QAM Mapper User Guide and Specification

May 31, 2025

Contents

1 Overview

1.1 Purpose

This document describes the specification and usage of the Quadrature Amplitude Modulation (QAM) Mapper hardware implementation in VHDL and its verification environment (testbench). The module maps input bits to QAM-16, QAM-64, QAM-128, or QAM-256 constellations, with optional Gray code conversion.

1.2 Scope

- **Target Audience**: FPGA designers, digital communication engineers, verification engineers.
- **Environment**: FPGA (e.g., Xilinx Zynq, Intel Stratix) for implementation and simulation.
- **Standards**: Normalized QAM constellations with Gray code mapping.

1.3 Related Documents

- Python Software Model: `utils.py` (`generate_constellation`, `GrayCoder`).
- VHDL Files: `qam_mapper.vhd`, `gray_decoder.vhd`, `constellation_lut.vhd`, `qam_mapper_tb.vhd`, `qam_expected_pkg.vhd`.
- Tools: Xilinx Vivado, Intel Quartus Prime.

2 System Overview

2.1 Functionality

The QAM Mapper provides:

- **Input Mapping**: Maps 8-bit input to QAM constellations (I/Q components).
- **QAM Sizes**: QAM-16 (4-bit), QAM-64 (6-bit), QAM-128 (7-bit), QAM-256 (8-bit).
- **Gray Code**: Optional conversion from Gray code to binary (`enable_gray_map`).
- **Error Handling**: Asserts `error_flag` for out-of-range inputs, outputs 0j.
- **Fixed-Point**: 16-bit Q4.12 format (4 integer bits, 12 fractional bits).
- **Clock-Synchronous**: Operates on a single clock edge (pipelining-ready).

2.2 Architecture

The QAM Mapper comprises:

1. **gray_decoder**: Converts Gray code to binary.
2. **constellation_lut**: Stores QAM constellation values in lookup tables (LUTs).
3. **qam_mapper**: Top-level module integrating input processing, Gray conversion, and LUT access.

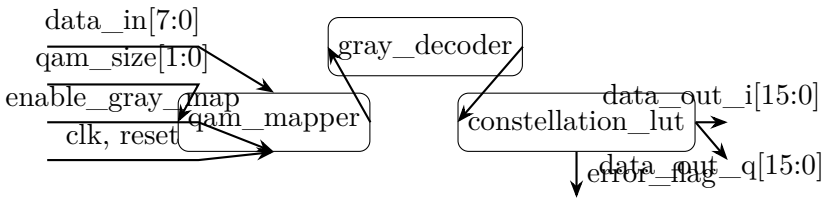

Figure 1: QAM Mapper Block Diagram

2.3 Block Diagram

3 Specification

3.1 Interfaces

3.1.1 qam_mapper

Table 1: qam_mapper Interface

Signal	Dir	Width	Description
`clk`	In	1	Clock (e.g., 100 MHz)
`reset`	In	1	Active-high synchronous reset
`data_in`	In	8	Input data (Gray or binary)
`qam_size`	In	2	QAM size (00: 16, 01: 64, 10: 128, 11: 256)
`enable_gray_map`	In	1	Gray code conversion (1: enabled, 0: disabled)
`data_out_i`	Out	16	I-component (Q4.12)
`data_out_q`	Out	16	Q-component (Q4.12)
`error_flag`	Out	1	Error detection (1 for out-of-range)

3.1.2 gray_decoder

Table 2: gray_decoder Interface

Signal	Dir	Width	Description
`data_in`	In	8	Gray code input
`bit_width`	In	4	Valid bit width (4, 6, 7, 8)
`enable`	In	1	Conversion enable (1: convert, 0: pass-through)
`data_out`	Out	8	Binary output

3.1.3 constellation_lut

3.2 Operational Specification

- **QAM Sizes and Bit Widths**:
  - QAM-16: 4 bits (`index[3:0]`).
  - QAM-64: 6 bits (`index[5:0]`).

Table 3: constellation_lut Interface

Signal	Dir	Width	Description
clk	In	1	Clock
index	In	8	LUT index
qam_size	In	2	QAM size
valid	In	1	Valid signal (1: valid)
data_out_i	Out	16	I-component (Q4.12)
data_out_q	Out	16	Q-component (Q4.12)

  - QAM-128: 7 bits (`index[6:0]`).
  - QAM-256: 8 bits (`index[7:0]`).

- **Gray Code Conversion**: When `enable_gray_map=1`, converts `data_in` from Gray code to binary using $b_n = g_n$, $b_k = g_k$ XOR b_{k+1}.
- **Constellation**: Normalized values from Python `generate_constellation` (average power = 1). QAM-128 uses an 8x16 asymmetric grid. Stored in Q4.12 format.
- **Error Handling**: Out-of-range inputs (e.g., `data_in` ≥ 16 for QAM-16) set `error_flag=1` and output 0j.
- **Timing**: 1 clock cycle latency (pipelining-ready).

3.3 Performance

- **Quantization Error**: ± 1 LSB (≈ 0.000244 in Q4.12).
- **SNR**: ≥ 30 dB (meets typical communication requirements).
- **Resource Usage**: FPGA-dependent (e.g., 500 LUTs, 2 BRAMs estimated).

4 Design Details

4.1 gray_decoder

- **Function**: Converts Gray code to binary.
- **Implementation**: Optimized parallel XOR circuit, limiting shifts to `bit_width-1` (e.g., 1, 2 for QAM-16).
- **Optimization**: Single-cycle processing, no redundant shifts.

4.2 constellation_lut

- **Function**: Outputs I/Q values from LUTs based on `qam_size` and `index`.
- **Implementation**: VHDL-2008 signal initialization for LUTs, generated by Python script.
- **Synthesis**: Inferred as BRAM/ROM.

4.3 qam_mapper

- **Function**: Integrates input processing, Gray conversion, and LUT access.
- **Implementation**: Uses `case` statement for QAM size selection, checks for out-of-range inputs.

5 Testbench

5.1 Overview

The testbench (`qam_mapper_tb.vhd`) verifies all QAM Mapper functionalities, using expected values from `qam_expected_pkg.vhd`.

5.2 Functionality

- **Test Cases**:
  - All QAM sizes (16, 64, 128, 256).
  - Gray code enabled/disabled.
  - Min (0), max (`qam_size-1`), mid (`qam_size/2`) indices.
  - Out-of-range inputs (`error_flag=1`, output 0j).
  - Reset behavior (output 0j, `error_flag=0`).
- **Verification**: `assert` statements check `data_out_i`, `data_out_q`, `error_flag` with ± 1 LSB tolerance.
- **Timing**: 10 ns clock (100 MHz), 2-cycle wait after `qam_size` change.

5.3 Usage

1. **Setup**:
   - Compiler: Vivado, ModelSim.
   - Files: `qam_mapper.vhd`, `gray_decoder.vhd`, `constellation_lut.vhd`, `qam_expected_pkg.vhd`.
2. **Generate Expected Values**:

```
python scripts/generate_expected_values.py
```

3. **Run Simulation**:
   - Compile `qam_mapper_tb.vhd`.
   - Execute (e.g., `vsim qam_mapper_tb`).
   - Inspect waveforms (`data_out_i`, `data_out_q`, `error_flag`).
4. **Verify Results**: Check `assert` errors and waveforms.

5.4 Timing Diagram

6 Installation and Setup

6.1 Required Tools

- **VHDL Compiler**: Vivado 2023.1, ModelSim 2022.3.
- **Python**: 3.8+ with `numpy`.
- **FPGA Board**: Xilinx Zynq-7000, Intel Stratix V (optional).

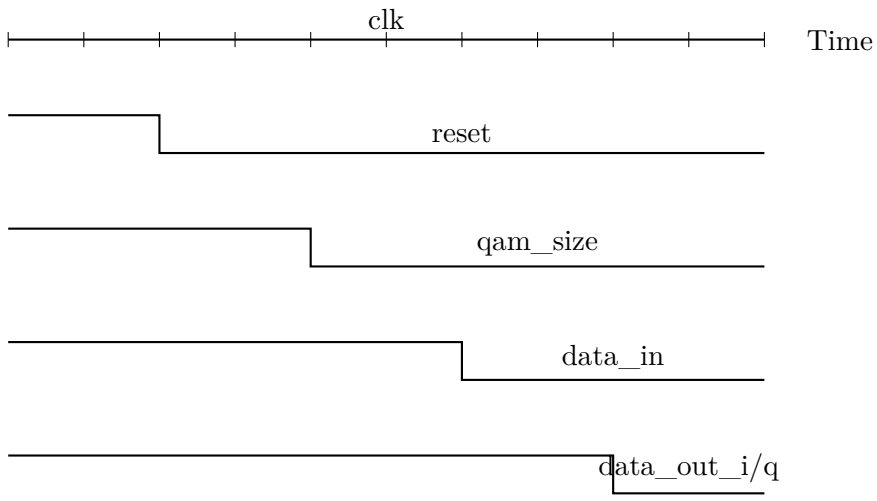

Figure 2: Simplified Timing Diagram

6.2 File Structure

```
project/
        src/
            qam_mapper.vhd
            gray_decoder.vhd
            constellation_lut.vhd
            qam_expected_pkg.vhd
        test/
            qam_mapper_tb.vhd
        scripts/
            generate_expected_values.py
        docs/
            qam_user_guide.pdf
```

6.3 Setup Instructions

1. Clone repository:

```
git clone <repository-url>
```

2. Install Python dependencies:

```
pip install numpy
```

3. Generate expected values:

```
python scripts/generate_expected_values.py
```

4. Create Vivado project, add VHDL files.

5. Run simulation or synthesis.

Table 4: Troubleshooting

Issue	Cause	Solution
`assert` errors	Expected value mismatch	Regenerate `qam_expected_pkg.vhd`.
Synthesis errors	VHDL-2008 unsupported	Enable VHDL-2008 in tool settings.
Timing violations	High clock frequency	Add pipelining (consult team).
Out-of-range errors	Input error	Check `error_flag`, fix `data_in`.

7 Troubleshooting

8 Maintenance and Extensions

8.1 Maintenance

- **Update Expected Values**: Rerun `generate_expected_values.py`.
- **Bug Fixes**: Modify VHDL, re-verify with testbench.
- **Documentation**: Update this guide for changes.

8.2 Extensions

- **Pipelining**: Add register stages for higher clock frequencies.
- **BER Testing**: Develop bit error rate testbench.
- **Floating-Point**: Support alternative formats (e.g., Q8.8).

9 Appendix

9.1 Glossary

- **QAM**: Quadrature Amplitude Modulation.
- **Gray Code**: Encoding where adjacent symbols differ by 1 bit.
- **Q4.12**: 16-bit fixed-point (4 integer, 12 fractional bits).

9.2 Sample Code

```
1  -- Example: gray_decoder (partial)
2  process (data_in, bit_width, enable)
3      variable binary_val : unsigned(7 downto 0);
4      variable masked_input : unsigned(7 downto 0);
5  begin
6      if to_integer(bit_width) < 1 or to_integer(bit_width) > 8 then
7          data_out <= (others => '0');
8      else
9          masked_input := unsigned(data_in) and
               (shift_left(to_unsigned(1, 8), to_integer(bit_width)) - 1);
10         if enable = '0' then
```

```
11              data_out <= std_logic_vector(masked_input);
12          else
13              binary_val := masked_input;
14              for i in 0 to 6 loop
15                  if i < to_integer(bit_width) - 1 then
16                      binary_val := binary_val xor
                           shift_right(binary_val, 2**i);
17                  end if;
18              end loop;
19              data_out <= std_logic_vector(binary_val and
                   (shift_left(to_unsigned(1, 8), to_integer(bit_width)) -
                   1));
20          end if;
21      end if;
22  end process;
```