

Simple GLITC User Manual

You need:

1. GLITC firmware (glitc_top_v2.bit)
2. ChipScope running with a connection to the device
3. Python

Program GLITC

Note: “dev = tisc.TISC()” selects /sys/class/uid/uid0 as the path to the device. You can also pass a path to the device (e.g. dev = tisc.TISC(“/sys/class/uid/uid1”). Make sure your permissions allow you to access “config”, “resource0”, and “resource1” under the path to the device.

```
Python 2.7.9 (default, Mar 1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import tisc
>>> dev = tisc.TISC()
>>> dev.gprogram(0, "glitc_top_v2.bit")
```

Initialize datapath

```
>>> dev.GA.datapath_initialize()
```

Enable the phase scanner

```
>>> dev.GA.write(0x40, 0x1)
```

Find a crude operating point

```
>>> dev.GA.rdac(0, 32, 3500)
>>> dev.GA.rdac(0, 31, 1700)
>>> dev.GA.rdac(1, 32, 3500)
>>> dev.GA.rdac(1, 31, 1700)
```

The “rdac” function loads the RITC DAC. The first parameter specifies the RITC, and the second specifies the DAC number. DAC #32 is VDD (the less sensitive parameter). DAC #31 is VSS (the sensitive parameter).

Verify crude operating point

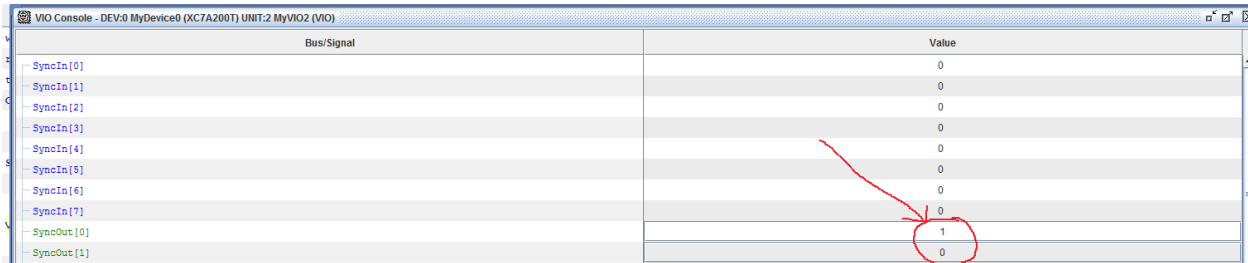
```
>>> dev.GA.counters()
Channel 0: 512
Channel 1: 512
Channel 2: 512
Channel 3: 512
Channel 4: 512
Channel 5: 512
```

If the counters are not identically 512, try raising/lowering VSS (DAC #31) until they are.

Load phase scanner ChipScope setup

Filename is "phasescanner.cpj" in "glitc_debug" in the firmware repository.

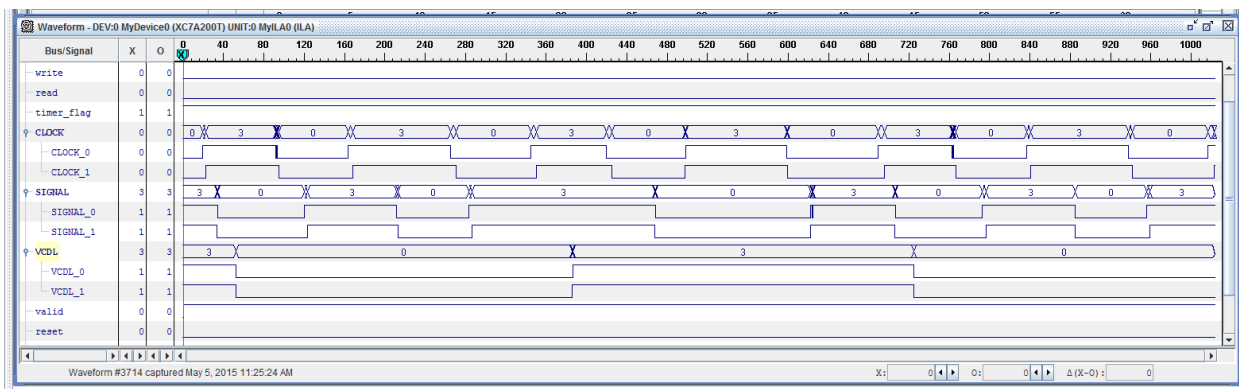
Switch to phase scanner debug inputs in ChipScope



Bus/Signal	Value
- SyncIn[0]	0
- SyncIn[1]	0
- SyncIn[2]	0
- SyncIn[3]	0
- SyncIn[4]	0
- SyncIn[5]	0
- SyncIn[6]	0
- SyncIn[7]	0
- SyncOut[0]	1
- SyncOut[1]	0

Enter "repetitive trigger mode" on phase scanner ILA (ILA0)

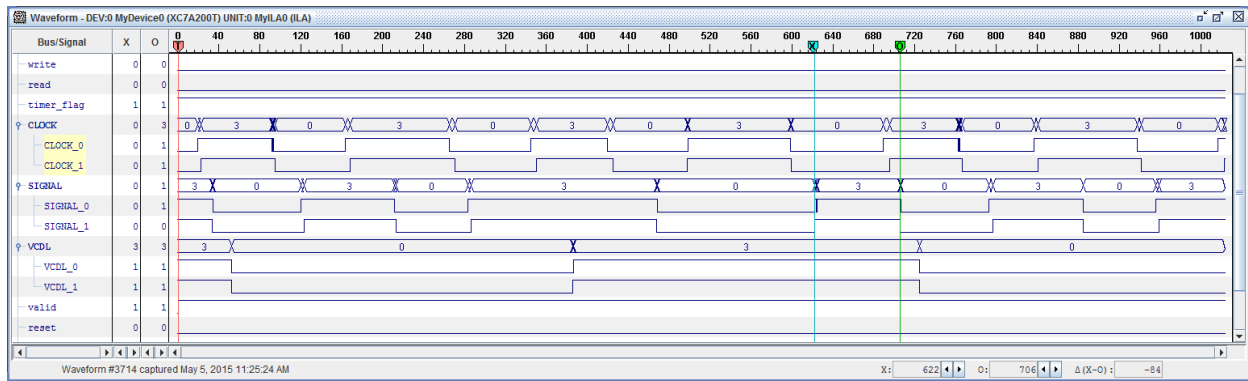
You should see something like this:



Fine tune the operating point

The goal of the fine tune is to make the *width of the last bit* in the training pattern equal to 84 samples. This is because each step is $(1/(975 \text{ MHz} * 56)) = 18.3 \text{ picoseconds}$, and $84 * 18.3 \text{ picoseconds}$ is 1537.2 ns, equal to 650 MHz (the bit rate of the data out of the TISC). Put another way, there are 672 (56 steps in 1 VCO period, and 12 VCO periods in 1 VCDL period) steps in a full VCDL cycle, and 8 total bits. $672/8 = 84$.

The last bit is indicated with the X/O cursors in the next diagram



Note that ChipScope specifies the X-O distance in the bottom right corner, which helps.

Raise/lower VSS (DAC #31) as needed until the bit width is 84 samples.

Autotune the data capture

Once a fine operating point has been found, the data capture can be automatically tuned:

```
>>> dev.GA.eye_autotune_all()
eye_autotune: setting to delay 11
eye_autotune: bitslipping 0 times
... (repeat for all bits/channels)
```

Note that in normal operation the bitslip count (“bitslipping 0 times”) will be 2 the *first time* after programming. After that the bitslip count should always be 0. This is because the ‘bitslip count’ is remembered.

Disable training

To actually view proper RITC data rather than training data, training must be disabled with:

```
>>> dev.GA.training_ctrl(0)
```

At this point you can view *individual samples* in the VCDL period with

```
>>> dev.GA.train_read(channel, sample, 1)
```

Channel must be (0,1,2) or (4,5,6). Sample must be between 0-31. The final ‘1’ puts “train_read” in ‘sample view’ mode, where the bits are captured as 3-bit samples rather than the 8 bits that a given LVDS pair puts out per VCDL cycle.

After power on, most likely all samples will read out “7”, since the comparators will all be set to 0.

Tune the thresholds

At this point the comparator thresholds can be tuned. The input value can be set using the DAC, via

```
>>> dev.GA.dac(channel, value)
```

“Channel” here should be 0,1,2 or 4,5,6. “Value” here should be 0-2000, which converts into millivolts via $(value * 2500 / 4095)$.

