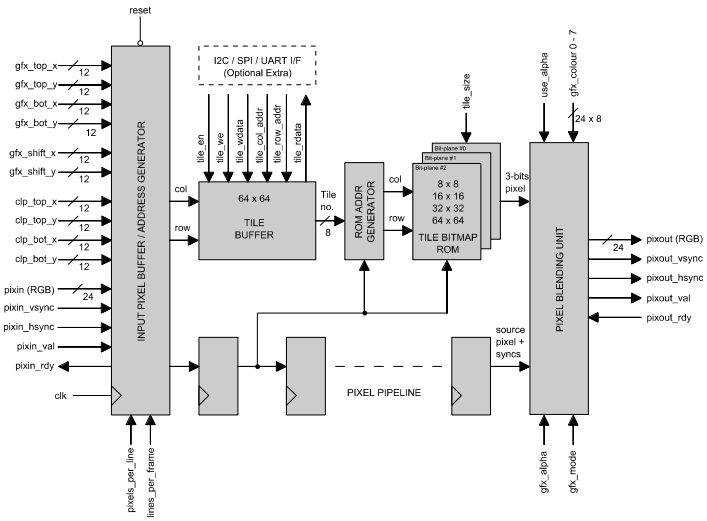

Figure 1: 2D Graphics overlay module architecture

General Description

GFX_OVERLAY is a highly versatile on-screen display that allows high-quality anti-aliased bitmap graphics 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 addressed by means of an 8-bit value stored in a 64x64 tile buffer. There are four tile sizes available - either 8x8, 16x16, 32x32 or 64x64.

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 stored in a user-defined ROM which can contain up to 256 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.

Pixels and syncs flow in and out of the overlay module in accordance with the valid-ready pipeline protocol. 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* is high and *pixout_rdy* is high. The pipeline protocol allows both input and output interfaces to be stalled independently.

The pipeline protocol is very versatile and permits any number of graphics overlay modules to be cascaded in series. By placing more than one module together, the user is able to achieve more complex 2D effects.

Figure 1 shows the architecture of the graphics overlay module in more detail.

Generic Parameters

Generic name	Description	Type	Valid Range
tile_size	Tile size selection	integer	0: small (8x8) 1: medium (16x16) 2: large (32x32) 3: xlarge (64x64)
use_alpha	Enable/disable alpha-blend support	boolean	True / False
gfx_mode	Graphics mode selection	integer	0: 1-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
gfx_alpha [7:0]	in	Alpha transparency of graphics window region	data
gfx_colour0-7 [23:0]	in	User defined palette of 8x24-bit colours	data
gfx_top_x [11:0]	in	Top-left x position of graphics-window	data
gfx_top_y [11:0]	in	Top-left y-position of graphics-window	data
gfx_bot_x [11:0]	in	Bottom-right x position of graphics-window	data
gfx_bot_y [11:0]	in	Bottom-right y position of graphics-window	data
gfx_shift_x[11:0]	in	Horizontal shift in pixels	data
gfx_shift_y[11:0]	in	Vertical shift in pixels	data
clp_top_x [11:0]	in	Top-left x position of clipbox	data
clp_top_y [11:0]	in	Top-left y-position of clipbox	data
clp_bot_x [11:0]	in	Bottom-right x position of clipbox	data
clp_bot_y [11:0]	in	Bottom-right y position of clipbox	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 [5:0]	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 valid-ready 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 *gfx_top_x*, *gfx_top_y*, *gfx_bot_x* and *gfx_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 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 *gfx_shift_x* and *gfx_shift_y* parameters. Applying a shift is useful for scrolling graphics (e.g. parallax scrolling) and moving banner displays.

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

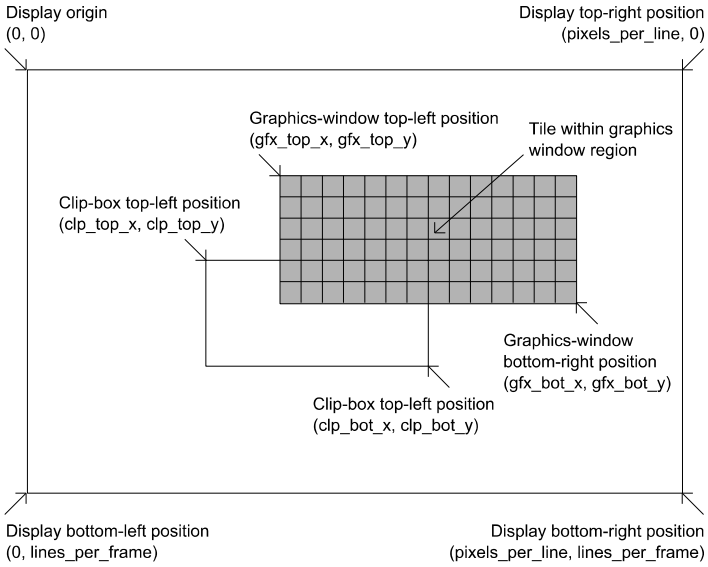

Figure 2: Graphics window and clip-box positioning and size

By modifying the graphics-window position, a similar effect to a 2D bit-blt operation is achievable. This is useful for the simple animation of 'sprites', pointers or the implementation of simple screen savers. 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 64 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.

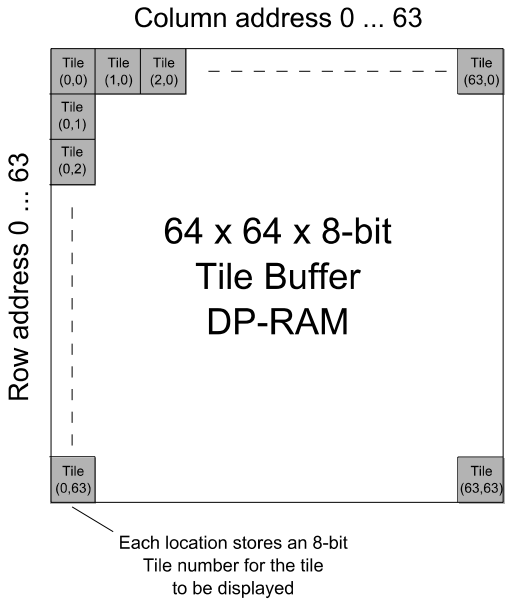

Figure 3: Tile buffer layout

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 numbers, 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 8x8, 16x16, 32x32 or 64x64 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 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.

The tile numbers in the buffer are subsequently passed to the ROM address generator circuit which looks up the current pixel in the bitmap ROM.

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: 8x8, 16x16, 32x32 and 64x64. The source files for these images are called: *gfx_rom_8x8.vhd*, *gfx_rom_16x16.vhd, gfx_rom_32x32.vhd* and *gfx_rom_64x64.vhd*.

By default, the ROM is left undefined and it is left to the user to define the contents of each tile. Up to 256 unique tiles may be defined - 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[1].

Figure 4 gives an example bitmap image of a series of concentric squares in an 8x8 tile. By combining the bits over three bit-planes, a value between 0x0 and 0x7 is generated for each pixel. Depending on the graphics mode, the pixel will be decoded as either an alpha value or one of 8 possible colours from the palette.

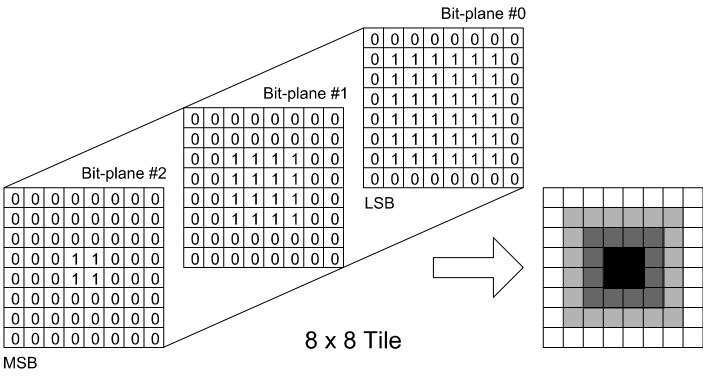

Figure 4: 8x8 tile bitmap as it´s stored in ROM

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

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

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. Figure 5 demonstrates the action of the blender graphically.

Figure 5: Pixel blending operations

When the parameter *gfx_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 *gfx_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. This is demonstrated in the first image of Figure 5.

When the parameter *gfx_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 *gfx_colour0* to *gfx_colour7*. For example, if the pixel value is “000” then the pixel will be displayed as gfx_colour0, if the pixel has the value “001” then the pixel will be displayed as *gfx_colour1* etc. In addition, when *gfx_mode* is set to '1', the parameter *gfx_aplha* defines an 8-bit alpha channel for the whole bitmap overlay. The second image in Figure 5 demonstrates this mode of operation in more detail. Each colour is a different shade of pink in the concentric circle pattern. An alpha value of 0xFF has been chosen to give a fully opaque bitmap overlay.

The next example image in Figure 5 shows the same series of circles, but this time, the alpha value has been set to 0xAA. This results in an overlay with 66% transparency. Dynamically modifying the transparency can be used to fade-in and fade-out graphics in the video display area. (Note that alpha blending is only supported with the generic parameter *use_alpha* set to *true*. When set to *false*, the alpha blending hardware is not generated).

The final example of Figure 5 shows the blending operation when *gfx_mode* is set to '2'. This mode is identical to mode '1' with the exception that *gfx_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.

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 64 array is uniquely addressable with separate column and row addresses. Figure 6 demonstrates a sequence of write and read operations. A write has a latency of two clock cycles before the buffer is updated. A read has a latency of three clocks.

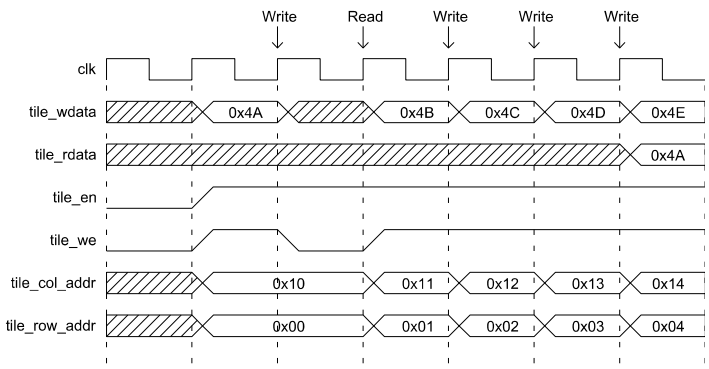

Figure 6: Writing and reading the tile buffer

RGB pixels are sampled according to the valid-ready pipeline protocol[2]. Figure 7 shows the signalling at the input of the graphics 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.

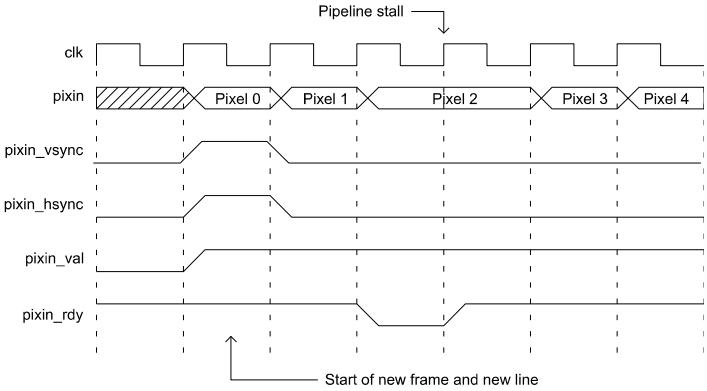

Figure 7: First line of a new frame

Figure 8 shows the signalling at the output of the graphics 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. Remember that transfers at a valid-ready interface are only permitted when valid and ready are both simultaneously high.

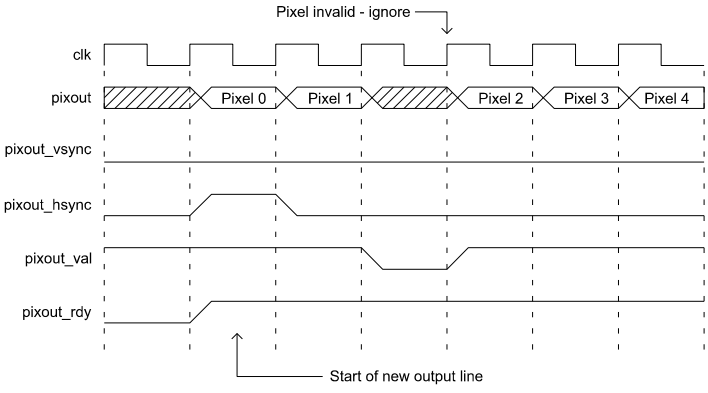

Figure 8: Graphics overlay output showing invalid pixel

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

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 graphics overlay module. It is the responsibility of the user to ensure that these parameters are updated once per frame in order to avoid the possibility of corrupted output video.

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
tile_in.txt	Tile buffer text file
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
gfx_tile_buffer.vhd	Tile buffer DP-RAM
gfx_input_buffer.vhd	Input pixel buffer
gfx_blend.vhd	Pixel blending unit
gfx_rom_8x8.vhd	Small tile-size bitmap ROM image
gfx_rom_16x16.vhd	Medium tile-size bitmap ROM image
gfx_rom_32x32.vhd	Large tile-size bitmap ROM image
gfx_rom_64x64.vhd	XLarge tile-size bitmap ROM image
gfx_overlay.vhd	Graphics overlay top-level component
gfx_overlay_bench.vhd	Top-level test bench

Functional Testing

An example VHDL 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. gfx_tile_buffer.vhd
6. gfx_input_buffer.vhd
7. gfx_blend.vhd
8. gfx_rom_8x8.vhd
9. gfx_rom_16x16.vhd
10. gfx_rom_32x32.vhd
11. gfx_rom_64x64.vhd
12. gfx_overlay.vhd
13. gfx_overlay_bench.vhd

The VHDL testbench instantiates the GFX_OVERLAY component and the user may modify the generic parameters as required. In the example testbench provided, a SVGA(1024x768) image is used as the source video and a simple checker board pattern is configured to appear in the top left of the display.

The generic parameter *gfx_mode* has been set to '1' and *tile_size* set to '2'. Figure 9, below shows the resulting video output from the testbench example.

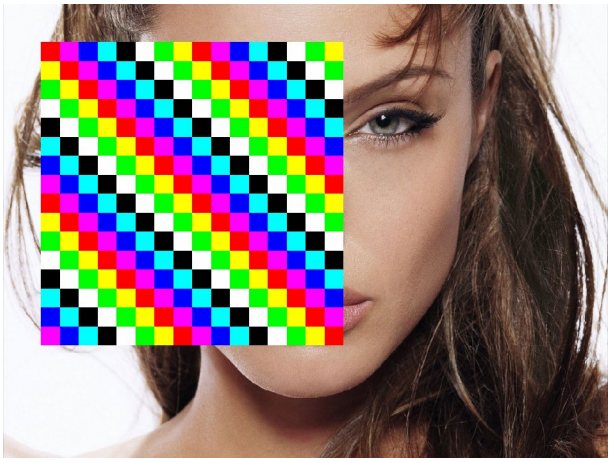

Figure 9: Output video from testbench example

The tile numbers to be written to the tile buffer are stored in the file *tile_in.txt*. This file 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 55  # pixel 1
0  0  0  00 00 00  # don't care!
1  0  0  66 77 88  # pixel 2
.
.
1  0  1  00 11 22 # pixel 0 line 1etc..
```

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*.

Example 2D-Graphics Overlay Outputs

Figure 10 is shows a 'target' example with alpha blending disabled. The overlay is inserted over RGB video at 512x512 resolution. The parameter *gfx_mode* is set to '1' meaning that all 8 colours in the palette are used. The palette has been programmed with varying shades of pink. The parameter *gfx_alpha* is set to 0xFF making the overlay fully opaque.

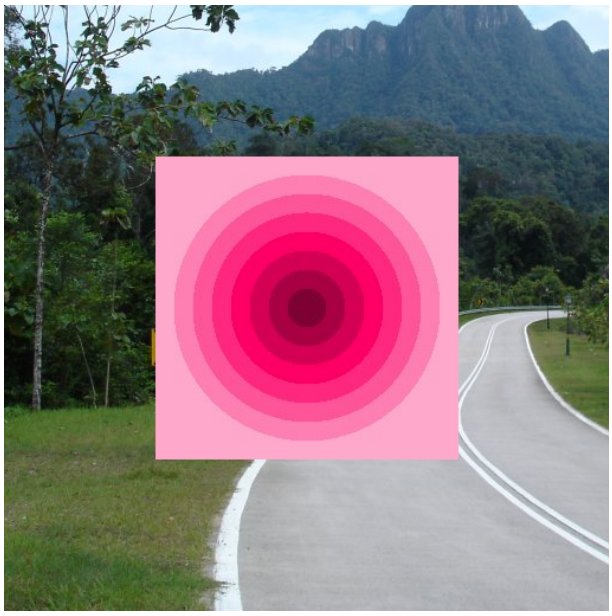

Figure 10: Pink 'target' with full use of colour palette

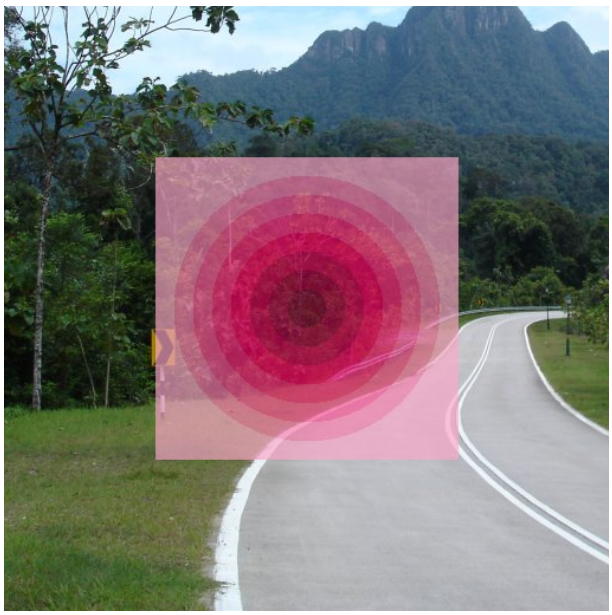

Figure 11: Pink 'target' with 66% alpha channel

Figure 11 demonstrates the same pink 'target' example, but this time the parameter *gfx_alpha* is set to 0xAA. This enables 66% alpha transparency for the the whole graphics overlay.

Figure 12 is an example of a more complex graphics overlay. In this particular case, *gfx_mode* has been set to '0', enabling mono-colour graphics with per-pixel alpha blending. The result is a smooth anti-aliased image avoiding jagged edges in the curves and lines.

In this particular example, a number of graphics overlay[3] modules have been cascaded in series to generate the different coloured components of the image. For the dial animation, different tiles are defined for the different dial positions. By dynamically updating the tile numbers in the buffer, the dials can be animated seamlessly on a frame-by-frame basis.

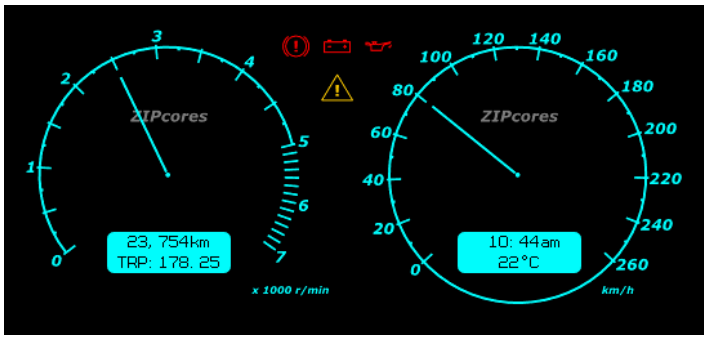

Figure 12: Animated 2D-graphics overlay

In Figure 12, the graphics were inserted over a black background. Figure 13 demonstrates the same overlay (with modified colours) over an RGB video source.

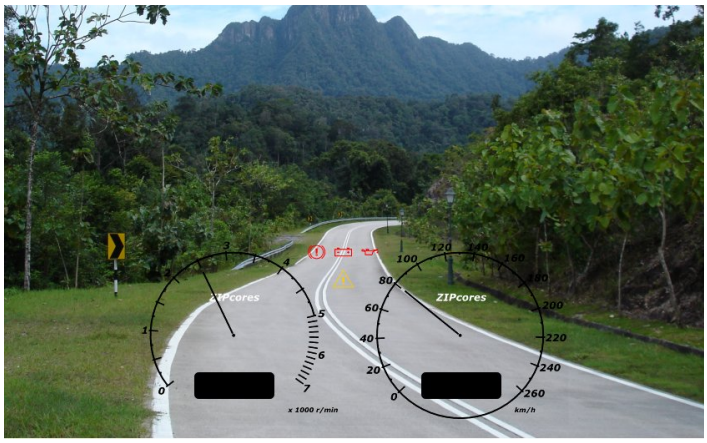

Figure 13: Animated dashboard overlay

Synthesis

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

- gfx_overlay.vhd
  - gfx_input_buffer.vhd
  - gfx_tile_buffer.vhd
  - gfx_rom_8x8.vhd
  - gfx_rom_16x16.vhd
  - gfx_rom_32x32.vhd
  - gfx_rom_64x64.vhd
  - gfx_blend.vhd
  - fifo_sync.vhd
    - pipeline_reg.vhd

The VHDL core is designed to be technology independent. However, as a benchmark, synthesis results have been provided for the Xilinx Virtex 5 and the Altera Stratix III series of FPGA devices. The lowest and highest speed grade devices have been chosen in both cases for comparison.

Note that the generic parameter *tile_size* will effect the number of embedded Block RAM components in the design. The largest size of 64x64 pixels will result in the greatest utilization of RAM. For further block RAM savings, then the character buffer can be made write only if necessary.

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.

Trial synthesis results are shown in the following tables. The design was synthesized with the generic parameters set as follows: tile_size = 1, use_alpha = true, gfx_mode = 0, pixels_per_line = 1024, lines_per_frame = 1024.

Resource usage is specified after Place and Route.

VIRTEX 5

Resource type	***Quantity used***
Slice register	405
Slice LUT	330
Block RAM	7
DSP48	6
Clock frequency (worst case)	205 MHz
Clock frequency (best case)	252 MHz

STRATIX III

Resource type	***Quantity used***
Register	549
ALUT	334
Block Memory bit	57552
DSP block 18	6
Clock frequency (worse case)	280 MHz
Clock frequency (best case)	322 MHz

3 The text overlay IP core was used for the text in the bottom panels. Please see the Zipcores website for more details.