Key Design Features

- Technology independent soft IP Core for FPGA, ASIC and SoC
- Supplied as human readable VHDL (or Verilog) source code
- Fully pipelined architecture with input/output flow-control and AMBA AXI4-stream compatible data interfaces
- Text overlays/On Screen Displays (OSDs) over RGB video
- Support for RGB pixels in/out (YCbCr formats may be provided on request)
- Real-time programmable character buffer maps instantaneously to the display
- No external memory or frame buffer required
- Supports all video resolutions up to 4096 x 4096 pixels
- Programmable text-box position and size
- Programmable window clipping region
- Independent horizontal and vertical scrolling
- Programmable foreground and background colours
- Programmable 8-bit alpha transparency
- Four sizes of text: 8x16, 16x32, 32x64 or 64x128
- Normal or highlighted text
- Used-defined ROM supports different fonts, characters and custom bitmap graphics
- Ships with four pre-defined character sets
- No complex programming required
- Optional I2C, SPI or UART interfaces for simple programming via micro-processor or micro-controller
- Support for 200 MHz+ operation on basic FPGA devices[1] and 500 MHz+ on Ultrascale

Example Applications

- Digital TV and home-media solutions including: interactive guides, menus, tables, lists, video games etc.
- Terminal and console windows
- Animated text and graphics such as hardware ‘sprites’, pointers, cursors, buttons etc.
- Window movement in the same manner as a 2D 'BitBlt'
- Scrolling text and moving wrap-around 'banner' displays
- Instrumentation and monitoring applications including animated gauges, charts, dials, meters, counters

Block Diagram

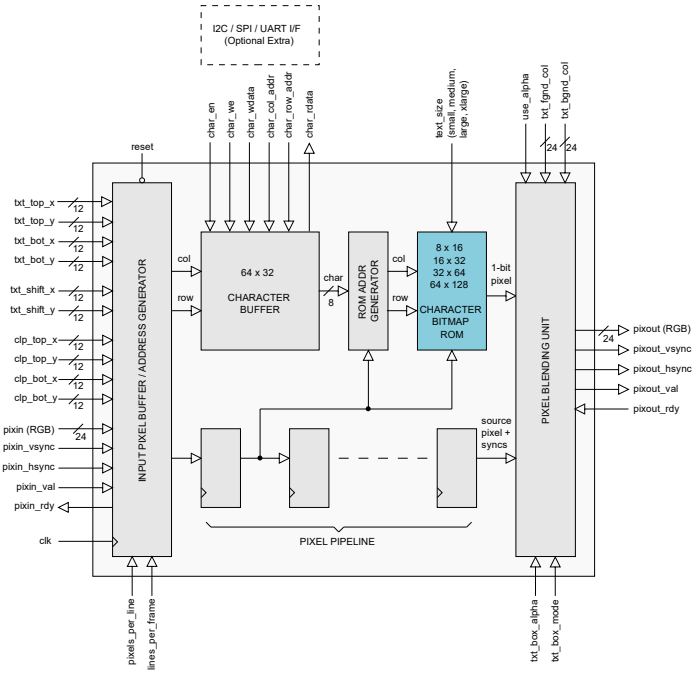

Figure 1: Text overlay module architecture

Generic Parameters

Generic name	Description	Type	Valid Range
text_size	Text 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
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

General Description

The TXT_OVERLAY IP Core is a highly versatile On Screen Display (OSD) module that allows text and bitmap graphics to be inserted over RGB video. The module supports a wide range of text effects and the programming interface is very simple. Text is written to a character buffer which is mapped (via a bitmap ROM) directly to the display.

The characters in the buffer are displayed in a 'text-box' which may be positioned anywhere in the video display area. Bitmaps for each character are stored in a ROM which may be modified to support different font styles or bitmap graphics.

1 Xilinx® 7-series used as a benchmark

Pixels and syncs flow in and out of the overlay module in accordance with the valid-ready pipeline protocol which is compatible with the popular AMBA AXI4-stream used widely in custom ASICs, FPGAs and SoCs[2].

Pixels and syncs are sampled at the module inputs on a rising clock-edge when *pixin_val* and *pixin_rdy* are both 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 if necessary. Any number of text overlay modules may be cascaded in series. By placing more than one overlay module together, the user is able to achieve more complex text and 2D effects.

Pin-out Description

Pin name	I/O	Description	Active state
clk	in	Synchronous clock	rising edge
reset	in	Asynchronous reset	low
txt_fgnd_col [23:0]	in	Text foreground colour as 24-bit RGB	data
txt_bgnd_col [23:0]	in	Text background colour as 24-bit RGB	data
txt_box_alpha [7:0]	in	Alpha transparency of text box region	data
txt_box_mode	in	Text background fill enable 0 = unfilled, 1 = filled	data
txt_top_x [11:0]	in	Top-left x position of textbox	data
txt_top_y [11:0]	in	Top-left y-position of textbox	data
txt_bot_x [11:0]	in	Bottom-right x position of textbox	data
txt_bot_y [11:0]	in	Bottom-right y position of textbox	data
txt_shift_x[11:0]	in	Horizontal shift in pixels	data
txt_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
char_en	in	Character buffer enable	high
char_we	in	Character buffer write enable	high
char_wdata [7:0]	in	Character buffer write data	data
char_col_addr [5:0]	in	Character buffer column address	data
char_row_addr [4:0]	in	Character buffer row address	data
char_rdata [7:0]	out	Character buffer read data	data

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

Pin-out Description cont ...

Pin name	I/O	Description	Active state
pixin [23:0]	in	24-bit RGB source pixel in	data
pixin_vsync	in	Vertical sync in (coincident with the first pixel of an input frame)	high
pixin_hsync	in	Horizontal sync in (coincident with the first pixel of an input 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 (coincident with the first pixel of an output frame)	high
pixout_hsync	out	Horizontal sync out (coincident with the first pixel of an output line)	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 the x and y dimensions 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 character buffer RAM. The circuit also detects whether the current pixel lies within the text-box region defined by the parameters *txt_top_x*, *txt_top_y*, *txt_bot_x* and *txt_bot_y*. If the current pixel lies outside the text-box region, the input pixel passes though unchanged. If the pixel lies inside the text-box, then the pixel is processed in the text overlay pipeline.

As well as the text-box, 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 text-box 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 text-box into view.

One final feature of the address generator is the implementation of a vertical or horizontal shift of the text in the text-box region. The desired shift in pixels is specified in the *txt_shift_x* and *txt_shift_y* parameters. Applying a shift is useful for scrolling text 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 text-box region and the clip-box.

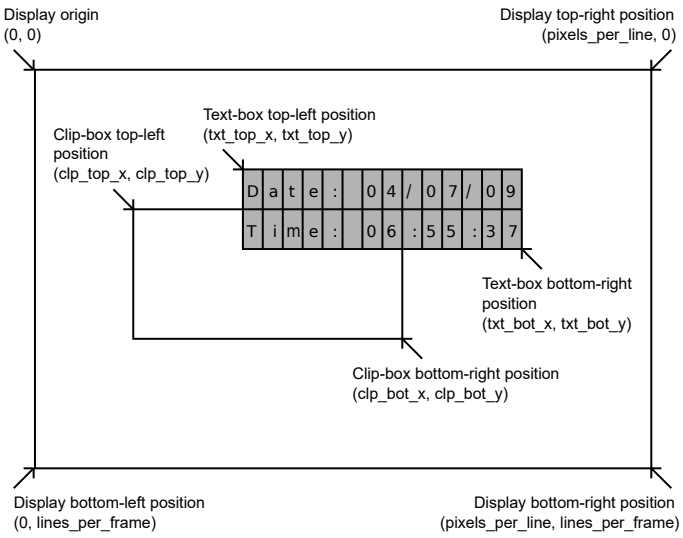

Figure 2: User-defined text-box and clip-box regions

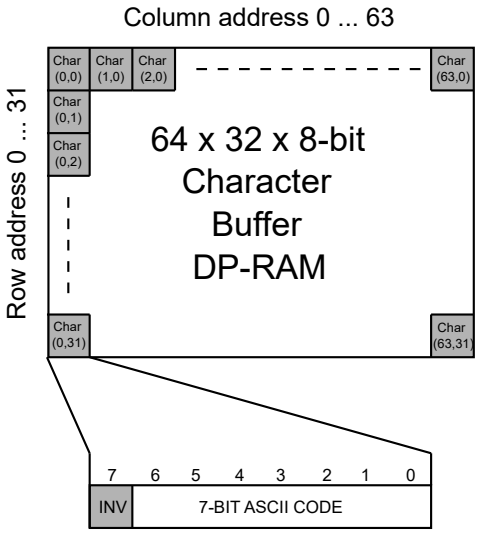

Figure 3: Character buffer layout

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

Numerous text overlay modules may be cascaded together in series for more complex animated effects.

Character buffer

The character buffer is a dual-port RAM organized as 64 columns x 32 rows of 8-bit characters (Figure 3). By default, the values written to the character buffer correspond to a 7-bit standard ASCII code. The MSB of the character value is an 'invert' bit which indicates to the blender unit that the character in that position must be inverted. Setting the invert bit is useful for highlighting text in tables and lists.

Each character in the 64x32 array is uniquely addressable and may be updated as and when required. A character is written to the buffer on the rising-edge of *clk* when *char_en* and *char_we* are both high. Likewise, a character is read from the buffer on the rising-edge of *clk* when *char_en* is high and *char_we* is low. A character write has a latency of 2 clock cycles and a read 3 cycles. By updating the buffer dynamically on a frame-by-frame basis, the animation of text and simple 2D shapes can be achieved.

Each character in the buffer maps to a bitmap stored in the character ROM. Depending on the parameter *text_size*, the character in the buffer will map to a different size bitmap: either 8x16, 16x32, 32x64 or 64x128 pixels.

Referring to Figure 3, the characters are logically arranged with char (0,0) positioned in the top-left corner of the text-box. Note that if the text-box size is smaller than than the space required to display the characters in the buffer, then then resulting text will be clipped to fit the text-box region.

The character codes are subsequently passed to the ROM address generator circuit which looks up the current pixel in the character bitmap ROM. The 'invert' bit bypasses the ROM and goes directly to the blender unit as a text invert control flag.

Character bitmap ROM

The bitmap ROM contains the bitmaps for the standard ASCII character set. By default, the printable characters from ASCII 32 (space) to ASCII 126 (~) are supported. The remaining non-printable characters are left 'blank'.

The bitmaps are defined in four separate ROM images corresponding to the four different text-sizes: 8x16, 16x32, 32x64 and 64x128. The source files for these images are called: *txt_rom_8x16.vhd*, *txt_rom_16x32.vhd, txt_rom_32x64.vhd* and *txt_rom_64x128.vhd*.

The text-overlay module is supplied with four different character sets as standard which are defined in the files 'XXX_rom.txt'. The user is free to modify the bitmap files as required to support different fonts and simple 2D shapes[3].

Figure 4: Standard character sets:
Original, Smallfnt, Fixedsys, Terminal
(from top to bottom)

3 Zipcores can supply scripts to parse bitmap images and generate the VHDL ROM image files. Please contact us for further details

Figure 5 gives an example layout of the capital letter 'A' in the bitmap ROM. By convention a '0' designates an 'active' pixel and a '1' designates an 'inactive' pixel. An active pixel will be visible in the output video display and an inactive pixel will be ignored[4].

The resulting output pixel from the ROM is passed onto the pixel blending unit where it will be blended with the source input pixel.

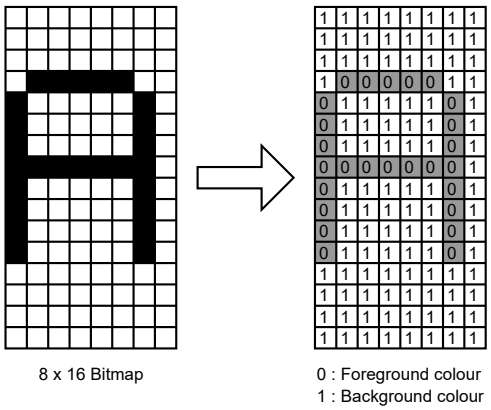

Figure 5: 8x16 Character bitmap as it's stored in ROM

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 text-box region and also the text-box attributes and chosen blending parameters. Figure 6 demonstrates the action of the blender graphically.

The parameters *txt_fgnd_col* and *txt_bgnd_col* specify the colours of the active and inactive pixels in the character bitmap. (Note that the background colour is only relevant when *txt_box_mode* is set to '1'). If the 'invert' bit is set in the respective character code then the state of the active pixels in the bitmap is changed. An active pixel now becomes logic '1' and an inactive pixel logic '0' resulting in an inverted character.

In the above examples, 'mode' refers to the parameter *txt_box_mode*. Setting mode to '0' results in a character without a filled background. When mode is '1' then the inactive pixels in the bitmap are filled with the chosen background colour.

The transparency of the character to be displayed may be modified by varying the 8-bit alpha value *txt_box_alpha*. Setting alpha to 0xFF results in a fully opaque character. Setting alpha to 0x00 makes it fully transparent. Dynamically modifying the transparency can be used to fade-in and fade-out text 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.

Figure 6: Pixel blending operations

Functional Timing

The internal character buffer has an independent read/write interface. Characters are written to the buffer on a rising clock edge when *char_en* and *char_we* are both high. A read occurs when *char_en* is high and *char_we* is low. Each character in the 64x32 array is uniquely addressable with separate column and row addresses.

Figure 7 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 clock cycles.

4 The actual appearance of the bitmap pixel will depend on the chosen blending parameters

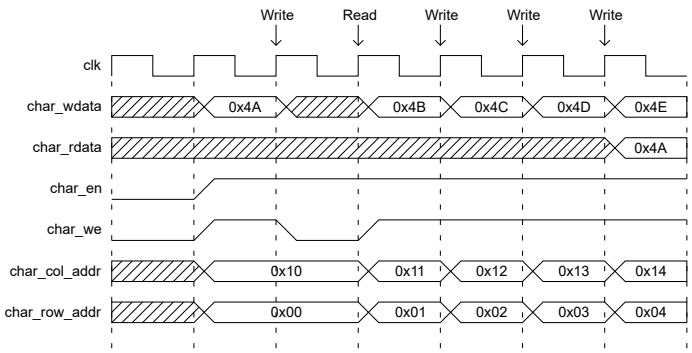

Figure 7: Writing and reading the character buffer

RGB pixels are sampled according to the valid-ready pipeline protocol[5]. Figure 8 shows the signalling at the input of the text 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 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.

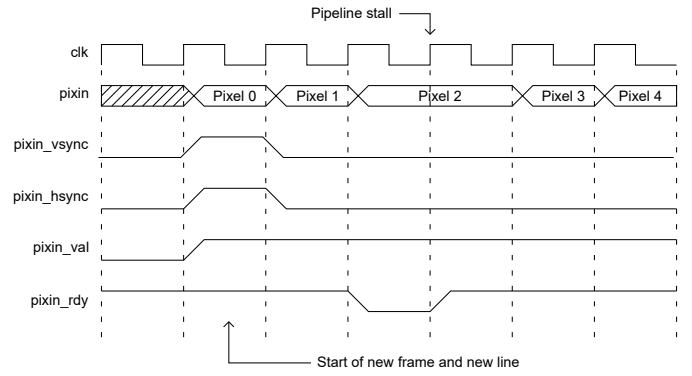

Figure 8: First line of a new frame

Figure 9 shows the signalling at the output of the text 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 a valid-ready interface are only permitted when valid and ready are both simultaneously high.

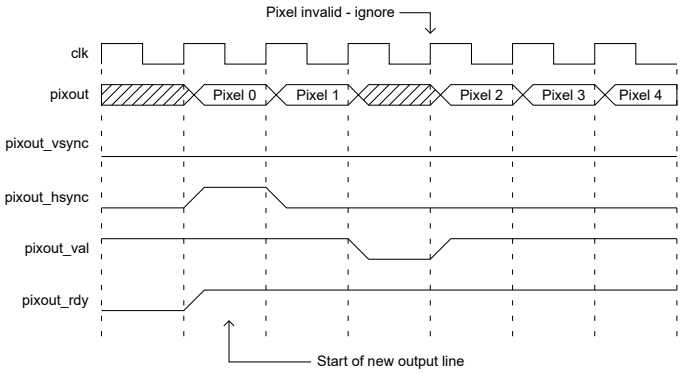

Figure 9: Text overlay output showing invalid pixel

All other parameters used by the text overlay module (including the text and clip-box coordinates and text-box attributes) are sampled on the rising edge of the system clock. In the following clock cycle, these attributes will be active and be ready for use in the text overlay module.

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
char_in.txt	Character buffer text file
video_file_reader.vhd	Source video file reader
char_file_reader.vhd	Character buffer input file reader
pipeline_reg.vhd	Pipeline register component
fifo_sync.vhd	Synchronous FIFO component
txt_char_buffer.vhd	Character buffer DP-RAM
txt_input_buffer.vhd	Input pixel buffer
txt_blend.vhd	Pixel blending unit
txt_rom_8x16.vhd	Small font bitmap ROM image
txt_rom_16x32.vhd	Medium font bitmap ROM image
txt_rom_32x64.vhd	Large font bitmap ROM image
txt_rom_64x128.vhd	Extra large font bitmap ROM image
txt_overlay.vhd	Text overlay top-level component
txt_overlay_bench.vhd	Top-level test bench

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

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. char_file_reader.vhd
3. pipeline_reg.vhd
4. fifo_sync.vhd
5. txt_char_buffer.vhd
6. txt_input_buffer.vhd
7. txt_blend.vhd
8. txt_rom_8x16.vhd
9. txt_rom_16x32.vhd
10. txt_rom_32x64.vhd
11. txt_rom_64x128.vhd
12. txt_overlay.vhd
13. txt_overlay_bench.vhd

The testbench instantiates the text overlay IP Core and the user may modify the generic parameters as required. In the example testbench provided, a 640x480 (VGA) image is used as the source video and a simple text-box is configured to appear in its centre.

The characters to be written to the character buffer are stored in the file *char_in.txt*. This file should be placed in the top-level simulation directory. Each line of this text file defines the state of the *char_en*, *char_we*, *char_wdata*, *char_col_addr* and *char_col_row* signals on a clock-by-clock basis. For example the line:

1 1 2B 02 04

Signifies a write of the value 0x2B ('+' in ASCII) 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 character 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 10 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*.

Figure 10 shows the resulting output video generated from the test provided. The output demonstrates the full 96 printable characters of the standard ASCII character set. A medium font has been chosen with a red foreground colour and text-box fill disabled. The second group of characters have their 'invert' bits set.

Figure 10: Output video from test bench example

Example Text-overlay outputs

Figure 11 is the output of four text-overlay modules cascaded in series, with each module configured with a different text size. The text: 'Date: 04/07/09' is in normal font. The text: 'Time: 16:55:37' is in inverted font. The first 3 text-boxes are configured with *txt_box_mode* set to '1' resulting in a filled text background. The last text-box is configured with *txt_box_mode* set to '0'.

Figure 11: Different sized text

The text overlay module is ideal for the generation of text windows. In this example, a single text overlay module was used to emulate a simple command window (Figure 12). For example, an external microprocessor could manage the character buffer in response to keyboard presses. Window movement and resizing could be mouse-driven and achieved simply by modifying the text-box top-left and bottom right x,y positions.

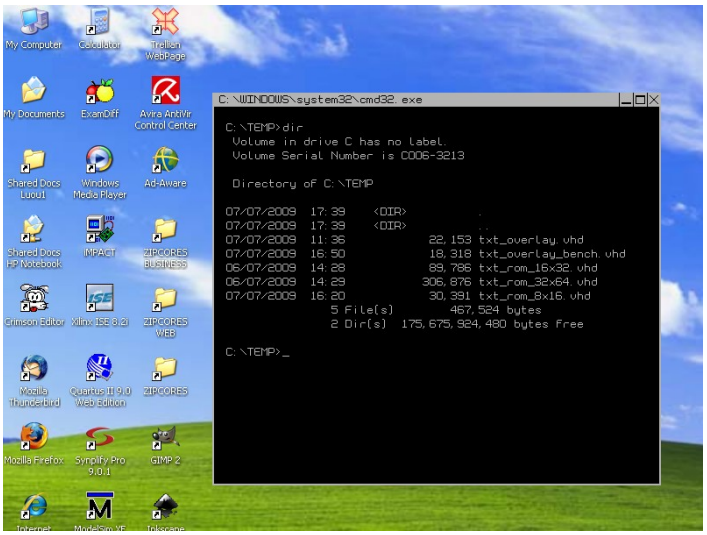

Figure 12: Console or Terminal window

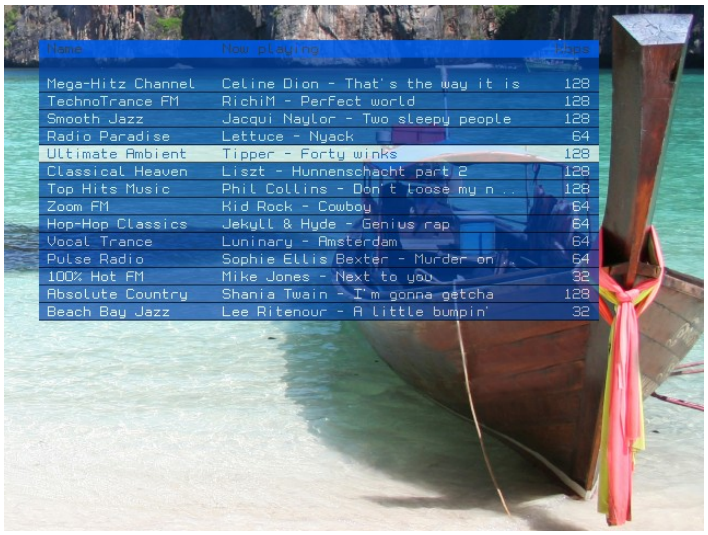

Figure 14: Menu with text-box alpha channel at 66%

Figure 13 is an example of an interactive style menu. Two text-overlay modules were cascaded in series with the first module giving the general menu layout and the second module adding some lines under the text to delineate each entry. The menu also demonstrates how rows of text may be highlighted using the character 'invert' feature. Interactive menus and lists are easy to implement for digital TV and home media solutions.

In the final example (Figure 15) a custom character set was programmed to include a series of shapes and icons. The result shows how the text overlay module can be adapted to produce charts, counters, dials and gauges. By defining a series of 2D shapes the charts and gauges may be animated in real time.

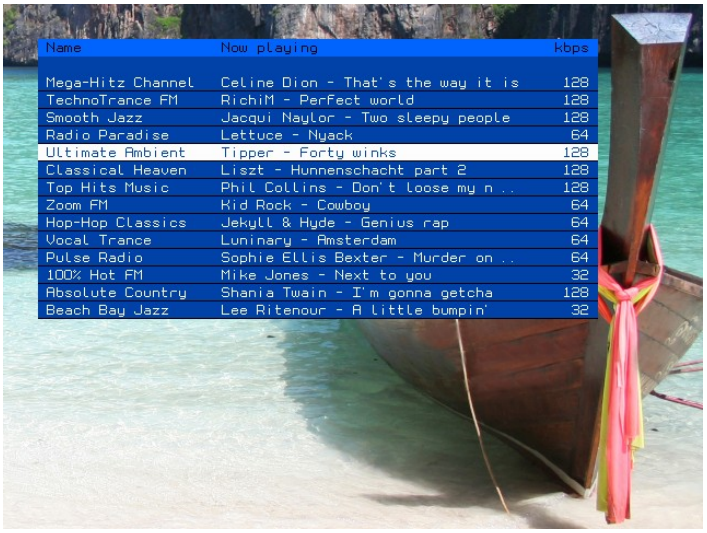

Figure 13: Interactive menus, tables and lists

Figure 14 demonstrates the same example as Figure 13, but this time with transparency enabled. For this example alpha was set to 0xAA (66%) to give the desired effect. The alpha channel is an ideal way to achieve smooth fade-in and fade-out video effects.

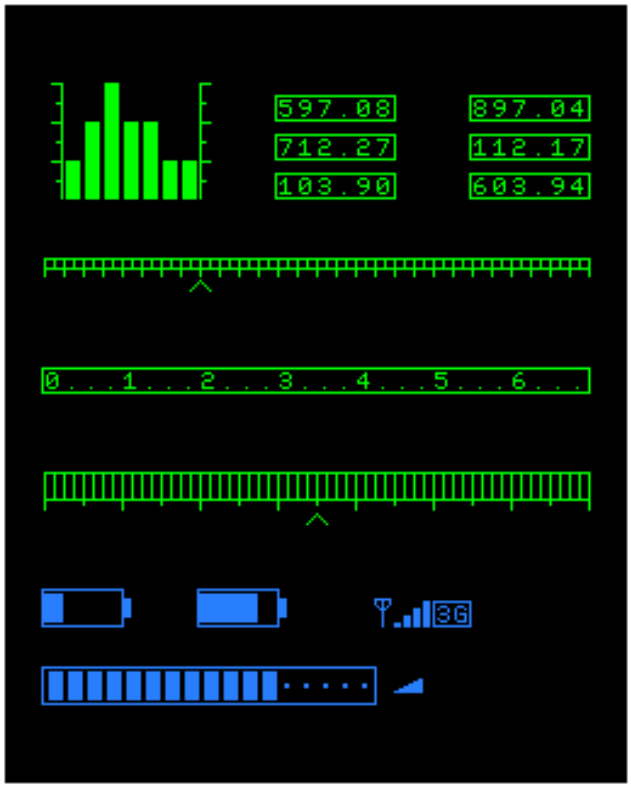

Figure 15: Bar charts, counters, gauges, dials etc..

Synthesis and Implementation

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

- txt_overlay.vhd
  - txt_input_buffer.vhd
  - txt_char_buffer.vhd
  - txt_rom_8x16.vhd
  - txt_rom_16x32.vhd
  - txt_rom_32x64.vhd
  - txt_rom_64x128.vhd
  - txt_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 *text_size* will effect the number of embedded Block RAM components in the design. The largest size of 64x128 pixels will result in the greatest utilization of block RAM. For further block RAM savings, 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: text_size = 2, use_alpha = true, pixels_per_line = 640, lines_per_frame = 480.

Resource usage is specified after place and route of the design.

XILINX® 7-SERIES FPGAS

Resource type	***A-7***	***K-7***	***V-7***	***US+***
Registers	561	561	561	561
LUTs	795	738	810	789
Block RAM	8.5	8.5	8.5	8.5
DSPs	0	0	0	0
Occupied Slices	301	310	334	196 (CLB)
Clk freq. (approx)	200 MHz	250 MHz	300 MHz	500 MHz

Revision History

Revision	***Change description***	***Date***
1.0	Initial revision	15/07/2009
1.1	Fixed various document typos and updated synthesis results	12/02/2010
1.2	Changed character buffer to 64x32 entries in keeping with hardware	13/09/2010
1.3	Included extra character sets Updated synthesis results	18/10/2011
1.4	Added horizontal/vertical shift	19/03/2012
1.5	Added clip-plane functionality	17/04/2012
1.6	Updated synthesis results for Xilinx® 7-series FPGAs and also for some minor sub-component changes	21/08/2015
1.7	Included support for the extra large character size of 64x128 pixels. Updated results for Xilinx® 7-series Ultrascale	04/11/2022