QREAL FPGA Report

Finn Moore
fdm@umich.edu

University of Michigan

Introduction

Quantum information science (QIS) has revolutionized communication, sensing, and computing capabilities by leveraging the unique properties of quantum states. Quantum receivers play a pivotal role in improving performance by enabling enhanced information processing tasks such as quantum-state discrimination and parameter estimation. Traditional analytical approaches to Quantum Receivers struggle to adapt to diverse operational conditions and are quickly overwhelmed by noise in practical scenarios. To address this challenge, we utilize the Quantum Receiver Enhanced by Adaptive Learning (QREAL) architecture. QREAL leverages reinforcement learning to adaptively design quantum receivers capable of tackling a wide range of QIP problems and environmental conditions. Our results showcase the ability of QREAL to adapt to noise and imperfections, thereby outperforming conventional quantum receivers and classical approaches.

Methodology

1 Receiver Architecture

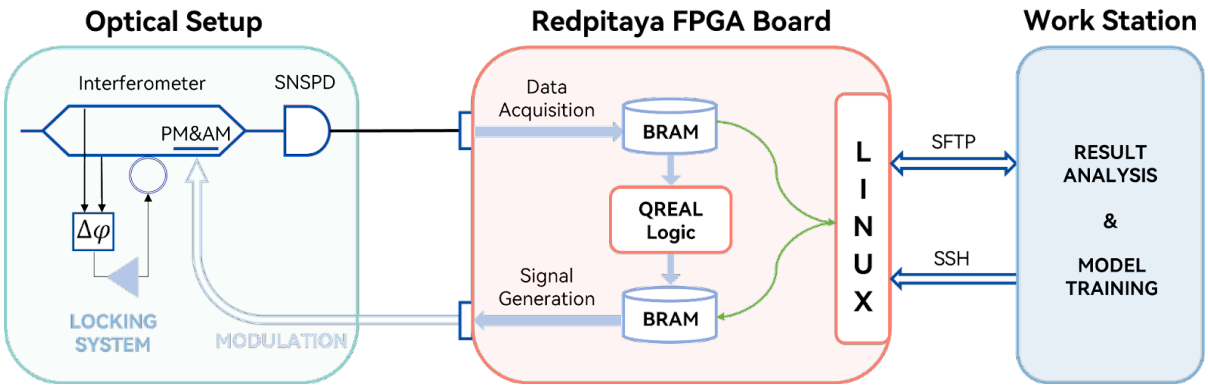

Figure 1: Receiver Architecture

The receiver was developed in three parts (1). The quantum hardware system was developed using a phase-locking interferometer and a superconducting nanowire single-photon detector (SNSPD). The FPGA control unit was developed in Verilog using Xilinx Vivado and deployed onto a Red Pitaya board. The adaptive learning model was developed in Python using the TensorFlow library.

2 FPGA Controller

2.1 Introduction

A field programmable gate array (FPGA) is an integrated circuit composed of many logic elements formed by look-up tables (LUTs). Due to the D-Flip-Flop in each of these logic elements as shown in 3,

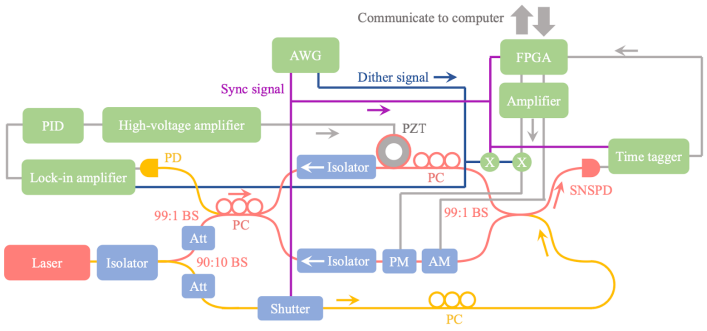

Figure 2: Optical Setup [1]

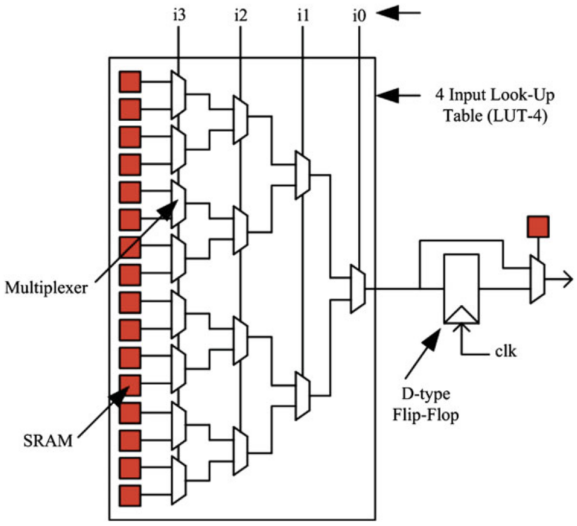

Figure 3: Basic Logic Element [2]

each LUT can either produce their output sequentially or combinationally, based on the programming of the FPGA. These basic logic elements are connected to form configurable logic blocks (4), and an array of configurable logic blocks connected to inputs and outputs forms an FPGA (5). An FPGA is programmed in a hardware description language (HDL) like Verilog or VHDL, where the programmer directly defines the digital logic that can be synthesized and translated into hardware. Once an HDL module is synthesized, the SRAM cells of the basic logic elements in the FPGA are populated with the values required to implement the programmed logic function.

2.2 Red Pitaya

The FPGA board selected for this project is the Red Pitaya STEMlab 125-14 as seen in 6. This board includes an Xilinx Zynq 7010 FPGA and a Dual-Core ARM Cortex-A9 MPCore processor for running Linux. The board also includes two SMA inputs and outputs. The inputs are used for synchronizing the locking circuit and for the SNSPD. The outputs control the phase and amplitude modulators in the local oscillator (LO) arm of the fiber optic circuit (2). The Red Pitaya is also equipped with a 125 MS/s 14-bit analog-to-digital converter (ADC). For this experiment, the Red Pitaya was programmed using Xilinx Vivado incorporating open-source and custom Verilog cores. The Linux processor was also used to run custom C programs for setting and reading BRAM values to communicate with the machine learning model.

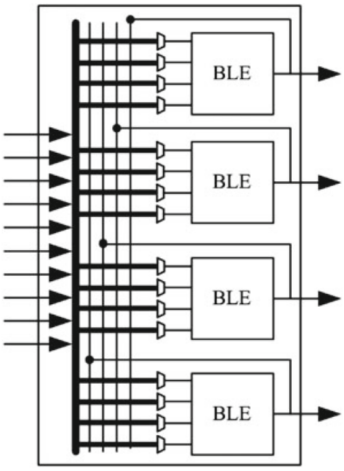

Figure 4: Configurable Logic Block [2]

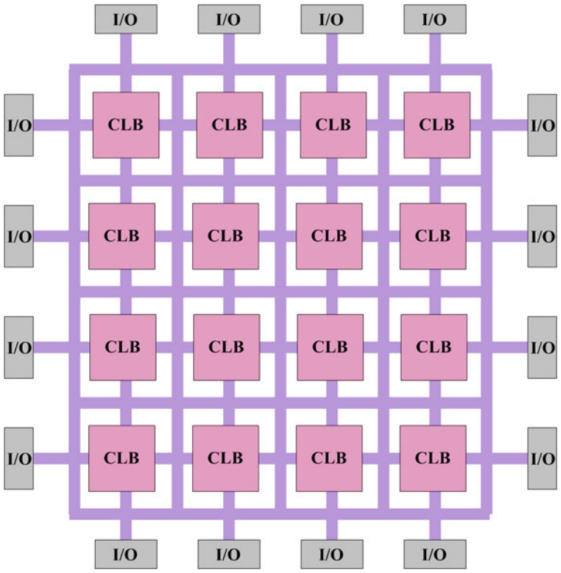

Figure 5: FPGA [2]

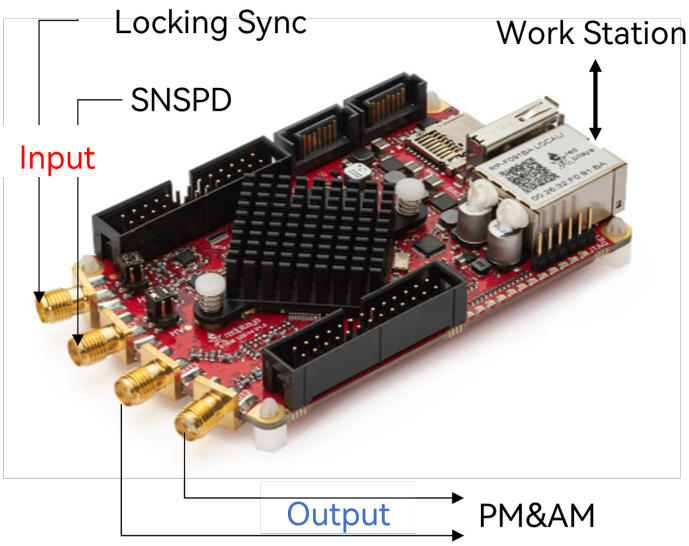

Figure 6: Red Pitaya STEMlab 125-14

2.3 Block Design

The block design (7) for the FPGA was developed in Xilinx Vivado and consists of four main components. Each of these modules works together to implement the data processing and communication between the formulator and the optical setup. The four main components are as follows:

2.3.1 DataAquisition (8)

This module uses the open-source AXI4-Stream core `axis_red_pitaya_adc_0` developed by Anton Potocnik [3] and a signal splitter to read the synchronization and SNSPD values from the ADC.

2.3.2 Xilinx Processing System (9)

Next is the processing system PS7 consisting of a ZYNQ IP core (`processing_system7_0`) allowing the FPGA to interface with the Linux processor.

2.3.3 Control (10)

The most important component, `control`, consists of many different sub-modules. The first of which (`axi_gpi_0`) is another open-source module enabling the use of a GPIO address on the Red Pitaya in conjunction with a C script to control whether block memory (BRAM) is being read or written to. This controls the bit enabling the `axis_bram_writer_0` module to write data to BRAM. In the case where the C program tells the FPGA to write BRAM, it takes the detection data from our custom Verilog module, `QREAL_42_BPSK_0` (A). This module consists of a state machine that implements the QREAL logic: for every cycle, for every bin, count the detections from the SNSPD and adjust the displacement for the signal generator. Finally, store the detections in BRAM to be sent to the machine learning model.

2.3.4 Signal Generator (11)

The fourth important module is the Singal Generation Module. This module actuates the displacement adjustment specified by the QREAL logic and creates signals to adjust the phase and amplitude modulators of the optical setup. It does this by reading BRAM from an address specified by the `control` unit and inputting that data to an open source DAC module [3] `axis_bram_reader_0` to be sent on the SMA outputs.

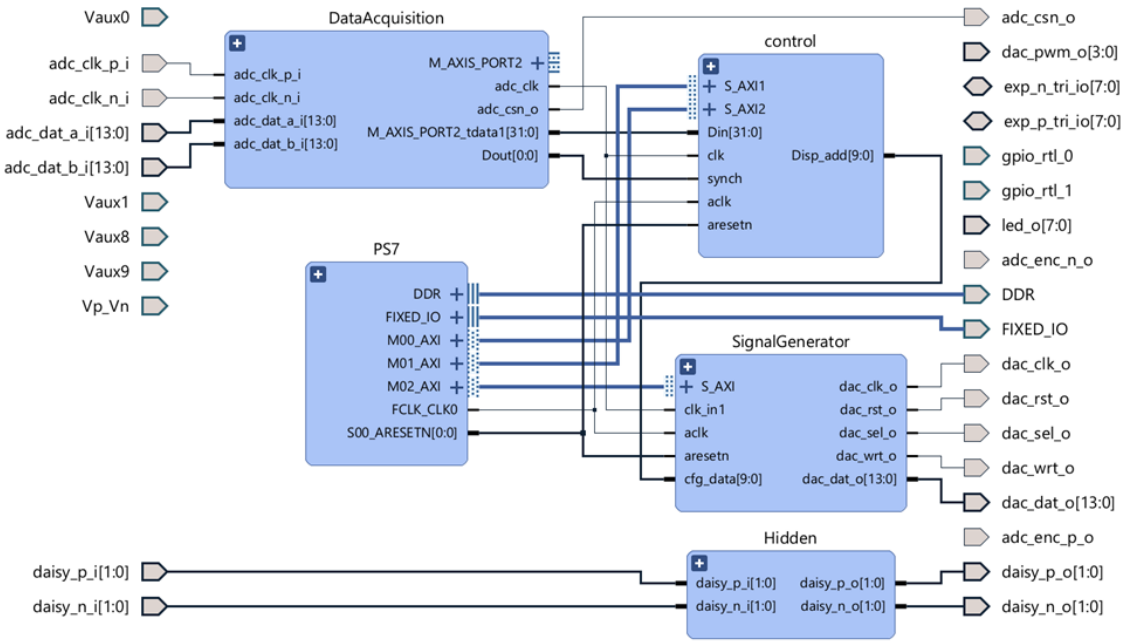

Figure 7: Overall Block Design

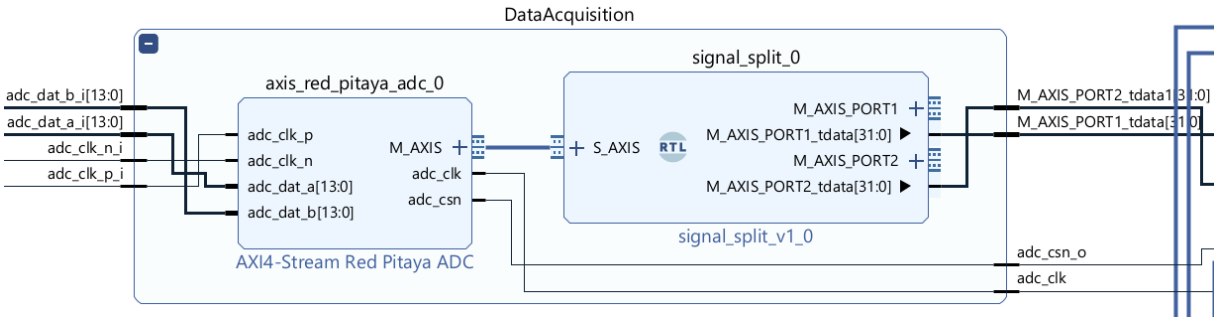

Figure 8: DataAcquisition Module

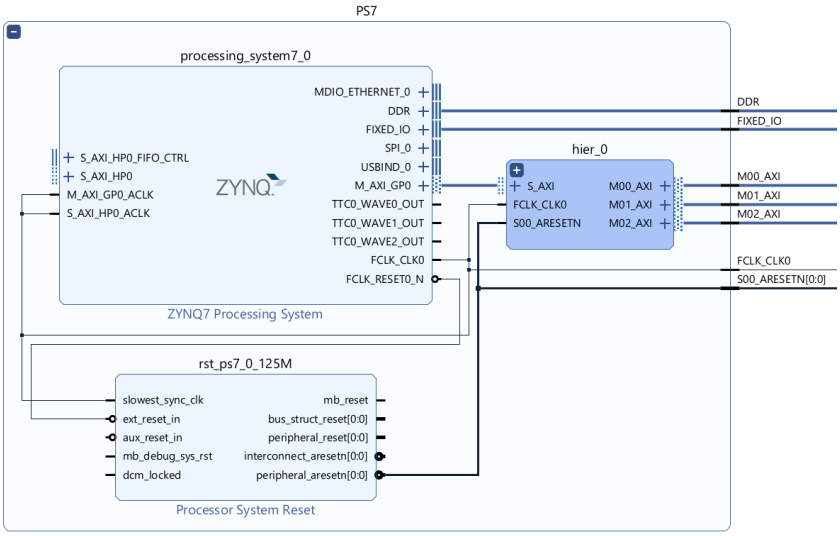

Figure 9: Processing System Module

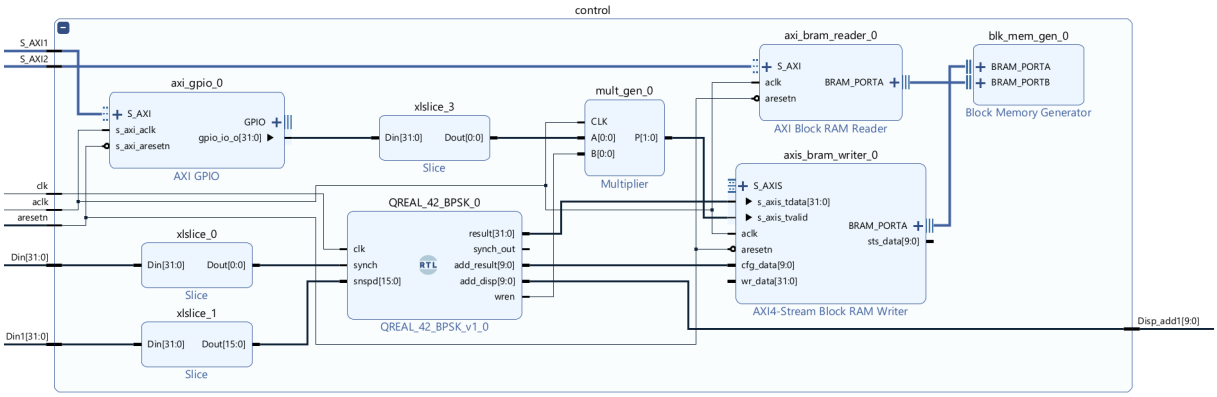

Figure 10: Control Module

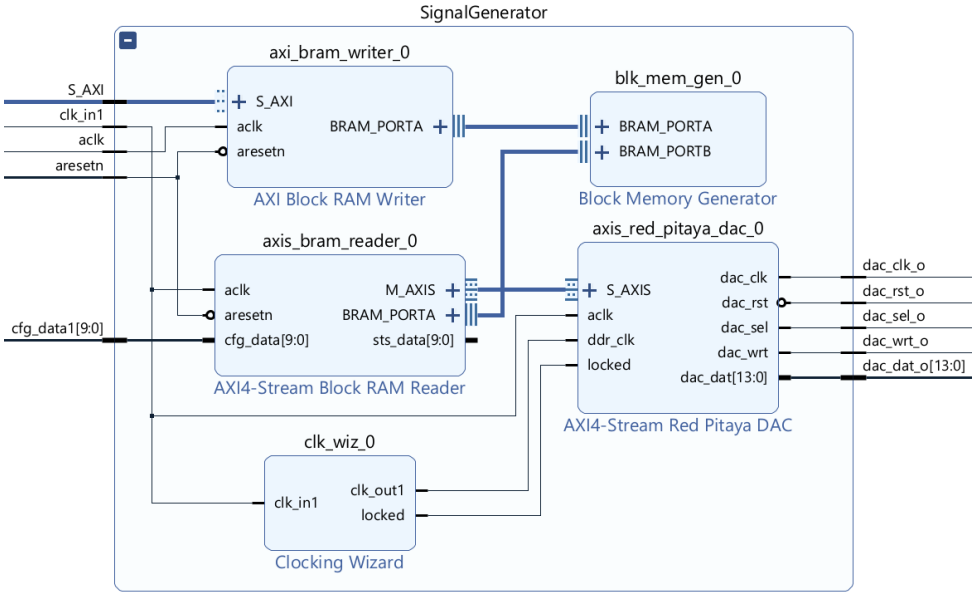

Figure 11: Signal Generation Module

2.4 C Programs

For communicating with the formulator, we need a way to take inputs and outputs from a regular operating system. To do this, we leverage the Linux chip present on the Red Pitaya board and its connection to the FPGA via the open-source AXI cores present in our block diagram allowing the reading and writing of GPIO and BRAM. To access the BRAM of the Red Pitaya board, the C scripts must utilize a pointer to the BRAM address specified in the block design for memory-mapped IO. The values in this range can be accessed through pointer arithmetic and bit shifting as seen in B.

Simulation Results

Our model shows significant performance benefits over heterodyne in simulation.

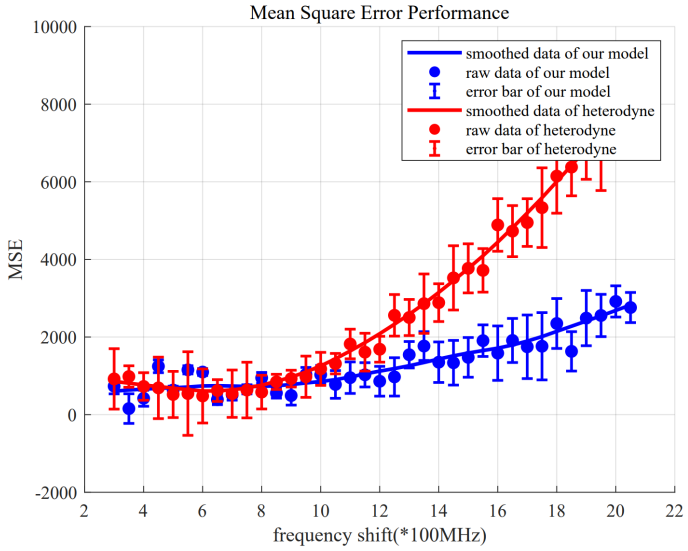

Figure 12: Velocimetry Mean Square Error Performance

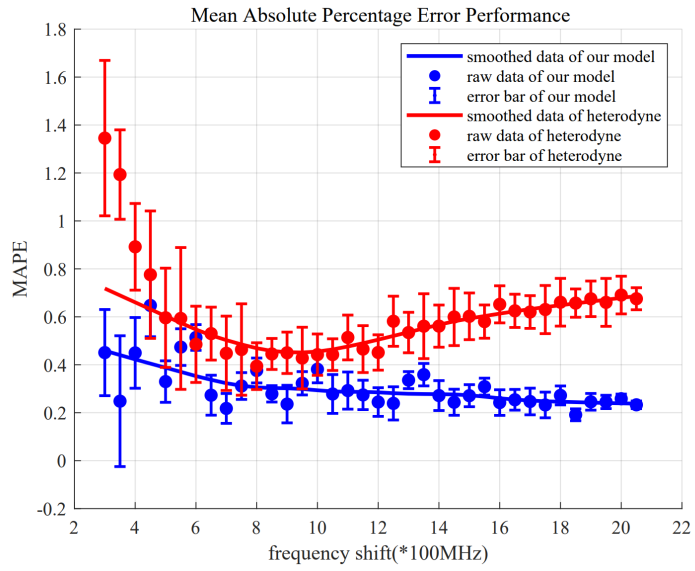

Figure 13: Velocimetry Mean Absolute Percentage Error Performance

Outlook

The outlook for this project is promising, marked by advancements in quantum receiver technology facilitated by the QREAL architecture. Leveraging reinforcement learning, QREAL has demonstrated adaptability to diverse operational conditions and noise environments, showcasing improved performance compared to traditional analytical approaches. Moving forward, key areas of focus for improvement include addressing noise sources in the optical setup and refining the code used to facilitate the experiment. By continuing to innovate and refine these technologies, this project is positioned well to contribute significantly to the advancement of QIS and its applications in communication, sensing, and computing domains.

References

[1] Chaohan Cui, William Horrocks, Shuhong Hao, Saikat Guha, Nasser Peyghambarian, Quntao Zhuang, and Zheshen Zhang. Quantum receiver enhanced by adaptive learning. *Light: Science & Applications*, 11(1):344, December 2022.

[2] Umer Farooq, Zied Marrakchi, and Habib Mehrez. FPGA Architectures: An Overview. In *Tree-based Heterogeneous FPGA Architectures*, pages 7–48. Springer New York, New York, NY, 2012.

[3] Anton Potocnik. redpitaya_guide. `https://github.com/apotocnik/redpitaya_guide`, 2019.

A QREAL_42_BPSK.v

```
module QREAL_42_BPSK(
input clk,
input synch,
input signed [15:0] snspd,
output [31:0] result,
output synch_out,
output [9:0] add_result,
output [9:0] add_disp,
output wren
//output done
    );
    //parameters
    reg signed [15:0] THRESHOLD =  16'd1600;
    parameter [9:0] ADD_LOCKING = 1022;
    //regs for values
    reg [31:0] reg_result = 0;
    reg signed [15:0] reg_snspd;
    reg reg_synch = 0;
    reg [9:0] reg_add_result = 0;
    reg [9:0] reg_add_disp = 0;
    reg [9:0] reg_add_disp_real = 0;
    reg reg_wren = 0;
    reg reg_photoncount = 0;
    reg [9:0] reg_Q = 10'b1011010010;
    //regs counters
    reg [10:0] reg_counter_framedelay = 0;
    reg [9:0] reg_counter_bin = 0;
    //regs states
    reg [1:0] reg_state = 0;
    reg [1:0] reg_binidx = 0;
    reg [1:0] reg_cycleidx = 0;

    reg [31:0] reg_test = 100;

    always @(posedge clk) begin
    reg_snspd <=snspd;
    reg_synch <= synch;
        case(reg_state)
            2'b00: begin // locking & modulation
                if(!reg_synch && synch) begin // rising edge of sync
                    reg_state <= 2'b01;
                    reg_add_result <= reg_add_result + 1;
                    reg_result <= reg_Q[9];
                end
            end

            2'b01: begin // waiting shutter goes off
                if(reg_counter_framedelay == 11'd2000) begin // when delay is over
                    reg_counter_framedelay <= 0;
```

```
                    reg_state <= 2'b10;
                    reg_add_disp <=0;
                    reg_add_disp_real <= reg_Q[9]*100;
                end
                else begin
                    reg_counter_framedelay <= reg_counter_framedelay + 1;
                end
            end

            2'b10: begin // qreal logic
                if(reg_counter_bin == 10'd625) begin // if the bin is over
                    reg_counter_bin <= 0;
                    reg_result <= (reg_result << 1) + reg_photoncount;
                    reg_photoncount <= 0;
                    if(reg_binidx == 2'b11) begin // if the cycle is over
                        reg_binidx = 2'b00;
                        reg_add_disp <= 0;
                        reg_add_disp_real <= reg_Q[9]*100;
                        if(reg_cycleidx == 2'b10) begin // if the last cycle is over
                            reg_state <= 2'b11;
                            reg_cycleidx <= 2'b00;
                            reg_add_disp <= ADD_LOCKING;
                            reg_add_disp_real <= ADD_LOCKING;
                        end
                        else
                            reg_cycleidx <= reg_cycleidx + 1;
                    end
                    else begin// if the cycle is not over
                        reg_binidx <= reg_binidx + 1;
                        reg_add_disp <= (reg_add_disp << 1) + 1 + reg_photoncount;
                        reg_add_disp_real <= (reg_add_disp << 1) + 1 + reg_photoncount + reg_Q[9]*100;
                    end
                end
                else begin // within a bin
                   if((reg_snspd < THRESHOLD) && (snspd > THRESHOLD) && (reg_counter_bin > 125))
                        reg_photoncount <= 1;
                   reg_counter_bin <= reg_counter_bin + 1;
                end
            end

            2'b11: begin // storing result
                if(reg_synch && !synch) begin // falling edge of sync
                    reg_state <= 2'b00;
                    reg_wren <= 0;
                    reg_Q <= (reg_Q << 1) + reg_Q[9]^reg_Q[6];
                end
                else begin
                    reg_wren <= 1;
                end
            end
        endcase
    end

assign result = reg_result;
assign add_result = reg_add_result;
assign add_disp = reg_add_disp_real;
assign wren = reg_wren;
endmodule
```

B QREAL_42_BPSK.c

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int fd;
  FILE* fp;
  void *rd;
  void *ctrl;
  void *wr;
  char *name = "/dev/mem";
  const int freq = 124998750; // Hz
```

```
  uint32_t buffer[1024];
  uint32_t j;
  uint32_t b;
  uint32_t i;
  uint32_t c;
  uint32_t result[17] = {0};
  uint32_t a[32];
  int32_t dispa[15] =
    {2981,3003,3542,3575,3487,3685,3542,3872,3366,3685,3685,3597,3036,3685,3630};
  int32_t dispp1[15] = {4664,4664,0,4664,0,0,4664,4664,0,0,4664,0,4664,4664,0};
int32_t dispp2[15] = {0,0,4664,0,4664,4664,0,0,4664,4664,0,4664,0,0,4664};

  printf("One\n");
  if((fd = open(name, O_RDWR)) < 0)
  {
    perror("open");
    return 1;
  }
  if (!(fp = fopen("result.txt", "w"))) {
      printf("Error opening file\n");
  }
  printf("Two\n");
  rd = mmap(NULL, sysconf(_SC_PAGESIZE), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x40000000
    );
  wr = mmap(NULL, sysconf(_SC_PAGESIZE), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x40010000
    );
  ctrl = mmap(NULL, sysconf(_SC_PAGESIZE), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0
    x41200000);
  printf("Three\n");

// -1~1V -8192~8191
  for(j = 0; j < 15; ++j)
    (*((int32_t *)(wr + 4*j))) = (dispa[j]<<16) + dispp1[j]  ;
  for(j = 0; j < 15; ++j)
    (*((int32_t *)(wr + 4*(100+j)))) = (dispa[j]<<16) + dispp2[j]  ;
  (*((int32_t *)(wr + 4*1022))) = 0 + (341<<16);
  (*((uint32_t *)(ctrl))) = 1<<0; // wren high
  sleep(2);
  (*((uint32_t *)(ctrl))) = 0<<0; // wren low

  for(j = 0; j < 1024; ++j)
  {
    buffer[j] = (*((uint32_t *)(rd + 4*j)));
  }

  printf("Five\n");
  for(j = 0; j < 1024; ++j)
  {
    b = buffer[j];
    fprintf(fp, "%d\n", buffer[j]);
    for(i = 0; i < 17; ++i)
    {
      result[i] = b%2;
      b = b/2;
    }
    printf("%u\t",result[16]);
    for(i = 0; i < 4; ++i)
      printf("%u",result[15-i]);
    printf("\t");
    for(i = 0; i < 4; ++i)
      printf("%u",result[11-i]);
    printf("\t");
    for(i = 0; i < 4; ++i)
      printf("%u",result[7-i]);
    printf("\t");
    for(i = 0; i < 4; ++i)
      printf("%u",result[3-i]);
    printf("\n");
  }

  munmap(rd, sysconf(_SC_PAGESIZE));
  munmap(ctrl, sysconf(_SC_PAGESIZE));
```

```
86   munmap(wr, sysconf(_SC_PAGESIZE));
87   return 0;
88 }
```