Gisselquist
Technology, LLC

WISHBONE CONTROLLED FM TRANSMITTER HACK SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) opencores.org

June 15, 2016

Revision History

Rev.	Date	Author	Description
0.1	6/15/2016	Gisselquist	First Draft

Contents

Tables

Preface

After watching someone demonstrate a Python hack that turned a Raspberry Pi into a poor man's FM transmitter, I decided that I should try to see if I could do the same with an FPGA. Indeed, it should be easier with an FPGA: the FPGA has complete control of the clock, as well as what the data line does. Therefore, this hack attempts to turn a GPIO line into an FM transmitter line for an antenna.

Dan Gisselquist, Ph.D.

1.

Introduction

This project is a hack.

It is not intended, nor appropriate, for any commercial or otherwise useful product. Broadcasting on commercial FM channels has legal implications associated with it. I am not recommending that you turn your FPGA into an illegal FM transmitter.

The purpose of this project is to show that an FPGA's outputs can be used to create a (nearly) analog FM output.

As the preface mentions, this project is also about one-upsmanship. Just because your Raspberry Pi can do something doesn't mean my FPGA can't. Here, let me prove to you that an FPGA can create and broadcast on a commercial FM radio channel.

As with any specification, this one is broken into sections or chapters. Chap. 2 will start off by explaining how to use this core. Chap. 3 will then discuss the registers in detail. This may seem like rehashing the Chap. 2 chapter, but the information is presented in a different order. Chap. 4 then presents the wishbone data sheet necessary for any wishbone compliant core. Finally, Chap. 5 walks through the I/O ports of the core.

2.

Operation

From a logical standpoint, the operation of this core is quite simple. Just follow the following steps:

1. Select the frequency "channel" to transmit on.
2. Adjust the sample rate to set how fast output samples will be sent to the device.
3. Send the first sample to the core
4. Wait for an interrupt, then send the next sample to the core
5. Repeat step 4 until the desired transmission is done.
6. Once transmission is complete, set the frequency "channel" slash NCO step size to zero.
7. Set the next sample to zero.
8. Disable, in your interrupt controller (external to this core) the interrupt generated by this core.

Internally, the core attempts to generate a square wave at a frequency given by the set frequency plus an amount given by the sample value times a constant. To do this, the core maintains a 32 bit counter which will roll over at the carrier frequency times per second. The top bit of this counter becomes the output bit for the transmitter. The counter is incremented every clock by an amount used to set the carrier frequency, plus an amount given by the input sample.

For example, let's assume that the FPGA is running with an 80 MHz clock. To toggle the output line at a rate of 20 Mhz, one need only set the counter increment to `0x40000000`. The top bit will, over time, trace through `0, 0, 1, 1` –creating a square wave at 20 MHz. As a rather interesting by product of the fact that this is a square wave is that this 20 MHz tone will have artifacts at odd harmonics of 20 MHz: 60 MHz, 100 MHz, 140 MHz, etc. The energy in each of these harmonics will decrease, dependent upon both the FPGA switching speed and the nature of a square wave. In particular, the 100 MHz harmonic will have 13.6 dB less power than the fundamental at 20 MHz.

Now, if we add a value of `0x7fff` times 32 to this counter increment, creating a new increment of `0x400fffe0`, the new counter will roll over as many times in 2^{32} clocks, creating a frequency of roughly 20.019 MHz. It's fifth harmonic, however, will be at 100.097 MHz, nicely at the edge, if not a little beyond, the frequency range of FM broadcast radio.

By changing the offset to the counter increment with each sample, we create a Frequency Modulation. This is what allows us to generate an FM waveform similar to that in the FM Broadcast band.

If only life were that simple, we'd be done at this point.

The next part of the operation of this hack is the antenna. For best performance, this output waveform needs to be fed into an antenna with a DC block and ground as the other lead and a DC block. Ideally, this antenna should be impedence matched to the board as well.

For the purposes of our hack, we will ignore these details and hope to demonstrate success with just the previously discussed logic.

2.1 Software Example

Before leaving our concept of operation, let's walk through some code which was used to demonstrate this board. The demonstration itself was done using a ZipCPU, together with a modified version of the XuLA2-LX25 SoC, both available from OpenCores.[1]

The first step is to set the frequency channel of the board. Here, we set it to 91.9 MHz, based upon an 80 MHz internal oscillator clock.

```
sys->io_fmtx_nco  =  0x26147ae1;
```
(2.1)

The next step is to set the sample rate of the device. In my case, I set this as a parameter to the module. However, it can also be set here as a run time configuration parameter:

```
sys->io_fmtx_audio  =  1814<<16;
```
(2.2)

For our example, we'll poll the interrupt controller to see when the `INT_FM` interrupt line goes high:

```
while((sys->io_pic & INT_FM)==0) ;
```
(2.3)

Once it goes high, we can send a sample to the transmitter,

```
sys->io_fmtx_audio  =  sample & 0x0ffff;
```
(2.4)

We now repeat the process of checking the transmitter for readiness to send the next sample, and sending samples, until we are done.

Once complete, we simply turn the module off:

```
sys->io_fmtx_nco  =  0;
```
(2.5)

```
sys->io_fmtx_audio  =  0;
```
(2.6)

That's it! It's really quite simple to use.

[1] That is, the XuLA2-LX25 SoC is available from OpenCores, as is the ZipCPU, but the modified version is not posted. It just didn't seem worth it to maintain a simple hack there.

3.

Registers

This FM Transmitter core supports two registers, as listed in Tbl. 3.1: a next sample register, `SAMPLE`, and a carrier frequency control register called `NCOSTEP`. Each register will be discussed in

Name	Address	Width	Access	Description
SAMPLE	0	32	R/W	Controls the sample value out of the transmitter, as well as the sample rate of the transmitters interrupts requesting further samples.
NCOSTEP	1	32	R(/W)	Controls the step size of the pseudo-oscillator controlling the RF frequency. Appropriate writes to this register will determine what channel the FM transmitter broadcasts on.

Table 3.1: List of Registers

detail in this chapter.

3.1 Sample Register

The bits in the control register are defined in Tbl. 3.2.

Basically, in sum, the top 16 bits determine the sample rate of the audio being sent to the device. Perhaps more accurately, they set the number of clocks between assertions of the CPU interrupt line. The core will internally run a timer at an interval given by these bits. When the timer is up, it will transmit its next sample, assert an interrupt, and restart the timer with this value. The CPU will then have until the timer expires to provide the next sample. Writing to this register with these bits set to zero will cause them to be ignored.

It should be possible to run this from a DMA controller, although I have not tried to do so.

The lower 16 bits of this register, when written to, control the next audio sample out of the device. When read from, they return the current audio sample being produced by the device, and in the low order bit whether or not an interrupt is currently being asserted.

3.2 Carrier Frequency Control Register

Based upon Nyquist principles, properly producing a sampled tone requires samples that are at least twice the frequency of the desired tone. In the case of commercial FM in the US, the highest

Bit #	Access	Description
16–31	R/W	This is the number of clocks between interrupts. Hence, to transmit from a waveform file sampled at a rate of R samples per second, from an FPGA with a clock rate of F Hz, set this value to F/R. For example, to transmit at 44.1 kHz from an FPGA with an 80 MHz clock, set this value to 1814. Writing a value of zero to this register has no effect, allowing a user to only write the sample value at each write without adjusting the sample rate.
0–15	W	Signed, twos complement, next sample to be broadcast.
1–15	R	Signed, twos complement, current sample being broadcast.
0	R	A 1'b1 if the interrupt is currently active, otherwise zero. The actual lowest bit of the data value in the transmitter cannot be read out.

Table 3.2: Sample Register

frequency may be roughly 110 MHz. This means that the FPGA must produce a sampled output using a clock of at least 220 MHz.

My FPGA boards don't clock that high. Instead, I can clock my Spartan–6 boards at 80 MHz. While this should be sufficient for transmitting in the Citizen's Band of 26 to 28 MHz, it is entirely insufficient for transmitting at commercial radio.

Instead, to reach these really high speeds, this core exploits what is normally an undesired consequence of sampling: aliasing. Basically, that means that it is possible to produce a tone at some frequency, such as 10 MHz, as well as your clock rate plus that frequency, or 90 MHz in my case. The 90 MHz output is often considered an undesirable artifact of the square wave outputs produced by the FPGA, but in our case we exploit this.

Now that all that is said, we can discuss setting the Carrier Frequency Control Reigster. This register is set when you wish to begin transmitting to:

$$\texttt{CFCR} \;=\; \left\lfloor \frac{2^{32} f_{ch}}{f_{\rm FPGA}} + \frac{1}{2} \right\rfloor \tag{3.1}$$

where f_{ch} is the center frequency you wish to transmit on, and $f_{\rm FPGA}$ is your FPGA clock frequency. Note that this value will be greater than 2^{32} for my setup, since the frequency of my FPGA is less than that of the channel I wish to transmit on. In this case, just throw away any bits above the lower thirty–two and continue.

As an example, my FPGA's clock runs at 80 MHz. In order to transmit at 91.9 MHz, I would then set the `CFCR` register to 0x26147ae1.

4.

Wishbone Datasheet

Tbl. 4.1 is required by the wishbone specification, and so it is included here. The big thing to notice

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Slave, Read/Write, pipeline reads supported
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	(Irrelevant)
Clock constraints	None.
Signal Names	Signal Name / Wishbone Equivalent i_wb_clk / CLK_I i_wb_cyc / CYC_I i_wb_stb / STB_I i_wb_we / WE_I i_wb_addr / ADR_I i_wb_data / DAT_I o_wb_ack / ACK_O o_wb_stall / STALL_O o_wb_data / DAT_O

Table 4.1: Wishbone Datasheet

is that this core acts as a wishbone slave, and that all accesses to any local registers become 32–bit reads and writes to this interface.

5.

IO Ports

The ports are listed in Table. 5.1. Of these ports, the `i_wb_*` and the `o_wb_*` ports are all defined

Port	Width	Direction	Description
`i_clk`	1	Input	The clock synchronizing the entire core.
`i_wb_cyc`	1	Input	Indicates a wishbone bus cycle is active when high.
`i_wb_stb`	1	Input	Indicates a wishbone bus cycle for this peripheral when high. (See the wishbone spec for more details)
`i_wb_we`	1	Input	Write enable, allows indicates a write to one of the two registers when `i_wb_stb` is also high.
`i_wb_addr`	1	Input	A single address line, set to zero to access the configuration and control register, to one to access the data register.
`i_wb_data`	32	Input	Data used when writing to the core. Valid when `i_wb_cyc`, `i_wb_stb`, and `i_wb_we` are all high, ignored otherwise.
`o_wb_ack`	1	Output	Wishbone acknowledgement. This line will go high on the clock after any wishbone access.
`o_wb_stall`	1	Output	Required by the wishbone spec, but always set to zero in this implementation.
`o_wb_data`	32	Output	Value read, whether the next sample register or the nco step register, headed back to the wishbone bus master. These bits will be valid during any read cycle when the `o_wb_ack` line is high.
`o_tx`	1	Output	A one wire output value to be sent to the "antenna" output pin of your FPGA.
`o_int`	1	Output	True whenever the next sample has transitioned to the current sample, until a new next sample is written.

Table 5.1: List of IO ports

by the wishbone specification. This leaves two ports of interest, `o_tx` and `o_int`.

The `o_tx` output is the FM transmitter output. This output needs to be wired off of your board to your FM transmit antenna. Should your board not have such an antenna, one can often be improvised by sending this output to any available output ports from your FPGA. The more

GPIO's that are set with this value, the more power the device will output and likewise the better the output may approximate an FM antenna.

Finally, the `o_int` line is an interrupt line to be sent to whatever controller is controlling the transmitter. This interrupt line will be set whenever the transmitter is ready for a new sample. It is also self clearing, so that sending a sample to the transmitter will turn this off until the next value is needed.