EE3DSD: Digital System Design (Coursework 2)

The following report details some of the methodology and reasoning behind the first work undertaken to implement the second and minute synchronisation components of a decoder for the UK MSF and German DCF77 low frequency radio signals.

Contents

List of Figures

Methodology

Before beginning development of the components, a log of the output generated by the black box model solution was recorded over several minutes with all switches enabled. This file (`allinputs.cap`) contained a set of ideal data which could be used for reference and as a baseline for measuring the success of my own components against. Once this data was recorded, each component could be developed in an iterative style:

1. Copy the component template into the source directory.
2. Prepare testbench stimuli for the component using data from `allinputs.cap`.
3. Write a testbench for the component which uses this stimuli.
4. Develop the VHDL implementation of the component.
5. Simulate the testbench, comparing the generated waveform against expected behaviour.
6. If the simulated design does not behave as expected, return to step 4.
7. Synthesise the VHDL implementation and test on FPGA board with real inputs.
8. If the synthesised design does not behave as expected, return to step 4.

Testbench stimuli

For the DCF and MSF synchronisation component testbenches, a set of signal data was created by filtering the relevant entries from the black box tracefile (Figure 1). These signals could be used to generate testbench stimuli, and the `std_logic_textio` library's `readline()` and `read()` procedures could be used to parse this stimuli in a testbench.

```
$ cat allinputs.cap | grep 'DCF Signal' > dcf-signal.cap
$ cat allinputs.cap | grep 'MSF Signal' > msf-signal.cap
```

Figure 1 Filtering DCF and MSF signal data from the log of the black box solution

My initial intention was to write a testbench for the sync components which could read these signal files directly and generate stimuli from this, however I was unable to find a way to parse the hexadecimal timestamps and signal values from the file directly, so it was necessary to write a program to act as an intermediary between the signal data and the data that the testbench would use as input. Appendix A (page 6) contains my solution for this problem: a small Python script which parses the input signal data and generates a set of testbench stimuli in the form `<clk > <di>` which could be read by the testbench and assigned directly to the component's `clk` and `di` input ports. In order to minimise the size of the stimuli files, the `clk_freq` of the component under test was stepped down from 125 MHz to 100 Hz, meaning that there was only 200 input stimuli per second (one entry per `clk` rising and falling edge), rather than 250 million.

```
$ head logs/dcf-signal.cap
0000016343D8E928 01 FE -- DCF Signal
0000016349A66B50 01 7F -- DCF Signal
000001637F59E718 01 C0 -- DCF Signal
000001638B2CC618 01 01 -- DCF Signal
00000163BADFFEB8 01 F0 -- DCF Signal
00000163C69A9068 01 7F -- DCF Signal
00000163F6A8AAC8 01 80 -- DCF Signal
0000016402 49BAB8 01 0F -- DCF Signal
00000164324E25A8 01 FF -- DCF Signal
000001643DD43478 01 07 -- DCF Signal
$ head dcf_sync_tb-stimulus.txt
0 11111110
1 11111110
0 11111110
1 11111110
0 11111110
1 11111110
0 11111110
1 11111110
0 11111110
1 11111110
```

Figure 2 An extract of the DCF signal and generated test stimuli

Testbenches

Once the DCF and MSF testbench stimuli were generated, the testbench processes needed only to read the files line by line and assign the `clk` and `di` signal values from each line. The implementation of each testbench is included at the bottom of the component's VHDL file, and Figure 3 shows the additions to the Makefile that were required to simulate and view the results.

```
TESTBENCHES = dcf_sync_tb msf_sync_tb ddrserdes_tb
VIEW_TB := dcf_bits_tb

all simulation sim:
	@for f in $(SOURCES); do \
		if [ -f $$f ]; then \
			echo "ghdl -a $$f"; \
			ghdl -a $$f; \
		fi; \
	done
	@for t in $(TESTBENCHES); do \
		echo "ghdl -e $$t"; \
		ghdl -e $$t; \
		echo "ghdl -r $$t --vcd=$$t.vcd"; \
		ghdl -r $$t --vcd=$$t.vcd; \
	done

view:
	gtkwave $(VIEW_TB).vcd $(VIEW_TB).sav >/dev/null 2>&1
```

Figure 3 Makefile targets for simulation and testbenches

Synchronisation components

The `dcf_sync` component uses two numerical signals as counters: the `cnt` signal counts the number of clock cycles since the start of the last second, and the `sec` signal counts the current second within each minute. In addition, two out-of-range second values are used: if the `sec` counter is set to 62, then the component is totally unsynchronised, i.e. it has just been reset, and hasn't detected a rising edge yet. Once the first rising edge is detected, the `sec` counter is set to 61, which means that the component is synchronised to a second, but it doesn't know where that second is within the minute. Once a missing 59^{th} second has been detected, the `sec` counter iterates within its valid range of 1-60, and is used to predict when to add in the second out signal on the 59^{th} second of each minute.

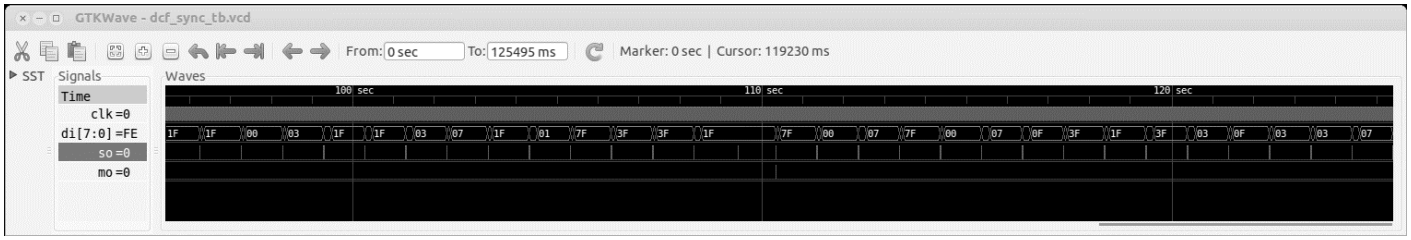

Figure 4 Testbench for the DCF sync component

Since the bits within the sampled data input byte are arranged in chronological order from least significant bit to most significant bit, a rising edge will have more bits set of a higher significance, and a falling edge will have fewer. Because of this, detection of rising and falling edges on the data input is done simply by performing a numerical greater than or less than comparison against the last sampled data input.

Additional logic is largely focused on minimising the chance of false positives generated by noise on the data input. Each second, there is a "window of opportunity" in which we expect to receive a start of second rising edge, and the `clk` counter has a reset value at which it assumes that the signal has been lost or that it is incorrectly synchronised, and causes the state to be reset. Outputting the missing 59^{th} second is done first by detecting a missing second (i.e. wait until the end of the window of opportunity and then assume that this is the 59^{th} second) and afterwards by prediction (by counting the seconds within the minute, it is possible to output the missing second signal exactly one second after the 58^{th} signal within the minute).

The `msf_sync` component is largely identical to the DCF component, however the logic for detecting the missing 59^{th} second has been replaced with logic which detects a 500ms pulse and then uses this to retrospectively determine the rising edge of the pulse as being the start of the minute.

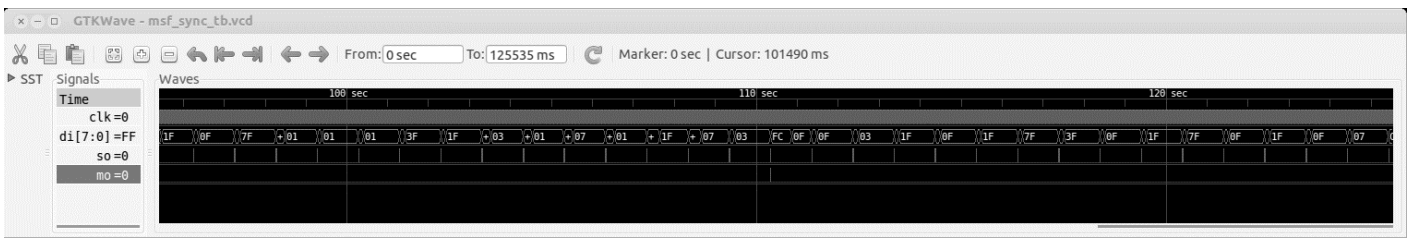

Figure 5 Testbench for the MSF sync component

Double Data Rate SER/DES

The double data rate logic is much simpler than in the DCF and MSF synchronisation components as it is not necessary to handle as many different states, and no external data was required to create the testbench stimulus. The `generate_ddrserdes_stimuli()` function on page 7 generates the input stimuli, which consists of values for the two clock inputs, and a randomly assigned `data_in` value.

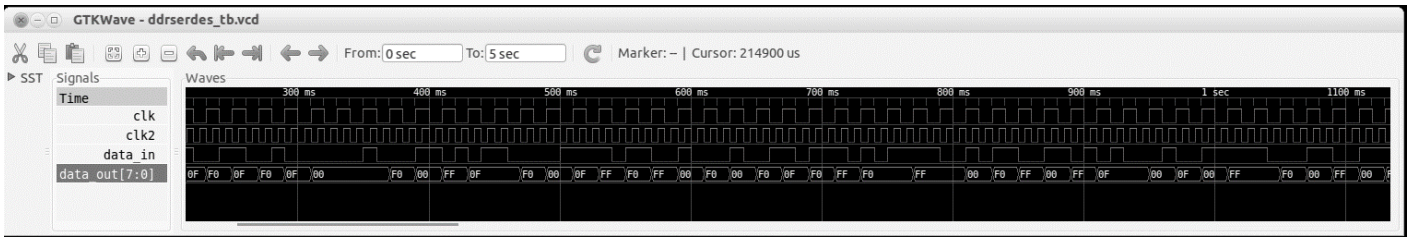

Figure 6 Testbench for double data rate serialiser and deserialiser

Conclusions

The implementations for each component behaves as expected under both simulated and synthesised conditions, however I suspect that my solutions for the MSF and DCF synchronisation components are overly complex, and this is indicated by their relatively large SLOC counts compared to the model answers, and the number of complex conditional and branching behaviour used. A lot of the logic within these components could be removed if we were assume that there was no chance of noise spikes (such as if the MSF and DCF signal inputs were low-pass filtered to around 12 Hz), and further logic could be removed if it was not required that the minute out signal must be given at the exact start of the minute, and if the DCF sync component was not required to add the missing the 59^{th} second signal.

The submitted source code for the MSF and DCF components represent my second attempt at solutions. My initial approach to developing the components was to try to abstract the logic into two processes as in the trigger component example, one for immediate logic and the other state logic; however, this was not successful and the limited time within the laboratory sessions forced me to adopt this single process approach. On a more positive note, the infrastructure for generating testbench stimuli will be useful in later development, and can be re-used when working on the `dcf_bits` and `msf_bits` components.

Appendix A – tb.py

```
#!/usr/bin/env python
#
# Generate testbench stimuli for coursework 2.
#
# WARNING:
#   Make sure that the clk_freq set here matches the clk_freq of your dcf and
#   msf VHDL components!

from itertools import cycle
from random import randint

clk_freq = 100 # Clock frequency (Hz)

def hex2int(hex):
	return int(hex, 16)

def hex2bin(hex):
	return "{0:b}".format(hex2int(hex)).zfill(8)

def process_file(input, output):
	file = open(input)
	out = open(output, "w")
	clk_period = 1000000000 / clk_freq # Clock period (ns)

	# Initialise starting state
	line = file.readline()
	components = line.split(' ')
	time_curr = hex2int(components[0])
	time_next = time_curr

	di_curr = hex2bin(components[2])
	di_next = di_curr

	clk_iter = cycle(range(2))

	line = file.readline()

	# Process all lines
	while line:
		components = line.split(' ')
		time_next = hex2int(components[0])
		di_next = hex2bin(components[2])

		while time_curr < time_next:
			clk = clk_iter.next()
			out.write(str(clk) + " " + str(di_curr) + "\n")
			time_curr += clk_period / 2

		di_curr = di_next
		time_curr = time_next

		line = file.readline()

	out.close()
```

```
def generate_ddrserdes_stimuli(output):
	out = open(output, "w")

	clk_iter = cycle(range(2))
	clk2_iter = cycle(range(2))
	clk = 0

	for i in range(0, 1000):
		clk2 = clk2_iter.next()

		if clk2 % 2 == 0:
			clk = clk_iter.next()

			di = randint(0, 1)

		out.write(str(clk) + " " + str(clk2) + " " + str(di) + "\n")

	out.close()


if __name__ == "__main__":
	process_file("logs/dcf-signal.cap",
			"dcf_sync_tb-stimulus.txt")
	process_file("logs/msf-signal.cap",
			"msf_sync_tb-stimulus.txt")
	generate_ddrserdes_stimuli("ddrserdes_tb-stimulus.txt")
```