AXIS_1553_ENCODER

May 21, 2025

Jay Convertino

Contents

1 Usage

1.1 Introduction

AXIS 1553 Encoder is a core for taking AXIS data and encoding for output to the PMOD1553 device. The output is a TTL differential signal. Meaning when diff[0] is 1 diff[1] is 0. This core also includes a diff enable which allows for mux switching to the transmit (encoder) when active.

1.2 Dependencies

The following are the dependencies of the cores.

- fusesoc 2.X
- iverilog (simulation)
- cocotb (simulation)

1.2.1 fusesoc_info Depenecies

- dep
  - AFRL:utility:helper:1.0.0
- dep_tb
  - AFRL:simulation:axis_stimulator
  - AFRL:utility:sim_helper

1.3 In a Project

Connect the device to your AXIS bus. TUSER is used to set various options such as command/data packet mode.

TDATA input should contain the 16 bit data payload. TUSER is a 8 bit command register that takes a description what type of data it is (command or data) and other options described below.

TUSER = TYY,NA,D,I,P (7 downto 0)

- TYY = TYPE OF DATA
  - 000 N/A
  - 001 REG (NOT IMPLEMENTED)
  - 010 DATA
  - 100 CMD/STATUS

- NA = RESERVED FOR FUTURE USE.
- D = DELAY ENABLED
  - 1 = 4 us delay enabled.
  - 0 = no delay between transmissions.
- I = INVERT DATA
  - 1 = Invert data.
  - 0 = Normal data.
- P = PARITY
  - 1 = ODD
  - 0 = EVEN

2 Architecture

This core is made up of a single module.

- **axis_1553_encoder** Interface AXIS to PMOD1553 device (see core for documentation).

2.1 Encoding Method

This core has 4 always blocks that are sensitive to the positive clock edge. They are

- **pause counter** Inserts delays between transmits to comply with 4us spacing.
- **axis data input** Deals with AXIS bus input data to the core based on its current state.
- **data processing** In charge of the state machine and processing of input data to 1553 non-differential bitstream.
- **differential data output** Output processed data from a non-differential bitstream into a differential one.

Pause counter simply gets reset at each transmit state to its initial value. Then once it is out of the transmit state it starts its countdown. The data processing block checks the pause counter, if it is not 0 we will not transition to the transmit state.

AXIS data input does exactly as it says. Take input data in the data capture state and register it. All other states simply ignore the input data.

Data processing block does most of the work. It is in charge of the state machine state and generating the 1553 non-differential bitstream. Essentially the state machine does the following:

1. Startup in error state, which does to the default handler that puts it into data capture.
2. Wait for data input from the AXIS streaming input, once valid data is input. Then go to the data invert state.
3. Check the TUSER command if a data invert is necessary. Always go to the parity generation state.
4. Generate parity bit based upon the input data using xor method. Then move on to process state.
5. In process state take the TUSER and TDATA input and process it into register. This register will contain a non-differential 1553 bitstream.
   - Check if pause check is enabled, if it isn't skip it.
   - Based on TUSER insert the needed sync pulse of command or data. 0 is inserted if a invalid selection is made.
   - Check parity option for odd or even and insert parity bit based upon selection and generated parity bit.
   - Generate machester data using XOR method, this is done with a synthetic clock xor with the input data over the number of samples needed. The sythetic clock is contained in the register used for the result.
6. Check for pause (if it wasn't skipped) when counter is 0, move to transmit state.
7. In transmit state wait for differential data output to process all of the register data. Once all conditions are meet, move to the data capture state to wait for more input data to encode.

Differential data output waits for the transmit state, once it is reached the core will begin outputing the registered data generated by data processing in the process state. It will output a TTL differential version of the data and keep enable diff high so a mux can be switched for transmit mode. Once the data, and by extension counters, are exhausted the block will wait for the next time transmit state is reached.

3 Building

The AXIS 1553 Encoder is written in Verilog 2001. It should synthesize in any modern FPGA software. The core comes as a fusesoc packaged core and can be included in any other core. Be sure to make sure you have meet the dependencies listed in the previous section. Linting is performed by verible using the lint target.

3.1 fusesoc

Fusesoc is a system for building FPGA software without relying on the internal project management of the tool. Avoiding vendor lock in to Vivado or Quartus. These cores, when included in a project, can be easily integrated and targets created based upon the end developer needs. The core by itself is not a part of a system and should be integrated into a fusesoc based system. Simulations are setup to use fusesoc and are a part of its targets.

3.2 Source Files

3.2.1 fusesoc_info File List

- src
  - src/axis_1553_encoder.v
- tb
  - 'tb/tb_1553_enc.v': 'file_type': 'verilogSource'
- tb_cocotb
  - 'tb/tb_cocotb.py': 'file_type': 'user', 'copyto': '.'
  - 'tb/tb_cocotb.v': 'file_type': 'verilogSource'

3.3 Targets

3.3.1 fusesoc_info Targets

- default

  Info: Default for IP intergration.
- lint

  Info: Lint with Verible
- sim

  Info: Simulation using icarus as the default.
- sim_cocotb

  Info: Cocotb unit tests

de

3.4 Directory Guide

Below highlights important folders from the root of the directory.

1. **docs** Contains all documentation related to this project.
    - **manual** Contains user manual and github page that are generated from the latex sources.
2. **src** Contains source files for the core
3. **tb** Contains test bench files for iverilog and cocotb
    - **cocotb** testbench files

4 Simulation

There are a few different simulations that can be run for this core.

4.1 iverilog

iverilog is used for simple test benches for quick verification, visually, of the core.

4.2 cocotb

To use the cocotb tests you must install the following python libraries.

```
$ pip install cocotb
$ pip install cocotbext-axi
$ pip install cocotbext-mil_std_1553
```

Then you must use the cocotb sim target. The targets above can be run with various bus and fifo parameters.

```
$ fusesoc run --target sim_cocotb AFRL:device_converter:axis_1553_encoder:1.0.
```

5 Module Documentation

- **axis_1553_encoder** Interfaces AXIS to the PMOD1553.

The next sections document the module in great detail.

axis_1553_encoder.v

AUTHORS

JAY CONVERTINO

DATES

2021/05/17

INFORMATION

Brief

AXIS MIL-STD-1553 ENCODER

License MIT

axis_1553_encoder

```
module axis_1553_encoder #(
parameter
CLOCK_SPEED
 =
2000000,
parameter
SAMPLE_RATE
 =
2000000
) ( input aclk, input arstn, input [15:0] s_axis_tdata, input s_axis_tvalid,
```

AXI streaming to MIL-STD-1553 encoder. This encoder can be used at 2 Mhz or above. TDATA is 16 bit data to be transmitted. TUSER sets how the core works.

Parameters

CLOCK_SPEED parameter — This is the aclk frequency in Hz, must be 2 MHz or above.

SAMPLE_RATE parameter — 2 MHz or above rate that is an even divisor of CLOCK_SPEED

Ports

aclk	Clock for all logic
arstn	Negative reset
s_axis_tdata	Input data for 1553 encoder.
s_axis_tvalid	When set active high the input data is valid.
s_axis_tuser	Information about the AXIS data {TYY,NA,D,I,P} Bits explained below:

```
- TYY = TYPE OF DATA
      - 000 = NA
      - 001 = REG (NOT IMPLIMENTED)
      - 010 = DATA
      - 100 = CMD/STATUS
- D   = DELAY ENABLED
- I   = INVERT DATA
- P   = PARITY
      - 1 = ODD
      - 0 = EVEN
```

s_axis_tready	When active high the device is ready for input data.
diff	Output data in TTL differential format.
en_diff	When diff is valid data, this is active high and can be used to switch a mux.

base_1553_clock_rate

```
localparam integer base_1553_clock_rate = 1000000
```

1553 base clock rate

samples_per_mhz

```
localparam integer samples_per_mhz = SAMPLE_RATE / base_1553_clock_rate
```

sample rate to caputre transmission bits at

cycles_per_mhz

```
localparam integer cycles_per_mhz = CLOCK_SPEED / base_1553_clock_rate
```

calculate the number of cycles the clock changes per period

samples_to_skip

```
localparam integer samples_to_skip = (
 (cycles_per_mhz > samples_per_mhz) ? cycles_per_mhz / samples_per_mhz -
1
:
0
)
```

calculate the number of samples to skip

bit_rate_per_mhz

```
localparam integer bit_rate_per_mhz = samples_per_mhz
```

bit rate per mhz

delay_time

```
localparam integer delay_time = cycles_per_mhz * 4
```

delay time, 4 is for 4 us (min 1553 time)

sync_pulse_len

```
localparam integer sync_pulse_len = bit_rate_per_mhz * 3
```

sync pulse length

bits_per_trans

```
localparam integer bits_per_trans = 20
```

bits per transmission

synth_bits_per_trans

```
localparam integer synth_bits_per_trans = (
bits_per_trans*bit_rate_per_mhz
)
```

sync bits per trans

bit_pattern

```
localparam [(
bit_rate_per_mhz
)-1:0]bit_pattern = {{bit_rate_per_mhz/2{1'b1}}, {bit_rate_per_mhz/2{1'b0}}
```

create the bit pattern. This is based on outputing data on the negative and positive. This allows the encoder to run down to 1 mhz.

synth_clk

```
localparam [synth_bits_per_trans-1:0]synth_clk = {
bits_per_trans{bit_pattern}
 }
```

synth clock is the clock constructed by the repeating the bit pattern. this is intended to be a representation of the clock. Captured at a bit_rate_per_mhz of a 1mhz clock.

sync_cmd_stat

```
localparam [sync_pulse_len-1:0]sync_cmd_stat = {
                                                                            {
sync_pulse_len/2{1'b0}},
                                                                            {
sync_pulse_len/2{1'b1}}
}
```

sync pulse command

sync_data

```
localparam [sync_pulse_len-1:0]sync_data = {
                                                                            {
sync_pulse_len/2{1'b1}},
                                                                            {
sync_pulse_len/2{1'b0}}
}
```

sync pulse data

cmd_data

```
localparam cmd_data = 3'b010
```

tuser decode for data

cmd_data

tuser decode for command

cmd_data

enable diff output

STATE MACHINE

Constants that makeup the encoder state machine.

data_cap

```
localparam data_cap = 3'd1
```

data capture

data_invert

```
localparam data_invert = 3'd2
```

invert data

parity_gen

```
localparam parity_gen = 3'd3
```

parity generator

process

```
localparam process = 3'd4
```

command processor

pause_ck

```
localparam pause_ck = 3'd5
```

check for pause (4us)

trans

```
localparam trans = 3'd6
```

transmit data

error

```
localparam error = 3'd0
```

someone made a whoops

tb_cocotb.py

AUTHORS

JAY CONVERTINO

DATES

2025/03/04

INFORMATION

Brief

Cocotb test bench

License MIT

FUNCTIONS

random_bool

```
def random_bool()
```

Return a infinte cycle of random bools

Returns: List

start_clock

```
def start_clock(
dut
)
```

Start the simulation clock generator.

Parameters

dut Device under test passed from cocotb test function

reset_dut

```
async def reset_dut(
dut
)
```

Cocotb coroutine for resets, used with await to make sure system is reset.

increment test no delay

Coroutine that is identified as a test routine. This routine tests by sending a incrementing value as a command and then as data, no delay between the two is inserted by the core.

Parameters

dut Device under test passed from cocotb.

increment test delay

Coroutine that is identified as a test routine. This routine tests by sending a incrementing value as a command and then as data, delay between the two is inserted by the core.

Parameters

dut Device under test passed from cocotb.

in_reset

```
@cocotb.test()
async def in_reset(
dut
)
```

Coroutine that is identified as a test routine. This routine tests if device stays in unready state when in reset.

Parameters

dut Device under test passed from cocotb.

no_clock

```
@cocotb.test()
async def no_clock(
```

```
dut
)
```

Coroutine that is identified as a test routine. This routine tests if no ready when clock is lost and device is left in reset.

Parameters

dut Device under test passed from cocotb.

tb_coctb.v

AUTHORS

JAY CONVERTINO

DATES

2025/03/04

INFORMATION

Brief

Test bench wrapper for cocotb

License MIT

tb_cocotb

```
module tb_cocotb #(
parameter
CLOCK_SPEED
 =
20000000,
parameter
SAMPLE_RATE
 =
20000000
) ( input aclk, input arstn, input [15:0] s_axis_tdata, input s_axis_tvalid,
```

This core is a MIL-STD-1553 to AXI streaming decoder. It uses the postive edge of a clock to sample data. This restricts the core to 2 Mhz and above for a sample clock.

Parameters

CLOCK_SPEED parameter — This is the aclk frequency in Hz, must be 2 MHz or above.

SAMPLE_RATE parameter — 2 MHz or above rate that is an even divisor of CLOCK_SPEED

Ports

aclk	Clock for all logic
arstn	Negative reset
s_axis_tdata	Input data for 1553 encoder.
s_axis_tvalid	When active high the input data is valid.
s_axis_tuser	Information about the AXIS data {TYY,NA,I,P} Bits explained below:

```
- TYY = TYPE OF DATA
      - 000 NA
      - 001 REG (NOT IMPLIMENTED)
      - 010 DATA
      - 100 CMD/STATUS
- NA  = RESERVED FOR FUTURE USE.
- D   = DELAY BEFORE DATA
      - 1 = Delay of 4us or more before data
      - 0 = No delay between data
- I   = INVERT
      - 1 = Invert input data before output
      - 0 = No inversion of data before output.
- P   = PARITY
      - 1 = ODD
      - 0 = EVEN
```

s_axis_tready	When active high the device is ready for data.
diff	Output data in TTL differential format.

INSTANTIATED MODULES

dut

```
axis_1553_encoder #(
                                                                        .
CLOCK_SPEED(CLOCK_SPEED),
                                                                        .
SAMPLE_RATE(SAMPLE_RATE)
) dut ( .aclk(aclk), .arstn(arstn), .s_axis_tdata(s_axis_tdata), .s_axis_tva
```

Device under test, axis_1553_encoder