304-487 Computer Architecture Laboratory

Assignment #2: Harmonic Frequency Synthesizer and FSK Modulator

Introduction

In this assignment, you are going to implement two designs in VHDL. The first design involves a digital modulation technique whereas the second one is a simple frequency tunable analog waveform generator. The goal of this assignment is twofold. First, it is meant to make you use pipelining to increase the performance of an otherwise poor design. Second, it is meant to give you additional practice with VHDL and an opportunity to gain more experience with the design and synthesis tools, and LPM modules that you will likely be using for your term project.

Background: Binary Modulation - FSK

There are three types of binary modulation techniques: amplitude-shift keying (ASK), phase-shift keying (PSK), and frequency-shift keying (FSK), which involve applying two-level changes to the amplitude, phase, and frequency of a sinusoidal carrier wave, respectively (see Figure 1 below). Note that no prior knowledge of digital modulation is necessary to complete this assignment.

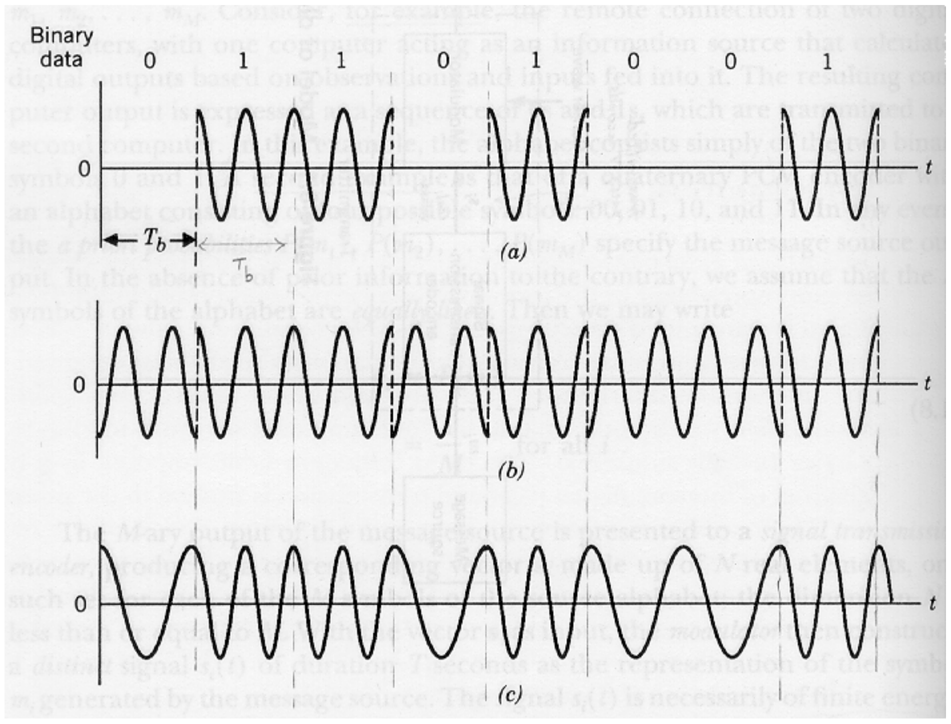

Figure 1: The three basic forms of signaling binary information. (a) Amplitude shift keying. (b) Phase-shift keying. (c) Frequency-shift keying with continuous phase.[1]

In this assignment, you are going to implement an FSK modulator, which will switch between two frequencies. In a binary FSK system, symbols 1 and 0 are distinguished from each other by transmitting one of two sinusoidal waves that differ in frequency by a fixed amount. A 10/11-MHz FSK Modulator is shown in Figure 2. You will now be walked through each of the steps required to implement the FSK modulator.

[1] Communication Systems 3rd ed., Simon Haykin.

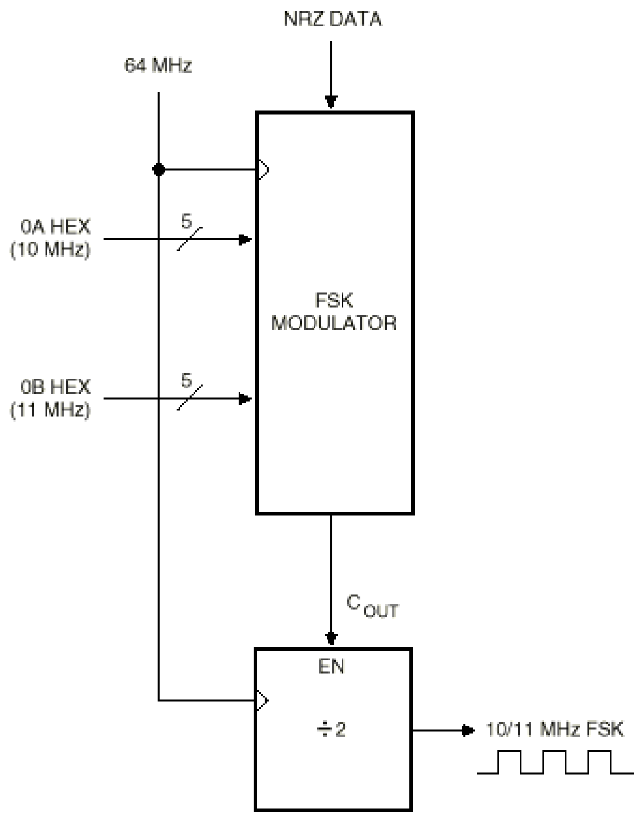

Figure 2: 10/11-MHz FSK Modulator

First, let's review a bit of theory. An FSK modulator is a modification of a *harmonic frequency synthesizer* (to be defined shortly) that automatically switches between two frequencies in accordance with an NRZ (non return to zero) input. The harmonic frequency synthesizer uses an accumulator technique to generate frequencies that are **evenly** spaced harmonics of some minimum frequency. Extensive pipelining is employed to permit high clock rates.

Most frequency synthesizers derive their outputs by using programmable counters to divide the clock frequency (see Figure 3). This results in a set of attainable output frequencies that are sub-harmonics of the clock, and are defined by the following equation.

Equation 1

$$f_{OUT} = \frac{f_{CLK}}{2^k}$$

where k in this case ranges from 1 to 6.

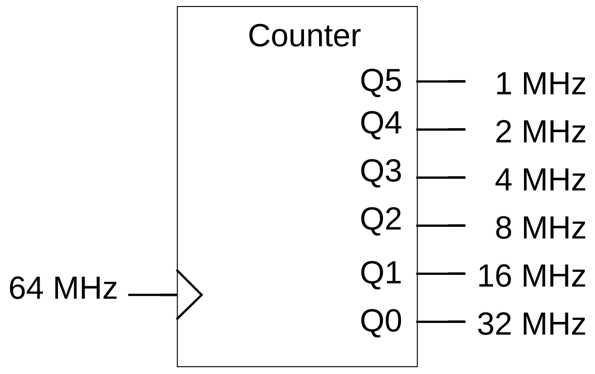

Figure 3: Frequency synthesizer deriving its outputs by using a counter to divide the clock frequency.

A better approach is to use an accumulator to generate the frequencies, as show in Figure 4. This results in a set of harmonic frequencies, defined by the equation:

Equation 2

$$f_{out} = N * f_{CLK}/2^n$$

Here the attainable frequencies are **evenly** spaced. If multiple frequencies are required, the clock need only be a binary multiple of a common factor of the frequencies. This requirement is often easier to satisfy. In particular, if the clock rate is a power of two, all **integer** frequencies up to the clock rate can be generated.

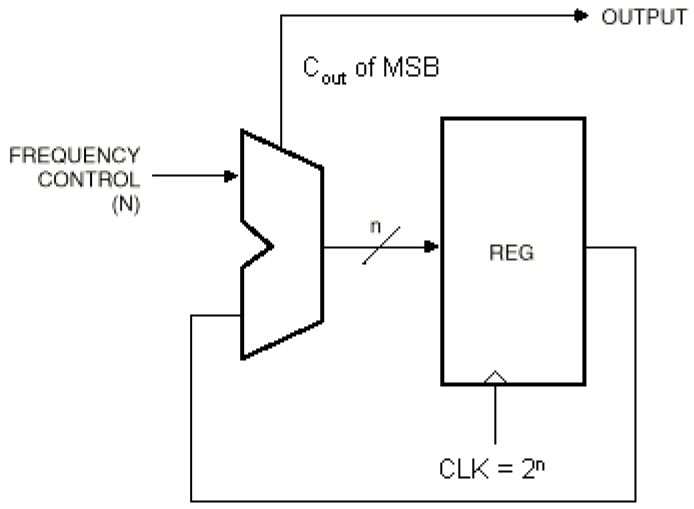

Figure 4: Accumulator-based Frequency Synthesizer

As an example, a 26-bit (i.e., n = 26 in Figure 4) frequency synthesizer, clocked at 67.108864 MHz (2^{26} Hz), generates every **integer-valued** frequency up to this clock rate. Refer back to Equation 2 and make sure you understand the example just outlined before you proceed any further.

We just mentioned that by choosing the appropriate clock frequency, all integer frequencies up to the clock rate can be generated. It must be recognized, however, that these frequencies describe the average rate at which output pulses are generated. Output transitions can only be generated on an integer number of clock periods apart, and this leads to jitter. As the output frequency approaches the clock rate, this jitter becomes severe. Hence, always make sure to chose a clock rate that is big enough to minimize jitter.

Exercise 1: Implement in VHDL the design shown in Figure 4 with n = 5 and a clock frequency of 64 MHz. With these values, we will be able to generate any even integer frequency up to 32 MHz – refer back to Equation 2 to convince yourself that this is right. Note that a bottom-up approach should be used.

1. First you should implement a ripple-carry adder **behaviorally** using only PRIMITIVES (AND, OR, XOR, ...) Yes, do it the hard way! You might find Figure 6 useful). You are NOT allowed to use an LPM module for the adder. The logic equations of a full adder are

   $S = C_{in} \oplus A \oplus B$

   ❖ $C_{out} = C_{in} \bullet (A + B) + A \bullet B$

   Please do not use the pipelining technique yet to implement the ripple-carry adder as this is the purpose of the next exercise.
2. Implement a 5-bit register. It is up to you to code it behaviorally or structurally.
3. Integrate your two designs in one file using component instantiation.
4. Simulate your design and show in the report that frequencies of 2 MHz, 4 MHz, 20 MHz, 22 MHz and 32 MHz can be generated from a 64 MHz clock. Note that the duty cycle of your waveforms won't be 50-50. That is fine for this exercise. You will find out later how to solve this problem. Show that jitter occurs when you attempt to generate frequencies closer to 64 MHz.
5. Run a timing analysis on your accumulator-based frequency synthesizer and report the maximum operational frequency for your design. You may find it helpful to refer to the video tutorial and simple SDC file example listed on the assignment web page.

Problem with the frequency synthesizer of exercise 1

A potential disadvantage of the scheme outlined in Figure 4 is the complexity of the adder and its effect on speed, when compared to the counter approach. However, this can be overcome through the use of pipelining. In the second exercise you should find out that pipelining the design implemented in exercise 1 increases the registered performance and allows higher frequencies to be generated.

The reason why the design of exercise 1 is so slow is because the adder requires serial propagation of the carry outputs from the lowest-order stage to the highest order stage (see Figure 6). The "rippling" of the carry from one stage to the next determines the adder's ultimate delay.

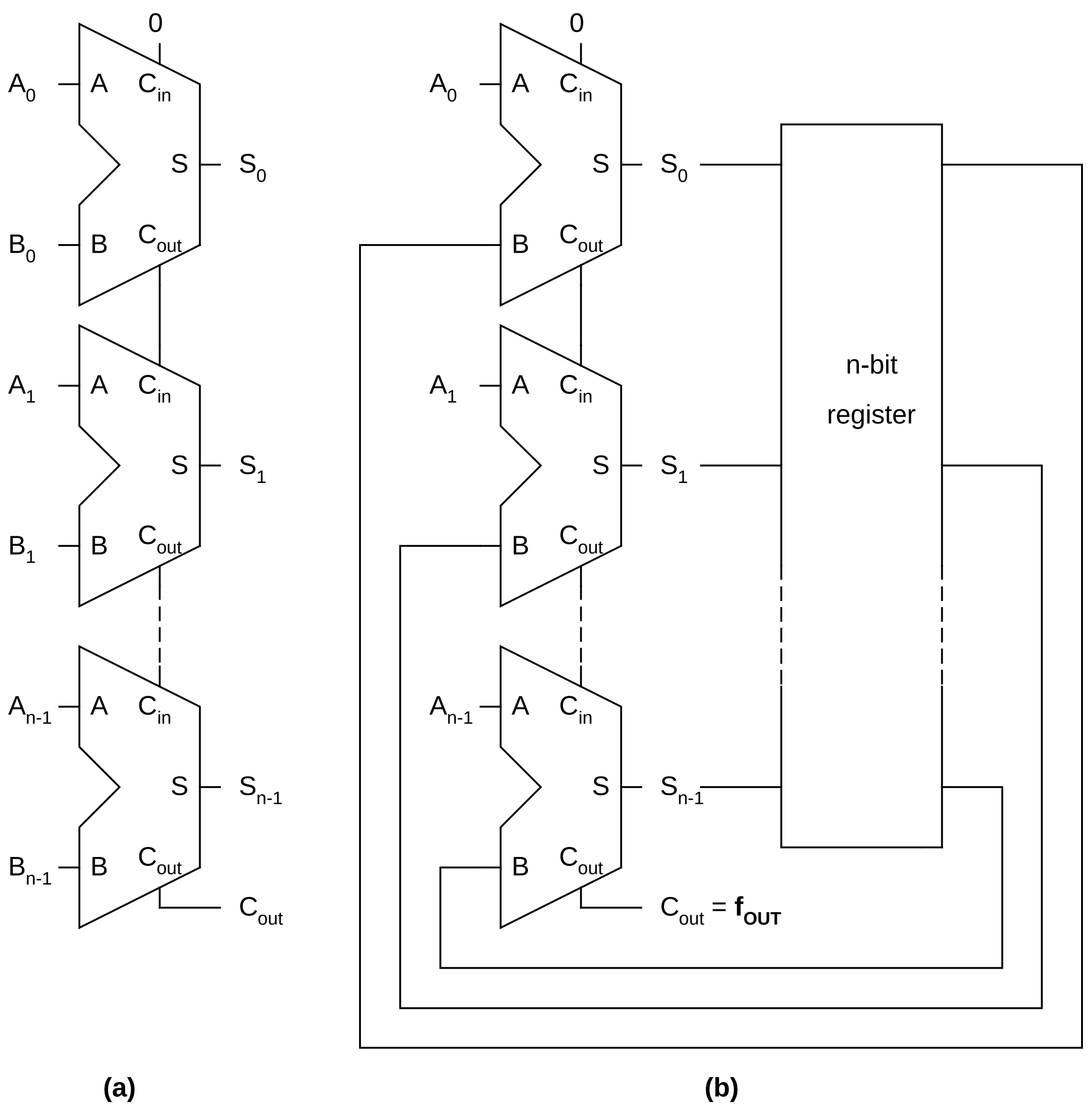

Figure 6: (a) n-bit full-adder w/o overflow detection (b) n-bit accumulator

In order to solve the long delay problem with the adder, a pipeline flip-flop is inserted between all its bits in the carry path. As can be seen on Figure 6 (b), the output skew this creates is not a problem as only the carry-out (f_{OUT}) is of interest.

Matching the pipeline delay at the input is also not an issue if only one frequency is required, as the input never changes. If multiple frequencies are required (two in the case of an FSK modulator), the input might simply be changed, but this would cause a phase discontinuity. Where this is unacceptable, a delay equalizer must be added, such that each addition into the accumulator is completed with the same input.

Conceptually, this requires a triangular array of registers, generating a 1-clock delay into the input of the second bit, a 2-clock delay into the third bit, and so on. However, this can be simplified greatly if the input only changes occasionally.

Figure 7 shows the accumulator cell with its delay equalizer. The accumulator cell is a simple full adder with its output registered and fed back to one of its inputs. A pipeline flip-flop is introduced into the carry path.

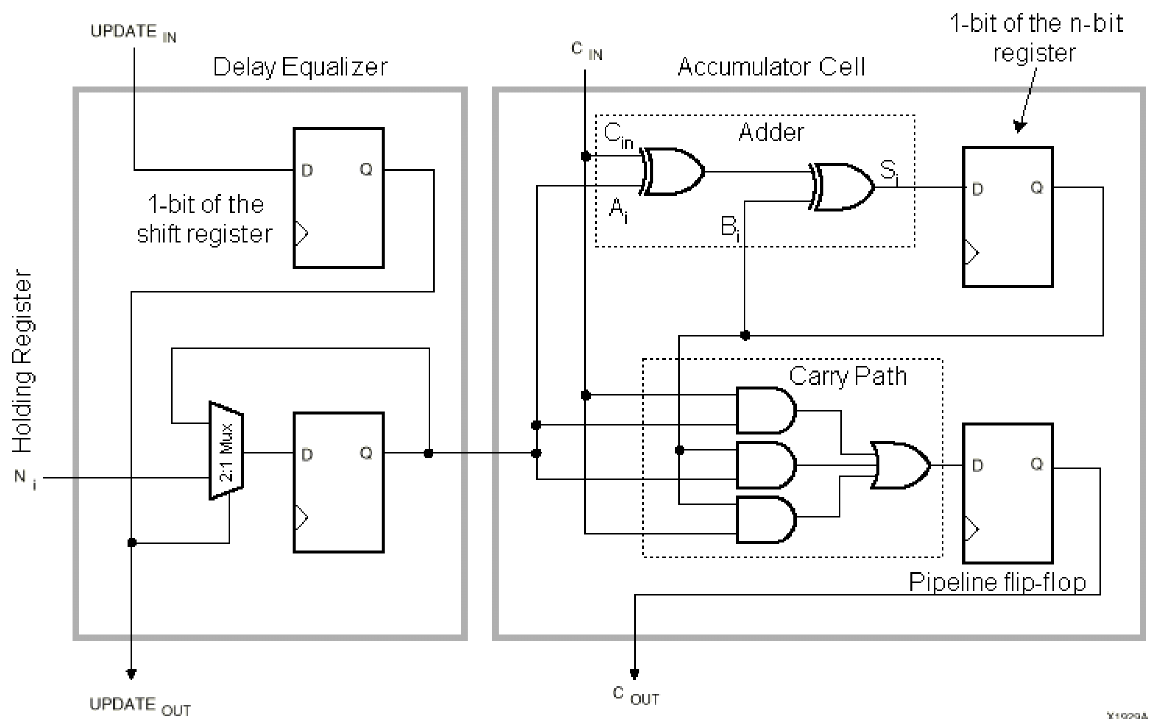

Figure 7: Bit-slice of the Frequency Synthesizer

The accumulator input that controls the frequency is stored in a register. Individual bits of this holding register are enabled from flip-flops that are connected as a shift register. When the frequency is to be changed, the appropriate number is input to a holding register (see Figure 7), and **a single '1'** introduced into the shift register. As this '1' propagates through the shift register, individual bits of the holding register are successively updated. This update occurs synchronously with an addition propagating through the pipelined adder.

For an n-bit accumulator, the data must be held at the input to the holding register for *n* clocks after the update pulse. This is the only restriction on how fast the frequency can be changed. Also, it takes *n* clocks from the update pulse before the frequency change is reflected at the output. At this time, however, the change is instantaneous and **phase continuity is maintained**.

Exercise 2: Implement in VHDL a pipelined version of the design built in exercise 1. Follow Figure 7 as a guideline.

1. First, you should implement the delay cell. Don't worry about the holding register at this point since you are only going to add it in your top level VHDL file.
2. Starting from your full adder design of exercise 1, modify it so that the register is incorporated into the cell (see Figure 7). You should also pipeline your adder by adding a pipeline flip-flop in the carry path.
3. Once you have simulated the delay equalizer and the accumulator cell and made sure they are both working, you can instantiate them, together with the holding register, in a file called bit-slice.vhd (or any other relevant name).
4. The last step will be to instantiate your bit-slice.vhd component five times in another file in order to come up with a pipelined version of the accumulator-based frequency synthesizer shown in Figure 4.
5. Simulate your design and show in the report that frequencies of 2 MHz, 4 MHz, 20 MHz, 22 MHz and 32 MHz can be generated.
6. Run a timing performance on your pipelined version of the accumulator-based frequency synthesizer and state in your report the frequency obtained. This frequency should be higher than what you obtained in exercise 1.

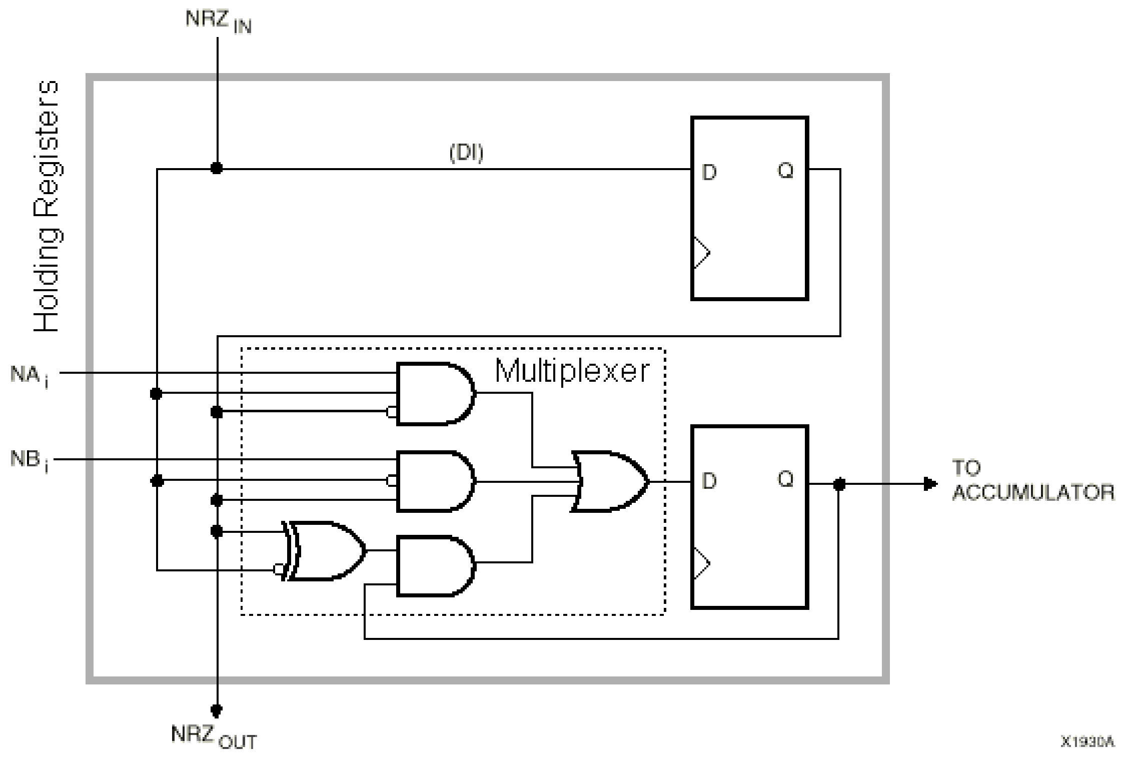

Figure 8: Delay Equalizer for an FSK Modulator

At this point, you should have a working pipelined harmonic frequency synthesizer. In the rest of the assignment, you will use your unit to implement an FSK Modulator and an Analog Waveform Generator.

10/11-MHz FSK Modulator

For FSK modulation, the synthesizer must alternate between two frequencies. This can easily be accommodated by modifying the delay equalizer, as shown in Figure 8.

Two numbers, appropriate to the two frequencies, are applied to the delay equalizer. If the frequencies must be programmable, these numbers can come from inputs or registers. In our case, they are going to come from two holding registers.

NRZ data is applied to the shift register. As this propagates through the shift register, multiplexers at the input to each bit of the holding register detect changes in the data. When a change is detected, the bit is reloaded from the appropriate number. Again, changes ripple through the holding register synchronously with the additions. The NRZ data may change every clock, if required.

A typical FSK modulator, as shown in Figure 2, might be required to switch between 10 and 11 MHz. To give a square output, a flip-flop is used to divide the synthesizer output by two. This flip-flop may be the carry pipeline of the final adder, modified to toggle with the carry rather than storing it.

The toggle flip-flop must be enabled at frequencies that are twice the output frequencies. The largest common factor of 20 and 22 is 2 MHz, so the clock frequency must be a binary multiple of this. Higher binary multipliers will result in lower jitter. In this case 64 MHz is chosen. This is 2^5 times 2 MHz, and a 5-bit accumulator must be used. Twenty and 22 MHz are 10 and 11 times 2 MHz, respectively, so these are the numbers that must be accumulated to generate the frequencies (0A Hex and 0B Hex).

Exercise 3: Implement in VHDL the 10/11-MHz FSK Modulator shown in Figure 2. Make sure you show all your simulations.

1. You should be able to demonstrate that your FSK Modulator can switch between a 10 MHz waveform and an 11 MHz waveform.
2. Show that there is no phase discontinuity when switching from one frequency to another.
3. Show that your waveforms exhibit a 50-50 duty cycle.

Analog Waveform Generator

If an analog output is required, you can use the unit that you built in exercise 2 to control a counter, as shown in Figure 9. The output of this counter is used to access a look-up table, which provides data to a DAC (which you won't implement). RAM or ROM can be used to implement the look-up table (LUT) internally. As a result, multiple wave-shapes (sine, cosine, triangular) can be supported by reconfiguring the RAM/ROM.

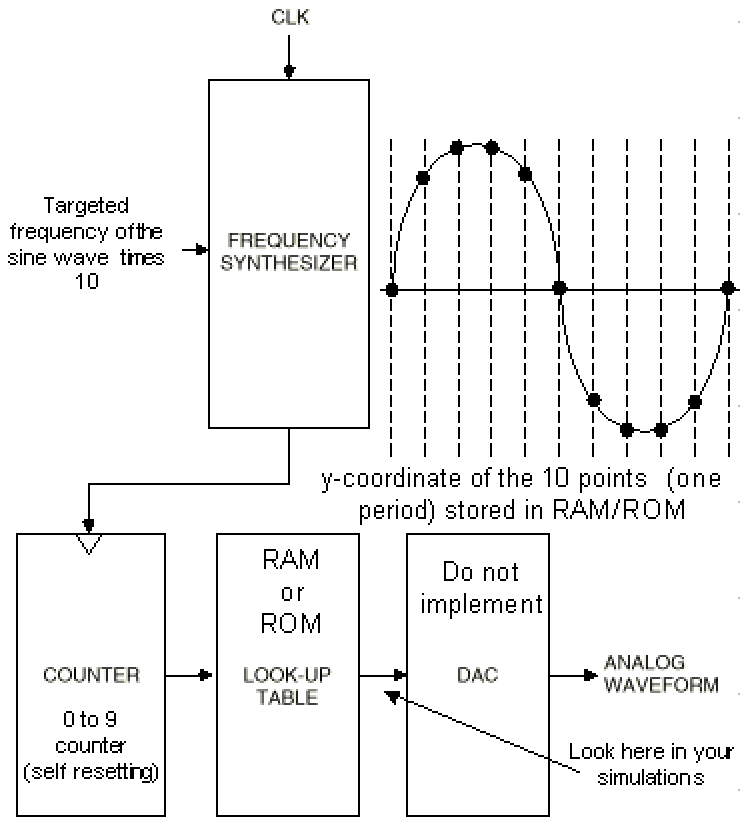
CLK
Targeted frequency of the sine wave times 10
FREQUENCY SYNTHESIZER
y-coordinate of the 10 points (one period) stored in RAM/ROM
COUNTER
0 to 9 counter (self resetting)
RAM or ROM
LOOK-UP TABLE
Do not implement
DAC
ANALOG WAVEFORM
Look here in your simulations

Figure 9: Analog Waveform Generator

Exercise 4 (Bonus): Implement in VHDL the Analog Waveform Generator shown in Figure 9. Make sure you show all your simulations.

1. You should be able to demonstrate that your Analog Waveform Generator can generate a sine wave of the appropriate frequency by loading the RAM/ROM with the proper values. RAM/RAM LPM modules can be loaded with default values upon power up through the use of *.mif* files. Look at the details of Figure 9. You will need to draw a sine wave and determine the y-coordinates of the 10 points approximating one period of the sine wave. For simplicity, use integer values only. No marks will be deducted for weird sine waves! If you you are asked for a sine wave of frequency 10 MHz, you should set the input of the frequency synthesizer to 100 MHz (i.e. 10 times the targeted frequency of the sine wave. This is so because each period of the sine wave is approximated with 10 points).

Laboratory Report

The report should consist of the following sections:

1. A front page (course number, lab title, student names, IDs, emails, date). This page must also indicate clearly which student submitted the electronic copy of the assignment (only one submission of both paper and electronic copy per group).

2. A second page containing the title and the student IDs of each member of your group, with the ID of the student who is submitting the electronic version clearly underlined. As the assignments are being marked by blind review it is imperative that your names do not appear on this page.

3. Lab description **--** in your own words, what is the purpose of this assignment?

4. A design description.

5. Documented VHDL source code for all entities.

6. Simulation results (traces) and any other necessary documentation to demonstrate that your design operates correctly. Please include documented vector files if they are used. Make sure your simulations sufficiently cover all significant cases. Each simulation should be legible and briefly described.

Grading Scheme

Yes, it is possible to obtain a mark greater than 100%. The breakdown is as follows:

- Exercise 1: 35%
- Exercise 2: 35%
- Exercise 3: 30%
- Exercise 4: 20%

Each of the exercises will be graded according to the following criteria:

- Simulation results & documentation – **most important**!
- Answers to questions posed
- Modularity of your design
- Clearly documented VHDL code
- Presentation