822H1: Reconfigurable Systems on Chip

Coursework and Mini project

Candidate Number: 218831

Due date: 30/04/2019

Contents

Introduction to FPGAs

A Field Programmable Gate Array is a device with programable hardware instead of programable software as a microcontroller which is more common to find in an embedded device. This type of devices allows a different kind of applications to be developed, as its higher clock speeds and the capability to implement combinational and sequential logic in code, makes possible for engineers to perform exhaustive Digital Signal Processing and Image Processing.

Previous to FPGAs, other devices such as Programmable Array Logic (PAL) which had a limited number of logic gate arrays, even fewer flip-flops and an array of connections; but are still used for their low cost. After PALs, Complex Programmable Logic Devices (CPLD) was developed to address the limitations PALs have by having several blocks containing a PAL in each one and a broader array of connections. [1]

FPGAs had a similar concept of multiple PALs but using Logic Blocks (LB) (also called Logic Element Blocks) instead of the earlier, adding programmable IO hardware and an increased number of programmable connection arrays.

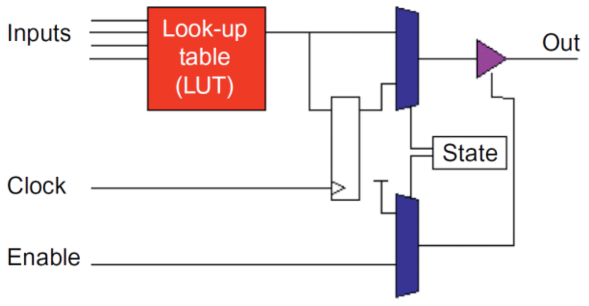


Figure 1. Typical CLB with a 4-input LUT. [1]

A CLB has much broader configuration options, as they implement Look-Up Tables (LUT) while avoiding fixed logic gates. Using CLBs, combinational and sequential systems can be optimally designed by using the LUT to emulate the function of logic gates and using multiplexors to toggle between the LUT and a Set-Reset (SR) Flip-Flop to get sequential circuits. Due to the modern advancements of CMOS technology, millions of LBs can be packed into a single silicon chip, allowing more complex designs to be developed and better performance. [1]

The second building block of FPGAs is the Input/output Block (IOB); which interfaces the LBs with the available IO in the FPGA package. This interface is like what is found in a microcontroller's GPIO peripheral with some additional components; pull-up, pull-down and tri-state arrangements, Flip-Flops in both input and output stages and multiplexors to select using these Flip-Flops or not. IOBs are meant to be used along with LBs to interface the programmed gate array with the rest of the embedded system. [2]

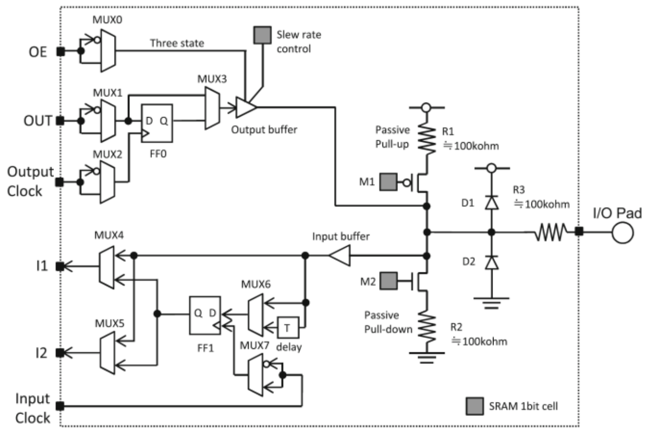


Figure 2. IOB of the Xilinx XC4000. [2]

The last fundamental characteristic is the programmable interconnection elements; this component allows to link several blocks inside the FPGA. This component consists of another three elements in the FGPA: Switching Blocks (SB), Connection Blocks (CB) and Routing Channels. A routing channel is a "wire node" in which several elements in the FGPA are connected. The Connection Channel is an interface for connecting an element in the FPGA to one or multiple Routing Channels or an IOB. Lastly, the Switching Block connects Routing Channels. [2]

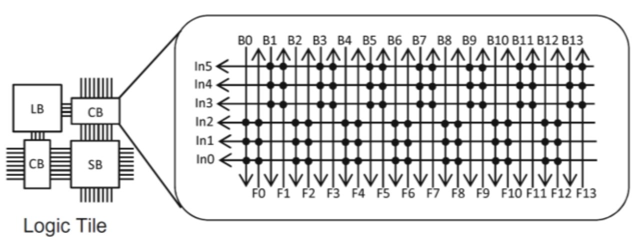

LB
CB
CB
SB
Logic Tile
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13
In5
In4
In3
In2
In1
In0
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13

Figure 3. Switch arrangement of a Connection Block. [2]

Other elements such as SRAM, DSP Slices, Phase-Locked Loops and Delay-Locked Loops are also available in some FPGAs, but these are commonly categorized as Dedicated Hard Block because these cannot be programmed.

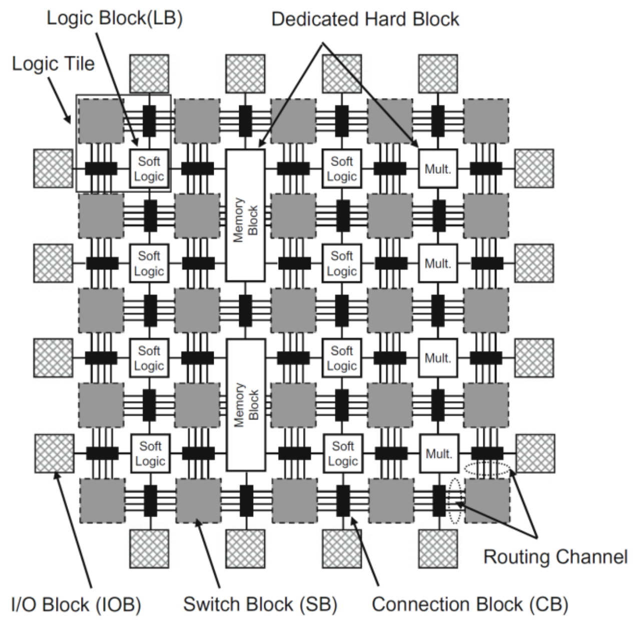


Figure 4. Island-style FPGA overview. [2]

FPGA Manufacturers

There are many FPGA manufacturers in the market at the time of the writing of this report, companies such as Xilinx, Altera (Recently absorbed by Intel), Lattice Semiconductor, Microsemi and QuickLogic offer engineers a competitive market of devices with a wide variety of characteristics and price.

From all the previous mentioned manufacturers, the two that have the most market share are Xilinx and Altera. Although both of these companies have a great community of users, Xilinx FPGAs are most likely to be found on high-end equipment such as Software Defined Radio units, while Altera FPGAs are known for their lower prices.

In a deeper level, Xilinx and Altera have some core differences starting with how they name the elements inside their FPGAs. For example, the LB in Altera's Architecture is called the Logic Array Block, while Xilinx's architecture calls them Complex Logic Blocks.

Altera's Adaptive Logic Module

Altera's architecture supports Adaptive Logic Modules (ALM) which are LABs with an 8-LUT, 2 Flip-Flops and two full adders joined using carry-in/out bits. This block allows the user to be used as 2 ½ LABs, as the 8-LUT can be split without losing input bits (Xilinx's CLBs cannot do this function) and the remaining elements are enough to configure 2 Logic Elements. [3]

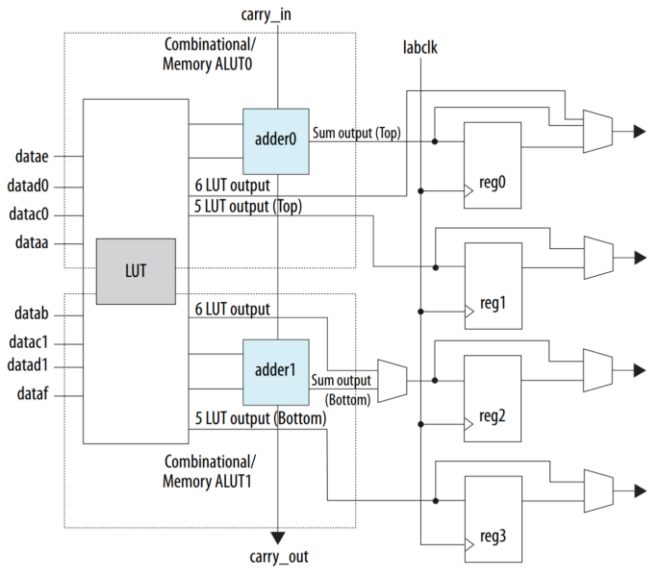


Figure 5. Intel Stratix 10 ALM High-Level Block Diagram. [3]

Xilinx FPGA Families

Xilinx 7 series of FPGAs include four families of devices: Spartan, Artix, Kintex and Virtex. These families have been designed to achieve a unique price and performance ratio to suit a wide range of applications, starting from the low-cost, mass-produced devices to the expensive high-performance devices. According to Xilinx [4], each family is optimized for a specific use case or budget.

- Spartan: Optimized for low-cost, low power, small footprint, and high IO applications.
- Artix: These devices are optimized for low power applications that require serial transceivers and DSP, and their cost is indirectly low due to the low quantity of external components required.
- Kintex: These devices are optimized for achieving a high performance/cost ratio. It features all the characteristics of the Artix family while increasing performance.
- Virtex: These devices are made for achieving the highest performance of all the families.

The 7-series devices feature the following characteristics:

- 6-input LUT configurable as distributed memory
- Support of DDR3 interfaces up to 1866 Mb/s
- High-speed serial connectivity (from 600 Mb/s to 28.05 Gb/s) while keeping low power consumption
- On-chip 12-bit 1 MSPS
- DSP Slices with 25x18 multipliers, 48-bit accumulators and pre-adder.
- Clock management based on combining PLL and mixed-mode clock manager blocks.
- Microblaze soft-processor support
- PCIe Gen3 support
- Wide variety of security encryption functions such as 256-bit AES and 256-bit SHA
- Low voltage requirement for low power consumption

Artix7 FPGA Family

The Artix7 family of FPGAs is part of the 7th generation of Xilinx's FPGA families. This family is optimized for low power DSP applications, and also requires the smaller number of external components for the FPGA to work. In comparison to the other 3 FPGA families Xilinx offers, the artix7 comes third on raw performance, which is compensated with other features such as price and implementation complexity.

Family Characteristics

Parameter	Value
Logic Cells	12K-215K
Block RAM	720 – 13140 Kb
DSP Slices	40 - 740
DSP Performance	929 GMAC/s
MicroBlaze CPU	303 DMIPs
Serial Transceivers	16
Serial Transceiver Speed	6.6 Gb/s
Serial Bandwidth	211 Gb/s
PCIe Interface	4 Gen2
Memory Interface	1066 Mb/s
I/O Pins	500
I/O Voltage	1.2-3.3V

Table 1. Artix 7 family characteristics.

Some other features of the Artix-7 family are the following:

- Mixed-Mode Clock Manager and Phase-Locked Loop: This block can be used as a frequency synthesizer or as a filter for input clocks.
- Block RAM: Dual-port 36 kb block RAM; this can be accessed by its two ports while only sharing the stored data.
- DSP Slices: 240 25x18 2's complement multipliers and pre-adder for DSP applications.
- Automotive temperatures certification.

Systems on Chip in FPGAs

FPGAs allows the design and use of Systems on Chip (SoC). These systems consist of a microcontroller and dedicated hardware to perform a specific operation. The microcontroller section of an SoC can be limited to the fundamental components, such as the logic unit and memory; removing the optional peripherals such as SPI, I2C UART, and many more. The specialized hardware on an SoC can be an analog or digital system that allows faster data processing or manipulation of particular ports or other devices; this can be improved by the possibility of inserting these systems into the main busses of the microcontroller, resulting in faster performance than having an off the shelf microcontroller with added external circuitry.

SoC can be found in many products and applications:

- **Mobile devices**: To process RF signals from cell carriers and WIFI.
- **Single Board Computers**: These devices hold a functional computer capable of running mainstream operative systems often in a single chip.
- **Datacenters**: Tasks from datacenters are routed into several SoC

System on Chip Design Workflow

The design cycle of a device based on an SoC is not as different as the one for a conventional embedded device. The most significant difference is the addition of the Register-Transfer Level, which is the high-level hardware abstraction layer for the circuit inside the device.

Since many platforms and vendors have their particular set of tools and architectures, the design cycle can be significantly affected just by deciding with which product the SoC is going to be developed. Manufacturers as Xilinx offer the Zynq SoC that already has one or two hardcoded ARM Cortex A9 processor with many AMBA (The highest performance bus in the ARM architecture) available to use in the programmable logic section of the SoC. Some other architectures let the designer select what architecture is going to be implemented using IP blocks that generate the RTL since these processors are coded into the FPGA, these are called soft processors.

Unlike standard embedded and computer programming, SoC design requires the engineering team to develop custom hardware blocks, meaning knowing at least one Hardware Description Language (HDL) on top of the embedded software language, which is typical for it to be C or C++.

The design flow of an SoC is commonly divided into two categories [5]:

- **Front End**: Includes the design decisions, RTL structure given by an HDL, software, logic flow and simulations. This step produces Structural RTL (Hardware and software behaviour and configuration)
- **Back End**: The SoC Structural RTL is synthesized, and the SoC silicon is manufactured if needed.

Front End

The design of the SoC is done in this step and mostly follow the following flow chart.

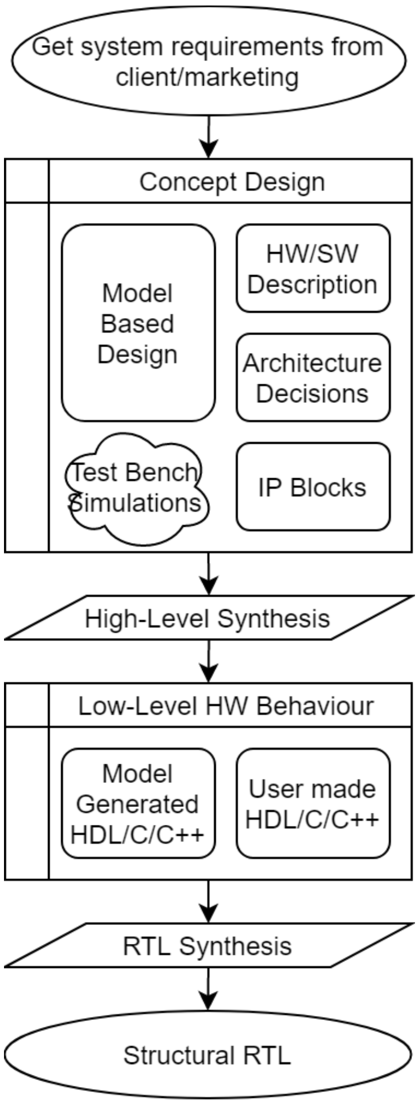


Figure 6. Design Workflow for SoC.

The design cycle starts when system requirements are defined by interpreting the requirements provided by the user or the marketing requirement; this is translating features into engineering performance goals and required characteristics. For example, the client can ask for a wireless closed circuit with a live feed from each camera; the engineering team must decide what wireless protocol can deliver the expected performance. In this scenario, Bluetooth is a wireless communications protocol, but it is not as fast, reliable, and flexible as Wi-Fi.

The design team can choose what architecture to use using these engineering goals and whether they are going to use third-party or in-house Intellectual Property (IP) blocks. A detailed description of the hardware and software sections of the SoC must be defined and (if possible) implemented in high-level synthesis (HLS) software to continue development using Model-Based Design.

Development using this design paradigm is more practical and faster to verify. A model has to be programmed using an HLS tool such as MATLAB/Simulink, SystemC or Python to describe the intended functionality of the SoC. Hardware blocks can be designed considering the real capabilities of the FPGA and SW can be compiled even using the soft processor's syntax if the IP developer supports the HLS tool. After creating a model and algorithm, the system can be simulated to see close to real performance before deploying a prototype. The effectiveness of the model-based design is as good as the accuracy of the model.

The following step can differ from many types of development paradigms. Assuming the development team has used Model-Based Design, both HDL and C/C++ code is generated by the HLS tool with just a few modifications, and fill-in-the-gaps are required from the developers. If Model-Based design was not used, then the development team must manually code both HDL and C/C++ code. Doing this can lead to longer development times but also more efficient code since HLS tools output generic and safe code reusable for many platforms.

Once both HDL and software are verified using Model-Based design or HDL simulation, the system can be synthesized again to get the structural RTL. Testing can be done in an FPGA development kit, with the required hardware to quickly link the FPGA and a PC for it to upload the structural RTL into the board; this is usually the last step of system verification.

Back End

Once the structural RTL is out, it can be used in two ways. The first one is for the design team to generate a bitstream file and then use this file in mass production, assuming an FPGA is used as the embedded target. This method can save a lot of time and money since cheap FPGAs can be found from different manufacturers at various costs as previously stated. The main disadvantage is that performance will be affected by the internal structure of the FPGA, and a memory device on the PCB board has to be included in order to store and load the configuration each time the device is powered.

The other alternative is to develop an SoC using the same procedure for an Application Specific IC (ASIC). The Structural RTL is used to generate a photomask to make SoC ICs in mass production; this single step is the most expensive in the order of tens of hundred-thousand GBP [5]. Photomask manufacturing is meant to be done only once due to the high cost, so validation and simulations in the Front-End step allow this to be faster.

Deciding whether using FPGAs of custom silicon is a matter of volume, performance and expected lifetime. Products with not critical performance requirements and low production volume may be manufactured using FPGAs. Meanwhile, products that benefit from the custom layout of an ASIC and large product volume are more likely to be manufactured using custom silicon.

Intellectual Property Blocks

IP blocks are hardware blocks that can be used in designs which are provided by the FPGA manufacturer or a third-party vendor. These blocks are meant to accelerate the design process of an embedded product by as they have a specific, complex and hard to develop functionality. The IP blocks available can fulfil many kinds of tasks, such as DSP, mobile comms, embedded systems, and many others.

It is common to find licencing costs of IP blocks, mostly depending on the complexity of the task it performs and the company that provides the block, as FPGA manufacturers often release IP blocks for free for engineers to choose their products.

Two IP blocks were researched in order to show the different types of blocks available. One of them is provided by an FPGA manufacturer and the other by a third-party supplier.

ARM Cortex-M1 DesignStart

The Cortex M1 DesignStart Xilinx edition from ARM is an IP block for implementing one or several Cortex-M1 soft processors in the Vivado environment. This IP block is provided by a third-party supplier (ARM) for free, and it is compatible with all the 7-series and Zynq devices from Xilinx. IP blocks like these are essential for engineers to design SoC, as embedded software engineers are more familiarized with the ARM architecture than any other soft processor as Xilinx's Microblaze and Picoblaze.

The Cortex-M1 processor is intended to be implemented in FPGAs. It is based in the ARM6-M architecture; similar to the Cortex-M0 and Cortex-M0+ processors, which are not intended for FPGA implementation. The following summary assumes the reader to have a basic understanding of the ARM Cortex architecture.

The Cortex-M1 DesignStart FPGA for Xilinx includes the following features:

- The Cortex-M1 soft processor.
  - 1, 8, 16 or 32 interrupts
  - Configurable endianness (Little by default)
  - JTAG debug port
  - Thumb-2 instruction set
  - AMBA, AHB and TCM (optional) buses
  - Big or Small multipliers
  - OS Extensions
- AHB to AXI bridge to interface the Cortex-M1 processor to the standard Vivado components.
- Support for the V2C-DAPLink debugger.
- ARM's CMSIS board support package compatibility.
- Instruction Set Simulation that can be performed in Vivado
- Example designs for two Xilinx development boards.

After installing the IP block, it will be available in Vivado's IP catalogue. Selecting the block will show the available settings, which are divided into four tabs:

- **Configuration:**
  - **Number of interruptions**: 1 to 32 interruption can be selected.
  - OS Extensions: Enables interrupt sources such as the NVIC, and Systick.
  - **Small multiplier**: This changes the multiplier setting from executing a 32-bit multiplication in a single cycle to 32 cycles.
  - **Big-endian**: This changes the order of the Most Significant Bit to Big Endian.
- **Debug**
  - **JTAG**: Enables the JTAG debug port.
  - Serial Wire: Enables the Serial debug port.
  - **Small Debug**: Reduces the debug performance to resources assigned to debugging will be reduced.
- **Instruction Memory**
  - **ITCM Size**: Sets the size of the Instruction-Tightly Coupled Memory for high-performance Direct Memory Access.
- **Data Memory**
  - **DTCM Size**: Sets the size of the Data-Tightly Coupled Memory for high-performance Direct Memory Access.

The settings selected will output an IP block with a different number of signals. An IP block with the most available signals is displayed below. The signal descriptions are available in [6] and are divided into the following categories.

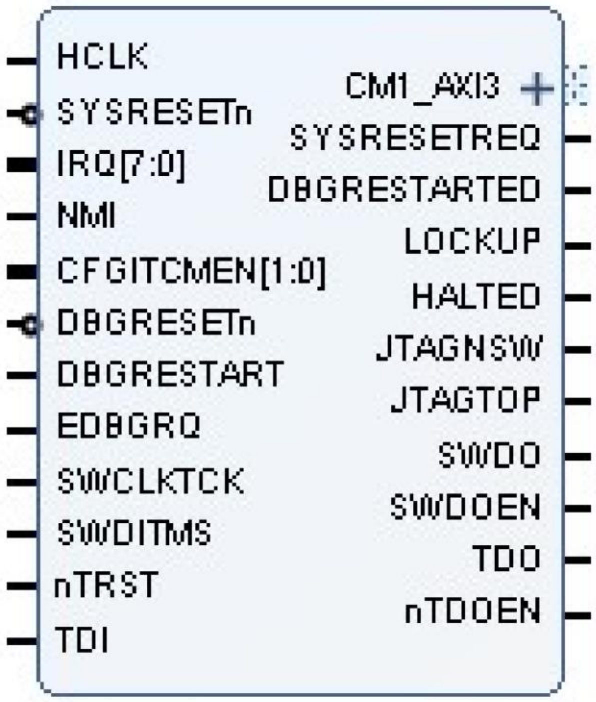


Figure 7. ARM Cortex-M1 IP Block.

- **Clocks and Resets**
  - **HCLK**: Main Processor clock.
  - **DBGResetn**: Reset for debug logic.
  - **SYSRESETn**: System Reset.
- **Miscellaneous**:
  - **LOCKUP**: Indicates core lockup.
  - **HALTED**: Indicates halting debug mode.
  - **SYSRESETREQ**: Requests that the system reset controller resets the core.
  - **EDBGRQ**: External debug request.
  - **DBGRESTART**: Requests that the system reset controller resets the core.
  - **DGBRESTARTED**: Handshake for DBGRESTART.
- **Interrupt Interface**:
  - **IRQ**: External interrupt signal.
  - **NMI**: Non-maskable interrupts.
- **Memory Interfaces**:
  - CFGITCMEN: ITCM Alias enable.
- **Debug**:
  - **JTAG~**: Input/Output lines for JTAG interface.
  - **SW~**: Input/Output lines for Serial Wire interface.
  - **nTRST/TDO/TDI/nTDOEN**: DAP Link lines.
- Bus:
  - CM1_AXI3: Interface to the AXI bus connected to the AHB.

As seen in Figure 7, the IP block does not provide access to the AHB bus, but to the AXI instead. Also, all the peripherals (GPIO, SPI, UART) and SRAM in the Cortex-M1 memory map have been moved to the AXI bus. All the previously mentioned components can be accessed from the CM1_AXI3 signal in the IP block.

A compiled binary file must be imported into Vivado, or a DAPLink must upload the same file in order to flash the Cortex-M1 processor. ARM's Keil MDK uses the CMSIS board support package to compile code intended for the specific soft processor the user has inserted into the FPGA.

FIR Compiler

Finite Impulse Response (FIR) filters are widely used in DSP related applications; features like unconditional stability and no feedback requirement are hugely useful for any kind of application. FIR filters can be used to develop Low-Pass, High-Pass, Band-Pass, Notch and many other kinds of filters using a vector consisting of the current and previous filter inputs. The significant disadvantage of this kind of filter is the increased filter order to achieve the same performance of an Infinite Impulse Response Filter (IIR), being this a difference in the order of tens of coefficients.

Programming FIR filters using a computing device (PC or microcontroller) with the increased number of coefficients a High-Performance FIR filter requires both computational speed to calculate the output given all the elements of the input vector and memory to allocate the input vector and the coefficient values. Although, more complex filters are available with multi-rate and multiple sets of coefficients.

As an example, adaptive FIR filters can be used to convert square wave signals (Common in digital systems) into analog mostly pure sine waves; this is often used in musical applications. Even filtering square waves are often found in digital synthesizers to tweak resonance, changing the overall sound.

FIR Filters can be developed and simulated in the Model-Based design; tools for implementation are often available in HLS software, such as MATLAB's Filter Designer App.

The FIR Compiler can implement and analyze several types of FIR filters. The IP block supports single-rate, multi-rate and oversampling variants of the Interpolation ($f_{out}>f_s$), Decimation ($f_{out}<f_s$) and Hilbert filters (IQ Filter). The frequency response of the filter, resource usage and transient performance are all shown for every previously mention filter implementation. Finally, optimization goals can be set for the RTL section to be faster or to have a smaller footprint.

FPGAs have DSP slices to perform the Multiply and Accumulate (MAC) operation. 7-series devices can perform 25x18 bits multiplications enabling higher accuracy calculations. As stated before, FIR filters require a large number of coefficients and multiplications, which can lead to less footprint efficiency; DSP slices use MAC engines in order to compensate for this, which store both input and coefficient vectors in memory to compute the output of filters of any order using a single MAC.

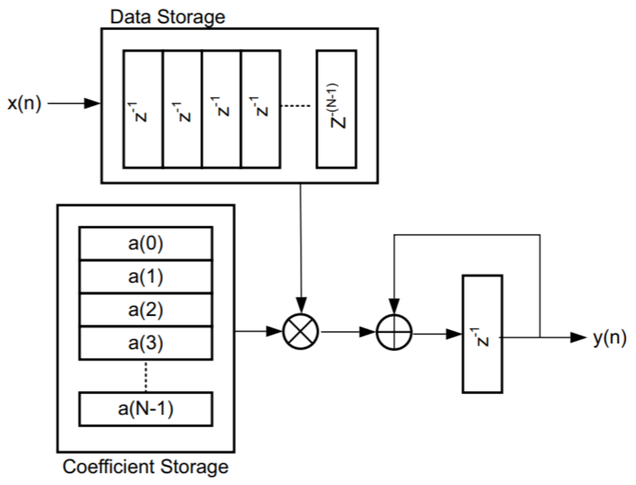


Figure 8. MAC engine block diagram. [7]

On top of this, two filter architectures are supported in Xilinx FPGAs. The systolic MAC consists of pipelined MAC operations, every clock cycle interleaved coefficients, and input signals are processed using MAC operations, and at the end, the filter equation is fulfilled; this configuration greatly enhances footprint efficiency. Transpose MAC consists of several simultaneous multipliers and only pipelined additions; this reduces latency at the cost of a more prominent footprint because multiplexers are needed in the MAC engine implementation.

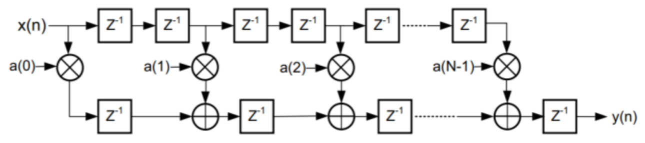


Figure 9. Common Systolic MAC Filter (Not using MAC Engines).

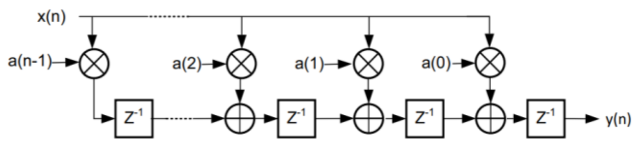


Figure 10. Common Transpose MAC Filter (Not using MAC Engines)

The Filter Compiler configuration window is divided into the following tabs. (Only fundamental parameters are detailed; the remaining are summarized)

- **Filter Options**
  - **Filter coefficients**
    - **Select Source**: Coefficient source selection.
    - **Coefficient Vector**: Coefficient list (Vector Source)
    - **Coefficient file**: .coe file with filter coefficients (File Source)
    - **Number of Coefficient sets**: Number of different coefficient configurations. Every set must have the same filter symmetry and length.
    - **Use Reloadable Coefficients**: Enables to input coefficients using an input signal.
  - **Filter Specification**

- **Filter Type**: FIR filter type such as Single-Rate, Interpolation, Decimation and Hilbert.
- **Rate Change Type**: Selects an integer multiple or fractional part of the sample rate. It is only used in multi-rate filters.
- **Interpolation Rate Value**: Specifies the sample rate factor for interpolation. It is only used in Interpolation filters.
- **Decimation Rate Value**: Specifies the sample rate factor for Decimation. It is only used in Decimation filters.
- **Zero Pack Factor**: Specifies the packing factor for interpolated filters.

- **Channel Specification**
  - **Interleaved Channel Specification**
    - Contains Configuration for interleaved filters
  - **Parallel Channel Specification**
    - **Number Paths**: Sets the number of parallel filter paths; This is to achieve lower latency filters.
  - **Hardware Oversampling**
    - **Input sample Frequency**: Sets the FIR filter sample frequency.
    - **Clock Frequency**: Sets the input clock frequency.
- **Implementation**
  - It sets the configuration of the input coefficient and input signal data types.
- **Detailed Implementation**
  - It sets optimization goals for the filter's implementation.

The IP block signals are divided into the following tabs. (Non-required signals are shown greyed)

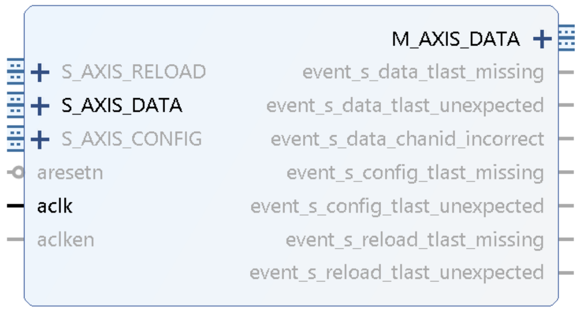

M_AXIS_DATA
S_AXIS_RELOAD
S_AXIS_DATA
S_AXIS_CONFIG
aresetn
aclk
aclken
event_s_data_tlast_missing
event_s_data_tlast_unexpected
event_s_data_chanid_incorrect
event_s_config_tlast_missing
event_s_config_tlast_unexpected
event_s_reload_tlast_missing
event_s_reload_tlast_unexpected

Figure 11. FIR IP block.

- **Required Signals**
  - **S_AXIS_DATA**: Filter's input signal.
  - **M_AXIS_DATA**: Filter's output signal.
  - **aclk**: Clock signal.
- **Non-required Signals**
  - **S_AXIS_RELOAD**: Filter coefficients vector input.
  - **S_AXIS_CONFIG**: Filter configuration and event flags, including enabling, valid input and input ready.
  - **aclken**: Clock signal enable.
  - **aresetn**: Filter reset.
  - **event_s_data**: Filter event data flags such as output ready, missing input, unexpected input and more.

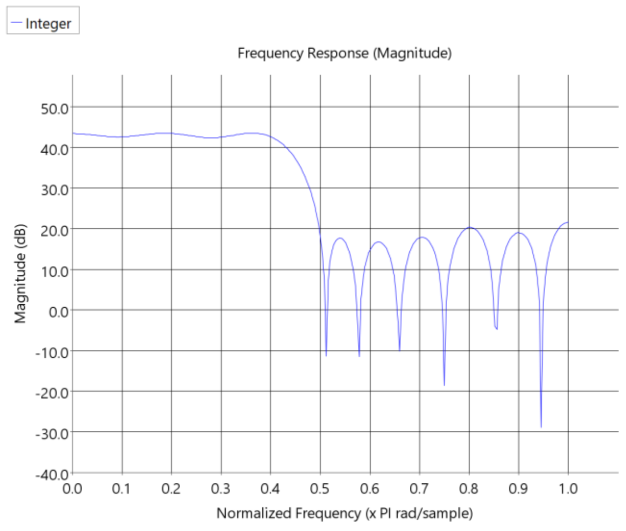
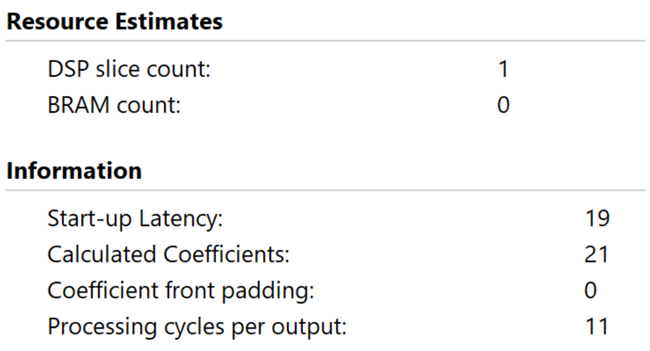


Figure 12. Frequency response for the default FIR filter in the FIR Compiler.


Resource Estimates

DSP slice count:	1
BRAM count:	0

Information

Start-up Latency:	19
Calculated Coefficients:	21
Coefficient front padding:	0
Processing cycles per output:	11

Figure 13. Implementation details for the default FIR filter in the FIR Compiler.

Mini-Project

The mini-project described in this report is a simple musical synthesizer using serial messages coming from the USB port on-board and hardware square wave generators with volume control.

The objectives of the synthesizer to meet are to be able to output at least two octaves of the chromatic scale (C, C#, D, D#, E, F, F#, G, G#, A, A# and B, twice), increase or lower volume and to change the output pitch according to the input serial message from a PC.

System Block Diagram

A proposed architecture was designed to fulfil the previously stated goals. A simple SoC was designed to process the input messages using software and to generate the output signal and volume control using hardware.

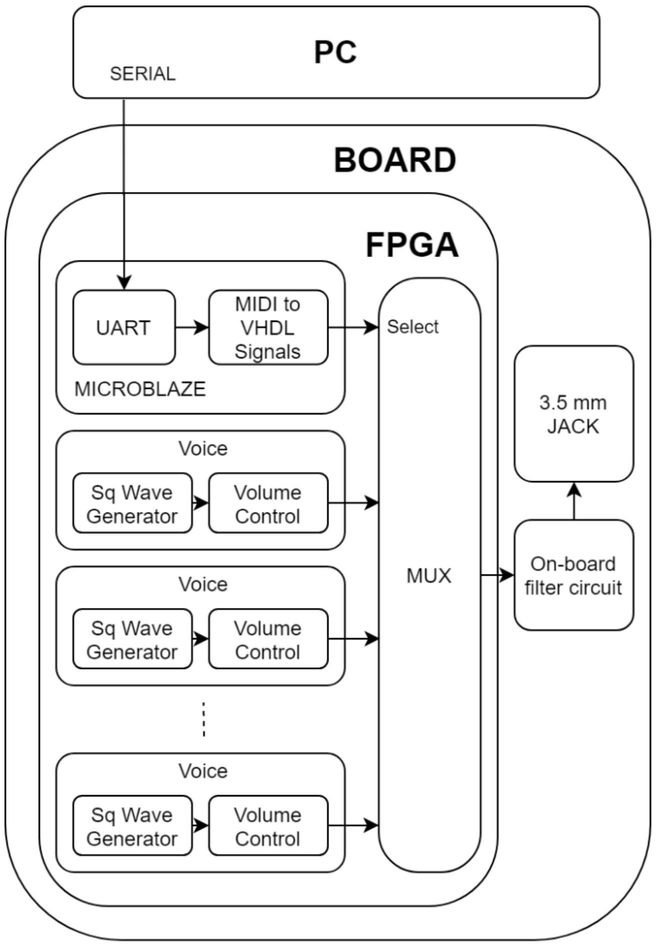


Figure 14. Music Synthesizer Block Diagram.

The system is divided into different sub-systems inside the FPGA section of the diagram. The FPGA block represents the Top-Level block of the SoC, which instantiates several other components, these being divided in Software and Hardware.

Top-Level System

COMMENTED AND FORMATTED CODE AVAILABLE IN THE APPENDIX

The hardware section of the SoC consists of many signal generator commonly known as Voices in music synthesiser terms (Or Sound Alchemy as experts like to call it), and a multiplexer routing the output of all Voices into a single output signal selected by the software.

The software system of the SoC consists of the soft processor and the software inside it. The soft processor is a Microblaze from Xilinx, which is a 32-bit architecture, licence free and easy to implement using the Xilinx Environment. This processor interfaces the PC with the system and controls the hardware side of the Top-Level design.

The Top-Level entity in the VHDL model requires many IO ports on-board defined in the board's constrain file. These ports are the switch input ports, on-board 100 MHz clock signal, onboard LEDs, GPIO for debugging, amplifier enable flag, amplifier input signal, Rx port from on-board UART, TX port from in-board UART, on-board CPU reset pin, respectively as shown in the following snippet.

```
ENTITY top IS
PORT (
sw : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
clk : IN STD_LOGIC;
led : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
JA : OUT STD_LOGIC;
ampSD : OUT STD_LOGIC;
ampPWM : OUT STD_LOGIC;
RsRx : IN STD_LOGIC;
RsTx : OUT STD_LOGIC;
btnCpuReset : STD_LOGIC
);
END top;
```

Script 1. Top-Level entity port definition.

The only architecture of the entity first defines the design constants and signals. The architecture was coded in a way that it is generic in the way that the number of octaves can be modified without changing anything else; this allows to have a single configuration constant representing the number of desired octaves in the system. Some other constants are the number of tones in the system; as an octave consists of 12 semi-tones, the number of constants is defined by 12 times the number of octaves. Lastly, an array containing the frequencies for the 12 semi-tones in the third octave (From C_3 to B_3), these will be used to calculate the frequencies in higher octaves.

The defined signals are the intermediary nodes between ports and other entities within the Top-Level entity. These are a vector that holds the signal of every tone in the system, the overall output signal going to the amplifier, the multiplexor's select signal, and the soft processor's reset signal, respectively as shown in the following snippet.

```
ARCHITECTURE Behavioral OF top IS

  CONSTANT N_OCTAVES : INTEGER := 5;
  CONSTANT N_TONES : INTEGER :=
                     12 * N_OCTAVES;
  TYPE FREQUENCIES IS ARRAY (0 TO 11) OF
                     INTEGER;
  CONSTANT FREQ_TONES : FREQUENCIES :=
  (131, 139, 147, 155, 165, 175,
  185, 196, 208, 220, 233, 245);

  SIGNAL voice_out : STD_LOGIC_VECTOR
                     (0 TO (N_TONES - 1));
  SIGNAL audio_out : STD_LOGIC;
  SIGNAL mux_ctrl : STD_LOGIC_VECTOR
                     (0 TO 31);
  SIGNAL GPIO : STD_LOGIC_VECTOR
                     (31 DOWNTO 0);
  SIGNAL reset : STD_LOGIC;

BEGIN
```

Script 2. Top-Level constants and signals.

The first part of the behavioural part of the architecture is the soft processor instantiation. The configuration of this IP block will be discussed in HEADER. The processor only requires a few connections, being these the input clock cycle, an active-low reset signal, UART RX and TX ports from the on-board UART receiver and the output GPIO signal to drive the multiplexer. These signals, as were described, are shown in the instantiation of the soft processor in the Top-Level entity.

Just before the instantiation, the processor's reset signal is defined as the logic inverse of the CPU reset button, as it has a pull-down resistor arrangement.

```
BEGIN
  reset <= NOT btnCpuReset;
  MB : ENTITY work.microblaze_mcs_0
    PORT MAP(
      Clk => Clk,
      Reset => reset,
      UART_rxd => RsRx,
      UART_txd => RsTx,
      GPIO1_tri_o => mux_ctrl
    );
```

Script 3. Soft Processor instantiation.

The following section is for instantiating the voices. Since this VHDL entity was written as a generic entity; every instantiation is done inside two for loops instead of hardcoding all the instantiations. The first for loop iterates through every octave, while the second for loop iterates through all the semi-tones.

Each voice requires a generic input, being this the desired voice frequency, obtained by doubling the base frequency N-1 times, N being the number of desired octaves. The following equations demonstrate and model how extrapolating frequencies work.

$$C_0 = \frac{1}{2}C_2 = \frac{1}{4}C_3 = \frac{1}{8}C_4 = \frac{1}{2^{N-1}}C_N$$
$$f_N = 2^{N-1}f_0$$
$$f_{C_3} = 2^{3-1}f_{C_0}$$

Equation 1. Frequency extrapolation equation.

The remaining ports are an input clock signal (Assumed to be of 100 MHz by the Voice entity), a volume control signal, being this the first three on-board switches and the output signal. The output signal is meant to be stored in the voice_out vector, which contains the output signal of every voice; both for loop variables are needed to route the output signal to the voice vector correctly.

```
octaves : FOR k IN 0 TO (N_OCTAVES - 1)
GENERATE
  voices : FOR i IN 0 TO 11 GENERATE
    voice : ENTITY work.voice
        GENERIC MAP(FREQ_TONES(i)*2**k)
        PORT MAP(
          clock => clk,
          volume => sw(3 DOWNTO 0),
        out_main => voice_out
                    (i + 12 * k));
    END GENERATE voices;
END GENERATE octaves;
```

Script 4. Top-Level voice instantiation.

As an example of how useful generic entities are, assume 4 octaves are going to be implemented in the synthesizer; this results in a total of $12 \times 4 = 48$ voice instantiations. Hardcoding this while modifying the desired voice signal would be a time-consuming task.

The multiplexer was coded as a behavioural process instead of using the WITH SELECT WHEN statement to keep the generic nature of this entity; and since coding a process allows sequential instructions, an if statement was introduced to filter between zero and non-zero select signals to silence the synthesizer when an unsigned integer representation of the select signal equals to zero. If Non-zero and less than the number of semi-tones in the system, the selected voice is the one given by the select signal minus one due to an offset between select and voice vectors.

```
PROCESS (mux_ctrl)
 CONSTANT MUX_INT : INTEGER :=
to_integer(resize(unsigned(mux_ctrl),32))
;
BEGIN
  IF (MUX_INT = 0) OR (MUX_INT > N_TONES)
THEN
    audio_out <= '0';
  ELSE
    audio_out <= voice_out(MUX_INT - 1);
  END IF;
END PROCESS;
```

Script 5. Top-Level Multiplexor process.

The multiplexer process routes the voice vector signals to the amplifier's input node. A more straightforward representation of the multiplexer process is given in the following equation.

$$mux = \begin{cases} \in [1, N_{ST}], & sound \\ \in [0] \cup (N_{ST}, \infty), & silence \end{cases}$$

$$Voice_{out} = \begin{cases} voice_{mux-1}, & mux > 0 \\ 0, & mux = 0 \mid mux > N_{ST} \end{cases}$$

Equation 2. Multiplexer and Select signal behaviour.

Finally, the remaining signals are the amplifier's enable and input signal, in addition to two debug signals (LEDs and a mirror of the amplifier's input signal)

```
ampSD <= sw(15);
ampPWM <= audio_out;

led <= mux_ctrl(0 TO 15);
JA <= audio_out;
END Behavioral;
```

Script 6. Top-Level output and debug signals.

Voice Entity

As stated in the Top-Level System description, the Top-Level instantiates Voice entities, which are responsible for both generating the voice signal and controlling volume. The following figure is a simplified block diagram of the voice entity since not every signal is represented.

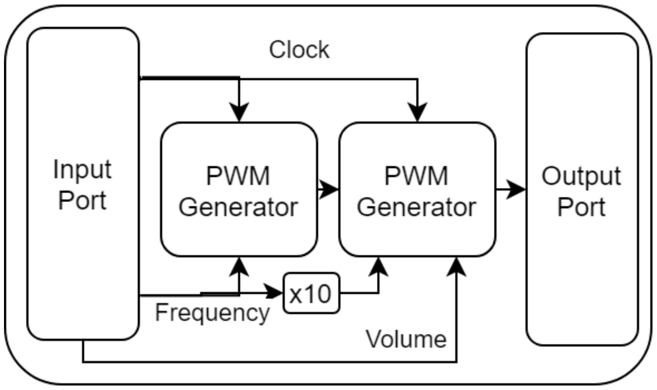


Figure 15. Voice entity block diagram.

The voice entity consists of two instances of a PWM generator. The first PWM generator receives the desired voice frequency, and the second PWM generator is for volume control only. The exciting part of this system is the combination of these two blocks, as the first one generates a PWM signal with frequency f_1 and duty-cycle of 50% and the volume control generates a second PWM signal with a frequency $f_2 = 10 \times f_1$ and duty-cycle defined by the on-board switches. The PWM entity has an active-low reset port which is used not used in the first instance, but it is used in the second to disable the second signal when the first is also disabled, creating an f_1 signal composed of another f_2 signal as if the two waves were being multiplied. Having ten times f_1 as f_2 is essential to avoid harmonics which frequencies are not multiples of f_1; this is for making the output signal as pure as possible.

The PWM entity has a generic argument of the number of bits in duty-cycle, which is used to define the resolution of the PWM's duty cycle; it was hardcoded to 4-bits in both instances. Meanwhile, the duty cycle of the first instance is set to "1000" being this 50% of "1111".

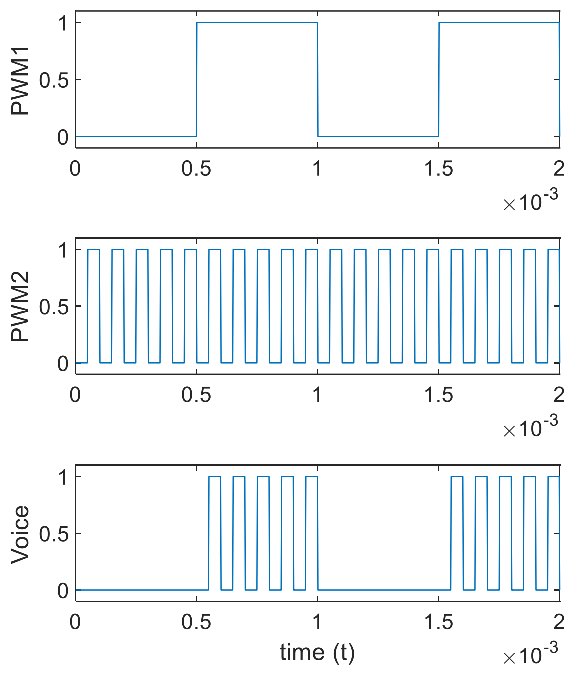


Figure 16. (Up) PWM signal from the first generator. (Middle) PWM signal from the second generator. (Down) PWM signal from the second generator using the first as a reset signal.

The main disadvantage of this volume control technique is that the output signal when the volume has been lowered using low duty-cycles and the signal's high-frequency harmonics are still hearable; this is somewhat compensated by the amplifier's filter attenuating these high frequencies, but at the end are still present.

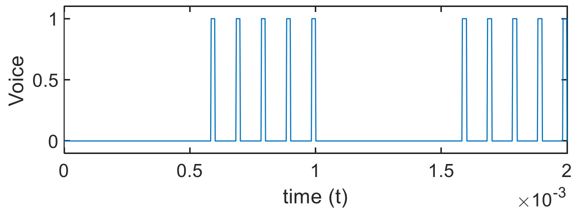


Figure 17. Voice signal using low duty-cycle.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY voice IS
  GENERIC (pwm_freq : INTEGER);
  PORT (
    clock : IN STD_LOGIC;
    volume : IN STD_LOGIC_VECTOR(3 DOWNTO
0);
    out_int : OUT STD_LOGIC;
    out_main : OUT STD_LOGIC
  );
END voice;

ARCHITECTURE Behavioral OF voice IS
  SIGNAL x : STD_LOGIC;
BEGIN
  PWM1 : ENTITY work.pwm
      GENERIC MAP(
      100_000_000,
      pwm_freq, 4, 1)
      PORT MAP(
        clk => clock,
        reset_n => '1',
        ena => '1',
        duty => "1000",
        pwm_out(0) => x
      );

        PWM2 : ENTITY work.pwm
            GENERIC MAP(
            100_000_000,
            pwm_freq * 10, 4, 1)
            PORT MAP(
              clk => clock,
              reset_n => x,
              ena => '1',
              duty => volume,
              pwm_out(0) => out_main
            );
              out_int <= x;
END Behavioral;
```

Script 7 Voice entity.

PWM Entity

The PWM generator is a common block for several embedded projects, so it is easy to find implementations of this for several different scenarios. PWM generators are available all over the internet, and since VHDL is a non-target-specific language, it is particularly easy to get online code to work in personal designs without dependencies. For this project, a VHDL file was downloaded from a repository belonging to Digikey; this entity has many features, and it is easy to implement. The contents of this entity have been left as is, but the functionality behind is commented in the following paragraphs. The use of the code blocks available in websites and books are an essential part of the development process to avoid wasting time and resources in developing something that may already be available for engineers to use. However, engineers must be able to code their blocks or, at least, be able to customize the code to fit their needs.

The PWM entity has both generic and port arguments, which are used for configuring the PWM signal and link it with the rest of the systems, respectively. Arguments are defined as the following.

- Generic
    - Clock signal: Input Clock signal frequency; 100 MHz by default.
    - PWM frequency: Desired PWM frequency.
    - Duty-Cycle resolution: Resolution in bits; steps are given by $2^N - 1$.
    - Number of phases: Number of different outputs with different phases but the same frequency.
- Port
    - Clock Signal: Clock signal input.
    - Reset: Active Low reset port.
    - Enable: Enable/Disable port.
    - Duty-Cycle: Duty-Cycle selector; must have as many bits as resolution bits.
    - PWM out: PWM output signals for all phases; it has as many elements as phases.
    - Inverse PWM out: Same as PWM out, but logic is inversed.

Keeping generic inputs, as seen in the top-level, makes entities reusable and scalable.

```
ENTITY pwm IS
  GENERIC (
    sys_clk : INTEGER := 100_000_000;
    pwm_freq : INTEGER := 1_000;
    bits_resolution : INTEGER := 4;
    phases : INTEGER := 1);
  PORT (
    clk : IN STD_LOGIC;
    reset_n : IN STD_LOGIC;
    ena : IN STD_LOGIC;
    duty : IN
STD_LOGIC_VECTOR(bits_resolution-1 DOWNTO
0);
    pwm_out : OUT
STD_LOGIC_VECTOR(phases-1 DOWNTO 0);
  pwm_n_out : OUT
STD_LOGIC_VECTOR(phases-1 DOWNTO 0));
END pwm;
```

Script 8. Ports and generic inputs.

The architecture of this entity defines signals and constants in order to make coding more accessible and more readable. Only one constant was used, being the period of the PWM signal defined as $N = \frac{f_{clk}}{f_{pwm}}$ (Cycles per Period). The following signals are not declared as constants although being integers because their contents are expected to change in execution. A counter vector is defined as an integer array with elements as phases and range from zero to the period. It is initially defined as zero. Finally, two arrays are declared having as elements as phases, and range from zero to half the period; these arrays are used to store half the current and next period's duty-cycles

```
ARCHITECTURE logic OF pwm IS
  CONSTANT period : INTEGER :=
  sys_clk/pwm_freq;
  TYPE counters IS ARRAY (0 TO phases-1)
OF INTEGER RANGE 0 TO period - 1;
  SIGNAL count : counters := (OTHERS=>0);
  SIGNAL half_duty_new : INTEGER RANGE 0
TO period/2 := 0;
  TYPE half_duties IS ARRAY (0 TO phases
- 1) OF INTEGER RANGE 0 TO period/2;
  SIGNAL half_duty : half_duties :=
(OTHERS => 0);
```

Script 9. PWM entity signals and constants.

The next thing is to begin the behavioural architecture of the entity. The first section of the architecture is to create a process triggered by either the input clock signal or the reset signal. If the process is triggered, the first thing it does is to evaluate which signal triggered its execution; if the reset signal changed to '0', the counter array and both output arrays are set to '0'. However, if the clock triggered the process, and the clock had a rising edge change, the PWM signal process is appropriately started.

```
BEGIN
PROCESS (clk, reset_n)
BEGIN
        IF (reset_n = '0') THEN
                count <= (OTHERS => 0);
                pwm_out <= (OTHERS => '0');
                pwm_n_out <= (OTHERS => '0');
        ELSIF (clk'EVENT AND clk = '1') THEN
                …
```

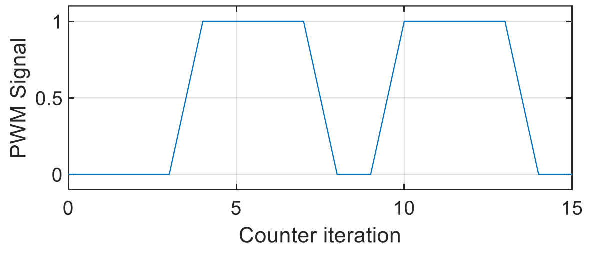
Script 10. PWM Entity process start and reset procedures.

When the past condition is met, the PWM duty-cycle in the next period is assigned, only if the enable signal '1'. Then, the cycle counter is incremented if it has not reached the last cycle of each phase's period. Each phase's last cycle is different for creating an offset between signals; for a period $N = 6$, and two phases, phase 0 last cycle would be $N - 1 - i\frac{N}{2} = (6) - 1 - (0)(3) = 5$, meanwhile phase 2 last cycle would be $N - 1 - i\frac{N}{2} = (6) - 1 - (1)(3) = 2$. This number only means at which value the counter will reset and apply the next duty-cycle value.

```
ELSIF (clk'EVENT AND clk = '1') THEN
IF (ena = '1') THEN
        half_duty_new <= to_integer(duty)*
        period/(2**bits_resolution)/2;
END IF;
FOR i IN 0 TO phases - 1 LOOP
        IF (count(0) = period-1-i*
                Period/phases) THEN
                count(i) <= 0;
                half_duty(i) <= half_duty_new;
        ELSE
                count(i) <= count(i) + 1;
        END IF;
END LOOP;
```

Script 11. PWM entity cycle counter.

The half Duty-Cycle signal stores half of the cycles when the signal is high. For example, assuming $N = 6$, and a duty-cycle of 66% the half duty-cycle signal would be $DC_{half} = DC \times \frac{N}{2} = \left(\frac{2}{3}\right) \times \frac{6}{2} = 2$. This procedure is done to have two trigger counter values instead of one to assign a hardcoded value to the output signal instead of inverting it every cycle, which can lead to uncertainties. This approach leads to a reduced accuracy only in the first period, which off-time is more extensive than it should; making the resulting look phased compared to the initial counter iteration. The resulting behaviour of the example can be seen in the following figure.



Script 12. PWM signal from example.

```
FOR i IN 0 TO phases - 1 LOOP
        IF (count(i) = half_duty(i)) THEN
                pwm_out(i) <= '0';
                pwm_n_out(i) <= '1';
        ELSIF (count(i) = period-half_duty(i))
THEN
                pwm_out(i) <= '1';
                pwm_n_out(i) <= '0';
        END IF;
END LOOP;
```

Script 13. PWM Entity output assignment stage.

Soft Processor Implementation

The SoC's processor is a Xilinx's Microblaze 32-bit soft processor. This processor is available in the Vivado environment through an IP block and programmed using Xilinx's Vitis IDE. The purpose of this processor is only for translating incoming UART messages from a PC to a binary representation of the message over a GPIO port.

IP Block Configuration

The configuration for this IP block is simple, as not many configurations settings are available, and only a few of them are essential. The most important settings for this project are listed as they appear in the IP block window's tabs.

- MSC
    - General
        - Input Clock Frequency: 100 MHZ
        - Memory Size: 32KB
    - Buses
        - Since no other buses are required, both are disabled
    - Debug
        - No debugging methods were used.
- UART
    - TX/RX Enabled
    - Baud Rate: 115200
    - The remaining UART parameters were left as default.
- Fixed Interval Timer
    - No timers were used
- Programable Interval Timer.
    - No timers were used.
- General Purpose Outputs
    - GPO1 was enabled.
    - GPO1 Port width: 32-bit.
    - GPO1 Initial value: 0.
- General Purpose Inputs
    - No GPI was used
- External Interrupts
    - No external interrupts were required.

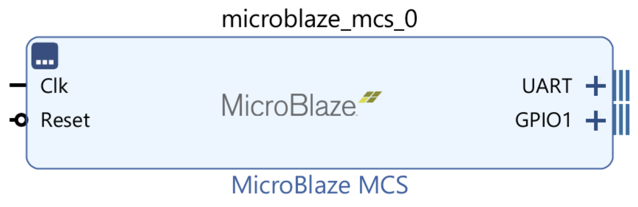


Figure 18. Microblaze IP block for Mini-Project.

The Microblaze processor is meant to be driven by the 100 MHz onboard clock signal, receive and send UART messages and to drive the multiplexor's output. The instantiation of this block has already been covered in the Top-Level System.

Microblaze Software

The Microblaze processor is meant to be programmed using Xilinx's Vitis IDE, which receives the configuration of the IP block for the IDE to know what features have been enabled by the user.

The program only has one task: Reading from UART and writing to the GPO port according to the content of the message. A single byte is sent through UART; this means ~255 possible tones are reachable.

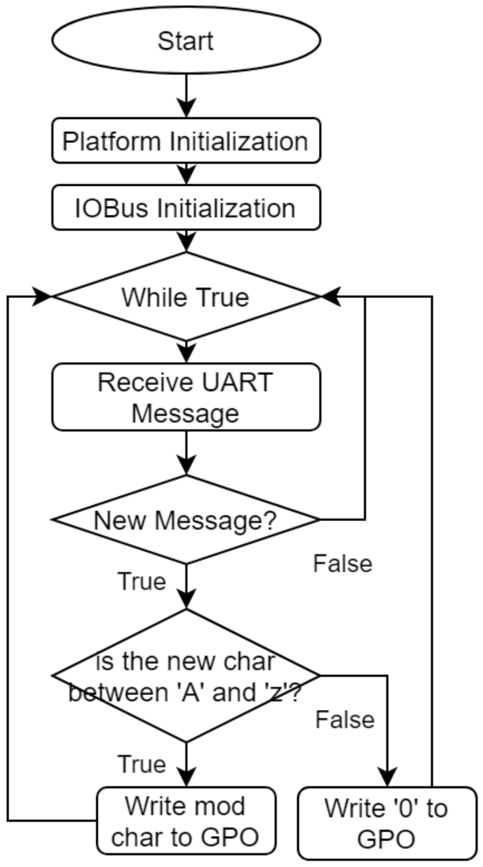


Figure 19. Program flow chart diagram.

The first step of the program is to initialize the processor and to initialize the bus that holds both GPO and UART peripherals.; This is done by executing the initialization function from the Microblaze Hardware Abstraction Layer drivers.

Also, three variables are declared: iomodule, note and data, which are of type XIOModule (From the Microblaze HAL), unsigned char and integer, respectively. The purpose of each variable is to access the IO bus to use the peripherals, store the return flag from the UART peripheral and store the UART message.

The iomodule is initialized with the ID 0, then started and configured, enabling all the peripherals.

```
XIOModule iomodule;
    uint8_t note;
    int data;

       init_platform();

       XIOModule_Initialize(&iomodule,
       XPAR_IOMODULE_0_DEVICE_ID);
       XIOModule_Start(&iomodule);
       XIOModule_CfgInitialize(&iomodule,
NULL, NULL);
```

Script 14. Platform and IOBus initialization.

Then, the while loop polls until a UART message is received, the program notices a new character by assigning the number of received characters to the data variable. Finally, the program processes the UART message by subtracting an arbitrary reference character, since the hardware multiplexor needs a select signal from zero to the number of semi-tones. If the processed message is between the reference character and the last alphanumeric ASCII character ('z'), the message is added one and written to the GPO port. For example, assume the incoming message is 70, and the reference character is 'A'; the ASCII representation of 70 is 'F', so the written value to the GPO port is $'F' - 'A' + 1 = 70 - 65 + 1 = 6$, triggering the 6th voice signal (element 5 of the voice array).

```
while(1){
data = XIOModule_Recv(&iomodule,&note,1);
if(data){
  xil_printf("Recv: %c, Tone #: %d, ASCII#:
  %d\n\r",note, note-REFERENCE_CHAR, note);

  XIOModule_DiscreteWrite(&iomodule,1,
(note >= REFERENCE_CHAR && note <= 'z')
 ? note-REFERENCE_CHAR+1
 : 0);
}
}
```

Script 15. While loop, UART receiving and GPO write.

The program was compiled using the Vitis compiler and the binary file added to the Vivado environment to be included in the bitstream generation.

Testbench Simulation

Since HDL describes the hardware behaviour, simulating it is easier than simulating microcontrollers with procedural programming languages. Testbench files are instantiations of the main entity or another and non-synthesizable HDL which describes the stimuli the system receives. The previously reviewed PWM entity was tested to verify its performance and features.

An empty test entity was created with an architecture containing the required signals to drive the PWM entity. Also, some configuration signals were created to create a clock signal and to flag the end of the simulation. Since the on-board clock is 100 MHz, a signal period of 10 ns is required to simulate the clock.

```
ENTITY tb_pwm IS
END tb_pwm;

ARCHITECTURE tb OF tb_pwm IS

       SIGNAL clk : std_logic;
       SIGNAL reset_n : std_logic;
       SIGNAL ena : std_logic;
       SIGNAL duty : std_logic_vector (3
DOWNTO 0);
       SIGNAL pwm_out : std_logic;
       SIGNAL pwm_n_out : std_logic;

       CONSTANT TbPeriod : TIME := 10 ns;
       SIGNAL TbClock : std_logic := '0';
       SIGNAL TbSimEnded : std_logic := '0';

BEGIN
```

Script 16. Testbench entity declaration and signal definition.

The next step is to instantiate the PWM entity and create the clock signal to drive it. The standard instantiation is required, while the clock requires a sequential procedure to simulate its behaviour. A PWM frequency of 10 MHz was specified in order to see all the simulated signals.

```
dut : pwm
GENERIC MAP(
        100_000_000,
        100_000_00, 4, 1)
PORT MAP(
        clk => clk,
        reset_n => reset_n,
        ena => ena,
        duty => duty,
        pwm_out => pwm_out,
        pwm_n_out => pwm_n_out);

clk <= NOT clk AFTER TbPeriod/2 WHEN
TbSimEnded /= '1' ELSE '0';
```

Script 17. Testbench entity DUT instantiation and clock generation.

Finally, the stimuli process is defined. Wait for statements can be used inside the scope of the stimuli process, which is non-synthesizable VHDL. There all the input signals of the entity can be changed in different instants. The intended test procedure was the following.

1. Enable the block by toggling both enable and reset signals high, while setting the duty-cycle to 50% (Assuming 4 bits of resolution).
2. After 400 ns, change the duty-cycle to ~90%.
3. After 200 ns, toggle the enable signal low to stop new duty-cycle values (as described in the PWM Entity) and decrease the duty cycle to 25%.
4. After 200 ns, toggle the enable signal high to change the duty-cycle.
5. After 400 ns, disable the reset signal to stop all the PWM functionality.

```
stimuli : PROCESS
BEGIN
        duty <= "1000";
        ena <= '1';
        reset_n <= '1';

        WAIT FOR 400ns;
        duty <= "1110";

        WAIT FOR 200ns;
        ena <= '0';
        duty <= "0100";

        WAIT FOR 200ns;
        ena <= '1';

        WAIT FOR 400ns;
        reset_n <= '0';

        TbSimEnded <= '1';
        WAIT;
END PROCESS;
END tb;
```

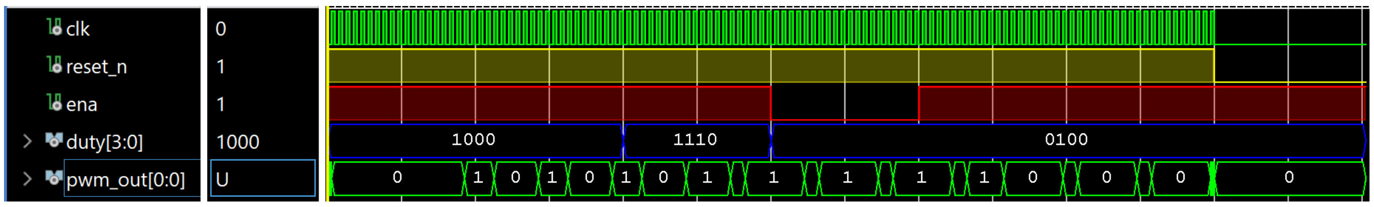
Script 19. Testbench stimuli process.

Hardware Deployment

After synthesizing all the HDL blocks and the compiled binary, the program works as expected. 'A' was set as the reference character, so only this character and above can trigger sound.

Volume control is noticeable if the switches are turned up from right to left the volume increases, and doing the opposite attenuates the sound, as expected. The unexpected result is that by turning down the switches from right to left keeps the same volume and pitch but adds some high-pitch harmonics; this is often called changing the resonance of a square wave in music terms.

The system is only capable of outputting up to 58 tones before being totally attenuated by the onboard filter. Having C_3 as the reference means that the highest note is A_7, 4.75 octaves above the reference.



Script 18. Testbench simulation output.

The total resource allocation for this implementation is 5649 LUT, 4087 Flip-Flops and 8 BRAM units (to store the frequency array).

Result Discussion

Although the design goals were achieved, the hardware cost of the architecture is enormous. Usually, in commercial synthesizers, a voice is marketed as a single Voltage Controlled Oscillator (VCO) or Numeric Controlled Oscillator (NCO) that can output a variable frequency signal. The implementation difference between the applied and the commercial architecture is significant, since coding a VCO or NCO is far more complicated than a PWM generator, and DSP slices are required to combine all the different voices.

Although, this architecture is expandable in order to have the same multi-simultaneous voices feature. If the Top-Level entity was rewritten to be instantiated, many of these entities could generate multiple voices simultaneously. Another enhancement would be to implement Direct Digital Synthesizers instead of PWM to have the same NCO features while outputting a sinusoidal wave instead of a square wave.

Appendix

The commented code and Vivado/Vitis project is located in the following repository.

https://github.com/JoseAmador95/RSOC_Mini_Project

References

[1] P. Wilson, Design Recepies for FPGA Using Verilog and VHDL, Elsevier, 2016.

[2] H. Amano, Principles and Structures of FPGAs, Springer, 2018.

[3] Intel, "Intel® Stratix® 10 Logic Array Blocks and Adaptive Logic Modules User Guide," 21 September 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf. [Accessed 20 April 2020].

[4] Xilinx, "7 Series FPGAs Data Sheet: Overview," 27 Febraury 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. [Accessed 18 April 2020].

[5] D. J. Greaves, "System on Chip Design and Modelling," 2011. [Online]. Available: https://www.cl.cam.ac.uk/teaching/1011/SysOnChip/socdam-notes1011.pdf. [Accessed 25 April 2020].

[6] ARM Holdings, "Cortex-M1 Reference Manual," 7 May 2008. [Online]. Available: https://static.docs.arm.com/ddi0413/d/DDI0413D_cortexm1_r1p0_trm.pdf. [Accessed 25 April 2020].

[7] Xilinx, "FIR Compiler v7.2," 31 January 2020. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/fir_compiler/v7_2/pg149-fir-compiler.pdf. [Accessed 26 April 2020].

[8] Arm Holdings, "Arm Cortex-M1 DesignStart FPGA XIlinx edition," 18 January 2019. [Online]. Available: https://static.docs.arm.com/100211/0001/arm_cortex_m1_designstart_fpga_xilinx_edition_ug_100211_0001_00_en.pdf. [Accessed 25 April 2020].

[9] S. Larson, "PWM Generator (VHDL)," DigiKey, 8 September 2017. [Online]. Available: https://www.digikey.com/eewiki/pages/viewpage.action?pageId=20939345. [Accessed 22 March 2020].