Development of a DDS-FPGA based frequency source for quantum optics experiments

Konrad Beck

Bachelorarbeit in Physik
angefertigt im Institut für Angewandte Physik

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität
Bonn

Dezember 2024

I hereby declare that this thesis was formulated by myself and that no sources or tools other than those cited were used.

Bonn, 04.12.2024 [illegible]

Date Signature

1. Reviewer: Prof. Dr. Sebastian Hofferberth
2. Reviewer: Dr. Matthias Frank

Contents

CHAPTER 1

Introduction

Radio-frequency waveform generators are indispensable tools in experimental physics, providing researchers with the ability to create precise and programmable signals required for a wide range of applications. These devices are essential in driving experimental setups, performing frequency sweeps, and modulating signals in real-time. Despite their importance, commercially available solutions often come with significant drawbacks, including limited flexibility, high costs, and challenges in meeting specific experimental requirements. This has motivated the development of custom-built solutions tailored to the unique needs of advanced physics experiments.

In this thesis, a custom waveform generator is designed and implemented by controlling a commercial Direct Digital Synthesizer (DDS) board. The DDS board operates at high speeds, allowing for the generation of almost arbitrary waveforms. Control of the DDS board is achieved through the use of a Field-Programmable Gate Array (FPGA), which is programmed specifically for this purpose. FPGAs offer distinct advantages in such applications, as they can reliably perform simple tasks at extremely high speeds and are capable of handling complex timing and control requirements.

The device developed in this thesis is designed to serve as a versatile waveform generator, with its primary use case being in the Ytterbium experiment conducted by the Nonlinear Quantum Optics (NQO) research group at the University of Bonn. In this experiment, the radio frequency output of the FPGA-controlled DDS system is intended to drive an Acousto-Optical Modulator (AOM), which is used to broaden the frequency of a laser. This frequency-broadened laser is a critical component for laser-cooling ^{174}Yb atoms in a Magneto-Optical Trap (MOT)[1], a fundamental step in preparing atoms for Rydberg experiments.

An FPGA-DDS-based solution is already in use within the Ytterbium experiment [2]. However, the current setup has limited functionality and lacks proper documentation, making it less reliable and harder to maintain or adapt for future needs. Despite these limitations, the existing system has enabled the successful trapping and laser cooling of Ytterbium atoms [3]. The success of this existing setup demonstrates the potential of using an FPGA-controlled DDS system in high-precision experimental scenarios.

This thesis begins by outlining the theoretical principles underlying FPGAs and DDS boards, providing the foundational knowledge necessary to understand the system's design. The main focus lies on the programming and implementation of the FPGA, detailing the methods and challenges involved in achieving precise control of the DDS board. The final section of the thesis discusses how specific signals can be reproduced, demonstrating its practical applications and flexibility for implementation in advanced physics experiments.

CHAPTER 2

Theory

As mentioned in the Introduction, the goal of this project is to build a waveform generator by controlling a DDS board using a custom programmed FPGA. This chapter provides a theoretical foundation for the project, detailing the operation of the AD9959 DDS board and its modes of operation. Additionally, it introduces the fundamental principles of FPGA architecture and programming, including an overview of Verilog, the hardware description language used in this project.

2.1 The AD9959 DDS Board

A Direct Digital Synthesizer (DDS) is a device that synthesizes an analog waveform with specified characteristics using a digital input clock and wave sample.

The core component of a Direct Digital Synthesizer (DDS) is an addressable memory, such as a Read-Only Memory (ROM), connected to a Digital-to-Analog Converter (DAC). This memory stores the waveform samples that the DDS reproduces, providing a Look-Up Table (LUT) for waveform generation. The phase accumulator is responsible for maintaining a digital representation of the signal's phase, which is incremented at a rate determined by the Frequency Tuning Word (FTW) stored in a dedicated register. The FTW controls the step size of the phase accumulator, thereby setting the output frequency.

The reference clock governs the rate at which the phase is incremented, ensuring a stable and precise output. By adding a constant value to the phase, the DDS can achieve phase modulation. The phase output is then mapped to the amplitude of the waveform via the LUT, which is subsequently converted into an analog signal by the DAC.

Figure 2.1 illustrates a block diagram of a DDS, showing the flow from the Frequency Tuning Word to the generation of a continuous sinusoidal waveform.

In this project, the AD9959 on the evaluation board Z by Analog Devices is used. It features 4 channels and can be interfaced via several Input/Output pins. The clock signal for the DDS must be provided externally. To write to a register, such as the FTW or the POW (Phase Offset Word), an instruction byte must be sent via the Serial Peripheral Interface (SPI) to the DDS board, followed by the value to be written to that register. All the registers and their instruction bytes are detailed in the AD9959 datasheet [4].

There are five pins for SPI communication: one SCLK (clock) pin and four Secure Digital Input/Output (SDIO) pins. This allows for single, dual, or quad SPI communication, depending on the board settings. The maximum SCLK speed is 200 MHz, resulting in a maximum communication speed of 800 Mbit/s when quad SPI is used. To apply newly configured registers, a digital high signal must be sent through the io_update pin on the DDS board, transferring the values from the buffer to the registers. In addition to these pins, there is a reset pin that resets all registers to their default values when set to a logical high.

The AD9959 can internally upscale the external reference clock to the system clock frequency f_{S} using a Phase-Locked Loop (PLL). The scaling factor is set in Function Register 1 (FR1) and can range from 4 to 20; any value lower than 4 results in no upscaling. The resulting system clock frequency is the externally provided reference frequency f_{REF} multiplied by the PLL factor. In addition to the system clock, the AD9959 also features a synchronization clock (`SYNC_CLK`), which is one-fourth of the system clock frequency. When sending an io_update signal, the pulse width has to be longer than one `SYNC_CLK` period to guarantee signal detection.

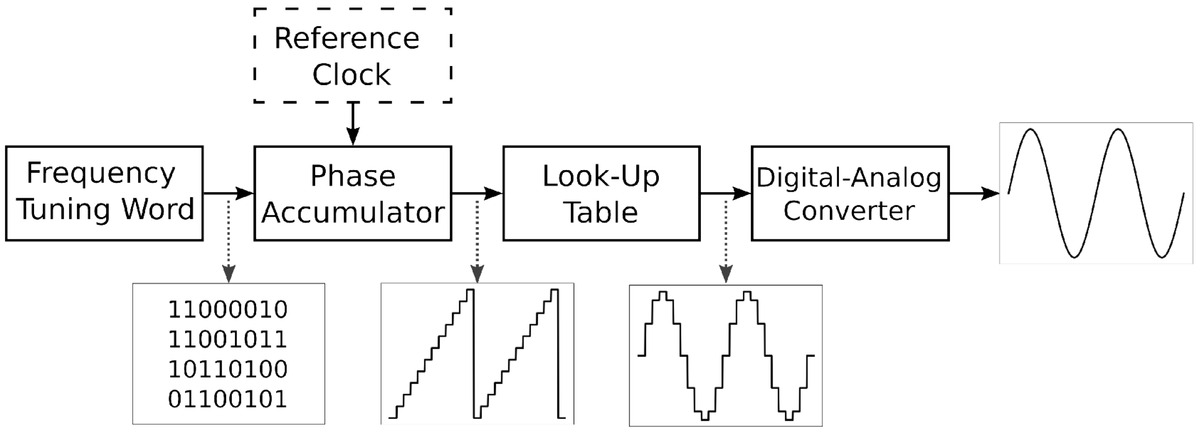


Figure 2.1: Block diagram of a Direct Digital Synthesizer (DDS). The Frequency Tuning Word (FTW) sets the rate at which the phase accumulator increments, controlling the frequency of the output signal. The phase accumulator outputs digital phase values, which are mapped to waveform amplitudes via a Look-Up Table (LUT). These digital amplitudes are then converted into an analog waveform by the Digital-to-Analog Converter (DAC). The reference clock provides timing for the entire process.

Source: [2]

The board supports multiple operating modes, two of which are utilized in this project. These modes are described in the following sections.

2.1.1 Single tone mode

In single tone mode the DDS board will output a sine wave with constant frequency, amplitude and phase. The frequency can be set in the FTW and is proportional to the system clock frequency f_{S}

$$f_{\mathrm{OUT}} = \frac{(\mathrm{FTW}) f_{\mathrm{S}}}{2^{32}}.$$

The phase offset between the channels is set by the POW and is given by

$$\phi = \frac{(\text{POW})}{2^{14}} \times 2\pi.$$

The full scale current of the DAC is given by

$$R_{\text{SET}} = \frac{18.91}{I_{\text{OUT}}(\text{max})}$$

where the resistance R_{SET} on the Evaluation board Z is given by $R_{\text{SET}} = 1{,}91\,\text{k}\Omega$ [4][5].

Phase and frequency can be set individually for each of the four channels.

2.1.2 Linear Sweep Mode

For applications such as frequency broadening, the linear sweep function of the AD9959 is highly useful. This feature allows for ramping frequency, phase, or amplitude in a linear manner. To achieve this, multiple registers must be configured. First, linear sweep mode must be enabled by setting a bit in the Channel Function Register (CFR) to logic high. Additionally, the parameter to be swept must be specified in the same register. Figure 2.2 shows the parameters that need to be set to perform a sweep. The lower boundary, S_0, is stored in the standard register for the parameter being swept (e.g., for frequency, this would be the Frequency Tuning Word or (FTW). The upper boundary, E_0, must be set in Channel Word 1 (CW1).

The slope of the sweep is determined by two values. The Rising/Falling Slope Ramp Rate (RSRR/FSRR) defines the time resolution for the rising or falling sweep. The Rising/Falling Delta Word (RDW/FDW) specifies the step size. The profile pins are used to switch between a rising or falling sweep. A logic high on a profile pin indicates a rising sweep, while a logic low indicates a falling sweep for the corresponding channel. There is one profile pin for each of the channels. This allows for individual configuration of sweeps for all channels.

The formulas to compute the frequency, phase, amplitude and time steps are given by [4]

$$\begin{aligned}
\Delta f &= \frac{(\text{RDW})f_{\text{S}}}{2^{32}} \\
\Delta\phi &= \frac{(\text{RDW})}{2^{14}} \times 2\pi \\
\Delta a &= \frac{(\text{RDW})}{2^{10}} \times 1024 \\
\Delta t &= \frac{(\text{RSRR})}{\text{SYNC\_CLK}}.
\end{aligned}$$

2.2 FPGA theory

This section discusses the fundamental building blocks of an FPGA and explains how they can be programmed. It also provides a brief introduction to the hardware description language Verilog.

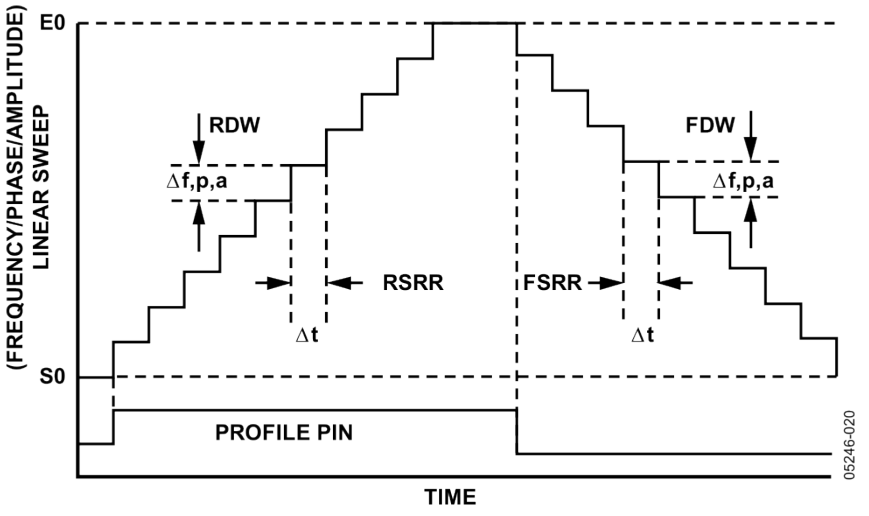


Figure 2.2: Schematic plot of frequency, amplitude, or phase over time in the linear sweep mode of the AD9959. The upper and lower limits are defined by E_0 and S_0. The rising slope of the sweep is determined by the RDW and RSRR, while the falling slope is determined by the FDW and FSRR. The state of the profile pin determines whether a rising or falling sweep is executed.

Source: [4]

2.2.1 Synthesis of Logical Circuits

Field Programmable Gate Arrays (FPGAs) consist of Configurable Logic Blocks (CLBs) that can be interconnected through routing channels and switch matrices, similar to regular gate arrays. However, unlike regular gate arrays, which are configured during manufacturing, FPGAs can be reconfigured through programming. This configuration allows the logic blocks and switch matrices to be arranged in any desired manner, enabling the creation of complex logic circuits in ‘the field’—hence the name. Unlike microcontrollers, where software configurations are primarily adjusted, FPGAs allow for the reconfiguration of the hardware itself. In microcontrollers, different processes may compete for computing resources, potentially leading to bottlenecks. In contrast, FPGAs enable fully parallel logic circuits, allowing multiple computations to be performed simultaneously.
The CLBs are interconnected through routing channels. The Input/Output blocks (I/O blocks) can be programmed as unidirectional or bidirectional interfaces between the FPGA and external devices. A schematic of such an array is shown in Figure 2.3.

Figure 2.4 shows a CLB from the company Xilinx as an example. It consists of multiple Look-Up Tables (LUTs), which are common for most FPGAs. The LUTs allow the realization of any binary function by writing specific values in each table entry. The entries, and thus the function, of the LUT can be configured by writing to memory such as Static Random Access Memory (SRAM) or Erasable Programmable Read-Only Memory (EPROM). This makes FPGAs easily programmable by writing to simple memory blocks. By using LUTs to realize the binary functions the propagation delay for every possible function does not vary. The LUTs connect through multiple multiplexers to outputs and flip-flops. These multiplexers can be configured to provide the desired signal path inside the CLB.

The CLBs are connected to the routing channels through configurable routing switches, as illustrated in

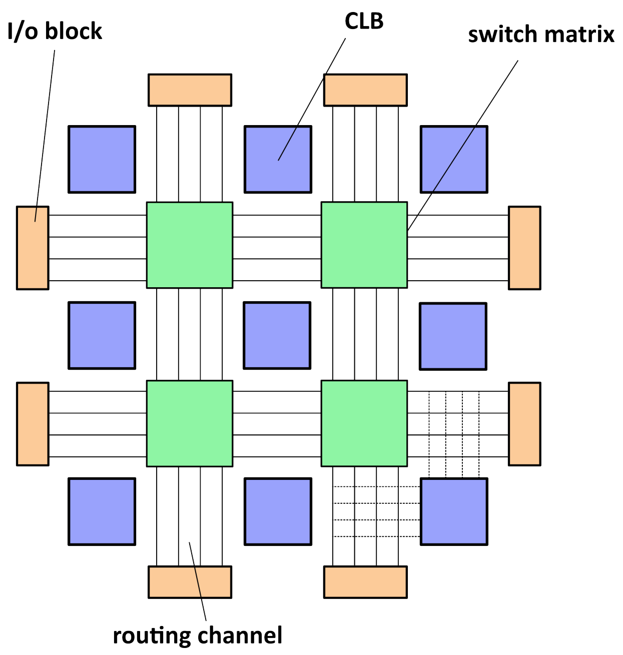


Figure 2.3: Schematic representation of an FPGA showing the arrangement of its components. The routing channels connect to the switch matrices, which together guide the signals to the CLBs and the I/O blocks.

Figure 2.5. These routing switches are implemented using transistors, which can be configured similarly to the CLBs by writing to memory, such as SRAM or EPROM. The horizontal and vertical routing channels are linked by programmable switch matrices that use the same type of routing switches. This design allows for highly flexible logic circuit design and signal routing.

The primary drawback of this architecture is that each programmable connection or routing through a CLB introduces additional signal delay, potentially limiting the operational speed. To minimize this effect on clock signals, most FPGAs feature low-impedance global clock lines that distribute the clock signal to clock buffers located throughout the chip. These clock buffers then provide the clock signals to the CLBs and other clocked elements, significantly reducing clock signal delays where timing is critical [7].

In some FPGAs, connections are created with anti-fuses, which form a permanent connection after a high voltage is applied for a short period. While these connections tend to be more stable, they are irreversible and thus not suitable for prototyping or flexible reprogramming [6, Chapter 14.3.3].

Since SRAM is a volatile memory, all programming is lost when the power is turned off. To address this limitation, many FPGA boards include a programmable memory device, such as an EPROM, which automatically loads the configured program from non-volatile memory into the SRAM upon power-up.

The configurable I/O blocks are used to send signals on and off the board. They consist of an input buffer and an output buffer. The polarity of the output can be set to either active high or active low. Additionally, there are flip-flops on both the input and output to reduce both delay and hold time requirements [7].

The FPGA used in this project is the Artix-7 35T on the CMOD A7 board. The board features 44 digital I/O pins, a USB connector, and an onboard 512 kB SRAM with an access time of 8 ns [9].

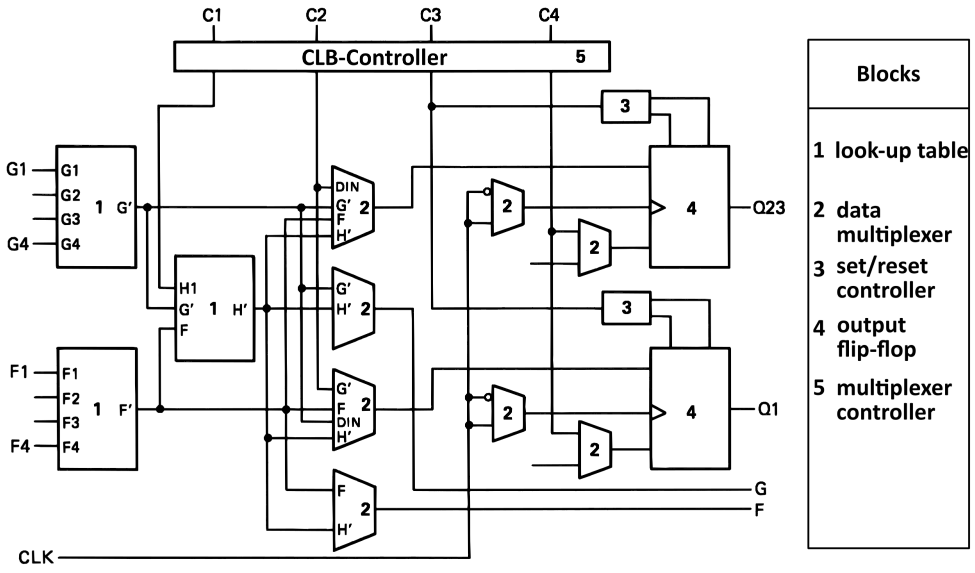


Figure 2.4: Block diagram of a Xilinx CLB. The look-up tables realize the desired logic and can be programmed. The multiplexers route the signals throughout the CLB. The flip-flops enable the option to store the signals and can be directly set and reset by the set/reset controller. The multiplexer controller contains the logic to controll the multiplexers and set/reset controllers.

Adapted from [6, Figure 14.24]

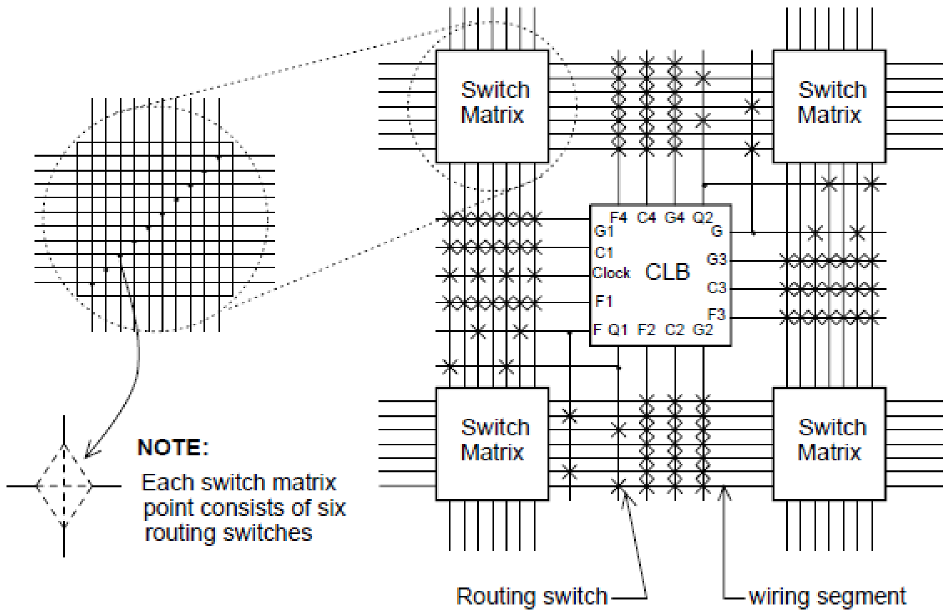


Figure 2.5: Schematic of a Programmable Switch Matrix. The CLBs inputs and outputs are connected to the wiring segment by routing switches that can be programmed by writing to a memory device. Each intersection of the switch matrices can be configured and consist of six routing switches.

Source: [8, Figure 8]

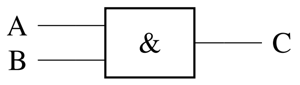
2.2.2 Introduction to Verilog

Manually configuring logic circuits on FPGAs can be very tedious. This is why FPGAs are typically programmed using Hardware Description Languages (HDLs). The HDL code is then processed through multiple steps to create a bitstream, which can be loaded onto the FPGA to realize the desired functionality. The process of constructing a logic circuit from HDL code is known as synthesis.

In this project, the FPGA is programmed using the IEEE-standard hardware description language Verilog [10]. This section will introduce basic Verilog syntax by showcasing a few examples. The examples will display the Verilog code on the left and the synthesized circuit on the right. These circuits represent only one possible way the logic might be synthesized, as the actual realization of the logic is highly hardware-dependent.

In Verilog, the logic is divided into modules, which can be thought of as logic circuits. These modules have inputs and outputs that are connected by the implemented logic. The code below demonstrates how a simple AND gate can be realized.

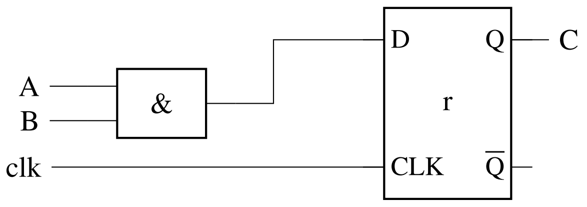
```verilog
module AND(
    input wire A,
    input wire B,
    output wire C
);
    assign C = A & B;
endmodule
```



The `assign` statement defines logic by driving the value on the left side of the statement with the expression evaluated on the right side. The & operator performs a bitwise AND operation on the two operands. This method of creating circuits works well for simple functions. However, when more complex and synchronized logic is required, these continuous assignments may not be suitable. A better option is to use `always` blocks. The example below demonstrates how such a block functions.

```verilog
module AND(
    input wire A,
    input wire B,
    input wire clk,
    output wire C
);
    reg r;
    always @(posedge clk)begin
        r <= A & B;
    end

    assign C=r;
endmodule
```

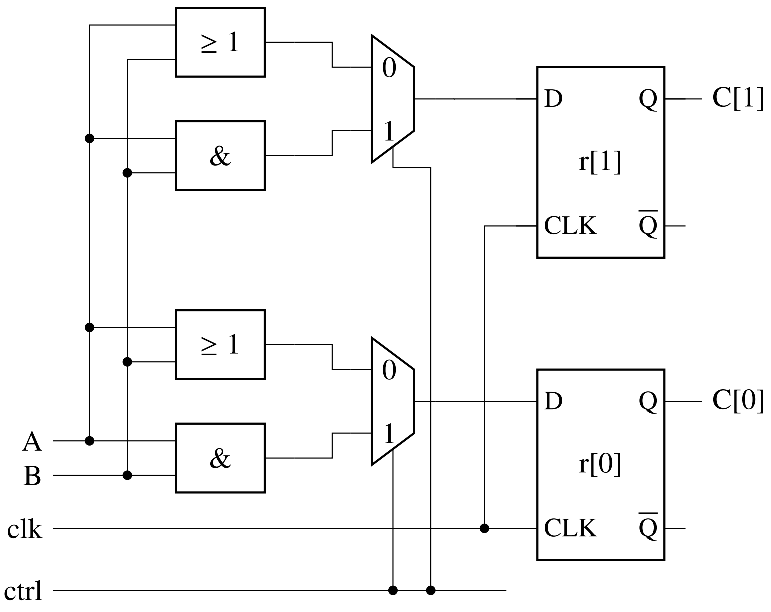


Both examples implement an AND gate. However, in the second example, the output is buffered into a latch that updates only upon detection of a rising edge (`posedge`) of the clock (`clk`) signal. This enables the design of synchronized circuits and sequential logic. The line `reg r;` declares a new 1-bit register. Registers can be defined in any desired size. This register is connected to the output `C` using the line `assign C = r;`. In addition to the AND operator, Verilog provides many different logical and arithmetic operators. A selection of these operators is listed in Table 1.

Conditional statements in Verilog can be used to realize logic based on a condition. An example of an `if` statement is shown below.

```verilog
module AND_OR(
   input wire A[1:0],
   input wire B[1:0],
   input wire ctrl,
   input wire clk,
   output wire C[1:0]
);
   reg r[0:1];
   always @(posedge clk)begin
      if(ctrl) begin
         r <= A & B;
      else begin
         r <= A | B;
      end
   end

   assign C=r;
endmodule
```



In this example, the inputs `A` and `B` are two bits wide. The notation `A[1:0]` refers to a two-bit wire, where the index of the Most Significant Bit (MSB) — the bit in a binary number that encodes the highest positional value — is 1. This essentially denotes in what order the bits are arranged in the register. This indexing also applies to registers. If only a single bit or a range of bits needs to be accessed, this can be done using the notation `r[<index>]` or `r[<index1>:<index2>]`. On the rising edge of the clock signal, the register `r` stores the result of either a bitwise AND or OR operation, depending on the state of `ctrl`. If the expression in the `if` block (in this case, `ctrl`) evaluates to true, the logic within that block is executed; otherwise, the `else` block is executed.

Verilog offers a range of logical operators for constructing more complex expressions within `if` statements. A selection of these operators is shown in Table 2.

The equivalent of the logic described above can also be achieved using conditional `assign` statements. For example, the expression `assign C = (ctrl) ?  A & B : A | B;` creates similar logic but operates asynchronously, like the first example. Conditional assignments are useful when a wire or register needs to be driven by multiple sources. One such example is an SRAM data bus that is driven externally when in write mode, or by the SRAM itself when in read mode.

In addition to `if` statements, Verilog also provides `case` statements, which enable different logic to be executed depending on the value of a given operand. An example of a `case` statement is shown below.

```
module AND_OR(
    input wire A[1:0],
    input wire B[1:0],
    input wire ctrl[1:0],
    input wire clk,
    output wire C[1:0]
);
    reg r[0:1] ;
    always @(posedge clk)begin
        case(ctrl)
            0: begin
                r2 <= A & B:
            end
            1: begin
                r2 <= A | B:
            end
            default: begin
                r2 <= 0;
            end
        endcase
    end
    assign C=r;
endmodule
```

This example realizes nearly the same logic as the previous one. The major difference is that `ctrl` is now a 2-bit wire, allowing it to represent four different states. In states zero and one, the logic behaves just like the previous example. In any other state, `r2` and thus `C` is set to zero. This behavior is achieved using the `default` statement, which is executed when none of the specified cases match.

To improve code readability, parameters can be used. The expression `parameter CONST = 7;` creates an alias for the number seven, named `CONST`. This does not affect the synthesized logic but is very useful for complex logic circuits with many possible states. For example, consider the line `if(state == IDLE) begin`. Defining the parameter `IDLE` once makes the code much more readable than having to remember the numerical values representing each state [10].

CHAPTER 3

Development

3.1 General Approach

The goal of this project is to program the FPGA to control the DDS board in a highly flexible manner. By enabling fast and efficient writing to the DDS registers, the system aims to produce an almost arbitrary function as the DDS output.

The general approach to achieving this goal is to configure the FPGA so that the user can write a sequence of instructions into the onboard SRAM of the CMOD A7. These instructions include tasks such as 'write value `x` to DDS register `a`' or 'wait `n` clock cycles'. Once the writing process is complete, the instructions are executed sequentially, mimicking the behavior of a microcontroller.

This allows for highly flexible control of the DDS and precise timing. The access time of the SRAM is 8 ns [11], which ensures that even at the maximum communication speed of 100 MB/s, the system has sufficient time to read data from memory.

Since each cell of the SRAM is one byte in size, it is logical to design the instructions to also be one byte long. As mentioned in Section 2.1, an instruction byte must be sent to the DDS, containing the address of the register to be written to. A bit field diagram of the DDS instruction byte is shown in Figure 3.1. The most significant bit (MSB), D7, determines whether the register should be written to or read from. A logic low represents a write operation, while a logic high indicates a read operation. Bits D4 to D0 encode the address of the register, while D6 and D5 don't carry any information and are therefore don't-care bits.

The instruction byte for the FPGA is constructed from the DDS instruction byte, which is illustrated in Figure 3.2. In this design, only write operations are performed on the DDS, meaning that the MSB can be used to encode something different. In this case, D7 is used to indicate whether an io_update signal should be sent after the execution of the instruction. Since D6 and D5 are don't-care bits of the DDS instruction byte, all bits from D6 to D0 represent an instruction on the FPGA. The instructions are encoded in such a way that if D6 and D5 are both logic low, the system will treat bits D6 to D0 as a write instruction for the DDS and will output data accordingly. If either D6 or D5 is high, then D6 to D0 represent an internal instruction such as 'wait'.

MSB **LSB**

D7	D6	D5	D4	D3	D2	D1	D0
R/W	-	-	A4	A3	A2	A1	A0

Figure 3.1: Bit field diagram of the instruction byte of the AD9959 DDS chip. The MSB at D7 indicates wether a read or a write operation should be performed. D6 and D5 are don't-care bits. Bits D4 to D0 encode the address of the register.

adapted from [4]

MSB **LSB**

D7	D6	D5	D4	D3	D2	D1	D0
U	I6	I5	I4	I3	I2	I1	I0

Figure 3.2: Bit field diagram of the instruction byte of the FPGA. D7 encodes whether an io_update signal should be sent after the execution of the instruction. D6 to D0 encode the instruction

This design allows for very efficient encoding of when to perform an io_update while still leaving enough space for instructions. In total, there are $2^7 = 128$ instructions, of which $2^5 = 32$ are used as write instructions for the DDS. This means that 96 internal instructions can be programmed.

Alongside the instruction itself, the parameters of the instruction must be stored in memory. This is achieved by starting each sequence at address 0x00 with an instruction followed by the parameter. The size of the instructions is, by design, one byte. However, the size of the parameter might vary. For example, the FTW of the DDS has a size of 4 bytes, while the POW is only 2 bytes long. This means that the system must know how many bytes each parameter consists of in order to be aware of when the next instruction is expected.

An example program, which demonstrates how the planned system should function, is shown in Table 3.1. The table outlines the addresses and contents of the first seven memory cells in the SRAM. The system is designed to execute programs sequentially, starting at address zero and progressing downward.

Address	Byte	Description
0	10000100	Set FTW and io_update
1	00011001	FTW[31:24]
2	10011001	FTW[23:16]
3	10011001	FTW[15:8]
4	10011001	FTW[7:0]
5	00100111	Wait
6	00000101	time

Table 3.1: Memory table of example program. The first column holds the address of the memory. The second column the content at that address. The description column comments, what information the byte represents.

In this design, the first byte at address 0 is intended to be interpreted as an instruction. The value `10000100` stored in this byte specifies two actions. Bits 0–6 (`0000100`) are meant to instruct the system to write to the Frequency Tuning Word (FTW) register of the DDS. Bit 7 (`1`) is planned to signal that an io_update should be sent after the instruction is executed.

The subsequent bytes, located at addresses 1 through 4, are used to store the 4-byte FTW that will be sent to the DDS. The system is designed to recognize that this parameter spans 4 bytes, and therefore, it expects the next instruction to begin at address 5.

At address 5, the second instruction (`00100111`) is planned to represent a `WAIT` instruction. In this case, bit 7 of the instruction is set to `0`, indicating that no `io_update` should be sent after the instruction is completed. Finally, the byte at address 6 (`00000101`) is meant to specify the number of clock cycles (`5`) that the system should wait before resuming program execution.

The system is planned to support five different types of instructions, each serving a specific purpose:

1. **Output:**
   Output instructions are used to write data to the registers of the DDS board, enabling configuration of its operating parameters.
2. **Wait:**
   Wait instructions pause the program execution for a specified number of clock cycles, allowing for precise timing.
3. **Synchronization:**
   Synchronization instructions halt the program until an external signal is detected, enabling synchronization with external systems.
4. **Address Manipulation:**
   Address manipulation instructions are designed to enable advanced program flow functionality, such as jumps, loops, and callable functions.
5. **Pin Control:**
   Pin control instructions allow the system to set or toggle external FPGA pins. These are primarily intended to control the profile pins of the DDS, which are necessary for performing linear sweeps.

When used efficiently, this set of instructions allows the user to create a wide variety of output RF functions, ranging from simple waveforms to more complex modulation schemes. The modular design of the instruction set ensures adaptability to different application scenarios.

The hardware setup used in this project is illustrated in Figure 3.3. The FPGA hardware is configured by the computer, which communicates with the FPGA via a USB to General Purpose Input/Output (GPIO) converter. Once configured, the FPGA serves as the interface between the computer and the Direct Digital Synthesizer (DDS). It provides the reference clock for the DDS, writes to its registers using Quad SPI, and manages the necessary control signals. Based on these inputs, the DDS generates the desired RF signal, which can be observed at its output.

An image of the prototype implementation of the setup is shown in Figure 3.4. The figure illustrates the connections between the various boards and the power distribution circuit, which is powered by a laboratory power supply. The USB to GPIO converter used in this project, is the FT232H. The reference

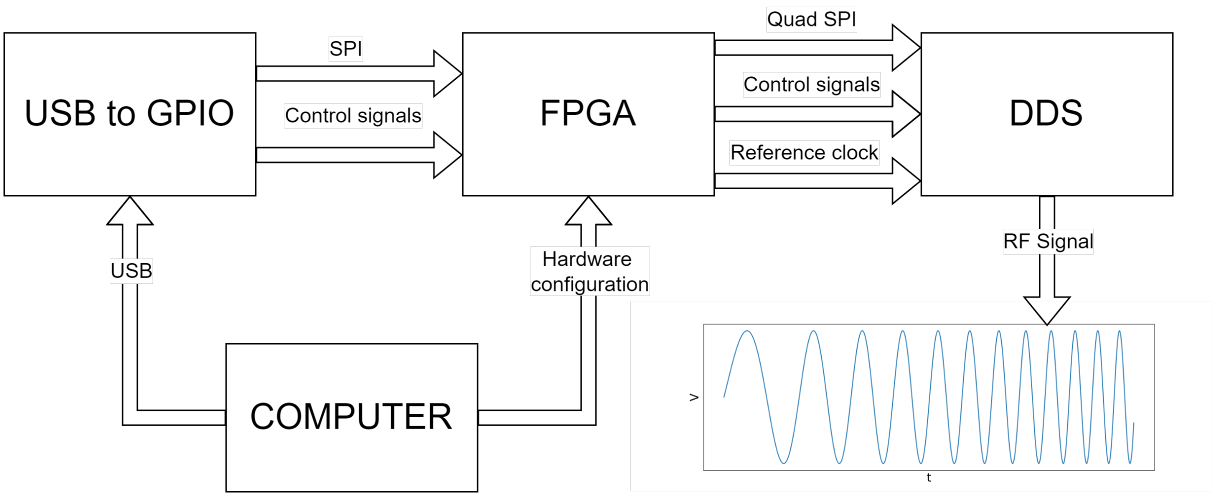


Figure 3.3: Flowchart of the hardware setup used in this project. The figure illustrates the interaction between the computer, FPGA, and DDS. The computer provides the hardware configuration of the FPGA. It also sends data and signals to the FPGA via a USB-to-GPIO converter, enabling the FPGA to act as an intermediary. The FPGA provides the reference clock, control signals, and Quad SPI commands to the DDS, which subsequently generates the desired RF signal at its output. The arrows in the diagram represent the flow of signals and data between components.

clock and analog power for the DDS are supplied via SubMiniature version A (SMA) cables, ensuring signal integrity. The FPGA is connected to the DDS pins using a single ribbon cable, while the digital power supply for the DDS is delivered through the green connector visible in the figure.

To limit current on the control pins, resistors have been added to all active lines. Additionally, an SMA connector has been soldered to relay the trigger signal from the FPGA to other devices such as oscilloscopes.

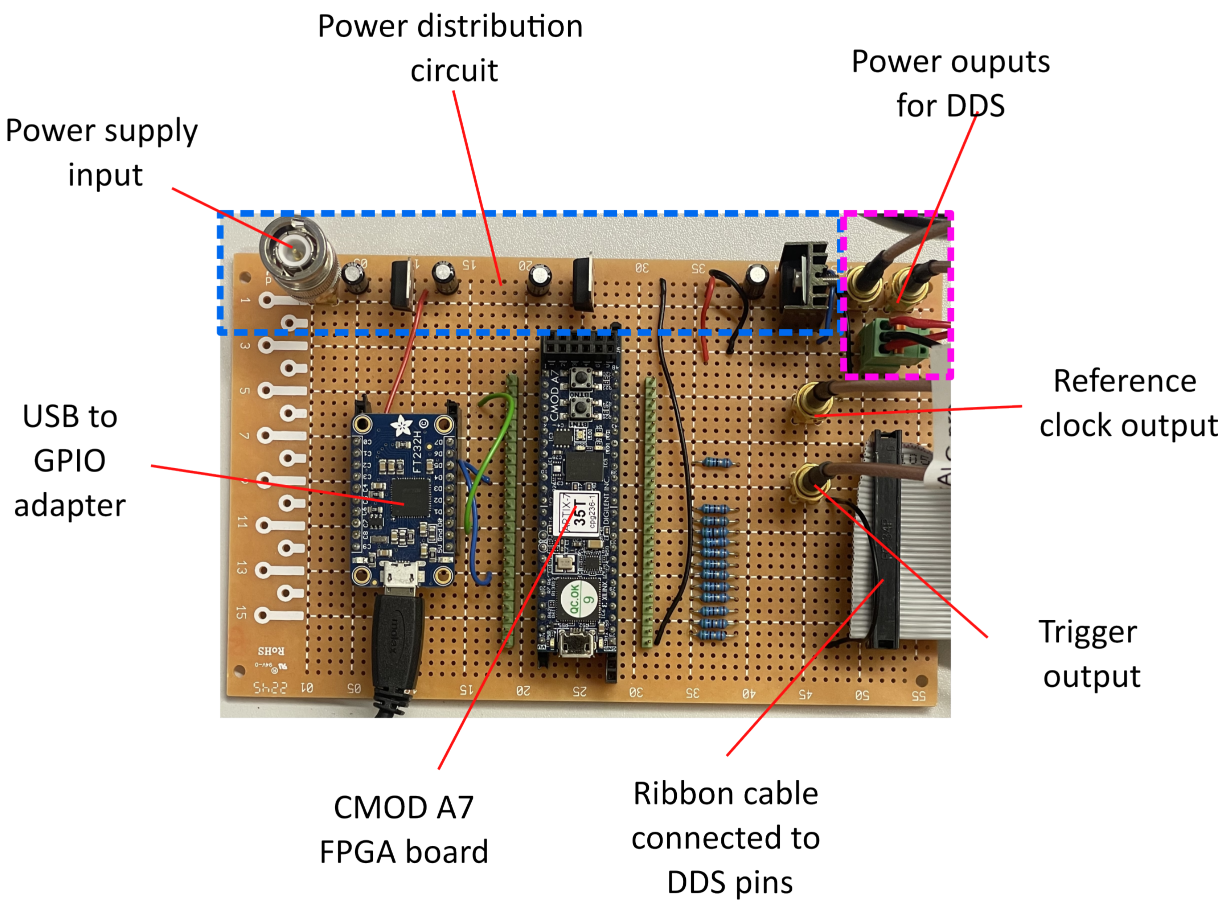


Figure 3.4: Prototype implementation of the FPGA-controlled DDS setup, highlighting the connections, power distribution, and additional features for signal routing and protection.

3.2 Implementation

This section explains how the functionality outlined in the previous section is implemented. It provides an overview of the Verilog modules specifically for this system , their individual roles, and how they interact to achieve the desired system behavior.

3.2.1 Writing process

To generate RF signals, a program must first be created on a computer and written to the SRAM of the CMOD A7 FPGA board. This subsection focuses on the implementation of the logic that processes the inputs and writes the data sent from the computer into the FPGA's SRAM.

A block diagram of the synthesized circuit that processes the inputs is shown in Figure 3.5. The program, transmitted through the USB-to-GPIO converter, is sent via SPI. The `input_spi` module is responsible for reading the SPI data. It receives the external clock signal on the `sclk` input and the data on the `sdio` input. The Verilog code for this module is presented in Listing 3.2.

The `index` register tracks which bit of the word register will be written to next. Whenever a rising edge of the sclk signal is detected, the module writes the value of sdio into the word register at the index. The index register is then decremented by one. When the index reaches zero, the output is updated, and DVLD is set to logic high. Since the index register is only three bit in size, decrementing it when its value is zero causes an overflow and resets it back to seven. When sending data, it is important to ensure that the transmission starts with the Most Significant Bit (MSB) first. If a rising edge of the reset signal is detected, all registers are reset.

The sram_controller handles the inputs and outputs of the FPGA that are connected to the SRAM chip on the CMOD A7 board. When we is enabled, it takes data_in and writes it to the RAM cell located at address_in on a rising edge of the clock. When we is low, it accesses the cell at address_in and outputs it to data_out.

Before a byte can be written to the SRAM, the write enable input (we) must be set to logic high via an external input. This will cause the system to enter the WRITE state, indicated by a blue status LED. While we is high, the sram_controller writes the data from the input_spi to the current address whenever a rising edge of the DVLD signal occurs.

The input_controller manages the address when the system is in WRITE mode. After each rising edge of DVLD, it increments the input address by one. This means that the RAM is filled byte by byte, starting from address 0x00, which matches the order in which the program will be read later.

The address and clock input of the SRAM must be driven differently, depending on the value of we. As mentioned earlier, when we is high, the address is driven by the input_controller, and the clock is driven by the DVLD signal of the input_spi module. When we is low, the address is driven by the data_loader module, which reads the program from the SRAM, (see 3.2.2) , and the clock is driven by the system clock. This behavior is achieved through conditional assignment, as shown in the code in Listing 3.1.

```
  assign ram_clk = (we) ? dvld_from_input : clk_1;
  assign address_bus = (we) ? input_address : output_address;
```

Listing 3.1: Conditional assignment for the sram_controller inputs in top.v. The address bus and clock signal are conditionally driven based on the value of the we signal. When we is high, the input_controller and dvld_from_input signals are used to drive the address and clock, respectively. Otherwise, the data_loader module and system clock take control.

During synthesis, this logic will behave like multiplexers (see Figure 3.5).
The writing speed of this setup is limited by the 8 ns access time of the SRAM [11]. This results in a maximum writing speed of 125 MB/s which means that the full SRAM could be written to in less than 5 ms.

Most of the modules featured in this and following sections have a reset input that will reset the module by resetting all registers to their default values, when a logic high is detected.

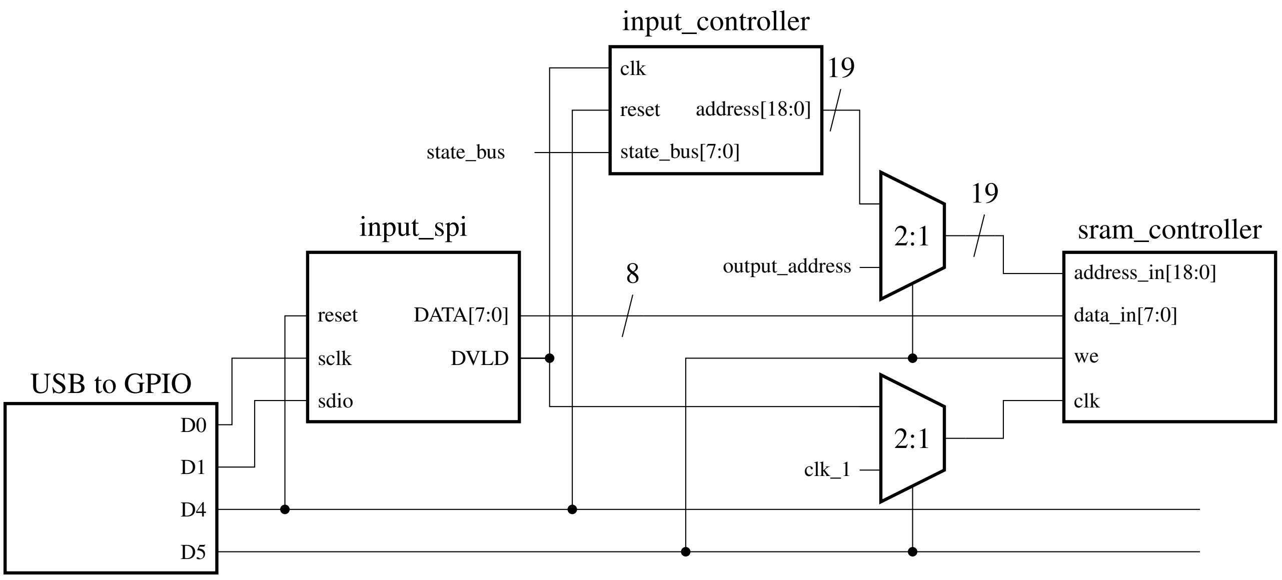


Figure 3.5: Block diagram of the synthesized circuit that processes the inputs and writes the data into the FPGA's SRAM. The program, transmitted via SPI, is read by the `input_spi` module and written to SRAM through the `sram_controller`. The diagram illustrates the role of the `input_controller` in managing the address during WRITE mode and the use of conditional assignment for addressing and clock inputs based on the state of the `we` signal.

```verilog
module input_spi_8bit(
input sclk,     //external sclk
input sdio,     //external data (MOSI)
input reset,       //external reset
output [7:0] DATA,    //data output
output DVLD     //data valid flag
);
   //INITIALIZE REGISTERS
   reg [2:0] index = 7;
   reg [7:0] data = 0;
   reg [7:0] word =0;
   reg finished=0;
   reg dvld=0;

   //ASSIGNEMENTS
   assign DATA = data;
   assign DVLD=dvld;

   //ALways Block on posedge of external sclk
   always @(posedge sclk or posedge reset) begin

      if(reset) begin //reset
         index=7;
         data =0;
         word=0;
      end else begin
         word[index] = sdio; //read current bit

         if(index==0) begin  //when byte is complete output
            data = word;
         end
         dvld=(index==0);    //data valid signal
         index <= index - 1; //decrement index
      end
   end
endmodule
```

Listing 3.2: Implementation of the `input_spi_8bit` module in Verilog. This module reads 8-bit data transmitted via SPI and outputs it alongside a data valid (`DVLD`) flag. The `sclk` signal drives the bit-by-bit reading of the SPI input (`sdio`), which is stored in a `word` register. Once a full byte is received, the `DATA` output is updated, and `DVLD` is set to logic high. The `reset` signal clears all internal registers, ensuring proper initialization.

3.2.2 Program execution

This section demonstrates how the program execution described in Section 3.1 is implemented by detailing the Verilog modules involved.

To execute the programs, multiple Verilog modules must work together. A diagram illustrating the modules and their dependencies is shown in Figure 3.6. The modules are represented as boxes, with small arrows indicating single-bit control signals driven by the module where the arrow originates. Buses are represented by larger arrows, which are also driven by their source modules. The I/O block represents the inputs and outputs of the FPGA.

For proper timing, multiple clock signals with different frequencies and relative phases are required. These clocks are generated by an Intellectual Property (IP) block called the 'Clock Wizard', a module provided by the Vivado Design Suite from AMD. The module is customizable and takes the built-in 12 MHz clock from the FPGA chip as an input. The Clock Wizard then upscales and shifts the clock to the desired frequencies and phases. All the clocks used in this project are listed in Table 3.2. The Clock Wizard IP block is located at the top of the diagram.

Name	**Frequency** / MHz	**Phase / degrees**	**Description**
clk_1	100	0	Data clock for quad SPI communication and reference clock for DDS.
clk_1_200dg	100	200	`sclk` for quad SPI communication.
clk_3	50	0	Main system clock.
clk_4	5	0	Data clock for single SPI communication.
clk_5	5	90	`sclk` for single SPI communication.
clk_6	50	90	90° delayed main system clock.

Table 3.2: Clock wizard configuration table. The table contains the names, frequencies, phases, and a short description of the clock signals provided by the clock wizard IP.

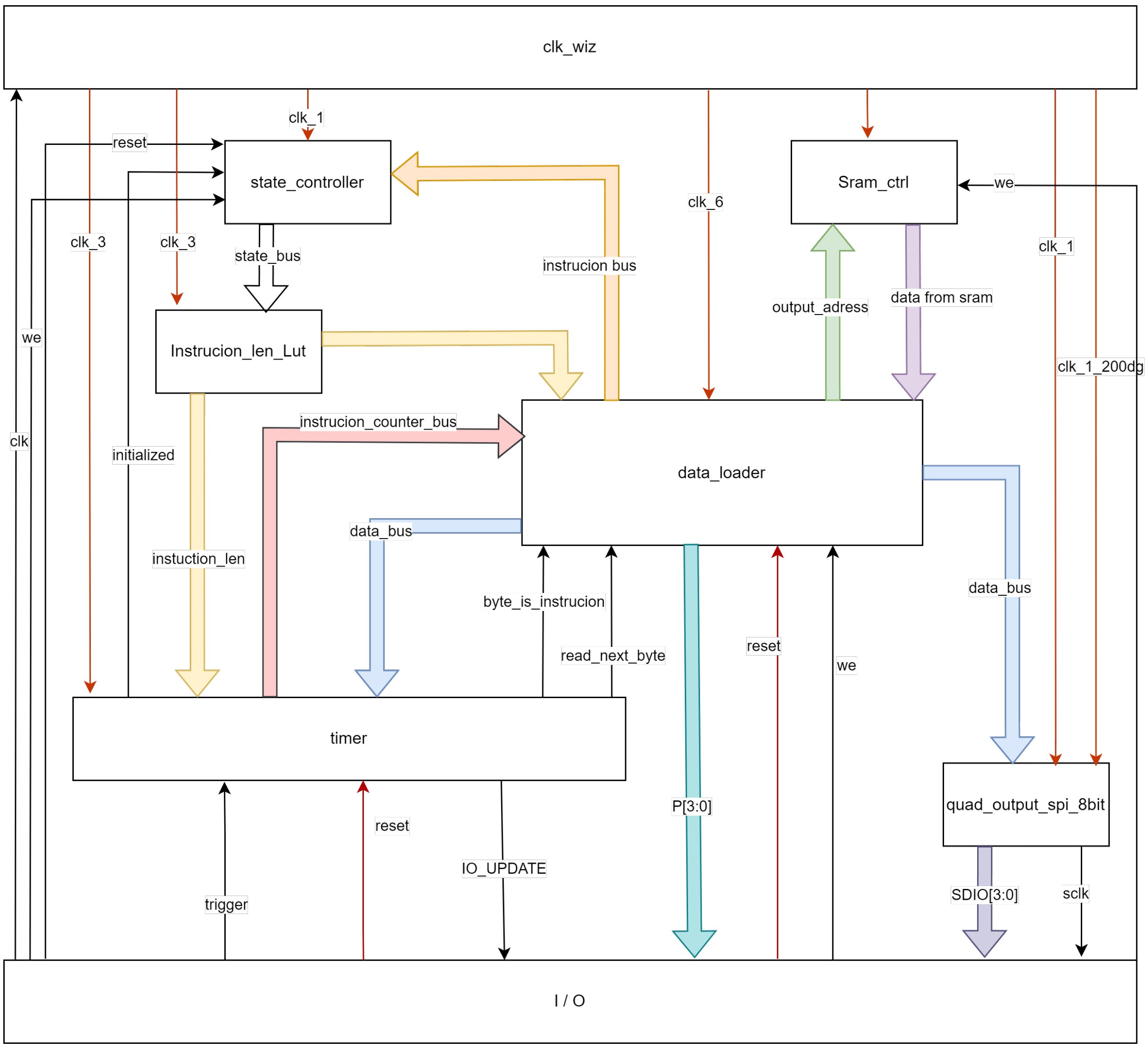


Figure 3.6: Diagram illustrating the Verilog modules and their interdependencies for executing programs on the FPGA. Each module is represented as a box, with single-bit control signals indicated by small arrows and multi-bit buses shown as larger arrows. The `state_controller` manages the state of the system and drives the `state_bus`, while the `data_loader` serves as the central module, interfacing with other components such as the `timer`, `Instruction_len_Lut`, and `Sram_ctrl`. The I/O block at the bottom represents the FPGA's inputs and outputs, including the clock and reset signals.

The state_controller module drives an 8 bit register that represents the state of the entire system. The state is determined by the reset, we, and instruction input signals of the module and is assigned to the state output. This output is wired to the state_bus.

When the reset signal is set to logic high, all other logic is overridden, and the state is forced into RESET. If the reset input is at logic low, the system will only transition to a new state if the initialized input is also at logic high. When these conditions are met, the module checks the we input. If we is at logic high, the state switches to WRITE. If we is at logic low, the state is set based on the value of the instruction input.

The code snippet that implements this logic is shown in Listing 3.3.

```
if(reset) begin
   state_register <= RESET;
end else begin
   if(initialized) begin
      if(we) begin
         state_register <= WRITE;
      end else begin
         // Logic to set the state based on the instruction
         case(instruction) // other instructions
            32: begin
               state_register <= WAIT;
            end
         //other cases
         endcase
      end
   end
end
```

Listing 3.3: Implementation of the state_controller module, which determines the system state based on the reset, we, instruction, and initialized signals. The logic prioritizes the reset signal to override all other inputs, followed by transitions determined by the we and instruction inputs when the system not initialized.

The logic that determines the state based on the instruction input is implemented using a straightforward case(instruction) statement. This statement assigns a predefined state to each instruction. For example, if the input instruction is a wait instruction, the state transitions to WAIT. All possible states of the system are listed in Table 3.

As shown in Figure 3.6, the data_module module has the most dependencies. Its primary purpose is to retrieve data from the SRAM controller and decode it. It also keeps track of the current program address, which is stored in the 19-bit address register and drives the output_address output. A portion of the code responsible for handling data reading and decoding in the data_loader module is shown in Listing 3.4.

When the module receives a read_next_byte signal, it reads a byte from the sram_controller, interprets it and stores it in registers based on the states of several control signals. If the byte_is_instruction signal is set to logic high, the byte just read is interpreted as an instruction. In this case, the module decodes the byte into the instruction and the update_at_end flag. The instruction is stored in a 7 bit register, which is directly assigned to the instruction_bus. If the byte_is_instruction flag is set to logic low, the data from the SRAM is accumulated in the 32 bit `data` register, which is assigned to the `data_bus`. The `data` register is cleared whenever a new instruction is read.

Instructions that direct the system to write to a register on the DDS board include information specifying the target register. If one of these instructions is read, the instruction byte is also stored in the `data` register.

When both the `is_instruction` and `update_at_end` signals are high, no data is read from the SRAM. Instead, the instruction is set to 127, which transitions the system into the `UPDATE` state.

The `input_address_bus` holds the address of the end of the program. If the current program address, stored in the `address` register, exceeds the program's end address, the instruction is set to 124, forcing the system into the `IDLE` state.

Whenever a byte is read from the SRAM, the program address is subsequently updated. In most cases, it is incremented by one. However, certain instructions, such as 'jump', manipulate the address in other ways. The logic that controls the program address is discussed in Section 3.2.3.

```verilog
// when read next byte flag is 1
if(read) begin
   //check if program has endet
   if(address >=input_address_bus) begin
      instruction=124;// IDLE instuction
   end else begin
      // when byte is instuction and update_at_end flag is high override instruction to IO_UPDATE
      if(is_instruction && update_at_end) begin
         instruction = 7'b1111111;  //update instrucion
         update_at_end <= 0;        //reset update_at_end
      end else begin
         //if byte is instuction set new instruction
         if(is_instruction) begin
            data=0;
            instruction = sram_data[6:0];
            update_at_end = sram_data[7:7];
            // if write to DDS instruction
            if(sram_data[6:5]==0) begin
               data[7:0] =  (sram_data & 8'b01111111); //mask first bit
            end
         end else begin
            data = data << 8; // shift data by 8
            data[7:0] = sram_data;// add new byte
         end
      end
   end
end
```

Listing 3.4: Logic implementation of the `data_loader` module, which reads and decodes bytes from the SRAM. The listing demonstrates the handling of instructions, data accumulation in the 32 bit `data` register, and program address management.

The `read_next_byte` and `byte_is_instruction` signals are controlled by the timer module. These signals are driven by the `read` and `is_instruction` registers. When reading data from the SRAM, the timer module keeps the `read_next_byte` signal at logic high until a complete instruction, including its parameters, is read. The lengths of the instructions are provided by the `instruction_len_LUT` module. The timer module tracks the number of bytes read for the current instruction using the `instruction_len_counter` register. When the end of the instruction is reached, the `read_next_byte` and `byte_is_instruction` signals are set based on the system's state.

This logic is implemented using a `case(state)` statement, a portion of which is shown in Listing 3.5. When the state is `WAIT`, the duration for which the system should pause is stored in the data register of the `data_loader` module, which drives the `data_bus`. The counter register tracks the number of clock cycles that have elapsed since the last instruction was read. This counter increments by one with each clock cycle and resets whenever a new instruction is read. If the `counter` exceeds the value provided by the `data_bus`, the `read` register—and consequently the `read_next_byte` wire—is set to logic high, allowing the program to continue.

```

case(state) // read next instruction based on state
   WAIT: begin
      read= counter >= data_bus;
   end
   UPDATE: begin
      read= counter >= update_length;
   end
   WAIT_TRIGGER: begin
      read=trigger;
   end
   WRITE: begin
      read=~we && counter>10;
   end
   \\ more cases
endcase
is_instruction = read;
```

Listing 3.5: Snipped of the `case` statement within the `timer` module that governs the `read` register based on the system's current state. In the `WAIT` state, the program pauses for a duration specified by the `data_bus`, while in the `UPDATE` state, the hold duration is fixed and determined by the `update_length` parameter. The `WAIT_TRIGGER` state halts program execution until the `trigger` signal, an external input, is set to logic high. In the `WRITE` state, program execution begins 10 clock cycles after `we` is set back to logic low. This case statement ensures synchronization with external devices and accurate timing for program flow.

The same logic applies in the `UPDATE` state; however, in this case, the hold duration is fixed and determined by the `update_length` parameter.

To synchronize with external devices, the system can be set to the `WAIT_TRIGGER` state using the corresponding instruction. In this state, program execution pauses until the `trigger` signal is set to logic high. The `trigger` signal is an external input to the FPGA. Additionally, some states cause the program to wait until an edge of the `trigger` signal is detected.

As previously discussed, setting `we` to logic high causes the system to enter the `WRITE` state. When this occurs, the program address is reset to zero. In this state, program execution begins 10 clock cycles after we is set back to logic low.

When the system is in the RESET state and the reset input returns to logic low, the system remains in that state for a specific duration determined by a Verilog parameter called reset_length. Since the reset signal also resets the DDS and its default communication protocol is single SPI, this time is used to reconfigure the DDS to quad SPI. To enable quad SPI, a DDS register must be written via single SPI. For this purpose, a single SPI module was implemented; however, it was omitted from Figure 3.6 for simplicity.

Once this process is complete, the initialized output of the timer module is set back to logic high, allowing the system to transition between states again.

The timer and data_loader modules are driven at the same clock speed but with different phases (clk_3 and clk_6). The phase shift ensures that signals are sent in the correct order. At this clock speed, the system takes one clock period, corresponding to 20 ns, to read a single byte. This must be considered when writing a timing-critical program to the SRAM, as it determines how long the instructions take to execute.

Both moudles reset when either of the reset or we inputs are set to logic high.

The module responsible for communication with the DDS is quad_output_spi_8bit. This module outputs the byte from its data input, which is driven by the least significant byte of the data bus (data_bus[7:0]), over quad SPI within two clock cycles. Transmission begins only when its start input signal is set to logic high. This start signal is asserted whenever the system is in the OUTPUT state. Any instruction that writes to a DDS register causes the system to transition to the OUTPUT state. Additionally, the IO_UPDATE output is set to logic high whenever the system is in the UPDATE state.

The timing diagram in Figure 3.7 illustrates how the registers and signals evolve over time during the execution of the example program shown in Table 3.1. The data was captured using an Integrated Logic Analyzer (ILA), an IP block that records signals, registers, or buses over time, functioning similarly to an onboard digital oscilloscope.

The diagram clearly shows that program execution begins at address 0, with the address incrementing on the rising edge of the system clock (clk_3) whenever read_next_byte is set to logic high. It can also be observed that the instruction_bus updates on the rising edge of the clock whenever both read_next_byte and byte_is_instruction are high.

Furthermore, the diagram demonstrates that the system state transitions occur after the instruction is updated. The program progresses through its states as expected, starting with the WRITE state, followed by OUTPUT, UPDATE, and WAIT.

The diagram also highlights how data is accumulated in the data register, which drives the data_bus. Although this feature is not utilized in this particular example, as the bytes are output one by one, it is crucial for other instructions where the complete parameter must be provided at once.

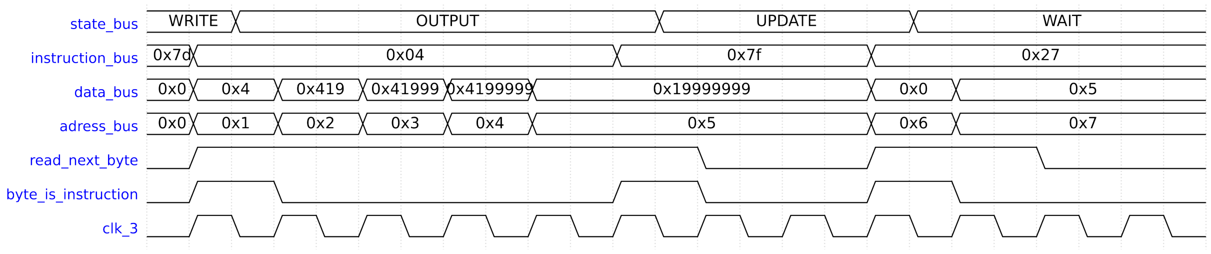


Figure 3.7: Timing diagram illustrating the evolution of registers and signals during the execution of the example program from Table 3.1. Captured using an Integrated Logic Analyzer (ILA), the diagram shows how the `address_bus` increments with the rising edge of `clk_3` whenever `read_next_byte` is high. The `instruction_bus` updates in sync with the rising clock edge when both `read_next_byte` and `byte_is_instruction` are high. State transitions, visible on the `state_bus`, occur after instruction updates, progressing through states like `WRITE`, `OUTPUT`, `UPDATE`, and `WAIT`. The diagram also highlights the role of the `data_bus`, which accumulates data in the `data` register.

3.2.3 Jumps, function calls and loops

The previous section discussed the general functioning of program execution. This section focuses on how the address manipulation instructions, as well as the instructions that toggle the profile pins of the DDS, are executed.

These instructions are directly handled within the `data_loader` module. Two separate `case` statements are used: one for instructions that manipulate the address (Listing 3.7) and another for those that do not. The `case` statements use the `instruction_code` register as their parameter. This register is updated every clock cycle using the lines shown in Listing 3.6.

```
instruction_code[6:0]=instruction;
instruction_code[10:7]=instruction_counter_bus;
```

Listing 3.6: Implementation of the `instruction_code` register, which combines the current instruction and the value from the `instruction_counter_bus`. This 11 bit register encodes both the instruction to be executed and the current position in the reading process, enabling the `data_loader` module to handle instructions effectively.

The 11 bit `instruction_code` register contains both the instruction
and the value from the `instruction_counter_bus`. The `instruction_counter_bus` is assigned the value of the `instruction_len_counter` from the timer module. This value represents the number of bytes that have been read for the current instruction. Thus, the `instruction_code` encodes both the instruction to be executed and the current position in the reading process.

This information is crucial for ensuring that the address is updated only after the full parameter of the instruction has been read and before the next instruction is executed. The process of combining the two binary values to create the `instruction_code` is equivalent to shifting the value of the `instruction_counter_bus` by seven bits and then adding it to the `instruction`. Consequently, the value of the `instruction_code` can be computed using the following formula:

$$\texttt{instruction\_code} = \texttt{instruction\_counter\_bus} \cdot 128 + \texttt{instruction}.$$

When no `instruction_code` is detected, the `address` register is incremented by one. This behavior is realized using the `default` statement.

```
case(instruction_code)
   //JUMP
   418:begin
      address <= data[18:0];
   end
   //END_LOOP
   47: begin
      if(loop_register != 0) begin
         address <=loop_return_address;
         loop_register <= loop_register-1;
      end else begin
         address <= address+1;
      end
   end
   //CALL_FUNC
   432: begin
      func_return_address<= address+1;
      address <= data[18:0];
   end
   //CALL_FUNC_FROM_BUFFER
   56: begin
      func_return_address<= address+1;
      address <=func_address;
   end
   //END_FUNC
   49: begin
      address<=func_return_address;
   end
   default: begin
      address <= address+1; // go to next address;
   end
endcase
```

Listing 3.7: Implementation of the address manipulation logic within the `data_loader` module using a `case` statement. The logic handles instructions such as `JUMP`, `END_LOOP`, `CALL_FUNC`, `CALL_FUNC_FROM_BUFFER`, and `END_FUNC`. Each instruction modifies the `address` register based on the current `instruction_code` and associated parameters, enabling complex program flow control.

The simplest instruction that changes the address is the `JUMP` instruction. This instruction takes a 3 byte parameter containing the address to which the program should jump. The decimal value of the `JUMP` instruction is 34. The address must be overwritten when the `instruction_counter_bus` holds the value 3, which corresponds to an `instruction_code` of 418. When this code is detected, instead of incrementing the `address`, the last 19 bits of the `data` register are written into the `address` register.

The `END_LOOP` instruction functions similarly to the `JUMP` instruction. When a `END_LOOP` instruction is detected (instruction_code 47), the system jumps to the address stored in the `loop_return_address` if the value in the `loop_register` is not equal to zero. If a jump occurs, the `loop_register` is decremented by one. When the value in the `loop_register` is zero, no jump is performed. This functionality allows for the implementation of `do-while`-like behavior by setting the `loop_return_address` and loop_register.

These and other internal FPGA registers can be set using designated instructions. Instructions that write to FPGA registers are handled in the second case statement within the data_loader module, parts of which are shown in Listing 3.8. The first case in the listing (instruction_code 555) handles the LOAD_LOOP_REGISTER instruction. When the full parameter of this instruction is read, it is stored in the loop_register. Other load register instructions are implemented in a similar manner.

As an alternative to setting both registers required for a LOOP instruction, the BEGIN_LOOP instruction (instruction_code 558) can be used. This instruction takes the number of loops as a parameter and loads it into the loop_register. The loop_return_address is automatically set to the succeeding address.

```
case(instruction_code)
   \\LOAD_LOOP_REGISTER
   555: begin
      loop_register <= data;
   end
   \\ other load register instruction
   \\BEGIN_LOOP
   558: begin
      loop_register <= data;
      loop_return_address <= address+1;
   end
   \\LOOP_FROM_BUFFER
   54: begin
      loop_register <= loop_buffer;
      loop_return_address <= address + 1 ;
   end
   //TOGGLE_P0
   35: begin
   profile_pin_register[0:0] =~profile_pin_register[0:0];
   end
   \\other toggle profile pin instructions
endcase
```

Listing 3.8: Implementation of the second case statement within the data_loader module, which handles instructions for writing to internal FPGA registers. The listing includes examples such as LOAD_LOOP_REGISTER (instruction_code 555), which sets the loop_register, and BEGIN_LOOP (instruction_code 558), which sets both the loop_register and loop_return_address. Additional instructions, such as toggling the profile pin register, are also demonstrated.

An example of how a loop can be implemented is shown in Table 3.3. The first instruction is BEGIN_LOOP, with its parameter stored at addresses 1–4. The next instruction is a WAIT command, which pauses the program for one clock cycle, followed by the END_LOOP instruction. Finally, an END instruction terminates the program and sets the system to the IDLE state. The time evolution of the registers during this program's execution is shown in Figure 3.8. It can be clearly observed that the program first sets the loop_register. Afterward, the WAIT instruction is executed, and the program address jumps back to address 5, decrementing the value of the loop_register. This process repeats one more time until the loop_register reaches zero. Once the loop_register is zero, the WAIT instruction is executed one final time before the program terminates.

Address	Byte	Description
0	0x2e	Begin loop
1	0x00	loop_register[31:24]
2	0x00	loop_register[23:16]
3	0x00	loop_register[15:8]
4	0x02	loop_register[7:0]
5	0x32	Wait one clock cycle
6	0x2f	End loop
7	0x7c	END

Table 3.3: Memory layout of an example program implementing a loop. The program begins with the `BEGIN_LOOP` instruction, followed by its parameter stored in the `loop_register`. The program executes a `WAIT` command, loops back using the `END_LOOP` instruction, and concludes with an `END` instruction, which transitions the system to the `IDLE` state.

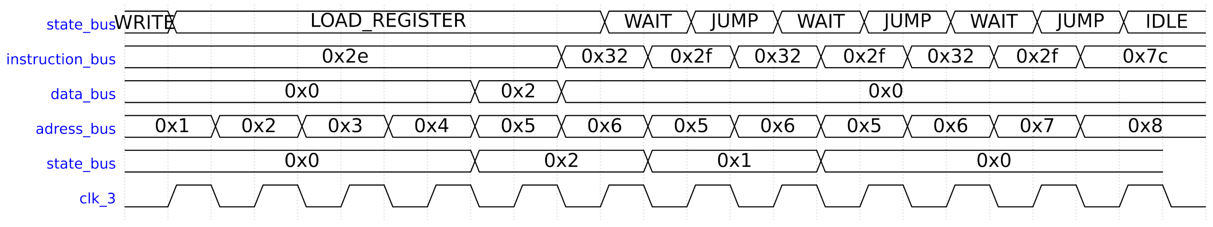


Figure 3.8: Time evolution of registers during the execution of the example loop program. The figure illustrates how the `loop_register` is initialized, decremented with each iteration, and how the program address loops back until the `loop_register` reaches zero. The final iteration executes the `WAIT` instruction before program termination. (Captured using an ILA)

The `CALL_FUNC` instruction (instruction_code 432) functions similarly to the `JUMP` instruction. However, in addition to changing the program address, it also saves the succeeding address in the `func_return_address` register. The `END_FUNC` instruction (instruction_code 49) causes the program to jump back to the address stored in the `func_return_address` register. Together, these two instructions enable the user to call and return from functions.

Since there is only one register that holds the return address, it is not possible to call a function from within another function, as this would overwrite the return address. An example program demonstrating the use of these instructions is shown in Table 3.4. In this example, the program calls a function located at address 0x05. The function contains a single `WAIT` instruction, which is executed before the program returns to the calling address.

The evolution of the registers involved during this process is shown in Figure 3.9. When the `CALL_FUNC` instruction is executed, the `func_return_address` register is updated to store the address of the next instruction (address 0x04). The program then jumps to the function address (0x05), as shown on the `address_bus`. Within the function, the `WAIT` instruction is executed, as indicated on the `instruction_bus`.

After the `WAIT` instruction, the `END_FUNC` instruction is executed. At this point, the system uses the value in the `func_return_address` register to return to the calling address (0x04). The figure also shows the

Address	Byte	Description
0	0x30	Call function at address 0x00005
1	0x00	address[23:16]
2	0x00	address[15:8]
3	0x05	address[7:0]
4	0x7c	END
5	0x32	Wait one clock cycle
6	0x31	End function

Table 3.4: Memory layout of an example program demonstrating the CALL_FUNC and END_FUNC instructions. The program begins by calling a function at address 0x00005. The function contains a WAIT instruction and concludes with an END_FUNC instruction, returning control to the calling address. The program ends with an END instruction, transitioning the system to the IDLE state.

transition of the state_bus back to the main program's state after the function completes.

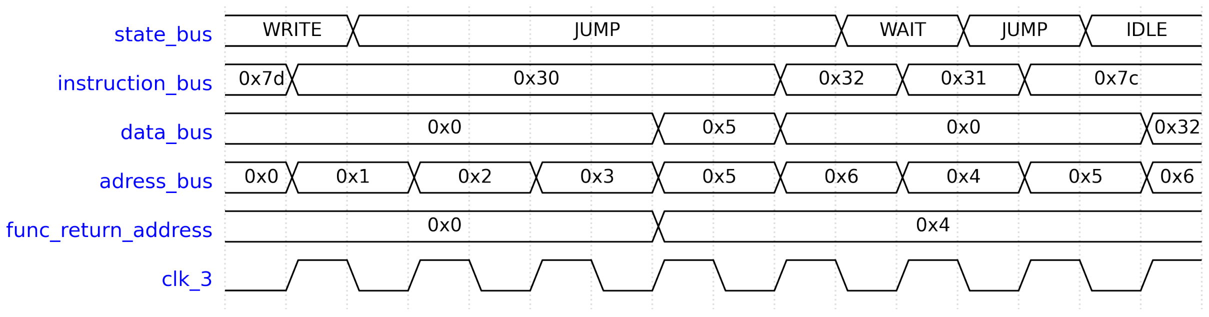


Figure 3.9: Time evolution of the registers during the execution of the example program using CALL_FUNC and END_FUNC instructions. The figure shows the saving of the return address in the func_return_address register, the execution of the WAIT instruction within the function, and the return to the calling address.(Captured using an ILA)

As can be seen in Listing 3.7 there is a second instruction that does the same as the CALL_FUNC instruction with the difference, that it does not take a parameter but loads the address from a register called func_address. This register can be written to with a designated instruction. This instruction is only one byte in size. Thus when a function is meant to be called mansy times in a row, one only has to set the func_address once. Doing this instead of passing the address as a paramater every function call, heavily reduces the RAM usage and execution time. These versions of instruction also exist for WAIT and BEGIN_LOOP that have version where they load the parameter from such a register.

As shown in Listing 3.7, there is a second instruction that functions similarly to the CALL_FUNC instruction and is only one byte in size. The difference is that it does not take a parameter but instead loads the address from a register called func_address. This register can be written to using a designated instruction.

When a function is intended to be called multiple times in succession, the func_address only needs to be set once. Using this approach, instead of passing the address as a parameter with every function call, significantly reduces both RAM usage and execution time. Similar versions of this instruction also exist for WAIT and BEGIN_LOOP, where the parameters are loaded from dedicated registers.

The final instructions to be mentioned are the `TOGGLE_PROFILE_PIN` instructions. These are handled in the `case` statement shown in Listing 3.8. When one of these instructions is detected, it toggles a bit in the `profile_pin_register`, which drives the profile pins. There are four such instructions, one for each profile pin of the DDS.

The full instruction set of the developed system is listed in Table 4.

CHAPTER 4

Reproduction of signal

This chapter demonstrates how the system developed in this thesis can be utilized to generate specific waveforms. It begins by introducing a Python class that was implemented to facilitate the generation of bytecode required for creating signals. This class simplifies the process by providing a more intuitive interface for defining and customizing waveforms.

4.1 Editing and writing sequences in python

Writing a program for a specific RF signal can be approached in various ways, provided the bytecode is correctly sent through the USB-to-GPIO adapter. However, writing instructions byte by byte manually is both tedious and error-prone. This section introduces a Python class designed to simplify the creation of such programs.

The Python class `Sequence` serves as a convenient tool for composing sequences that can be transmitted to the FPGA. It was implemented as part of this project and is a user-friendly tool that does not require extensive knowledge of the underlying system.

The class primarily manages an array called `program`, which contains byte arrays, each representing an instruction along with its parameters. The methods `load` and `save` enable users to save programs to a file or load existing ones. Additionally, the `append_to_program` method appends a byte string to the `program` array while ensuring that the program size does not exceed the available memory capacity.

Functions such as `set_FTW`, `disable_sweep_mode`, and `select_channels` generate the corresponding byte strings for each instruction. Parameters for functions that set a register corresponding to a physical value, such as a frequency, must be provided in SI units. For example, a function call like `set_FTW(100e+6)` appends a byte string to the program that, when executed, configures the FTW of the DDS to the value corresponding to 100 MHz. Functions that compute hexadecimal values for phases, frequencies, and amplitudes follow a naming convention of `<type>_to_hex`.

The class contains multiple fields that store the reference and system frequencies of the DDS, as well as the clock period of the FPGA's main clock. To ensure that the hexadecimal values for the various ructions are computed correctly, these fields must be initialized to match the FPGA's clock settings.

Most functions also include an optional boolean parameter, `update`, which can be set to `true` if the FPGA should issue an `io_update` after executing the instruction.

Debugging these programs in bytecode form is challenging. To address this, all functions that generate byte strings are designed to detect potential errors and raise exceptions when issues are identified.

A significant advantage of using this Python class is its ability to minimize user errors. For instance, writing an instruction to the FPGA with an incorrect byte length can lead to unpredictable system behavior. Such errors, which are nearly impossible to detect manually, are effectively mitigated by the error handling of the provided functions.

The `print_program` function outputs the bytecode to the console, displaying each instruction along with its starting address and a brief comment explaining its purpose.

To upload a program to the FPGA, the Python class `FPGA_DDS` is provided. Creating an instance of this class establishes a connection to the FT232H via USB. The class includes functions to send control signals, such as the `write_program` function, which takes an instance of the `Sequence` class and writes the corresponding program to the FPGA.

The following example demonstrates the use of the Python classes discussed. Listing 4.1 presents the example Python code on the left and the resulting bytecode on the right in (Listing 4.2).
This program creates a function at address `0x04` that toggles the profile pin `p0` ten times, with a fixed delay between each toggle. The main program starts at address `0x11` and configures the DDS into linear sweep mode after setting a single tone of 10 MHz. Following a rising edge of the trigger signal, the program calls the function at address `0x04` with different values in FTW and CW1.The expected output signal generated by this program is a frequency that sweeps five times between two specified frequency ranges for two sets of upper and lower limits. The signal was measured with the MSO-X 2024A oscilloscope.

```python
t=Sequence()
t.jump(17)
t.begin_loop(4)
t.toggle_profile_pin(0)
t.wait(5e-6)
t.toggle_profile_pin(0)
t.wait(5e-6)
t.end_loop()
t.end_func()
t.set_DDS_PLL(5, True)
t.set_FTW(10e+6, True)
t.enable_sweep_mode("frequency")
t.set_LSRR(8e-9, 8e-9)
t.set_FDW(16000 , "frequency")
t.set_RDW(16000, "frequency", )
t.set_CW_01(1, 20e+6, "frequency")
t.wait_trigger("posedge", True)
t.call_func(4)
t.set_FTW(20e+6)
t.set_CW_01(1, 30e+6, "frequency",True)
t.call_func(4)
t.end_programm()
```

Listing 4.1: Python test program to create an example RF signal.

```
address: bytes              | comment
0x00000: 22 00 00 11        | jump
0x00004: 2e 00 00 00 04     | begin loop
0x00009: 23                 | toggle P0
0x0000a: 27 fa              | short wait
0x0000c: 23                 | toggle P0
0x0000d: 27 fa              | short wait
0x0000f: 2f                 | end loop
0x00010: 31                 | end func
0x00011: 81 94 00 00        | set FR1, io_update
0x00015: 84 05 1e b8 51     | set FTW, io_update
0x0001a: 03 80 43 00        | set CFR
0x0001e: 07 01 01           | set LSRR
0x00021: 09 00 02 18 de     | set FDW
0x00026: 08 00 02 18 de     | set RDW
0x0002b: 0a 0a 3d 70 a3     | set CW1
0x00030: a8                 | wait on posedge trigger, io_update
0x00031: 30 00 00 04        | call func
0x00035: 04 0a 3d 70 a3     | set FTW
0x0003a: 8a 0f 5c 28 f5     | set CW1, io_update
0x0003f: 30 00 00 04        | call func
0x00043: 7c                 | END
```

Listing 4.2: Hexadecimal byte representation of the test program.

To analyze the signal, a Hilbert transform was applied, which decomposes the signal into its instantaneous phase and amplitude as functions of time [12]. To obtain the frequency as a function of time, the numerical derivative of the phase was computed after applying a smoothing technique. The smoothing was achieved using a moving average, which effectively reduced the impact of noise present in the raw phase data. The resulting frequency-over-time plot is presented in the lower half of the figure.

A plot of the raw signal can be observed in the appendix in Figure 1.

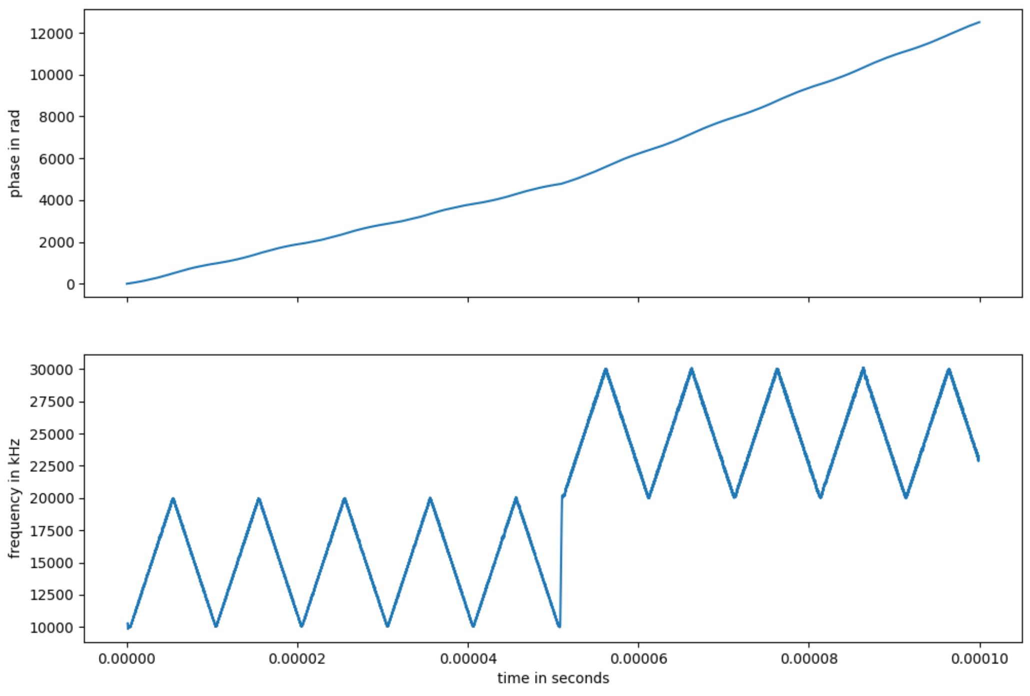


Figure 4.1: Hilbert transform analysis of the DDS signal generated by the test program. The upper plot shows the instantaneous phase over time, derived using the Hilbert transform. The lower plot illustrates the frequency evolution over time, obtained by calculating the numerical derivative of the smoothed phase data.

The graph shows the phase and the frequency over time, where $t = 0$ corresponds to the time of the external trigger signal. The upper half of Figure 4.1 shows the phase of the resulting signal over time. The signal resembles the ramps that sweep first from 1 MHz to 2 MHz for five repetitions before sweeping between 2 MHz and 3 MHz for another five times. This example illustrates that with this waveform generator that is programmed using the presented python class,it is possible to create complex waveforms with little effort.

4.2 Sawtooth for laser frequency broadening

One application considered during the design of the system was using the waveform generator as a driver for an Acousto-Optical Modulator (AOM), to control the frequency of laser beams. Such application would be implemented in the Ytterbium experiment conducted by the Nonlinear Quantum Optics (NQO) research group at the University of Bonn. In this experiment a similar solution is already in place but with limited functionality. This experiment utilizes a Magneto-Optical Trap (MOT) to cool and trap Ytterbium atoms (^{174}Yb), enabling the study of nonlinear quantum optics with Rydberg atoms .

In a MOT, the atoms are cooled using the principle of laser Doppler cooling [13]. Ytterbium possesses two atomic transitions that are utilized for laser cooling, each offering distinct properties: a broad blue transition with a large capture range and a narrow green transition with a low cooling limit, as illustrated in Figure 4.2. In this Ytterbium experiment the operation of the MOT is divided into three distinct phases: the blue MOT phase, the blue-green transfer phase, and the green MOT phase[3].

To bridge the very different regimes of the two MOTs and avoid atoms loss, the green MOT light is frequency-broadened during the blue-green transfer phase. This broadening is achieved by modulating the laser frequency using an AOM driven by an RF signal with sawtooth frequency sweeps. This section will demonstrate how the signal required to drive the AOM is reproduced using the system implemented in this project.

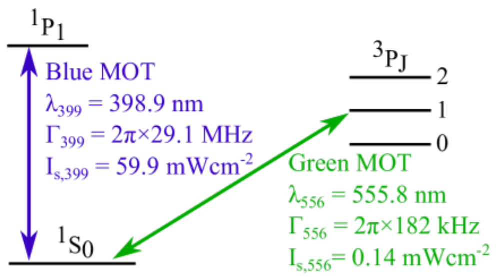

$^1P_1$
Blue MOT
$\lambda_{399} = 398.9$ nm
$\Gamma_{399} = 2\pi\times29.1$ MHz
$I_{s,399} = 59.9$ mWcm^{-2}
^3P_J
2
1
0
Green MOT
$\lambda_{556} = 555.8$ nm
$\Gamma_{556} = 2\pi\times182$ kHz
$I_{s,556} = 0.14$ mWcm^{-2}
$^1S_0$

Figure 4.2: Atomic transitions used for laser cooling of ^{174}Yb in the Ytterbium experiment. The broad blue transition (λ_{399} = 398.9 nm) is used for initial cooling, while the narrow green transition (λ_{556} = 555.8 nm) is employed in the final MOT phase.

Source: [3]

The required RF signal to broaden the laser is illustrated in Figure 4.3. The signal clearly exhibits three distinct sections. Initially, the frequency remains constant at f_0. Upon detection of an external trigger, the frequency rapidly increases to a higher value and then sweeps downward in a sawtooth pattern. This process repeats multiple times.

The sawtooth pattern is bounded by the lines $\overline{AB}$ and $\overline{CD}$, with the period of the sawtooth denoted as Δt. After the sweeping phase completes and the endpoints of these lines are reached, the signal transitions back to a single tone at the frequency f_1.

Before a program for such frequency modulation can be created with the `Sequence` class, a Python function must be implemented to compute the intersections between the sawtooths and the enveloping lines, as well as the slopes of the individual sawtooths. This function, called `old_sawtooth`, is not further discussed but is included in Listing 1 in the appendix.

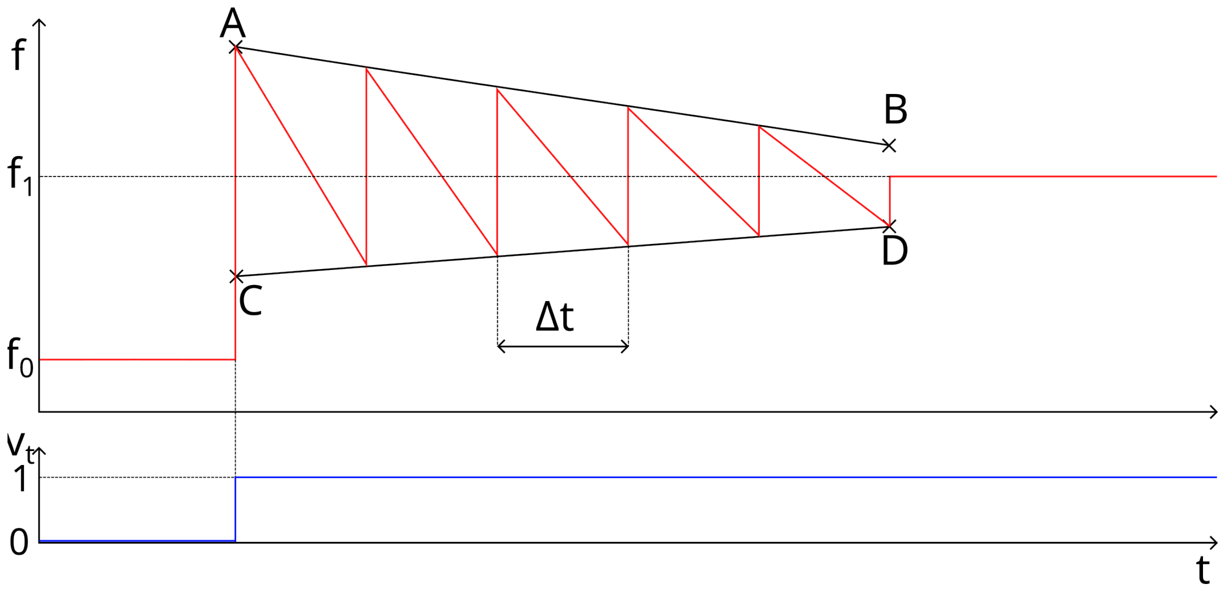


Figure 4.3: The RF signal required to broaden the laser, divided into three sections: an initial constant frequency phase at f_0, followed by a rapid increase in frequency triggered by an external signal, and subsequent downward sweeps in a sawtooth pattern. The sawtooth is bounded by lines $\overline{AB}$ and $\overline{CD}$ with a period Δt. After completing the sweeping phase, the signal transitions to a single tone at frequency f_1.

The function `write_old_sawtooth`, shown in Listing 4.3, takes the four points that define the enveloping lines, the start and end frequencies (`f0` and `f1`), and the sawtooth period `dt`. Additionally, it accepts a parameter called `step`, which determines the number of sawtooth periods after which the limits and the slope should be updated. Decreasing this value increases the time resolution of these parameters but also increases memory usage. The implemented function is described in the following.

The function starts by instantiating a new `Sequence` object. The new sequence starts with a `Jump` instruction that skips to address 23. At this address, the main program begins. It starts by setting the PLL divider of the DDS to 5. Next, the FTW is set to `f0` with the update flag set to `true`. This part of the code realizes the first section of the signal to be reproduced.

For the second section, the DDS is configured into linear sweep mode. Subsequently, the sweep registers are set. As explained in the Theory section 2.1.2, the slope of the linear sweeps is determined by two parameters (see Figure 2.2). To simplify the computation of these registers and achieve maximum time resolution, the time step Δt of the linear sweep is kept constant at its minimum value.

The `RDW` is set close to its maximum value to approximate the vertical rise of the sawtooth frequency. The `FDW` is calculated from the first entry of the `slopes` array. Afterward, the upper and lower sweep boundaries are configured. Once all these settings are complete, the system waits until a rising edge of the trigger signal is detected.

Note that no `io_update` signal has been sent yet. This ensures that the system remains in single-tone mode, with all recent settings still held in the DDS buffer.

When a rising edge is detected, an `io_update` is performed, and the function at address 4 (immediately following the first `JUMP` instruction) is called. This function toggles the profile pin 0 of the DDS, causing the frequency of the DDS to sweep to the upper limit almost instantaneously. Immediately afterward, the pin is toggled again to initiate the downward sweep of the sawtooth.

To maintain the period of the sawtooth at `dt`, the system must wait for this duration. However, since the execution of these instructions takes time, the execution time must be subtracted from the waiting period to ensure precise timing. Figure 4.4 illustrates how the execution time must add up with the waiting time to the period of one sawtooth. The diagram shows an example set of instructions that are executed in during one sawtooth period.

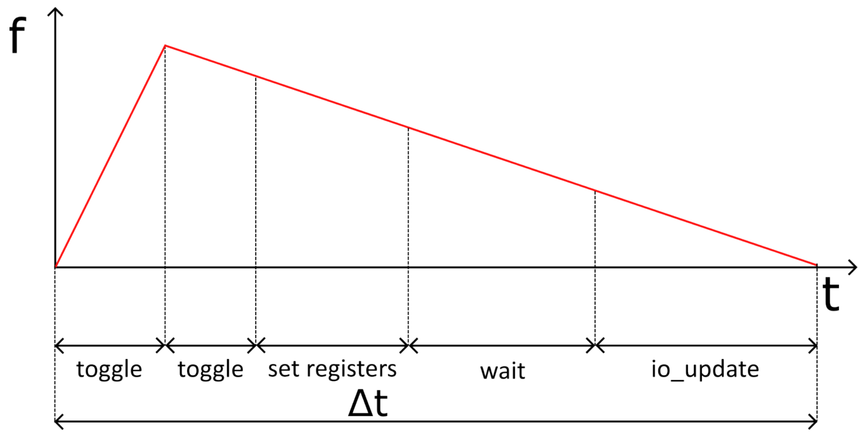


Figure 4.4: Timing diagram illustrating the relationship between the execution time of instructions and the waiting time required to maintain the period of the sawtooth at Δt. The diagram shows an example set of instructions executed during one sawtooth period, highlighting how precise timing is achieved by subtracting the execution time from the waiting period.

Following this, a loop is created to repeat the process of toggling the profile pin and waiting between toggles. After the loop, this process is executed one final time before the function returns.

AOnce the function returns to the calling address, sawtooth modulations of the frequency were performed `step` times. The program then updates the upper and lower limits as well as the slope, and the function is called again. This process is repeated for all values in the `lower_points` array.

Finally, the DDS is returned to single-tone mode, and the frequency is set to `f1` before the program terminates.

```python
def write_old_sawtooth(A, B, C, D, dt, f0, f1, step):
   upper_points, lower_points, slopes = self.old_sawtooth(A, B, C, D, dt)  # Compute intersections and slopes
   sawtooth = Sequence()  # Instantiate Sequence
   sawtooth.jump(23)  # Jump to beginning of program

   # Start of function
   sawtooth.toggle_profile_pin(0)  # Sweep up
   sawtooth.toggle_profile_pin(0)  # Sweep down
   sawtooth.wait(dt - 7 * sawtooth.FPGA_clk_period)  # Wait

   sawtooth.begin_loop(step - 3)  # Begin loop
   sawtooth.toggle_profile_pin(0)  # Sweep up
   sawtooth.toggle_profile_pin(0)  # Sweep down
   sawtooth.wait(dt - 3 * sawtooth.FPGA_clk_period)  # Wait
   sawtooth.end_loop()  # End loop

   sawtooth.toggle_profile_pin(0)  # Sweep up
   sawtooth.toggle_profile_pin(0)  # Sweep down
   sawtooth.wait(dt - 25 * sawtooth.FPGA_clk_period)  # Wait
   sawtooth.end_func()  # Return

   # Main program start
   sawtooth.set_DDS_PLL(5, True)  # Set PLL to 5
   sawtooth.set_FTW(f0, True)  # Set FTW, io_update
   sawtooth.enable_sweep_mode("frequency")  # Enable linear sweep mode of frequency
   sawtooth.set_LSRR(8e-9, 8e-9)  # Set RSRR and FSRR
   sawtooth.set_FDW(np.abs(slopes[0]) / sawtooth.sync_frequency, "frequency")  # Set FDW
   sawtooth.set_RDW(490000000, "frequency")  # Set RDW
   sawtooth.set_CW_01(1, upper_points[0], "frequency")  # Set upper limit
   sawtooth.set_FTW(lower_points[0])  # Set lower limit
   sawtooth.wait_trigger("posedge", True)  # Wait for rising edge of trigger

   sawtooth.call_func(4)  # Call function
   for i in np.arange(step, len(lower_points), step):
      # Change function parameters
      sawtooth.set_CW_01(1, upper_points[i], "frequency")  # Set upper limit
      sawtooth.set_FTW(lower_points[i])  # Set lower limit
      sawtooth.set_FDW(np.abs(slopes[i]) / sawtooth.sync_frequency, "frequency", True)  # Set slope
      sawtooth.call_func(4)  # Call function

      sawtooth.disable_sweep_mode()  # Set DDS to single tone mode
      sawtooth.set_FTW(f1, True)  # Set FTW
      sawtooth.end_programm()  # End program
```

Listing 4.3: The function `write_old_sawtooth` writes the bytecode to create a sawtooth frequency modulation. It computes the necessary parameters for the signal, such as intersections, slopes, and limits, and creates a sequence that handles initialization, sweep configurations, and iterative signal generation. The function also dynamically updates parameters during runtime and ensures precise timing for the generated waveform.

The system developed during this work, was tested by measuring the signal created by `write_old_sawtooth` function using the MSO-X 2024A oscilloscope for the following example parameters.

$$
\begin{aligned}
A &= (0\,\mu\text{s}, \quad 88{,}8\,\text{MHz}) \\
B &= (60\,\mu\text{s}, \quad 82{,}8\,\text{MHz}) \\
C &= (0\,\mu\text{s}, \quad 80{,}8\,\text{MHz}) \\
D &= (60\,\mu\text{s}, \quad 82{,}8\,\text{MHz}) \\
f_0 &= 78{,}8\,\text{MHz} \\
f_1 &= 82{,}8\,\text{MHz} \\
\texttt{step} &= 10
\end{aligned}
$$

The raw signal is shown in the appendix in Figure 2. The resulting frequency over time, as well as the trigger signal, can be observed in Figure 4.5. The frequency was obtained by performing the Hilbert transform and calculating the numerical derivative of the smoothed instantaneous phase as explained in Section 4.1.

In the plot shown in Figure 4.5, the three stages of the signal are clearly visible. Initially, the frequency remains constant. As soon as the trigger signal transitions from low to high, the frequency rapidly sweeps upwards. Following this, the frequency oscillates between two values in a sawtooth-like pattern for 10 repetitions, corresponding to the `step` parameter that was set. After each set of sweeps, the lower limit, upper limit, and slope are updated, and the sweeping continues with the new parameters.

This behavior demonstrates how the frequency range progressively decreases until the system returns to single-tone mode 60 µs after the trigger signal is set to high. Figure 4.6 provides a zoomed-in view of the frequency plot, where the individual sawtooth patterns are clearly distinguishable.

Since the measured signal exhibited a high level of noise, a large bin size of 300 was chosen for the moving average. This smoothing process causes the sawtooth peaks and rising edges to be smoothed out and not fully reach the limits expected from the parameters set.

One potential cause of the high noise, aside from the limitations of the oscilloscope, is time jitter in the system clock of the DDS. The reference clock is generated by the clock wizard in the FPGA, which scales up the 12 MHz FPGA clock to 100 MHz. This signal is then further upscaled in the DDS to the 500 MHz system clock. Each of these consecutive upscalings increases the time jitter, potentially contributing to the observed noise.

In future implementations, the reference clock is planned to be replaced with a more stable source, such as a crystal oscillator, to reduce time jitter and improve signal quality.

In conclusion, this chapter showcased the capabilities of the developed system in generating precise and complex waveforms. These results demonstrate the potential of the system to meet the stringent requirements of modern experimental physics.

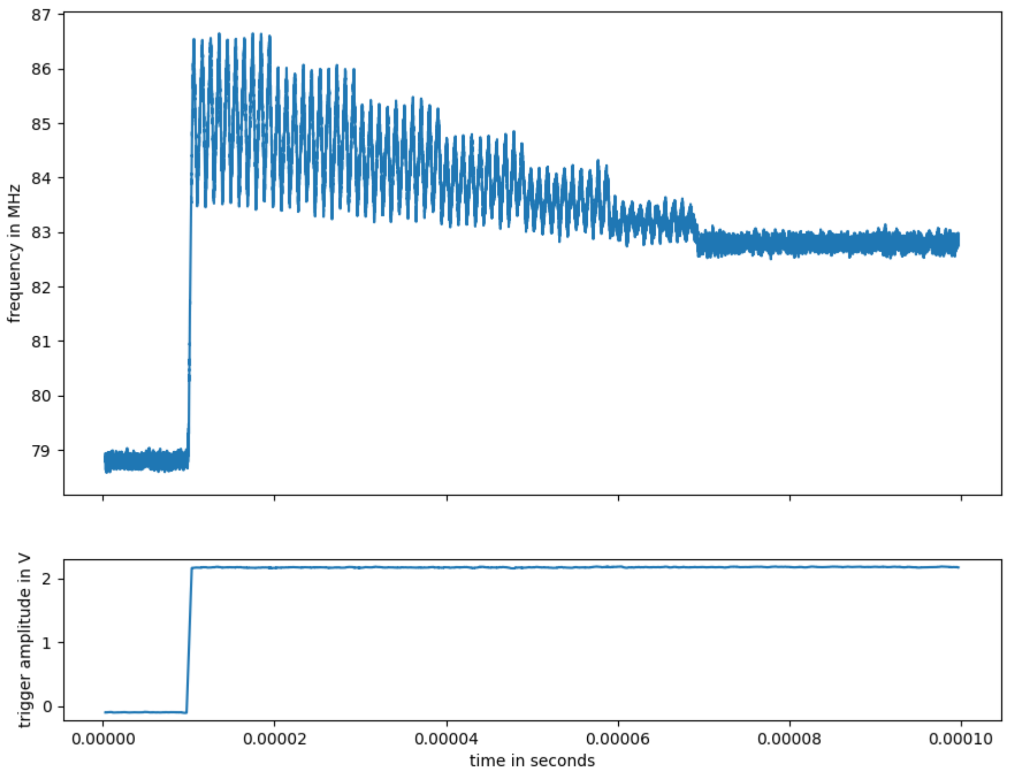


Figure 4.5: The frequency evolution over time for the generated sawtooth signal, as obtained by performing the Hilbert transform and calculating the numerical derivative of the smoothed instantaneous phase.

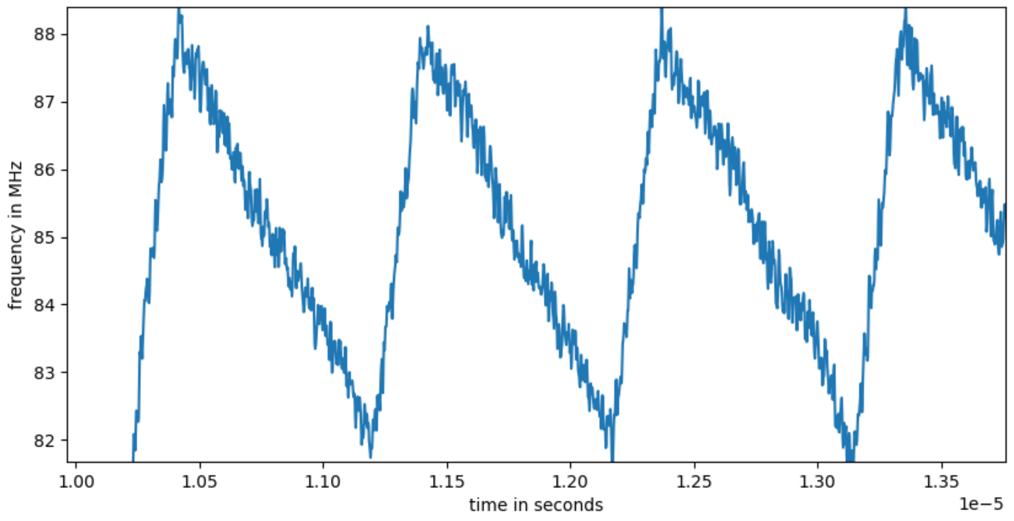


Figure 4.6: Zoom of the frequency evolution over time for the generated sawtooth signal, as obtained by performing the Hilbert transform and calculating the numerical derivative of the smoothed instantaneous phase.

CHAPTER 5

Conclusion and Outlook

Conclusion

In this thesis, the development of a waveform generator that works by controlling a commercial Direct Digital Synthesizer (DDS) board via an FPGA was presented. The work began with a theoretical discussion of the principles underlying FPGAs and DDS devices. These components were highlighted for their essential roles in generating high-precision, programmable signals for applications in experimental physics. Additionally, a brief introduction to Verilog was provided, enabling a basic understanding of the hardware description language used to program the FPGA.

The implementation of the system was described in detail, including the programming of the FPGA to execute instructions in a similar manner to a microcontroller, which enabled efficient and flexible control of the DDS. Additionally a Python class was developed and presented to simplify the creation and editing of programs for generating signals. This Python class significantly enhanced the usability of the system, providing a more user-friendly interface for waveform generation and reducing the complexity of programming tasks.

Examples of generated waveforms were presented, including those specifically created for the Ytterbium experiment conducted by the Nonlinear Quantum Optics (NQO) research group at the University of Bonn. In this experiment, the waveform generator was used to drive an Acousto-Optical Modulator (AOM), which was used to broaden the frequency of a laser in a Magneto-Optical Trap (MOT). This process is critical for laser-cooling and trapping ^{174}Yb atoms. The successful reproduction of these signals validates the device's capability to meet the requirements of this experimental setup.

The design approach used in this project has proven to be successful. The combination of FPGA-based control and DDS technology has demonstrated its ability to reliably generate complex waveforms. This underscores the system's potential as a versatile and cost-effective alternative to commercial waveform generators, which are often limited in customization and significantly more expensive.

Outlook

Even though the system has demonstrated its functionality, it remains in a prototype state, and additional of work is required before it can be fully integrated into experimental setups. While the current Verilog code operates reliably, it could be optimized and simplified in many areas to improve readability. Simplifying the code will also make it easier for other users or researchers to adapt and expand the system for additional use cases.

A custom Printed Circuit Board (PCB) is currently under development. This PCB will consolidate the necessary electrical components, handle power distribution for the various devices, and route signals between the components more efficiently. By integrating these elements into a single board, the overall system design will become more compact and robust for long-term experimental use.

Currently, the speed at which programs are written to the SRAM is significantly slower than the capabilities of the devices involved. In many experimental setups, the time between measurements available for resetting and reconfiguring devices is extremely limited. Therefore, it is crucial to substantially increase the writing speed to ensure the system does not slow down the experimental cycle.

To ensure the system meets the precision requirements of experimental physics, a complete characterization of its performance is essential. This includes verifying the accuracy of generated frequencies and the timing precision of signals.

Additional efforts are also needed to provide thorough documentation and a user manual. These resources will make the system more accessible to other researchers. A well-documented system ensures that future users can easily understand, operate, and modify the setup as needed.

For integration into the existing experiments of the NQO research group, Python scripts must be developed to perform the communication between the waveform generator and the experiment control systems. These scripts will be responsible for fetching parameters from the group's databases and passing them to the functions that generate the RF signals.

Currently, the FPGA communicates with the DDS board at half the maximum data transfer speed supported by the DDS. By optimizing the Verilog code and redesigning certain modules, the communication speed and time resolution of the system could be doubled. These improvements would significantly enhance the performance and precision of the waveform generator. However, the priority remains on finalizing a fully functional first version of the system before approaching these upgrades.

Appendix

Mathematical Expression	Verilog Syntax	Description
$C = A \wedge B$	`C = A & B;`	Performs a bitwise AND operation between corresponding bits of operands A and B.
$C = A \vee B$	`C = A | B;`	Performs a bitwise OR operation between corresponding bits of operands A and B.
$C = A \oplus B$	`C = A ^ B;`	Performs a bitwise XOR (exclusive OR) operation between corresponding bits of operands A and B.
$C =\sim A$	`C = ~A;`	Performs a bitwise NOT operation, inverting each bit of operand A.
$C = A \times 2^B$	`C = A « B;`	Shifts bits of operand A to the left by B positions, effectively multiplying A by 2^B.
$C = \left\lfloor \frac{A}{2^B} \right\rfloor$	`C = A » B;`	Shifts bits of operand A to the right by B positions, effectively performing integer division by 2^B.
$C = A + B$	`C = A + B;`	Adds operand A and operand B together.
$C = A - B$	`C = A - B;`	Subtracts operand B from operand A.

Table 1: Selection of logic and arithmetic operators in Verilog. This table showcases the mathematical expression in the left and the corresponding Verilog syntax in the middle column as well as a brief description in the right column

Verilog Syntax	Description
&& (logical AND)	Evaluates to true if bot Robotics Exercises Summary Spelling and Grammar Check Plotting Timing Diagram h operands are true; otherwise, evaluates to false.
| | (logical OR)	Evaluates to true if at least one of the operands is true; otherwise, evaluates to false.
! (logical NOT)	Inverts the logical value of the operand; true becomes false and false becomes true.
< (less than)	Compares two operands and evaluates to true if the left operand is less than the right operand.
> (greater than)	Compares two operands and evaluates to true if the left operand is greater than the right operand.

Table 2: Selection of logic operators in for conditional expressions in Verilog. This table showcases the Verilog syntax on the left and a brief description on the right

State Name	Number	Description
IDLE	0	System is in Idle
OUTPUT	1	System Outputs data to the DDS
UPDATE	2	System sends IO_Update signal
WRITE	3	System is written to
WAIT	4	System waits for time
RESET	6	System is resetting
WAIT_TRIGGER	7	System waits for trigger
JMP	8	System Jumps address
TOGGLE	9	System toggles Pin
WAIT_POS_TRIGGER	10	System waits for rising edge of trigger
WAIT_NEG_TRIGGER	11	System waits for falling edge of trigger
WAIT_EDGE_TRIGGER	12	System waits for either edge of trigger
LOAD_REGISTER	13	System loads internal register

Table 3: State parameter definitions. The table lists the names of the different states, the values they represent, and brief descriptions.

Name	Hex Value	Size in bytes	Description
OUTPUT	0x00-0x1F	2-5	Write to DDS register
WAIT	0x20	5	Wait n clock cycles
WAIT_ON_TRIGGER	0x21	1	Wait until trigger is high
JMP	0x22	4	Jump to parameter
TOGGLE_P0	0x23	1	Toggle profile pin 0
TOGGLE_P1	0x24	1	Toggle profile pin 1
TOGGLE_P2	0x25	1	Toggle profile pin 1
TOGGLE_P3	0x26	1	Toggle profile pin 3
SHORT_WAIT	0x27	2	Wait n clock cycles
WAIT_POSEDGE_TRIGGER	0x28	1	Wait for rising edge of trigger
WAIT_NEGEDGE_TRIGGER	0x29	1	Wait for falling edge of trigger
WAIT_EDGE_TRIGGER	0x2A	1	Wait for either edge of trigger
LOAD_LOOP_REGISTER	0x2B	5	Load `loop_register`
LOAD_LOOP_ADDRESS	0x2C	4	Load `loop_address`
LOAD_FUNC_ADDRESS	0x2D	4	`func_address`
BEGIN_LOOP	0x2E	5	starts a loop
END_LOOP	0x2F	1	end of loop
CALL_FUNC	0x30	4	calls function
END_FUNC	0x31	1	end of function (return)
WAIT_1	0x32	1	waits 1 clock cycle
LOAD_WAIT_REGISTER	0x33	5	Loads `wait_register`
WAIT_FROM_REGISTER	0x34	1	Waits amount of clock cycles stored in `wait_register`
LOAD_LOOP_BUFFER	0x35	5	Loads `loop_buffer`
LOOP_FROM_BUFFER	0x36	1	Loops `loop_buffer` times
LOAD_FUNC_ADDRESS_BUFFER	0x37	4	Loads `func_address_buffer`
CALL_FUNC_FROM_BUFFER	0x38	1	Calls function at `func_address_buffer`
STOP_IDLE	0x7C	1	Sets state to `IDLE`
WRITE	0x7D	1	sets state to `WRITE`
RESET	0x7E	1	sets state to `RESET`
UPDATE	0x7F	1	sets state to `UPDATE`

Table 4: Full instruction set of the developed FPGA system. The table lists the instruction names, their numeric representations, and sizes, along with brief descriptions. Detailed lengths and values for the `OUTPUT` instructions corresponding to all DDS registers can be found in the AD9959 reference manual [4].

```python
def old_sawtooth(self, A, B, C, D, dt):
    t_l = np.arange(0, D[0], dt)
    t_h = t_l + self.FPGA_clk_period
    A = np.array(A, dtype=np.float64) #typecasting
    B = np.array(B, dtype=np.float64)
    C = np.array(C, dtype=np.float64)
    D = np.array(D, dtype=np.float64)

    a1 = (B[1] - A[1]) / (B[0] - A[0])  # Slope for the first segment
    a2 = (D[1] - C[1]) / (D[0] - C[0])  # Slope for the second segment
    b1 = A[1] - A[0] * a1               # Intercept for the first segment
    b2 = C[1] - C[0] * a2               # Intercept for the second segment

    lower_points = a2 * t_l + b2        # Compute lower intersection points
    upper_points = a1 * t_h + b1        # Compute upper intersection points
    slopes = (lower_points[1:] - upper_points[:-1]) / dt  # Compute slopes

    return upper_points[:-1], lower_points[1:], slopes
```

Listing 1: Listing 2: Definition of the `old_sawtooth` function.

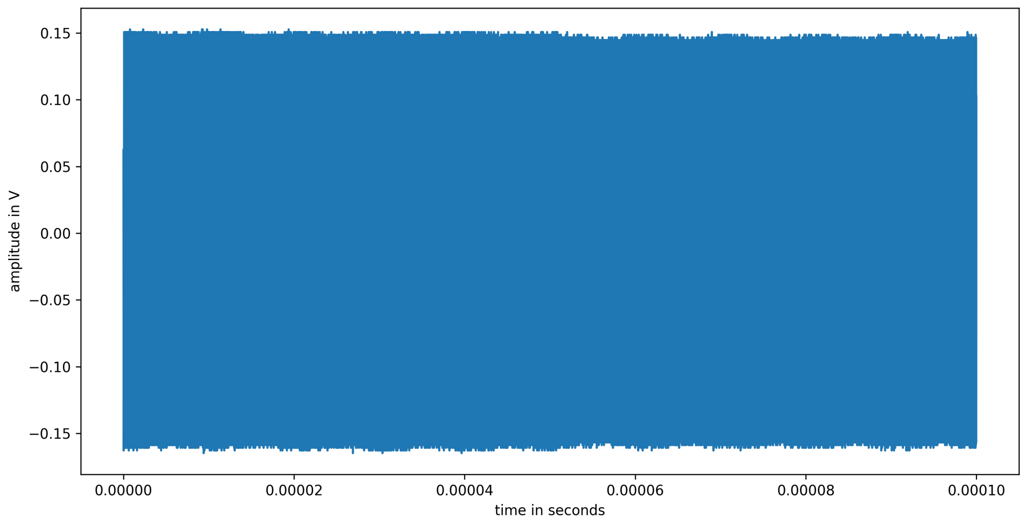


Figure 1: RF signal created by the test program in section 4.1. Measured using the MSO-X 2024A oscilloscope.

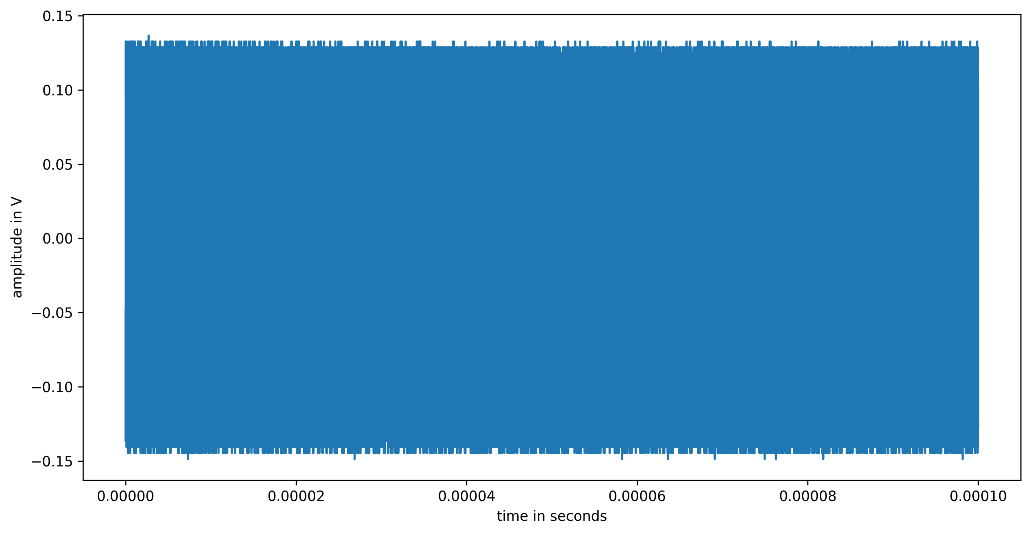


Figure 2: RF signal created by the `old_sawtooth` program in section 4.2. Measured using the MSO-X 2024A oscilloscope.

Bibliography

[1] Jeongwon Lee et al. ‘Core-shell magneto-optical trap for alkaline-earth-metal-like atoms’. In: *Physical Review A* (2015). DOI: 10.1103/PhysRevA.91.053405. URL: https://journals.aps.org/pra/abstract/10.1103/PhysRevA.91.053405.

[2] Mogens Henrik. ‘Towards Rydberg quantum optics with ultra-cold Yb atoms’. masters thesis. Denmark, Odense: University of southern Denmark, 2021.

[3] Xin Wang et al. ‘Two-color Ytterbium MOT in a compact dual-chamber setup’. In: *arXiv preprint* (Aug. 2024). Preprint, available on arXiv. arXiv: 2408.03310. URL: https://arxiv.org/abs/2408.03310.

[4] Analog Devices. ‘Data Sheet AD9959’. In: (2016). URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ad9959.pdf.

[5] Analog Devices. ‘User Guide Evaluation Board for 4-Channel 500 MSPS DDS with 10-Bit DACs’. In: (2016). URL: https://www.analog.com/media/en/technical-documentation/user-guides/eval-ad9959.pdf.

[6] Jürgen Gutekunst Ekbert Hering Julian Endres. *Elektronik für Ingenieure und Naturwissenschaftler*. German. 8. ed. 2021. ISBN: 978-3-662-62697-9.

[7] Electrical Design News (EDN). ‘All about FPGAs’. In: (Mar. 2006). URL: https://www.edn.com/all-about-fpgas/.

[8] DR. SHIRSHENDU ROY. ‘FPGA: Basic Overview’. In: (May 2019). URL: https://digitalsystemdesign.in/fpga-basic-overview.

[9] Digilent. ‘Cmod A7 Reference Manual’. In: (Aug. 2019). URL: https://digilent.com/reference/_media/reference/programmable-logic/cmod-a7/cmod_a7_rm.pdf.

[10] J. Bhasker. *Verilog HDL Synthesis A Practical Primer*. en. Star Galacy Publishing, 1998. ISBN: 0-9650391-5-3.

[11] Inc. Integrated Silicon Solution. ‘512K x 8 HIGH-SPEED ASYNCHRONOUS CMOS STATIC RAM data sheet’. In: (2009). URL: https://www.issi.com/WW/pdf/61-64WV5128Axx-Bxx.pdf.

[12] Frederick W. King. *Hilbert Transforms*. Vol. 1. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011. ISBN: 9780521887625. URL: https://www.cambridge.org/us/universitypress/subjects/mathematics/abstract-analysis/hilbert-transforms-volume-1?format=AR&isbn=9780511888977.

[13] William D. Phillips. ‘Nobel Lecture: Laser cooling and trapping of neutral atoms’. In: *Reviews of Modern Physics* (July 1998). DOI: 10.1103/RevModPhys.70.721. URL: https://link.aps.org/doi/10.1103/RevModPhys.70.721.