

VHDL Based QAM Modulation

Abdelrahman S.A

June 15, 2019



Hint

All source code have been listed and referenced in each assosciated part in the document. However the full of list of source codes can be seen in github: https://github.com/astro7xRTL_QAM The system is developed using Xilinx Vivado Webpack, GHDL and Scansion

Abstract

The Motivation behind this project is to investigate the low-level "RTL Based" implementation and the building blocks of the QAM Modulator therefore, to provide a flexible way of testing the transmission and modulation properties later in real-time. A higher level hierarchical implementation based on **GNURadio** was provided to investigate future application of offloading QAM Modulation into FPGA, to accelerate the development, testing, and verification of **SDR** Software-Defined Radio application in real-time environment. On the Otherhand, each system module will be presented, and verified with testbenches along with **Matlab** scripts to verify the functional behavior of the QAM Waveform including the constellation diagram and upconvertedPass-Band waveform transition, then with respect to all system components. Finally, a future implementation overview is illustrated with respect to Improving and expanding the implementation into higher order Modulation and Multi-Channel Modulation. In this system no target board for implementation has been specified but only our system address the Zynq-7000 FPGA.

Contents

1	System Specification	1
2	Design Flow	3
3	System Components	4
3.1	Digital Oscillator	4
3.2	4-QAM Constellation Mapper	5
3.3	QAM Modulator	6
4	System Verification	7
5	System Synthesis	10
6	Conclusion and Future Work	15
	References	16
7	Source Files: Listing	17

Listings

1	QAM_Hierarchical.vhd	17
2	qam_mod.vhd	20
3	qam_mapper.vhd	24
4	ddfs.vhd	27
5	PhaseGenerator.vhd	32
6	tb_QAM_Hierarchical.vhd	34
7	Timing_Report_Summary.md	37

1 System Specification

This document presents an implementing for Quadrature Amplitude Modulation (**QAM**) signal modulators using Field Programmable Gate Array (FPGA). QAMs are widely used in DSP applications for the frequency conversion in digital transceivers, filters, modems, radars, sonars, mobile phones, satellite receivers, etc. And more interestingly, that it has been used in telecommunication standards starting from the third generation mobile cellular system **UMTS** Universal Mobile Telecommunications until the moment, the fifth generation, **LTE-A Pro** [Release 16] [1] where the only observed difference is the order of modulation, and this is quite interested where getting a higher modulation order, gives higher data rate, but of course this leads to higher Bit-Error Rate, but it's not the topic which we would like to address here. QAM has different types of constellations. Some of the constellations are called 4-QAM, 8-QAM, 16-QAM and 64-QAM .. etc based on the nearest point to origin. For example the 4-QAM constellation is limited to the 4 nearest points around the origin. Once the constellation is increased, the number of amplitude and phase also increased. It's also affected to frequency utilization where it is more efficient and is expressed in bits per second per hertz. In this document we focus on the implementation of 4-QAM, however, it's easy to extend the order by only replace the mapper module.

The basic operation of modulator is based on Signal mixing which means multiplying the input signal and a reference oscillator signal, to produce spectral images at the sum and difference frequencies. The implemented QAM modulator addressed here is a 4-QAM based modulator where each symbol is represented by 2 bits, with respect to the following formula, where M is the modulation order which is equal to 4, we got 2 bits per each symbol.

$$Bits_Per_Symbol = \log_2(M)$$

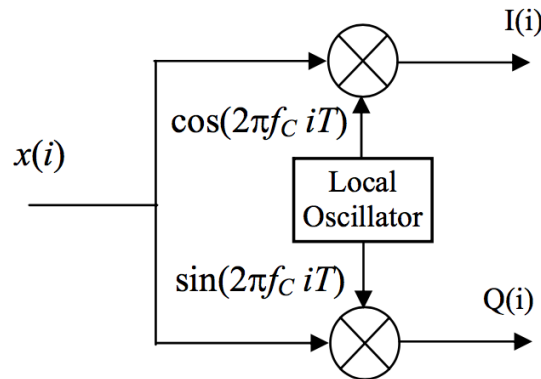


Figure 1. QAM Modulator Block Diagram

One of the main tasks in implementing any digital transmitter including QPSK, 4-QAM, 16-QAM .. etc is the generation of the sine wave carrier. In order to do that, a 12-bit phase accumulator and Look Up Table (LUT) has been used based on Direct Digital Frequency Synthesizer (DDFS). After getting the carrier generated, the basic QAM modulated signal is nothing more than the carrier signal itself at different phases and constant amplitude. For the implementation of 4-QAM modulators, two sinusoidal carriers are needed a sine wave and cos wave as described in 1

To get the two carriers with 90-degree phase shifts, Only 1 Phase accumulator were used but with different addressing ways to fetch the samples for both the sine and cosine as illustrated in 13

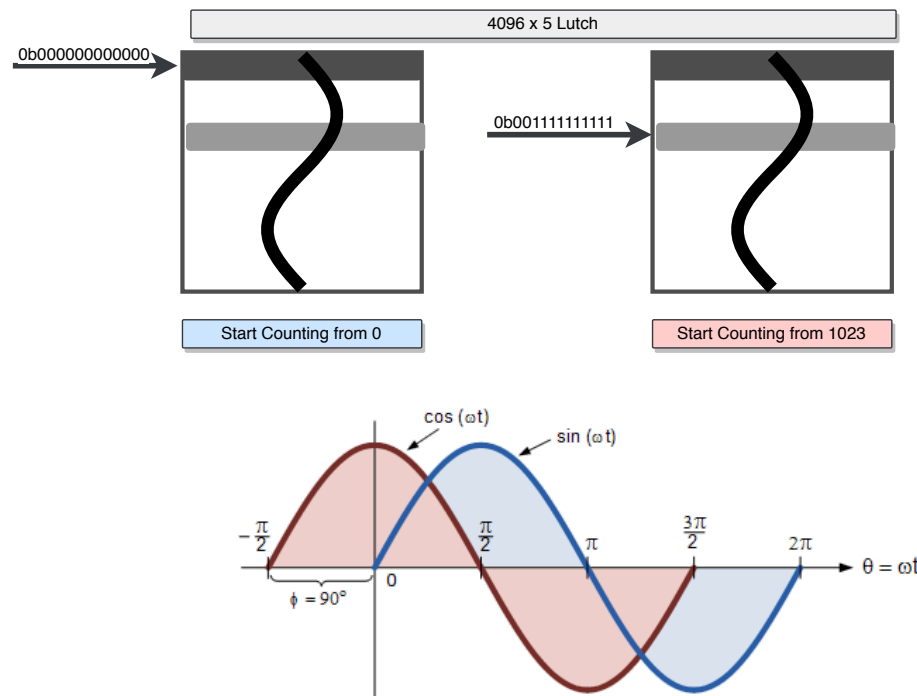


Figure 2. Waveform Oscillator

For I balanced modulator, 2 phases are possible is

$$+\sin(\omega ct) \text{ and } -\sin(\omega ct)$$

For Q balanced modulator, 2 phases are possible is

$$+\cos(\omega ct) \text{ and } -\cos(\omega ct)$$

Linear summer will combine the 2 quadrature signal and yet 4 possible phases acts as an output as shown in 3

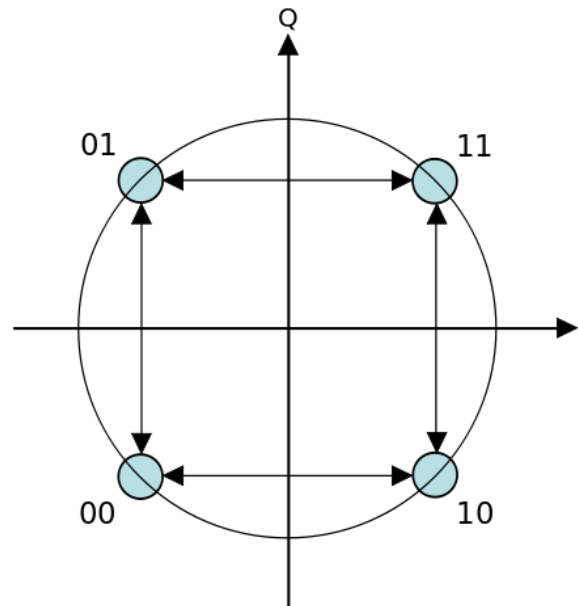


Figure 3. 4-QAM Constellation

2 Design Flow

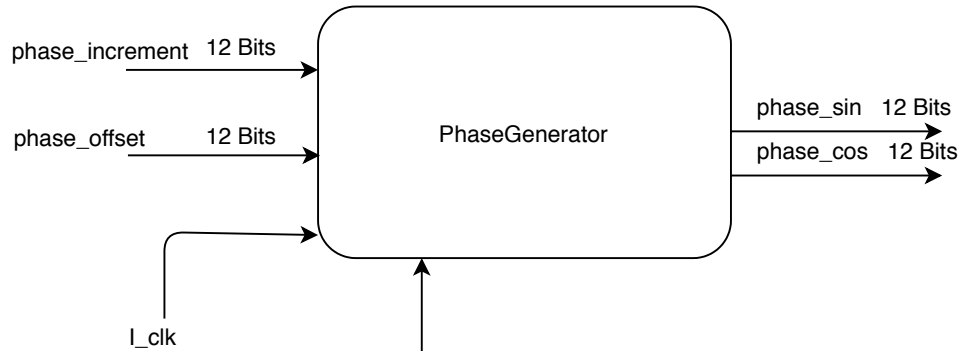
The implementation of the entire systems was done in Very high speed integrated circuit Hardware Description Language (VHDL) without the help of IP-Core Package Xilinx System Generator or DSP Builder Tools.

Design flow comprises of the following steps: architecture design, VHDL design entry, behavioral simulation, synthesis, implementation, timing analysis and [download to ZYBO board*](#) We are not addressing generating the bitstream for the board, the synthesis has been performed without the IO planning.

- **The behavioral simulation** have been achieved using GNU-Radio and Matlab along with test-benches developed using ghdl and scansion to fully cover all of the test corner cases in each module (which were actually bounded with respect to the number of test-vectors)
- **Synthesis and Implementation** have been achieved using Xilinx Vivado

3 System Components

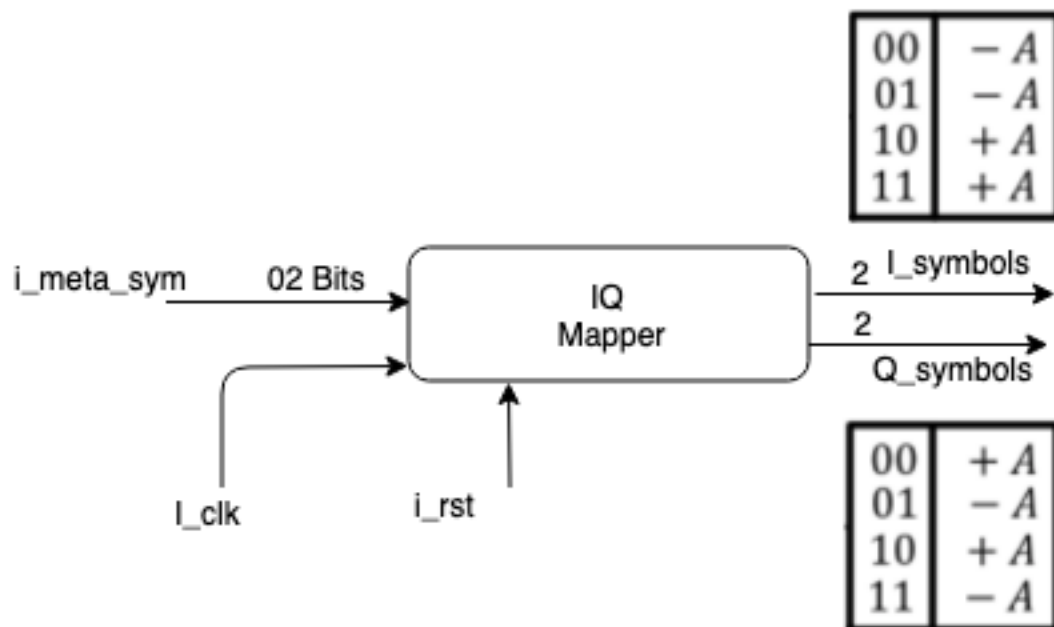
3.1 Digital Oscillator



Description This module is basically a phase accumulator which is responsible for adding a constant value FCW (Frequency Control Word) which is addressed here under the name `phase_increment`. The accumulator wrap around every time it reaches its maximum value. For example, in our configuration if the maximum value is 4096 (12-bit accumulator) and the `phase_increment = 4`, the accumulator will count: 0, 1, 2, 3, 4, ..., 4095. The system In this configuration counts from 0 to address later the sinusoidal waveform from a latch. As the address counter steps through each memory location, the corresponding digital amplitude of the signal at each location. On the other hands, it also start counts from 1023 to generate the cosine wave as illustrated before in [13](#).

Source_code: Can be found in listing [5](#)

3.2 4-QAM Constellation Mapper



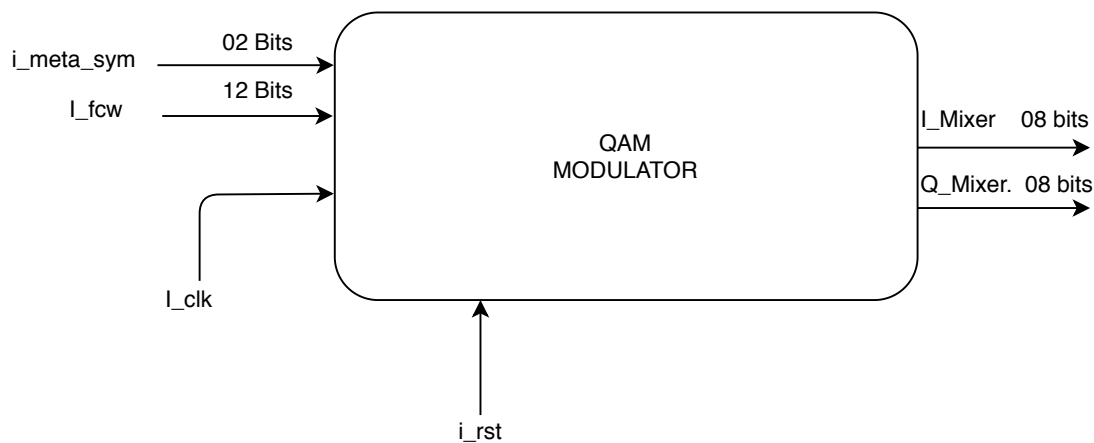
Description This module is responsible for constellation mapping of 4-QAM symbols. We want to have one single change of bits from one symbol to the other; thus, using fixed point number[gray code] representation 00, 01, 11, 10 to represent $1+j$, $-1+j$, $-1-j$, $1-j$ as shown in 1 See code 3.

Constellation[Binary Input Sequence]	Symbols Mapping
00	+1+J
01	-1+J
10	+1-J
11	-1-J

Table 1. 4-QAM Symbol Mapping

Source_code: Can be found in listing 3

3.3 QAM Modulator

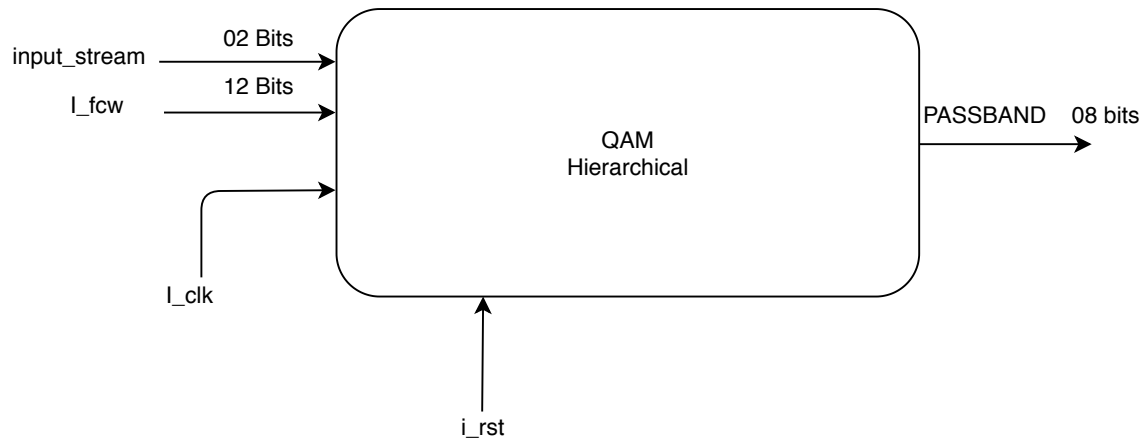


Description The QAM Modulator module takes as an Input the a pair sequence of bits and the frequency control word and then perform the multiplication or in other words, the mixing process to produce the PassbandUpconverted signals for both I componenet and Q components. For I balanced modulator, 2 phases are possible is

$$+sin(\omega ct) \text{ and } -sin(\omega ct)$$

For Q balanced modulator, 2 phases are possible is

$$+cos(\omega ct) \text{ and } -cos(\omega ct)$$



Description This module is nothing except it's an encapsulator for the QAM Modulator Module but perform Linear summationaddition by combining the up-converted signals.

Source_code: For qam_mod can be found in listing 2 and for qam_encapsulator in listing 1

4 System Verification

The measurement system consists of several building block that previously had been verified. Below in Figure 4 illustrated the waveform of the Phase Accumulator Counter where each counter is associated with a waveform, i.e. for cosine wave generation we count from 1023 which is equivalent to $\pi/2$ to later index the latch and get the cosine data points as shown. On the Other hand, to generate a sine wave we start the default counting from 0 and therefore we fetch the sine wave data points.

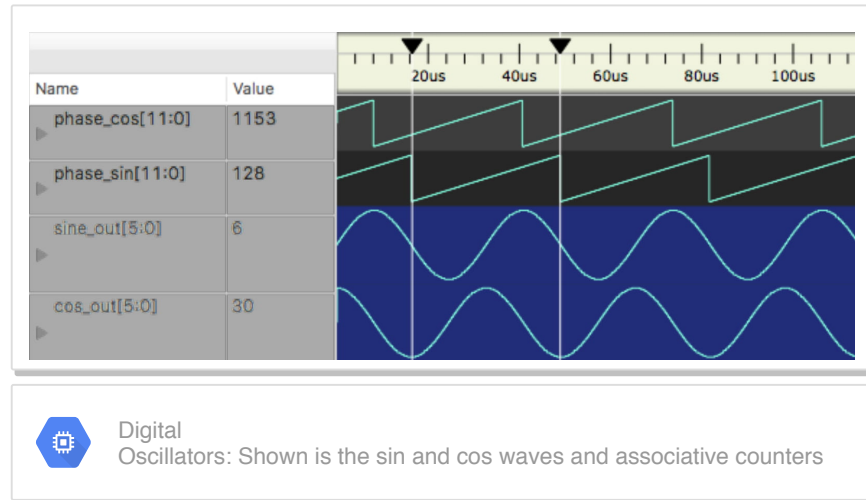


Figure 4. Digital Oscillators: DDFS

In Figure 5, the Individual modulated waveforms for I and Q component are illustrated after performing the mixing process. Hence: Since we mix (multiply) a waveform of size **N** and a symbol containing 2 bits, size **M** we got an output vector of size **M+N**.

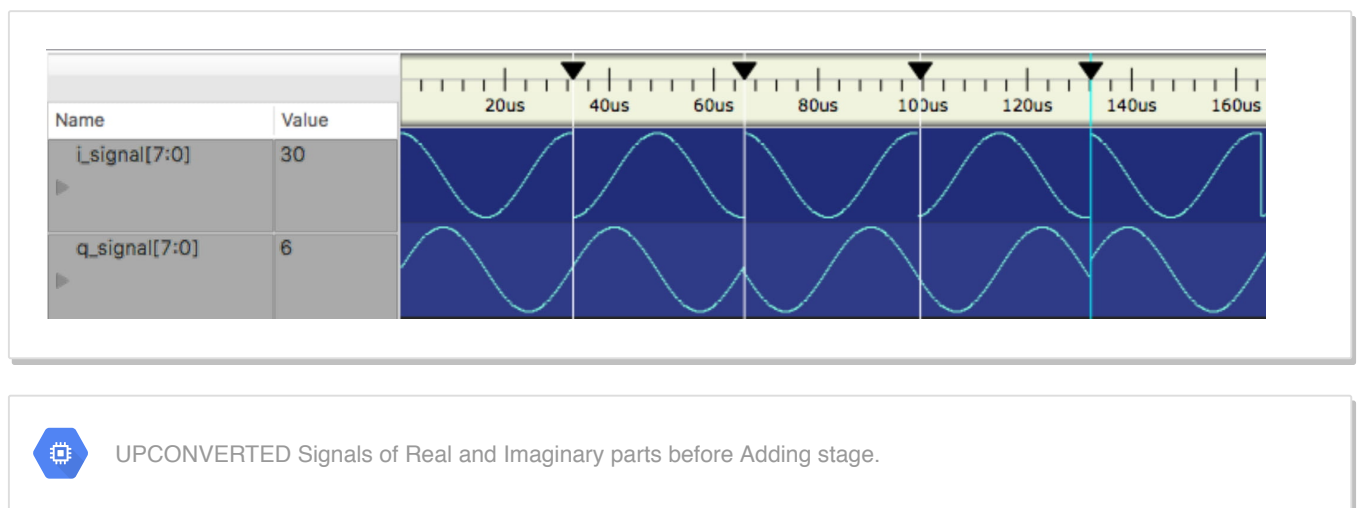


Figure 5. PassPand: Upconveted waveforms

Below in figure 6 is shown the constellation behaviour with respect to the input symbols as described before in table 1 versus the matlab equivalent verification.

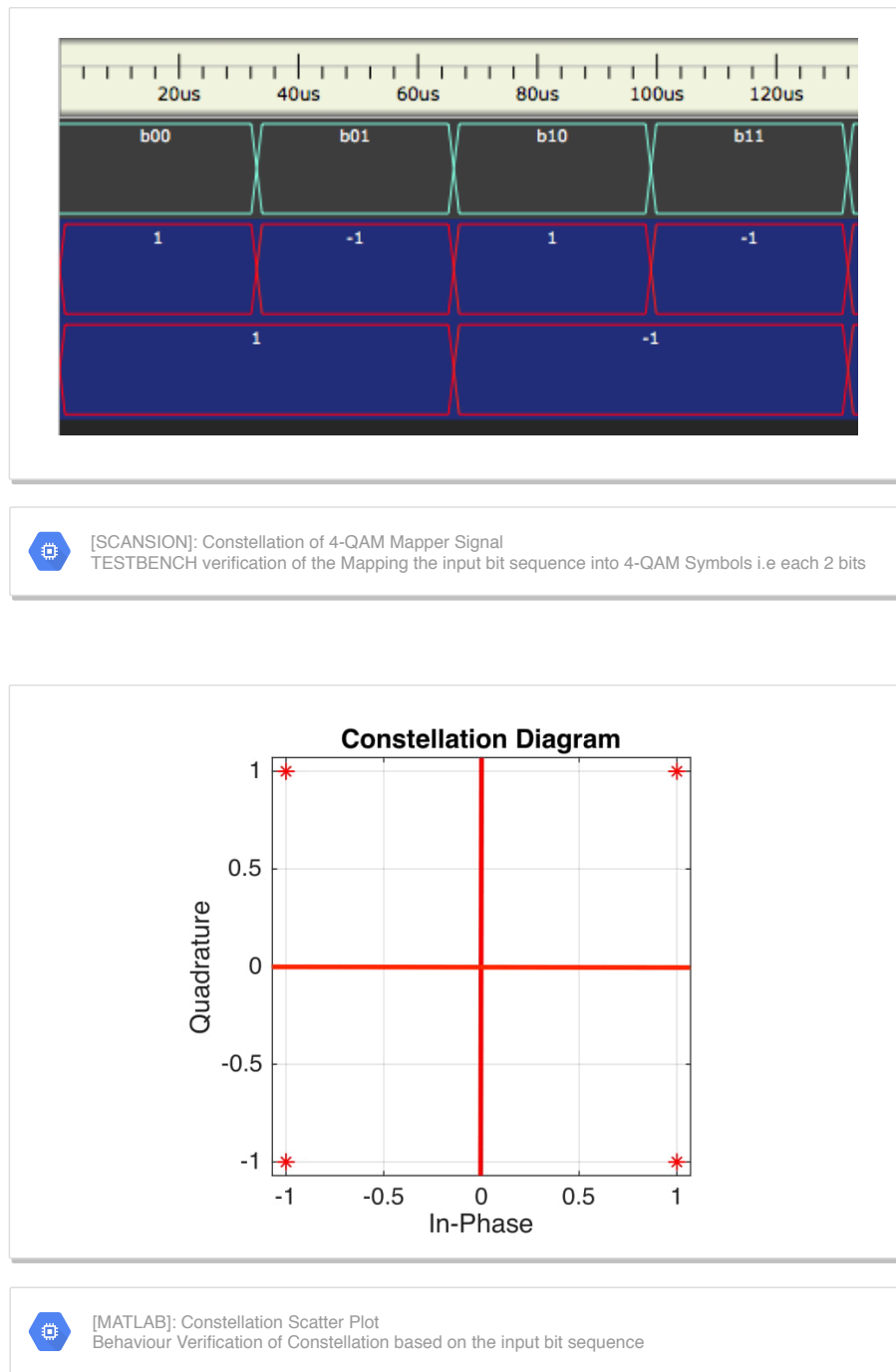


Figure 6. [VHDL vs MATLAB] 4-QAM Constellation Diagram

In Figure 7 is shown the final stage of behaviour verification using the testbench and matlab for the waveform transition with respect to the input symbols.

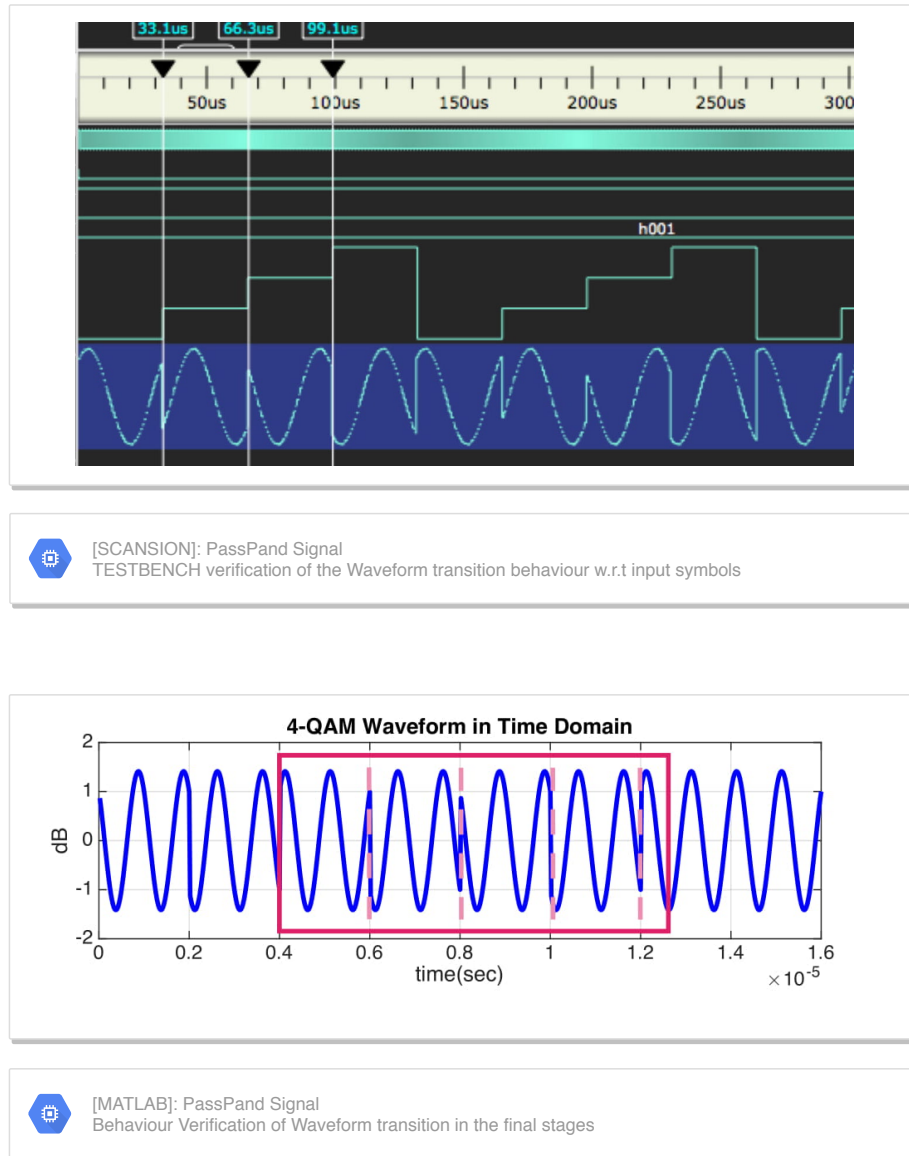


Figure 7. [VHDL vs MATLAB] 4-QAM Modulated Signal

5 System Synthesis

At the beginning of design, the synthesis has been performed versus different clock based criterion. The main based selection for the clock constrains [2] is the assumption that we want to system to operate in **125MHz**. However different tests have been done to try to achieve a maximum clock frequency of 200MHz and 614.4 MHz. The schematic of the system after elaborating the design is shown below in 8, 9 and 10. which is basically, reading RTL files (VHDL files) and recognizing of code syntax that represent real hardware structures. Once recognized, these are converted (in Vivado synthesis case) into generic technology cells - abstract things like registers, adders, compactors, multiplexers, gates, etc...

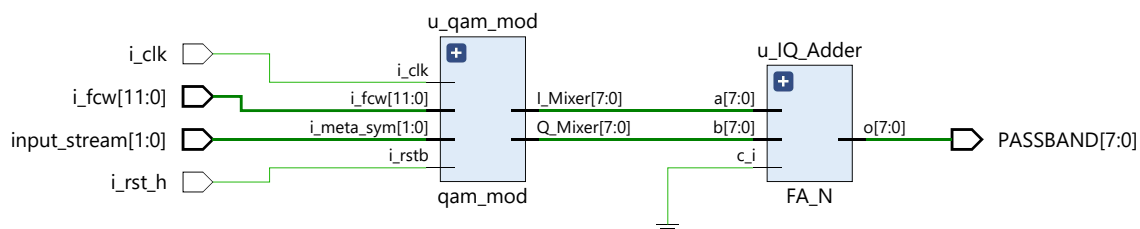


Figure 8. QAM higher level schematic

??

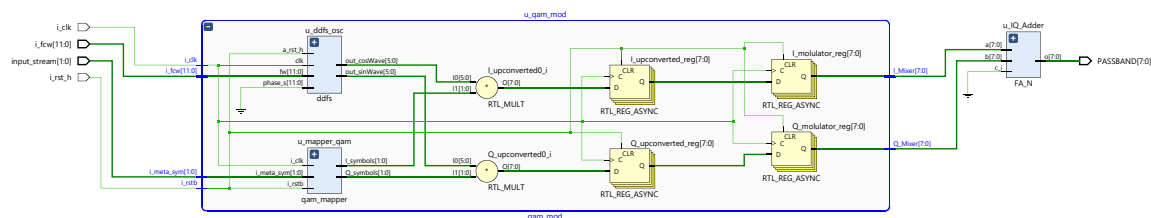


Figure 9. qam_mod block schematic

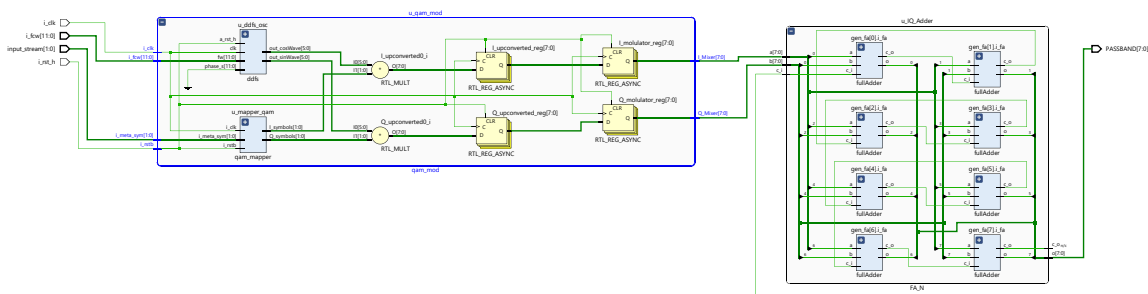


Figure 10. IQ_Adder block schematic

After elaborating the design, the synthesis is performed with no constraints first to verify that

every thing is working correctly and that the design is bug-free. Then we defined the constraints by defining the clock period to be **8 nano sec** to achieve a maximum clock frequency of 125 MHz. Then the implementation is performed, where implementation stage is intended to translate netlist into the placed and routed FPGA design. Xilinx design flow has three implementation stages: translate, map and place and route. (These steps are specific for Xilinx.

below in figure 11 is reported utilized resources. Hint: Extended timing summary is listed in 7

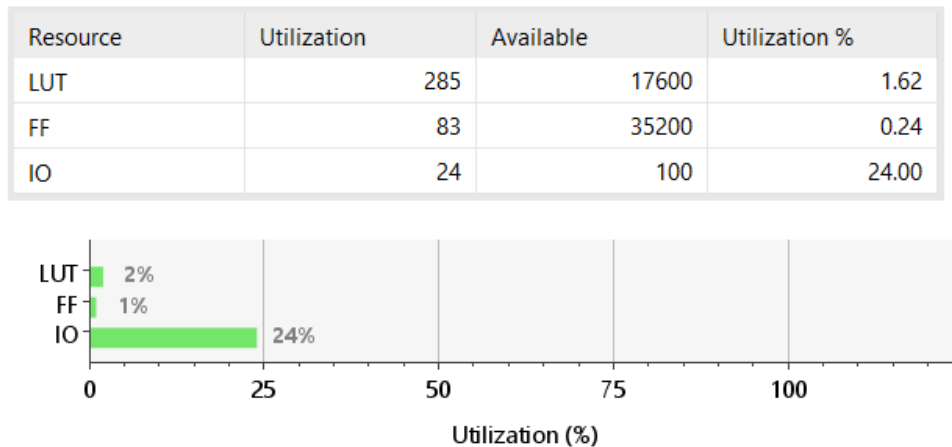


Figure 11. Utilization Report Summary

Below in 12 is reported the critical path due to the minimum period which in turns indicate the maximum operating frequency. Also shown all possible paths in ??

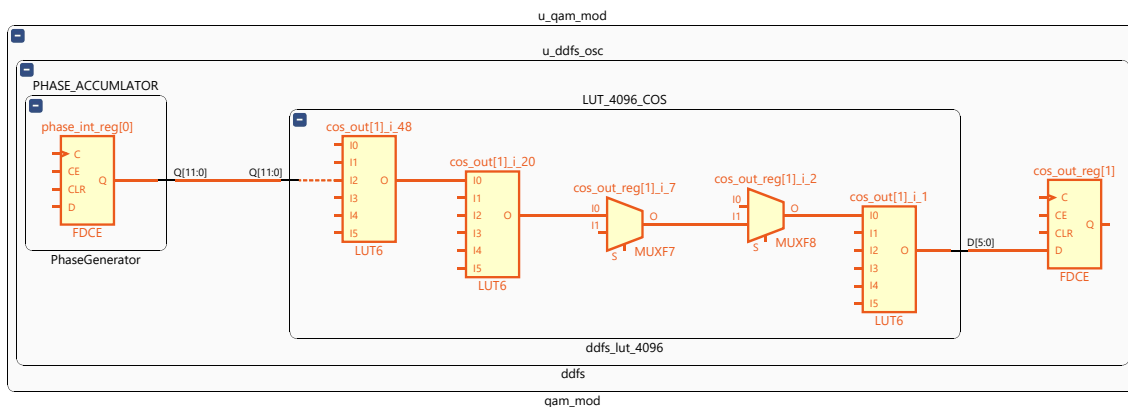


Figure 12. Critical Path Schematic

Name	Slack ^{^1}	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	2.386	5	3	52	u_qam_mod/u_nt_reg[0]/C	u_qam_mod/u_t_reg[1]/D	5.516	1.519	3.997	8.0
Path 2	2.603	5	4	78	u_qam_mod/_g_rep[2]/C	u_qam_mod/u_t_reg[3]/D	5.323	1.565	3.758	8.0
Path 3	2.691	5	3	78	u_qam_mod/u_nt_reg[2]/C	u_qam_mod/u_t_reg[0]/D	5.193	1.727	3.466	8.0
Path 4	2.717	5	4	78	u_qam_mod/_g_rep[2]/C	u_qam_mod/u_t_reg[2]/D	5.207	1.565	3.642	8.0
Path 5	2.734	5	4	82	u_qam_mod/u_nt_reg[3]/C	u_qam_mod/u_t_reg[3]/D	5.114	1.396	3.718	8.0
Path 6	2.770	4	3	78	u_qam_mod/u_nt_reg[2]/C	u_qam_mod/u_t_reg[4]/D	5.189	1.335	3.854	8.0
Path 7	2.811	5	3	82	u_qam_mod/u_nt_reg[3]/C	u_qam_mod/u_t_reg[2]/D	5.089	1.612	3.477	8.0
Path 8	2.912	4	4	78	u_qam_mod/_g_rep[2]/C	u_qam_mod/u_t_reg[1]/D	5.013	1.182	3.831	8.0
Path 9	3.076	5	3	52	u_qam_mod/_g_rep[0]/C	u_qam_mod/u_t_reg[0]/D	4.881	1.427	3.454	8.0
Path 10	3.426	4	3	78	u_qam_mod/_g_rep[2]/C	u_qam_mod/u_t_reg[4]/D	4.534	1.273	3.261	8.0

Figure 13. paths

Finally; In 14 We report the power consumption and the Worst Negative slack by making a sanpshot comparison between working on 125MHz Clock and 200MHz clock and clearly we can observe the increase of dynamic power consumption by increasing the the operating system frequency.

$$P = \frac{1}{2} CVF$$

where P is the power, C is the load capacitance and V is the supply voltage level. The frequency refer to the clock frequency and data toggles once every clock cycle.

On the other hand, The Worst Negative Slack (WNS) reported in 14 is actually the worst positive slack. If WNS is positive then it means that the path passes. If it is negative, then it means the path fails. The "Total Negative Slack (TNS) is the sum of the (real) negative slack in our design. If 0, then the design meets timing. If it is a positive number, then it means that there is negative slack in the design (hence your design fails). It cannot be negative. In both clock configuration the system have been met the requirements constraints but further increasing in clock frequency i.e 614.4 MHz will fail where different optimization techniques must be performed to meet the clock requirements like adding pipelines in the system design or the use of DSP Blocks on ZYNQ where it's observed from timing report in 7 that no DSP Blocks have been used. Hence, the choice of clock 614.4 MHz was based on the waveform frequency resolution of 150KHz

$$\log_2(fclk/DesiredFreq) = N$$

where N is the number of points needed to represent the waveform, Desired Freq is 150KHz and fclk is the system clock constraints.

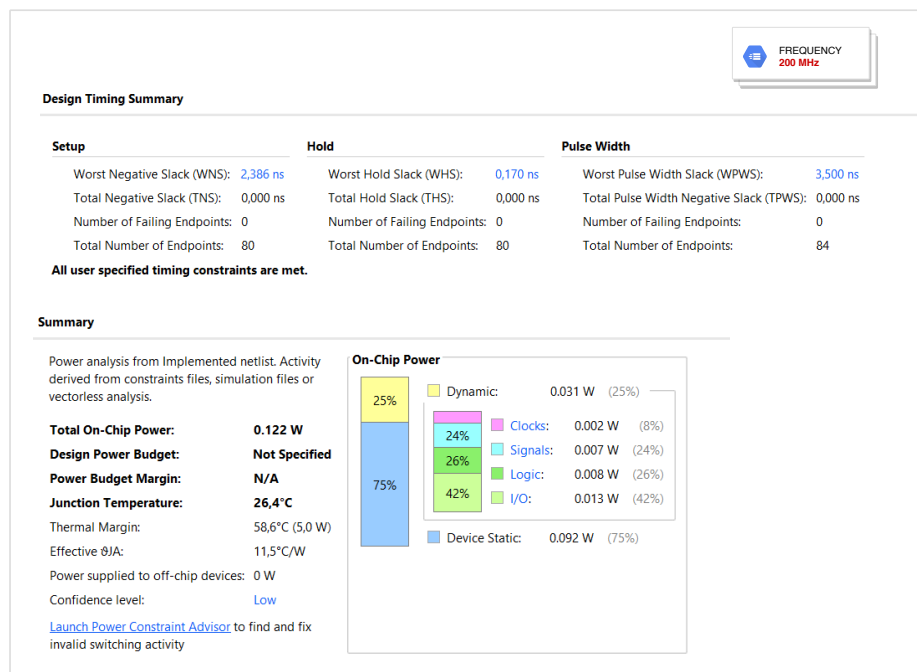
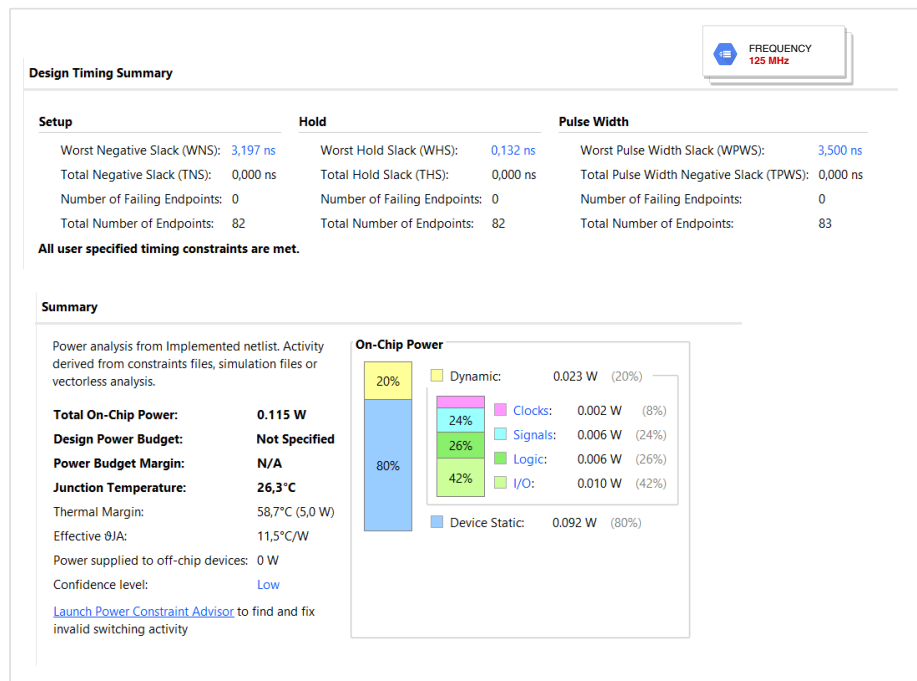


Figure 14

Finally, in 15 and 16 is shown the device implementation.

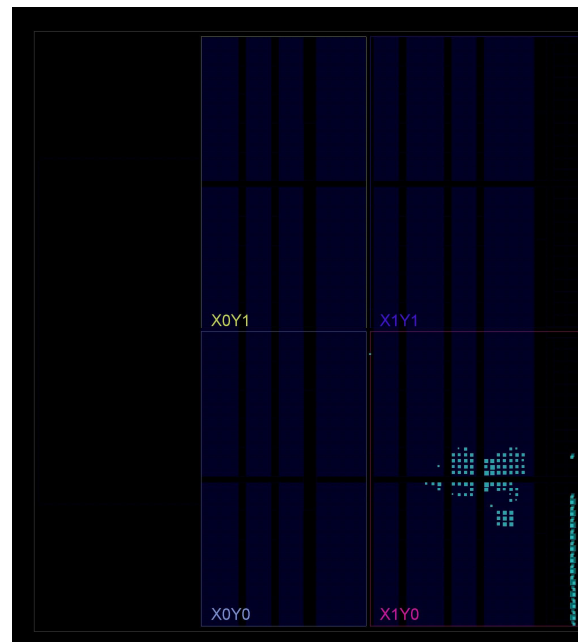


Figure 15. Device Implementation

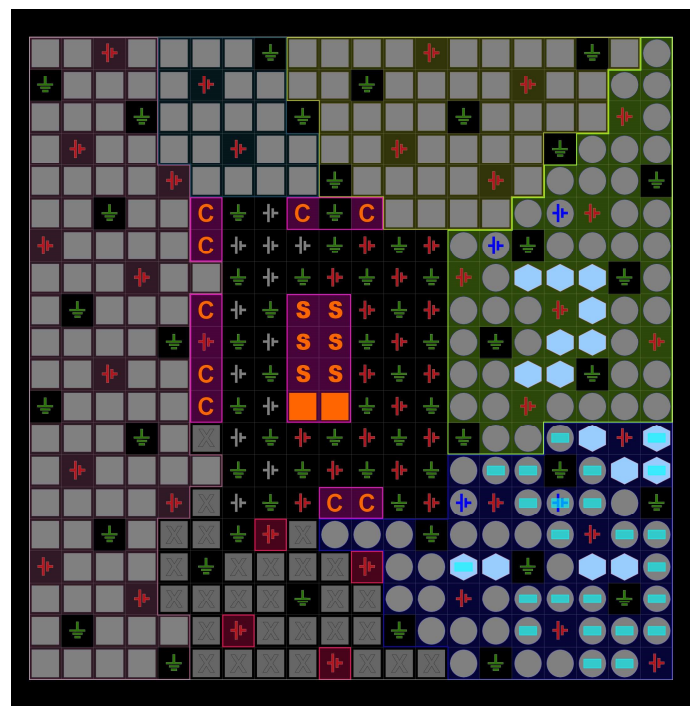


Figure 16. IO Planning

6 Conclusion and Future Work

Nowadays the the basic idea of development, testing, and verification of communication standards is to accelerate the production phases by exploiting the existence of FPGAs and MUCs as prices decrease and performance increase. The technology is referred to as SDR, Software Defined Radio. This technology provides a way of changing physical and virtual properties of a radio within re-configurable blocks, such as modulation, frequencies, amplitudes and algorithms amongst others. SDR technology opens up new ways of adaptive, and flexible communication techniques to develop, verify and test new telecommunication standards in real-time. Where SDR itself composed by an FPGA and ARM Based processor and where the majority of processing is on host based processing (done by softwares and then samples are transmitted to the SDR device using ethernet or PCI interface), further optimization and acceleration could be done by offloading the processing to FPGA [3]. For example, the modulator block of OFDM system could be further offloaded to FPGA therefore achieving a higher performance. Below in 17 and 18 is reported a simple system designed by GNURadio software to illustrate the constellation of 4-QAM modulator. [4]

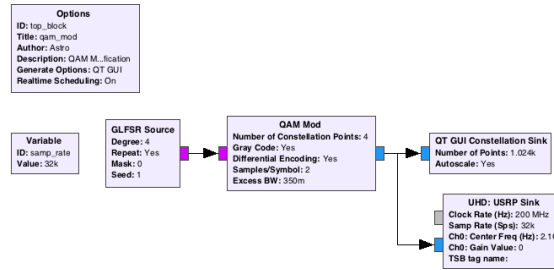


Figure 17. GNURadio QAM flowgraph

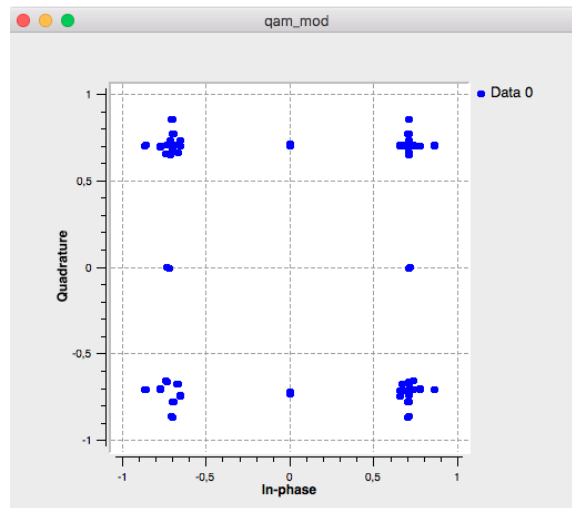


Figure 18. Constellation Plot in real-time



References

- [1] Evolved universal terrestrial radio access (e-utra); physical channels and modulation. 3GPP Specification Groups.
- [2] Xilinx design guide ;url=http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx2019_1/ug1118-vivado-creating-packaging-custom-ip.pdf.
- [3] S. S. Hanna, A. A. El-Sherif, and M. Y. Elnainay. Maximizing usrp n210 sdr transfer rate by offloading modulation to the on-board fpga. 2016 International Conference on Wireless Networks and Mobile Communications (WINCOM), 2016.
- [4] R. source Code. https://github.com/mastro7xrtl_qam, Jun 2019.



7 Source Files: Listing

Listing 1. QAM_Hierarchical.vhd

```
-----  
--@DESCRIPTION  
-- This Module Is responsible for adding the two generated  
-- PassBand Waverforms:  
-- I_Symbol MixedWith cosWave + Q_Symbol MixedWith sinWave  
-----  
-- Module: QAM_Hierarchical  
-- Author: Astro  
-- Project: QAM Modulation  
-- Delievered to: Digital System Design  
-- Supervised by: Prof. Luca Fanucci  
-----  
-----  
  
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;  
USE IEEE.NUMERIC_STD.ALL;  
USE IEEE.MATH_REAL.ALL;  
  
ENTITY QAM_Hierarchical IS  
    PORT (  
        -- @Input Bits  
        input_stream : IN std_logic_vector(1 DOWNTO 0);  
        -- @System clock  
        i_clk : IN std_logic;  
        -- @Asynchronous active – high reset  
        i_rst_h : IN std_logic;  
        -- @sin Mixer Output  
        PASSBAND : OUT std_logic_vector(7 DOWNTO 0);  
        -- @Frequency Control Word [Phase Increment]  
        i_fcw : IN std_logic_vector(11 DOWNTO 0)  
    );  
END ENTITY; -- IQ_Adder  
ARCHITECTURE Behaviour1 OF QAM_Hierarchical IS  
  
    SIGNAL I_signal : std_logic_vector(7 DOWNTO 0);  
    SIGNAL Q_signal : std_logic_vector(7 DOWNTO 0);  
    SIGNAL adder_out : std_logic_vector(7 DOWNTO 0);  
    -----  
    -- QAM Modulator  
    -----
```



```
COMPONENT qam_mod IS
  PORT (
    — @System clock
    i_clk : IN std_logic;
    — Asynchronous active – high reset
    i_rstb : IN std_logic;
    — @Frequency Control Word [Phase Increment]
    i_fcw : IN std_logic_vector(11 DOWNTO 0);
    — @Input Symbols for 4-QAM
    i_meta_sym : IN std_logic_vector(1 DOWNTO 0);
    — @cos Mixer Output
    I_Mixer : OUT std_logic_vector(7 DOWNTO 0);
    — @sin Mixer Output
    Q_Mixer : OUT std_logic_vector(7 DOWNTO 0)
  );
END COMPONENT qam_mod;
```

—FULL ADDER

```
COMPONENT FA_N
  GENERIC (N : INTEGER := 2);
  PORT (
    — @Input of the full-adder A
    a : IN std_logic_vector(N – 1 DOWNTO 0);
    — @Input of the full-adder B
    b : IN std_logic_vector(N – 1 DOWNTO 0);
    — @Carry input
    c_i : IN std_logic;
    — @Output of the full-adder
    o : OUT std_logic_vector(N – 1 DOWNTO 0);
    — @Carry output
    c_o : OUT std_logic
  );
END COMPONENT FA_N;
```

BEGIN

```
—@qam_mod Component Wiring/Mapping
u_qam_mod : qam_mod
PORT MAP(
  i_clk      => i_clk ,
  i_rstb     => i_rst_h ,
  i_fcw      => i_fcw ,
  i_meta_sym => input_stream ,
  I_Mixer    => I_signal ,
  Q_Mixer    => Q_signal
```



```
);

--@u_IQ_Integrator Component Wiring/Mapping
u_IQ_Integrator : FA_N
    GENERIC MAP(N => 8)
PORT MAP(
    a    => I_signal ,
    b    => Q_signal ,
    c_i  => '0' ,
    o    => adder_out ,
    c_o  => OPEN
);
    PASSBAND <= adder_out;
END Behaviourl;
```



Listing 2. qam_mod.vhd

```

__*****
__
__ QAM Modulator: Modulation of 4-QAM
__
__ The system takes the input I and Q symbols each 2 bits
__ and produce The modulated waveforms for each the I and
__ Q components.
__
__ Module: qam_mod.vhd
__ Author: Astro
__ Project: QAM Modulation
__ Delievered to: Digital System Design
__ Supervised by: Prof. Luca Fanucci
__
__*****

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.MATH_REAL.ALL;

__*****
__*****ENTITY*****
__*****
ENTITY qam_mod IS
    PORT (
        __ @System clock
        i_clk : IN std_logic;
        __ @Asynchronous active – high reset
        i_rstb : IN std_logic;
        __ @Frequency Control Word [Phase Increment]
        i_fcw : IN std_logic_vector(11 DOWNTO 0);
        __ @nput Symbols for 4-QAM
        i_meta_sym : IN std_logic_vector(1 DOWNTO 0);
        __ @cos Mixer Output
        I_Mixer : OUT std_logic_vector(7 DOWNTO 0);
        __ @sin Mixer Output
        Q_Mixer : OUT std_logic_vector(7 DOWNTO 0));
END qam_mod;
__*****
__*****ARCHITECTURE*****
__*****

ARCHITECTURE rtl OF qam_mod IS

```




```
— *****
— START EMBEDDING AND DECLARING
— @COMPONENTS OF THE SYSTEM
— *****

-----
— QAM MAPPER
-----

COMPONENT qam_mapper
  PORT (
    — @Input DATA
    i_meta_sym : IN std_logic_vector (1 DOWNTO 0);
    — @In-Phase Symbols
    I_symbols : OUT std_logic_vector (1 DOWNTO 0);
    — @Quadrature-Phase Symbols
    Q_symbols : OUT std_logic_vector (1 DOWNTO 0);
    — @system clock
    i_clk : IN std_logic;
    — @Asynchronous active – high reset
    i_rstb : IN std_logic
  );
END COMPONENT;

-----
— Digital Oscillator -----
COMPONENT ddfs IS
  PORT (
    — @input frequency word
    fw : IN std_logic_vector(11 DOWNTO 0);
    — @Phase Offset/Starting Phase
    phase_s : IN std_logic_vector(11 DOWNTO 0);
    — @clock of the system
    clk : IN std_logic;
    — @Asynchronous active – high reset
    a_rst_h : IN std_logic;
    — @output waveform sin
    out_sinWave : OUT std_logic_vector(5 DOWNTO 0);
    — @output waveform cos
    out_cosWave : OUT std_logic_vector(5 DOWNTO 0)
  );
END COMPONENT ddfs;

-----
—FULL ADDER
-----

COMPONENT FA_N
  GENERIC (N : INTEGER := 2);
```



```

    PORT (
        -- @Input of the full-adder A
        a : IN std_logic_vector(N - 1 DOWNTO 0);
        -- @Input of the full-adder B
        b : IN std_logic_vector(N - 1 DOWNTO 0);
        -- @Carry input
        c_i : IN std_logic;
        -- @Output of the full-adder
        o : OUT std_logic_vector(N - 1 DOWNTO 0);
        -- @Carry output
        c_o : OUT std_logic
    );
END COMPONENT FA_N;

```

```

-- *****
--START WIRING THE SUB-SYSTEM COMPONENT
-- *****
SIGNAL phase_ctr : std_logic_vector(11 DOWNTO 0);
SIGNAL Q_sine    : std_logic_vector(5 DOWNTO 0);
SIGNAL I_cos     : std_logic_vector(5 DOWNTO 0);

SIGNAL I_mapper  : std_logic_vector(1 DOWNTO 0);
SIGNAL Q_mapper  : std_logic_vector(1 DOWNTO 0);
-- @modulator outputCos (-- 6(Waveform) + 2(Symbol))
SIGNAL I_molulator : signed(7 DOWNTO 0);
-- @modulator outputSin (-- 6(Waveform) + 2(Symbol))
SIGNAL Q_molulator  : signed(7 DOWNTO 0);
SIGNAL I_upconverted : signed(7 DOWNTO 0);
SIGNAL Q_upconverted : signed(7 DOWNTO 0);

BEGIN

    I_Mixer <= std_logic_vector(I_molulator);
    Q_Mixer <= std_logic_vector(Q_molulator);

```

```

-- Purpose:
-- I/Q Complex Data Generation [one symbol at a time]
-- since it's 4-QAM the symbol is containing 2 bits

```

```

u_mapper_qam : qam_mapper
PORT MAP(
    i_meta_sym => i_meta_sym,
    I_symbols  => I_mapper,
    Q_symbols  => Q_mapper,
    i_clk      => i_clk,
    i_rstb     => i_rstb

```



```
);

-- Purpose:
-- Sinsoidal Oscillator: Generate the Carrier
-- [Generate SIN/COS WAVE] simultenuously
-- Could be modified to be more generic by
-- adding a starting phase and a MUX to select
-- the output waveform (carrier)

u_ddfs_osc : ddfs
PORT MAP(
    fw          => i_fcw ,
    phase_s     => phase_ctr ,
    clk         => i_clk ,
    a_rst_h     => i_rstb ,
    out_sinWave => Q_sine ,
    out_cosWave => I_cos
);
p_mod_qam : PROCESS (i_clk , i_rstb)
BEGIN
    IF (i_rstb = '1') THEN
        I_upconverted <= (OTHERS => '0 ');
        Q_upconverted <= (OTHERS => '0 ');
        I_molulator   <= (OTHERS => '0 ');
        Q_molulator   <= (OTHERS => '0 ');
    ELSIF (rising_edge(i_clk)) THEN
        I_upconverted <= signed(I_cos) * signed(I_mapper);
        Q_upconverted <= signed(Q_sine) * signed(Q_mapper);
        I_molulator   <= I_upconverted;
        Q_molulator   <= Q_upconverted;

    END IF;
END PROCESS p_mod_qam;

END rtl;
```



Listing 3. qam_mapper.vhd

```

--- *****
---
--- QAM MAPPER: Constellation mapper of 4-QAM
---
--- we want to have one single change of bits from one symbol to the other;
--- thus, using fixed point number[gray code] representation {00, 01, 11, 10}
--- to represent {1+1j, -1+1j, -1-1j, 1-1j}
--- Constellation | Symbols
--- {00} | {+1+J}
--- {01} | {-1+J}
--- {10} | {+1-J}
--- {11} | {-1-J}
---
--- Module: qam_mapper
--- Author: Astro
--- Project: QAM Modulation
--- Delievered to: Digital System Design
--- Supervised by: Prof. Luca Fanucci
---
--- *****

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY qam_mapper IS
    PORT (
        -- @Input DATA
        i_meta_sym : IN std_logic_vector (1 DOWNTO 0);
        -- @In-Phase Symbols
        I_symbols : OUT std_logic_vector (1 DOWNTO 0);
        -- @Quadrature-Phase Symbols
        Q_symbols : OUT std_logic_vector (1 DOWNTO 0);
        -- @system clock
        i_clk : IN std_logic;
        -- @Asynchronous active - high reset
        i_rstb : IN std_logic
    );
END qam_mapper;

ARCHITECTURE Behaviour1 OF qam_mapper IS
    -- define array for both I and Q parts (4 constellation points)
    -- Constellation Size; to be generic whatever the modulation order
    CONSTANT const_size : INTEGER := 4;
    --++++++RESERVED++++++
    -- Could be tested with any frame Size (Binary Message Stream)

```



```
— Here: We test against all possible input combination:
— e.g 04-QAM: [00:11]
— 16-QAM [0000: 1111] ...etc
— constant frame_size : integer := 8;
— Frame Length
— TYPE frame is array (0 to frame_size-1) of std_logic;
—++++++RESERVED++++++

— Constellation Type
TYPE constellation IS ARRAY (0 TO const_size - 1) OF signed (1 DOWNTO 0);
— I array
— to_signed: Converts an INTEGER to a SIGNED vector of the specified SIZE.
CONSTANT I_data : constellation := (
    to_signed(1, 2),
    to_signed( - 1, 2),
    to_signed(1, 2),
    to_signed( - 1, 2)
);
— Q array
CONSTANT Q_data : constellation := (
    to_signed(1, 2),
    to_signed(1, 2),
    to_signed( - 1, 2),
    to_signed( - 1, 2)
);
SIGNAL bits_in_unsigned : unsigned (1 DOWNTO 0);

BEGIN

— to get from std_logic_vector to integer: > we go to unsigned then to integer
— s.t input data is needed to index the constellation array
bits_in_unsigned <= unsigned(i_meta_sym);
— Mapping of Input Data/Bits into QAM symbols

I_symbols <= std_logic_vector(I_data(to_integer(bits_in_unsigned)));
Q_symbols <= std_logic_vector(Q_data(to_integer(bits_in_unsigned)));

p_mapper_qam : PROCESS (i_clk, i_rstb)
BEGIN
    IF (i_rstb = '1') THEN
        I_symbols <= (OTHERS => '0');
        Q_symbols <= (OTHERS => '0');
    ELSIF (rising_edge(i_clk)) THEN
        I_symbols <= std_logic_vector(I_data(to_integer(bits_in_unsigned)));
        Q_symbols <= std_logic_vector(Q_data(to_integer(bits_in_unsigned)));
    END IF;
END PROCESS p_mapper_qam;
```



END Behaviourl;



Listing 4. ddfs.vhd

```
-----
-- *****
--
-- DDFS: Direct Digital Frequency Synthesizer
--
-- Module: ddfs
-- Author: Astro
-- Project: QAM Modulation
-- Delievered to: Digital System Design
-- Supervised by: Prof. Luca Fanucci
--
-- *****

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.MATH_REAL.ALL;
ENTITY ddfs IS
    PORT (
        -- @input frequency word
        fw : IN std_logic_vector(11 DOWNTO 0);
        -- @Phase Offset/Starting Phase
        phase_s : IN std_logic_vector(11 DOWNTO 0);
        -- @clock of the system
        clk : IN std_logic;
        -- @Asynchronous active – high reset
        a_rst_h : IN std_logic;
        -- @output waveform sin
        out_sinWave : OUT std_logic_vector(5 DOWNTO 0);
        -- @output waveform cos
        out_cosWave : OUT std_logic_vector(5 DOWNTO 0)
    );
END ddfs;

-- ARCHITICTURE OF DDFS SYSTEM

ARCHITECTURE struct OF ddfs IS

    -- Internal signals

    -- Output of of the phase accumulator counter
    SIGNAL S_phase_sine : std_logic_vector(11 DOWNTO 0);
    SIGNAL S_phase_cos  : std_logic_vector(11 DOWNTO 0);

    -- Output of the LUT table
```



```

SIGNAL lut_output_cos : std_logic_vector(5 DOWNTO 0);
SIGNAL lut_output_sin : std_logic_vector(5 DOWNTO 0);
— Output register for the output synchronization
SIGNAL sine_out : std_logic_vector(5 DOWNTO 0);

SIGNAL cos_out : std_logic_vector(5 DOWNTO 0);
— RESERVED
—SIGNAL start_phase : std_logic_vector(11 downto 0);

—*****
— @Used With Optimization Configuration
— Complemented output of the LUT
— signal lut_output_muxed_osc0 : std_logic_vector(5 downto 0);
— signal lut_output_muxed_osc1 : std_logic_vector(5 downto 0);
— LATCH ADDRESS:
— SIGNAL S_phase_sine_add : std_logic_vector(11 downto 0);
— SIGNAL S_phase_cos_add : std_logic_vector(11 downto 0);
—*****

— START EMBEDDING AND DECLARING
— @COMPONENTS OF THE SYSTEM

—PhaseGenerator

COMPONENT PhaseGenerator IS
    PORT (
        — @System clock
        clk : IN std_logic;
        — @Asynchronous active – high reset
        rst : IN std_logic;
        — @Phase Increment
        phase_increment : IN std_logic_vector(11 DOWNTO 0);
        — @starting phase of the counter
        phase_offset : IN std_logic_vector(11 DOWNTO 0);
        — @wave1: sin Oscillator
        phase_sin : OUT std_logic_vector(11 DOWNTO 0);
        — @wave2: cos Oscillator
        phase_cos : OUT std_logic_vector(11 DOWNTO 0)
    );
END COMPONENT PhaseGenerator;

—FULL ADDER

```




```

COMPONENT FA_N
  GENERIC (N : INTEGER := 2);
  PORT (
    — @Input of the full-adder A
    a : IN std_logic_vector(N - 1 DOWNTO 0);
    — @Input of the full-adder B
    b : IN std_logic_vector(N - 1 DOWNTO 0);
    — @Carry input
    c_i : IN std_logic;
    — @Output of the full-adder
    o : OUT std_logic_vector(N - 1 DOWNTO 0);
    — @Carry output
    c_o : OUT std_logic
  );
END COMPONENT FA_N;

— Sinsoidal LUTCH/ROM

COMPONENT ddfs_lut_4096 IS
  PORT (
    — @lut address/Index
    address : IN std_logic_vector(11 DOWNTO 0);
    — @lut output (wave values)
    dds_out : OUT std_logic_vector(5 DOWNTO 0)
  );
END COMPONENT;

— END EMBEDDING AND DECLARING
— @COMPONENTS OF THE SYSTEM

— START
— WIRING THE SUB-SYSTEM COMPONENT

BEGIN
  PHASE_ACCUMLATOR : PhaseGenerator
  PORT MAP(
    clk          => clk ,
    rst          => a_rst_h ,
    phase_increment => fw ,
    phase_offset  => phase_s ,
    phase_sin     => S_phase_sine ,
    phase_cos     => S_phase_cos
  );

```



```

LUT_4096_SIN : ddfs_lut_4096
PORT MAP(
    address => S_phase_sine ,
    dds_out => lut_output_sin
);

LUT_4096_COS : ddfs_lut_4096
PORT MAP(
    address => S_phase_cos ,
    dds_out => lut_output_cos
);
--*****
--ENABLE OPTIMIZED LATCH
--*****
-- Selecting the lut output
--lut_output_muxed_osc0 <= lut_output_cos when (S_phase_cos(11) = '0') else not
-- lut_output_muxed_osc1 <= lut_output_sin when (S_phase_sine(11) = '0') else n
-- Selecting the lut_address
-- S_phase_sine_add <= S_phase_sine(11 downto 0) when (S_phase_sine(10) = '0')
-- S_phase_cos_add <= S_phase_cos(11 downto 0) when (S_phase_cos(10) = '0') else
--*****

--
--
-- END
-- WIRING THE SUB-SYSTEM COMPONENT

--
--
-- PROCESS DIFFINITION
-- For each rising edge.

DDFS_OUTPUT_REG : PROCESS (a_rst_h, clk)
BEGIN
    IF (a_rst_h = '1') THEN
        sine_out <= (OTHERS => '0');
        cos_out  <= (OTHERS => '0');
        --start_phase <= (others => '1');
    ELSIF (rising_edge(clk)) THEN
        sine_out <= lut_output_sin;
        cos_out  <= lut_output_cos;
    END IF;
END PROCESS;

-- Mapping the output

```



```
        out_sinWave <= sine_out;  
        out_cosWave <= cos_out;  
END struct;
```



Listing 5. PhaseGenerator.vhd

```

-----
-- PhaseGenerator/Phase Accumulator
-----
-- The system is responsible for start counting once
-- system started; with different offset
-- e.g count from 0 to generate the sine wave (index the latch)
-- count from 1024 (equivalent to pi/2) to generate a cos wave
-----
-- Module: PhaseGenerator
-- Author: Astro
-- Project: QAM Modulation
-- Delievered to: Digital System Design
-- Supervised by: Prof. Luca Fanucci
-----

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY PhaseGenerator IS
    PORT (
        -- @system clock
        clk : IN std_logic;
        -- @Asynchronous active – high reset
        rst : IN std_logic;
        -- @FCW: Freq Control Word: phase increment
        phase_increment : IN std_logic_vector(11 DOWNTO 0);
        -- @starting phase of the counter
        phase_offset : IN std_logic_vector(11 DOWNTO 0);
        -- @wave1: sin Oscillator
        phase_sin : OUT std_logic_vector(11 DOWNTO 0);
        -- @wave2: cos Oscillator
        phase_cos : OUT std_logic_vector(11 DOWNTO 0)
    );
END PhaseGenerator;

-- ARCHITICTURE OF PHASE ACCUMLATOR

ARCHITECTURE arch OF PhaseGenerator IS

    SIGNAL accum          : unsigned(11 DOWNTO 0);
    SIGNAL phase_int      : unsigned(11 DOWNTO 0);
    SIGNAL phase_offset_d : unsigned(11 DOWNTO 0);

```



BEGIN

—Main Phase Accumulation process

main : **PROCESS** (clk, rst)

BEGIN

IF rst = '1' **THEN**

accum <= (**OTHERS** => '0');

— @pi/2 to index the cosine waveform so count from 1023

phase_offset_d <= "010000000010";

phase_int <= (**OTHERS** => '0');

ELSIF (rising_edge(clk)) **THEN**

accum <= accum + unsigned(phase_increment);

phase_int <= accum + phase_offset_d;

END IF;

END PROCESS;

— @fetch the cosine waveform by addresses

phase_cos <= std_logic_vector(phase_int);

— @fetch the cosine waveform by addresses

phase_sin <= std_logic_vector(accum);

END arch;



Listing 6. tb_QAM_Hierarchical.vhd

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

ENTITY tb_QAM_Hierarchical IS
END tb_QAM_Hierarchical;
ARCHITECTURE testbench OF tb_QAM_Hierarchical IS

    COMPONENT QAM_Hierarchical IS
        PORT (
            — @Input Bits
            input_stream : IN std_logic_vector(1 DOWNTO 0);
            — @System clock
            i_clk : IN std_logic;
            — @Asynchronous active – high reset
            i_rst_h : IN std_logic;
            — @sin Mixer Output
            PASSBAND : OUT std_logic_vector(7 DOWNTO 0);
            — @Frequency Control Word [Phase Increment]
            i_fcw : IN std_logic_vector(11 DOWNTO 0)
        );
    END COMPONENT;

    — constants declaration

    — CLK period (f_CLK = 125 MHz)
    CONSTANT T_CLK : TIME := 8 ns;
    — Maximum sine period
    CONSTANT T_max_period : TIME := 4096 * T_CLK;
    — Simulation time
    CONSTANT T_sim : TIME := 100 * T_max_period;
    — Time before the reset release
    CONSTANT T_reset : TIME := 10 ns;

    — signals declaration

    — clk signal (intialized to '0')
    SIGNAL clk_tb : std_logic := '0';
    — Active high asynchronous reset (Active at t = 0)
```



```
SIGNAL a_rst_h_tb : std_logic := '1';
— signal to stop the simulation
SIGNAL stop_simulation : std_logic := '1';
— signal for symbol duration
SHARED VARIABLE T_symbol_duration : TIME := 33 us;

— APPLICATION SPECIFIC SIGNALS

— freq_Word
SIGNAL T_FCW      : std_logic_vector(11 DOWNTO 0);
SIGNAL in_stream  : std_logic_vector(1  DOWNTO 0);
SIGNAL tb_signal_out : std_logic_vector(7  DOWNTO 0);

BEGIN

— clk variation
clk_tb <= (NOT(clk_tb) AND stop_simulation) AFTER T_CLK / 2;
— reset control
a_rst_h_tb <= '0' AFTER T_reset;
— stopping the simulation after T_sim
—stop_simulation <= '0' after T_sim;
— instead control the simulation via command line argument
— ghdl -r tb_IQ_Adder —stop-time=10000000ns —vcd=qam_mod.vcd

i_DUT_QAM_H : QAM_Hierarchical
PORT MAP(
    — @2 bits input
    input_stream => in_stream,
    — @clock of the system
    i_clk => clk_tb,
    — @Asynchronous active – high reset
    i_rst_h => a_rst_h_tb,
    — @output waveform
    PASSBAND => tb_signal_out,
    — @Frequency Control Word
    i_fcw => T_FCW
);

input_process : PROCESS

BEGIN
    T_FCW      <= (OTHERS => '0');
    in_stream  <= (OTHERS => '0');
    a_rst_h_tb <= '0';
    —wait until a_rst_h_tb = '0';
```



```
T_FCW <= std_logic_vector(to_unsigned(1, 12));

--TESTING VECOTRS
--report "test passed for input combination 00" severity error;
--Dummy Loop To do not Halt Simulation
FOR I IN 0 TO 100 LOOP

    in_stream <= "00";
    WAIT FOR T_symbol_duration;
    in_stream <= "01";
    WAIT FOR T_symbol_duration;
    in_stream <= "10";
    WAIT FOR T_symbol_duration;
    in_stream <= "11";
    WAIT FOR T_symbol_duration;

END LOOP;

END PROCESS;

END testbench;
```




Listing 7. Timing_Report_Summary.md

Copyright 1986–2018 Xilinx, Inc. All Rights Reserved.

```
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec 6 23:38:27 MST 2018
| Date        : Fri Jun 14 15:52:30 2019
| Host       : DESKTOP-0K7MQOL running 64-bit major release (build 9200)
| Command    : report_utilization -file C:/Users/rewal/Desktop/RTL_QAM-master/QAM_Viv
| Design     : QAM_Hierarchical
| Device     : 7z010clg400-1
| Design State : Routed
```

Utilization Design Information

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	285	0	17600	1.62
LUT as Logic	285	0	17600	1.62
LUT as Memory	0	0	6000	0.00
Slice Registers	83	0	35200	0.24
Register as Flip Flop	83	0	35200	0.24
Register as Latch	0	0	35200	0.00
F7 Muxes	34	0	8800	0.39
F8 Muxes	8	0	4400	0.18

3. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	60	0.00
RAMB36/FIFO*	0	0	60	0.00
RAMB18	0	0	120	0.00

4. DSP



Site Type	Used	Fixed	Available	Util%
DSPs	0	0	80	0.00

6. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	8	0.00
MMCME2_ADV	0	0	2	0.00
PLLE2_ADV	0	0	2	0.00
BUFMRCE	0	0	4	0.00
BUFHCE	0	0	48	0.00
BUFR	0	0	8	0.00

7. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

8. Primitives



Ref Name	Used	Functional Category
LUT6	174	LUT
FDCE	83	Flop & Latch
LUT5	68	LUT
LUT4	35	LUT
MUXF7	34	MuxFx
LUT2	33	LUT
LUT3	19	LUT
IBUF	16	IO
CARRY4	13	CarryLogic
OBUF	8	IO
MUXF8	8	MuxFx
LUT1	2	LUT
BUFG	1	Clock