The home-made oscilloscope on the DE1-SoC

The hardware project for laboratory 3 (UARTHW) contains an home-made oscilloscope to display the UART signals using a VGA monitor connected to the DE1-SoC board. Here are some instructions on how to use it and how to change its configuration (screen resolution and number of samples).

How to use the oscilloscope

The hardware project contains all blocks that are needed for the UART laboratory sessions (both for the second and for the third), plus additional hardware to implement the oscilloscope. Connect a VGA monitor to the VGA connector on the DE1-SoC board and the USB-mini cable to the USB-mini connector (the other side of the USB-mini cable goes to the PC running the terminal emulator program). You also need the standard USB cable for programming the board (connected to the PC running Quartus Prime and NiosII Software Build Tools for Eclipse) and the power supply.

Open the hardware project in Quartus Prime and start the Programmer. Program the board using the provided .sof file in the output_files directory. Set the SW9 to the 0 position for the second laboratory session (software UART) and to the 1 position for the third laboratory session (UART Peripheral). Then open the Nios II Software Build Tools for Eclipse, write your software (create a project and the corresponding BSP) and download it to the processor. Using the terminal emulator you can send and receive data over UART.

In the standard implementation the monitor is driven at 1024x768 resolution at 60 Hz refresh frequency, with a pixel clock of 65 MHz. The oscilloscope on the VGA monitor shows 4 channels:

- The UARTRX signal (i.e. data from the PC to the DE1-SoC board) in yellow (top waveform)
- The UARTTX signal (i.e. data from the DE1-SoC board to the PC) in blue (second waveform)
- The GPIO1(0) signal (that you can control with the LSB of the value written to the NIOS_HEADER_CONNECTOR PIO) in red (third waveform)
- The GPIO1(1) signal (that you can control with the LSB+1 of the value written to the NIOS_HEADER_CONNECTOR PIO) in green (bottom waveform)

The oscilloscope always operates in Single Mode. When it is ready the red LED9 is on and all waveforms showing on the monitor are constant. When LED9 is off, waveforms are present on the monitor and the oscilloscope is stopped. The enable the Single Mode again, press the pushbutton KEY3, and LED9 should become on again, waiting for waveforms.

The trigger is on both UARTRX and UARTTX. Whenever one of the two signals shows a high to low transition, the acquisition is started for a certain amount of time, and the waveforms are displayed. The other two channels are not used for triggering, so if there is no UART transmission, you should not see anything on the oscilloscope (you may actually see something, as the system is acquiring data anyway but not synchronizing). If pressing KEY3 does not enable the Single Mode (i.e. the LED9 does not light up), try to reprogram the board using the Programmer (and stop any running software before running the Programmer again).

The monitor shows a grid, with squares divided into 4 segments on each side. Each square is 64 pixels (regardless of the resolution of the monitor), and each segment is 16 pixels. To control the time scale, i.e. how many seconds correspond to a square, you should write a number in a DIVISOR register used for sampling and showing the data. The expressions to be used are the following:

- $pixel_{time} = \frac{DIV+1}{f_{clk}} = \frac{DIV+1}{50 \cdot 10^6}, \qquad DIV = f_{clk} \cdot pixel_{time} - 1 = 50 \cdot 10^6 \cdot pixel_{time} - 1$

- $seg_{time} = \frac{DIV+1}{f_{clk}} \cdot 16 = \frac{DIV+1}{50 \cdot 10^6} \cdot 16, \qquad DIV = \frac{f_{clk} \cdot seg_{time}}{16} - 1 = \frac{50 \cdot 10^6 \cdot seg_{time}}{16} - 1$
- $square_{time} = \frac{DIV+1}{f_{clk}} \cdot 64 = \frac{DIV+1}{50 \cdot 10^6} \cdot 64, \qquad DIV = \frac{f_{clk} \cdot square_{time}}{16} - 1 = \frac{50 \cdot 10^6 \cdot square_{time}}{64} - 1$

So to have a square equal to 1ms it should be:

- $DIV = \frac{f_{clk} \cdot square_{time}}{16} - 1 = \frac{50 \cdot 10^6 \cdot 1 \cdot 10^{-3}}{64} - 1 = 780$

And to have a square equal to 10µs it should be:

- $DIV = \frac{f_{clk} \cdot square_{time}}{16} - 1 = \frac{50 \cdot 10^6 \cdot 10 \cdot 10^{-6}}{64} - 1 = 7$

To write the DIVISOR register use the following code at the beginning of your main() function:

```
int osc_divisor_10ms = 780;
IOWR_ALTERA_AVALON_PIO_DATA(NIOS_OSCDIVISOR_BASE, osc_divisor_10ms);
```

or you can put the value directly into the macro (but I suggest to define a few different variables for different time scales with proper names, so that it is easy to choose the right one). You can change the timescale dynamically by writing the DIVISOR register in your software.

Note that when you write the GPIO1 to show pulses when performing some operation, the pulses are usually pretty short. If the timescale is long, then you may completely miss these pulses. Reduce the DIVISOR in this case. Sometimes it is sufficient to change it by a single unit.

In the standard implementation, the oscilloscope acquires 2048 samples of the input signals, regardless of the monitor resolution. However, the horizontal resolution of the monitor is usually less, so the entire acquired data does not fit on the screen completely. On the top of the monitor you see a mini version of the waveforms, with the area currently being displayed with a grey background. Pressing KEY2 and KEY1 allows to move the displayed area to the left and to the right, to see other parts of the acquired data.

Important: when using this oscilloscope the data is taken directly from the inside of the FPGA. So you don't need any female/male cable, like you do when using a normal oscilloscope. You obviously need the VGA cable. Note also that the data acquired is digital (and each sample occupies a single bit), so this implementation is not adequate to show analog signals on the monitor (but it can be changed to do it, using the ADC on the board).

Changing the amount of data acquired

The standard number of samples acquired is 2048. However, you can increase it (or decrease it, but 2048 is already a low value) by changing the hardware project and recompiling it. To change it, open the file uarthwsw.vhd and look at the lines 267 and 268, which are:

```
constant depth    : integer range 0 to 16383 := 2048;
constant bitdepth : integer range 0 to    14 :=   11;
```

The constant depth is the number of samples acquired. You can increase it for instance to 4096, and change the bitdepth to 12. Or change it to 8192 and change the bitdepth to 13. In general, the bitdepth should be the number of bits which are necessary to represent the number depth-1. Once you change this value, recompile the entire system (it takes a while, especially with many samples, up to one hour), and reprogram the board.

You will see that the mini waveforms on the top are now longer, and you need to pan over it with KEY2 and KEY1 to see them all. Increasing the number of samples allows to use a lower DIVISOR (which corresponds to a higher zoom level) while still being able to show an entire transmission.

Having more data increases the routing requirements, and this can lead to not meeting timing. The consequence is that you see some glitches on the waveforms displayed (the data acquired is correct, but the display fails). If you find a way of avoiding these glitches, please let me know! It might be an error of mine in the design of the oscilloscope (but at 1024x768 with 2048 samples I never see any glitch).

Changing the display resolution

Your monitor may not be happy to show a picture at 1024x768 and may require a higher resolution. If you want to change it, look at the file vgaoscilloscope.vhd. This file contains the main entity for the oscilloscope, and defines all the required timing as constants at the beginning of the architecture. Just comment out the constants defined for 1024x768 and uncomment the constants for the wanted resolution. I managed to make it work up to 1680x1050, and I didn't try 1920x1080 (try it out if you have time). For other resolutions, look on the internet for the timing specification (see for instance http://tinyvga.com/vga-timing).

Once you change the constants for timing, you also need to change the pixel clock. This is in the uarthwsw.vhd file. The pixel clocks are generated by a second PLL in Platform Designer and the outputs are called: clock65, clock108, clock86, clock162 and clock144. When instantiating the vgaoscilloscope entity you need to pass the correct clock. Look at line 419 in the port map and change it accordingly to your needs. For instance, for 1280x1024 resolution you need to change it to:

```
pixelclock  => clock108,
```

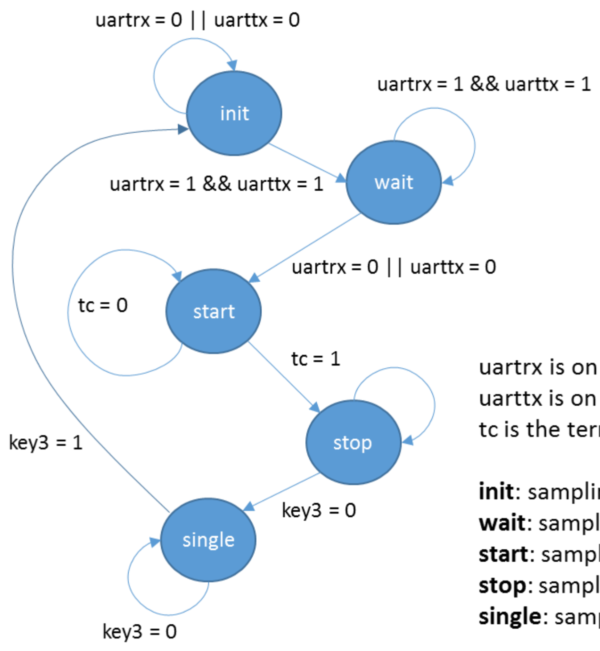
After making these changes, you need to recompile the entire hardware project and reprogram the board using the Programmer. Again, it takes several minutes or several tens of minutes to complete compilation, so be prepared to wait.

Structure of the hardware oscilloscope

The oscilloscope consists of a sampler (sampler.vhd, actually instantiated 4 times, since I'm acquiring 4 channels in parallel) which is driven by the clock obtained after the divisor[1] (divisor.vhd, a simple counter with terminal count and maximum value). The sampler writes the data in a shift register with a number of elements equal to depth, so that the first element is always the most recent, while the last is the oldest; when displayed, the last element is shown on the left of the monitor, and the most recent towards the right. The sampler is always active, but after the trigger a counter counts (depth – 64) clocks (samples_counter.vhd, again a simple counter with terminal count and maximum value) and then disables the sampler, until the Single mode is again activated. In this way, after the trigger the data in the shift register is filled in and then remains to be displayed. All operations are managed by a Finite State Machine (osc_fsm.vhd) whose state diagram is in the picture.

To display the waveform, the entity sync is used (sync.vhd). This entity is a process that runs at every clock cycle of the pixel clock, i.e. it runs for each pixel. It is responsible of creating the required timing (sync, back porch, display data, front porch) to keep the monitor synchronized. Inside the process the two signals hpos and vpos scan all pixels, including those outside the visible area, where the sync happens. It drives a triple DAC that generates the analog signals for the monitor.

[1] The clock is always the 50MHz clock of the board, but the sampler is enabled by the terminal count of the divisor, so the effective clock frequency is lower.



uartrx is on the trigsrc1 input of the FSM
uarttx is on the trigsrc2 input of the FSM
tc is the terminal count of the samples counter

init: sampling is active, the samples counter is cleared
wait: sampling is active, the samples counter is cleared
start: sampling is active, the samples counter is active
stop: sampling is disabled, the samples counter is cleared
single: sampling is disabled, the samples counter is cleared

What a typical module of this sort would do is to read a memory that contains the image to be displayed. However, this case is different. The module takes the data directly from the shift registers, and reads them only when the pixel is at the correct position. Reading is implemented using a procedure, to simplify the code allow making changes in a single place. The shift registers contains values 0 or 1, and to draw vertical transitions the module checks if the data changes from one element of the shift register to the next (vertical transition are not plotted in the mini version).

The module also implement the mini waveforms at the top of the monitor, and the grid. During the vertical blank, it also checks the pushbuttons key2 and key1 to implement panning, by changing a single variable that identifies the beginning of the display area.

Possible improvements

- Increase the number of channels (pretty easy, just duplicate everything that refers to a channel).
- Make an auto triggering, so that a periodic input can be displayed correctly without activating Single mode.
- Perform sampling at high frequency and then implement a zooming function.
- Use on-chip memories to hold the data. This is especially useful if you want to show analog data acquired by the ADC, because in this case the shift register becomes huge. The on-chip memory shall be used with two ports, one from the sampler, the other for the sync module.