Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

PWM + eQEP module
In FPGA zybo base
experiment

강사 : Innova Lee(이 상훈)

학생 : 장 성환

A상 B상 두 신호를 입력 받거나 Z상을 이용한 방법을 사용하는 eQEP 패리페럴은 기존 PWM 펄스파형에 비해서 4배의 신호 데이터를 획득하기 때문에 더욱 정밀한 측정을 요구될 때 사용된다.
 eQEP는 기본적으로 저속 측정 모드와 고속 측정 모드로 나눌 수 있다. 고속 측정 모드를 지원하는 Custom IP 를 제작하여 SDK 실험을 하였다.

과정은 다음과 같다.

1. Custom IP 제작에 들어가서 원하는 IP를 제작한다.
탑 모듈의 코드는 다음과 같다.

```
`timescale 1 ns / 1 ps

        module My_EQEP_Core_v1_0 #
        (
                // Users to add parameters here

                // User parameters ends
                // Do not modify the parameters beyond this line


                // Parameters of Axi Slave Bus Interface S00_AXI
                parameter integer C_S00_AXI_DATA_WIDTH      = 32,
                parameter integer C_S00_AXI_ADDR_WIDTH      = 4
        )
        (
                // Users to add ports here
                input wire pwm_sig_a,
                input wire pwm_sig_b,
                // User ports ends
                // Do not modify the ports beyond this line


                // Ports of Axi Slave Bus Interface S00_AXI
                input wire  s00_axi_aclk,
                input wire  s00_axi_aresetn,
                input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
                input wire [2 : 0] s00_axi_awprot,
                input wire  s00_axi_awvalid,
                output wire  s00_axi_awready,
                input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
                input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
                input wire  s00_axi_wvalid,
                output wire  s00_axi_wready,
                output wire [1 : 0] s00_axi_bresp,
                output wire  s00_axi_bvalid,
                input wire  s00_axi_bready,
                input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
                input wire [2 : 0] s00_axi_arprot,
                input wire  s00_axi_arvalid,
```

```verilog
            output wire  s00_axi_arready,
            output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
            output wire [1 : 0] s00_axi_rresp,
            output wire  s00_axi_rvalid,
            input wire  s00_axi_rready
        );
// Instantiation of Axi Bus Interface S00_AXI
        My_EQEP_Core_v1_0_S00_AXI # (
                .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
                .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
        ) My_EQEP_Core_v1_0_S00_AXI_inst (
            .pwm_sig_a(pwm_sig_a),
            .pwm_sig_b(pwm_sig_b),
                .S_AXI_ACLK(s00_axi_aclk),
                .S_AXI_ARESETN(s00_axi_aresetn),
                .S_AXI_AWADDR(s00_axi_awaddr),
                .S_AXI_AWPROT(s00_axi_awprot),
                .S_AXI_AWVALID(s00_axi_awvalid),
                .S_AXI_AWREADY(s00_axi_awready),
                .S_AXI_WDATA(s00_axi_wdata),
                .S_AXI_WSTRB(s00_axi_wstrb),
                .S_AXI_WVALID(s00_axi_wvalid),
                .S_AXI_WREADY(s00_axi_wready),
                .S_AXI_BRESP(s00_axi_bresp),
                .S_AXI_BVALID(s00_axi_bvalid),
                .S_AXI_BREADY(s00_axi_bready),
                .S_AXI_ARADDR(s00_axi_araddr),
                .S_AXI_ARPROT(s00_axi_arprot),
                .S_AXI_ARVALID(s00_axi_arvalid),
                .S_AXI_ARREADY(s00_axi_arready),
                .S_AXI_RDATA(s00_axi_rdata),
                .S_AXI_RRESP(s00_axi_rresp),
                .S_AXI_RVALID(s00_axi_rvalid),
                .S_AXI_RREADY(s00_axi_rready)
        );

        // Add user logic here

        // User logic ends

        endmodule
```

하위모듈은 다음과 같다.

`timescale 1 ns / 1 ps

```
module My_EQEP_Core_v1_0_S00_AXI #
(
        // Users to add parameters here
        // User parameters ends
        // Do not modify the parameters beyond this line
        // Width of S_AXI data bus
        parameter integer C_S_AXI_DATA_WIDTH        = 32,
        // Width of S_AXI address bus
        parameter integer C_S_AXI_ADDR_WIDTH        = 4
)
(
        // Users to add ports here
        input wire pwm_sig_a,
        input wire pwm_sig_b,

        // User ports ends
        // Do not modify the ports beyond this line

        // Global Clock Signal
        input wire  S_AXI_ACLK,
        // Global Reset Signal. This Signal is Active LOW
        input wire  S_AXI_ARESETN,
        // Write address (issued by master, acceped by Slave)
        input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
        // Write channel Protection type. This signal indicates the
        // privilege and security level of the transaction, and whether
        // the transaction is a data access or an instruction access.
        input wire [2 : 0] S_AXI_AWPROT,
        // Write address valid. This signal indicates that the master signaling
        // valid write address and control information.
        input wire  S_AXI_AWVALID,
        // Write address ready. This signal indicates that the slave is ready
        // to accept an address and associated control signals.
        output wire  S_AXI_AWREADY,
        // Write data (issued by master, acceped by Slave)
        input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
        // Write strobes. This signal indicates which byte lanes hold
        // valid data. There is one write strobe bit for each eight
        // bits of the write data bus.
        input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
        // Write valid. This signal indicates that valid write
        // data and strobes are available.
        input wire  S_AXI_WVALID,
        // Write ready. This signal indicates that the slave
```

```verilog
        // can accept the write data.
        output wire  S_AXI_WREADY,
        // Write response. This signal indicates the status
        // of the write transaction.
        output wire [1 : 0] S_AXI_BRESP,
        // Write response valid. This signal indicates that the channel
        // is signaling a valid write response.
        output wire  S_AXI_BVALID,
        // Response ready. This signal indicates that the master
        // can accept a write response.
        input wire  S_AXI_BREADY,
        // Read address (issued by master, acceped by Slave)
        input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
        // Protection type. This signal indicates the privilege
        // and security level of the transaction, and whether the
        // transaction is a data access or an instruction access.
        input wire [2 : 0] S_AXI_ARPROT,
        // Read address valid. This signal indicates that the channel
        // is signaling valid read address and control information.
        input wire  S_AXI_ARVALID,
        // Read address ready. This signal indicates that the slave is
        // ready to accept an address and associated control signals.
        output wire  S_AXI_ARREADY,
        // Read data (issued by slave)
        output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
        // Read response. This signal indicates the status of the
        // read transfer.
        output wire [1 : 0] S_AXI_RRESP,
        // Read valid. This signal indicates that the channel is
        // signaling the required read data.
        output wire  S_AXI_RVALID,
        // Read ready. This signal indicates that the master can
        // accept the read data and response information.
        input wire  S_AXI_RREADY
);

// AXI4LITE signals
reg [C_S_AXI_ADDR_WIDTH-1 : 0]        axi_awaddr;
reg      axi_awready;
reg      axi_wready;
reg [1 : 0]      axi_bresp;
reg      axi_bvalid;
reg [C_S_AXI_ADDR_WIDTH-1 : 0]        axi_araddr;
reg      axi_arready;
reg [C_S_AXI_DATA_WIDTH-1 : 0]        axi_rdata;
reg [1 : 0]      axi_rresp;
reg      axi_rvalid;

// Example-specific design signals
```

```verilog
// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
// ADDR_LSB is used for addressing 32/64 bit registers/memories
// ADDR_LSB = 2 for 32 bits (n downto 2)
// ADDR_LSB = 3 for 64 bits (n downto 3)
localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
localparam integer OPT_MEM_ADDR_BITS = 1;
//----------------------------------------------
//-- Signals for user logic register space example
//----------------------------------------------
//-- Number of Slave Registers 4
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg3;
wire    slv_reg_rden;
wire    slv_reg_wren;
reg [C_S_AXI_DATA_WIDTH-1:0] reg_data_out;
integer  byte_index;
reg      aw_en;

// user register ****************************************************

reg [31:0] QUPRD; // Input period Hz_val_data. default is 10000000 hz (7s zero)
reg [31:0] QPOSLAT; // Get count_val_data.
reg [31:0] phase_a;
reg [31:0] phase_b;
reg [31:0] vclk_count;
reg phase_a_flag;
reg phase_b_flag;

//input wire pwm_sig_a, pwm_sig_b




// user register end ***********************************************



// I/O Connections assignments

assign S_AXI_AWREADY   = axi_awready;
assign S_AXI_WREADY    = axi_wready;
assign S_AXI_BRESP     = axi_bresp;
assign S_AXI_BVALID    = axi_bvalid;
assign S_AXI_ARREADY   = axi_arready;
assign S_AXI_RDATA     = axi_rdata;
assign S_AXI_RRESP     = axi_rresp;
assign S_AXI_RVALID    = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
```

```verilog
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awready <= 1'b0;
      aw_en <= 1'b1;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
        begin
          // slave is ready to accept write address when
          // there is a valid write address and write data
          // on the write address and data bus. This design
          // expects no outstanding transactions.
          axi_awready <= 1'b1;
          aw_en <= 1'b0;
        end
        else if (S_AXI_BREADY && axi_bvalid)
          begin
            aw_en <= 1'b1;
            axi_awready <= 1'b0;
          end
      else
        begin
          axi_awready <= 1'b0;
        end
    end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awaddr <= 0;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
        begin
          // Write Address latching
          axi_awaddr <= S_AXI_AWADDR;
        end
```

```verilog
      end
  end

  // Implement axi_wready generation
  // axi_wready is asserted for one S_AXI_ACLK clock cycle when both
  // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
  // de-asserted when reset is low.

  always @( posedge S_AXI_ACLK )
  begin
    if ( S_AXI_ARESETN == 1'b0 )
      begin
        axi_wready <= 1'b0;
      end
    else
      begin
        if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID && aw_en )
          begin
            // slave is ready to accept write data when
            // there is a valid write address and write data
            // on the write address and data bus. This design
            // expects no outstanding transactions.
            axi_wready <= 1'b1;
          end
        else
          begin
            axi_wready <= 1'b0;
          end
      end
  end

  // Implement memory mapped register select and write logic generation
  // The write data is accepted and written to memory mapped registers when
  // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
  // select byte enables of slave registers while writing.
  // These registers are cleared when reset (active low) is applied.
  // Slave register write enable is asserted when valid address and data are available
  // and the slave is ready to accept the write address and write data.
  assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID;

  always @( posedge S_AXI_ACLK )
  begin
    if ( S_AXI_ARESETN == 1'b0 )
      begin
        slv_reg0 <= 0;
        slv_reg1 <= 0;
        slv_reg2 <= 0;
        slv_reg3 <= 0;
```

```verilog
        end
    else begin
      if (slv_reg_wren)
        begin
          case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
            2'h0:
              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                  // Respective byte enables are asserted as per write strobes
                  // Slave register 0
                  QUPRD[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                end
            2'h1:
              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                  // Respective byte enables are asserted as per write strobes
                  // Slave register 1
                  slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                end
            2'h2:
              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                  // Respective byte enables are asserted as per write strobes
                  // Slave register 2
                  slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                end
            2'h3:
              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                  // Respective byte enables are asserted as per write strobes
                  // Slave register 3
                  slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                end
            default : begin
                        slv_reg0 <= QUPRD;
                        slv_reg1 <= slv_reg1;
                        slv_reg2 <= slv_reg2;
                        slv_reg3 <= slv_reg3;
                      end
          endcase
        end
    end
  end

  // Implement write response logic generation
  // The write response and response valid signals are asserted by the slave
```

```verilog
// when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
// This marks the acceptance of address and indicates the status of
// write transaction.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_bvalid  <= 0;
      axi_bresp   <= 2'b0;
    end
  else
    begin
      if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
S_AXI_WVALID)
        begin
          // indicates a valid write response is available
          axi_bvalid <= 1'b1;
          axi_bresp  <= 2'b0; // 'OKAY' response
        end              // work error responses in future
      else
        begin
          if (S_AXI_BREADY && axi_bvalid)
            //check if bready is asserted while bvalid is high)
            //(there is a possibility that bready is always asserted high)
            begin
              axi_bvalid <= 1'b0;
            end
        end
    end
end

// Implement axi_arready generation
// axi_arready is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_awready is
// de-asserted when reset (active low) is asserted.
// The read address is also latched when S_AXI_ARVALID is
// asserted. axi_araddr is reset to zero on reset assertion.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_arready <= 1'b0;
      axi_araddr  <= 32'b0;
    end
  else
    begin
      if (~axi_arready && S_AXI_ARVALID)
        begin
```

```verilog
        // indicates that the slave has accepted the valid read address
        axi_arready <= 1'b1;
        // Read address latching
        axi_araddr  <= S_AXI_ARADDR;
      end
    else
      begin
        axi_arready <= 1'b0;
      end
    end
end

// Implement axi_arvalid generation
// axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_ARVALID and axi_arready are asserted. The slave registers
// data are available on the axi_rdata bus at this instance. The
// assertion of axi_rvalid marks the validity of read data on the
// bus and axi_rresp indicates the status of read transaction.axi_rvalid
// is deasserted on reset (active low). axi_rresp and axi_rdata are
// cleared to zero on reset (active low).
always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_rvalid <= 0;
      axi_rresp  <= 0;
    end
  else
    begin
      if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
        begin
          // Valid read data is available at the read data bus
          axi_rvalid <= 1'b1;
          axi_rresp  <= 2'b0; // 'OKAY' response
        end
      else if (axi_rvalid && S_AXI_RREADY)
        begin
          // Read data is accepted by the master
          axi_rvalid <= 1'b0;
        end
    end
end

// Implement memory mapped register select and read logic generation
// Slave register read enable is asserted when valid address is available
// and the slave is ready to accept the read address.
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
```

```verilog
        case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
          2'h0  : reg_data_out <= QUPRD;
          2'h1  : reg_data_out <= QPOSLAT;
          2'h2  : reg_data_out <= slv_reg2;
          2'h3  : reg_data_out <= slv_reg3;
          default : reg_data_out <= 0;
        endcase
    end

    // Output register or memory read data
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_rdata  <= 0;
        end
      else
        begin
          // When there is a valid read address (S_AXI_ARVALID) with
          // acceptance of read address by the slave (axi_arready),
          // output the read dada
          if (slv_reg_rden)
            begin
              axi_rdata <= reg_data_out;    // register read data
            end
        end
    end

    // Add user logic here


    always @(posedge S_AXI_ACLK) begin

      vclk_count <= vclk_count + 32'b1;

      if(vclk_count == QUPRD - 1)begin
        QPOSLAT <= phase_a + phase_b;
      end

      if(vclk_count == QUPRD)begin
      phase_a <= 0;
      phase_b <= 0;
      vclk_count <= 0;
    end

        if(phase_a_flag == 1 && pwm_sig_a == 0)begin
      phase_a <= phase_a + 32'b1;
    end
```

```verilog
if(phase_a_flag == 0 && pwm_sig_a == 1)begin
   phase_a <= phase_a + 32'b1;
end

if(phase_b_flag == 1 && pwm_sig_b == 0)begin
   phase_b <= phase_b + 32'b1;
end

if(phase_b_flag == 0 && pwm_sig_b == 1)begin
   phase_b <= phase_b + 32'b1;
end

phase_a_flag <= pwm_sig_a;
phase_b_flag <= pwm_sig_b;

   end

   // User logic ends
   endmodule
```
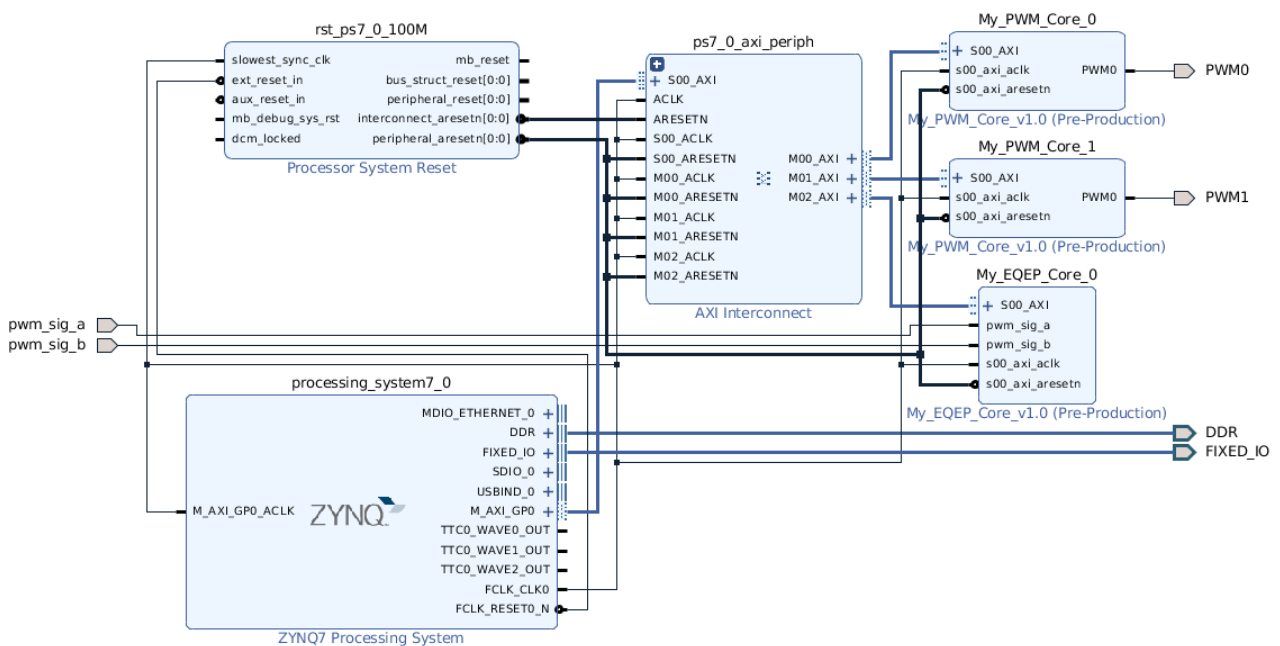
해당 IP를 Verilog로 작성한 뒤에 CUSTOP IP를 리패키지 하여 생성한다.

Open Block Design을 통하여 생성된 최종 결과물은 다음과 같다.

PWM 발생 CUSTOM IP 를 2개를 달아서 실험적으로 A상, B상 신호로 사용할 목적이다.
실제 구현을 할 때는 필요 없는 부분이므로 생략해도 무방하다.

Constraint.XDC 파일을 생성하도록 하자 Scalar Port 지정을 위하여 필요하다.

 set_property PACKAGE_PIN V12 [get_ports PWM0]
set_property IOSTANDARD LVCMOS33 [get_ports PWM0]
set_property PACKAGE_PIN W16 [get_ports PWM1]
set_property PACKAGE_PIN J15 [get_ports pwm_sig_a]
set_property PACKAGE_PIN H15 [get_ports pwm_sig_b]
set_property IOSTANDARD LVCMOS33 [get_ports PWM1]
set_property IOSTANDARD LVCMOS33 [get_ports pwm_sig_a]
set_property IOSTANDARD LVCMOS33 [get_ports pwm_sig_b]

JE 포트의 1번을 A상, 2번을 B상, 3번을 PWM_A input, 4번을 PWM_B input으로 사용하였다.

Cell	Slave Interface	Base Name	Offset Address	Range		High Address
∨ ⊕ processing_system7_0						
∨ ⊞ Data (32 address bits : 0x40000000 [1G])						
▭ My_EQEP_Core_0	S00_AXI	S00_AXI_reg	0x43C2_0000	64K	▾	0x43C2_FFFF
▭ My_PWM_Core_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K	▾	0x43C0_FFFF
▭ My_PWM_Core_1	S00_AXI	S00_AXI_reg	0x43C1_0000	64K	▾	0x43C1_FFFF

미리 각각의 코어의 베이스 어드레스를 확인하여 보자.
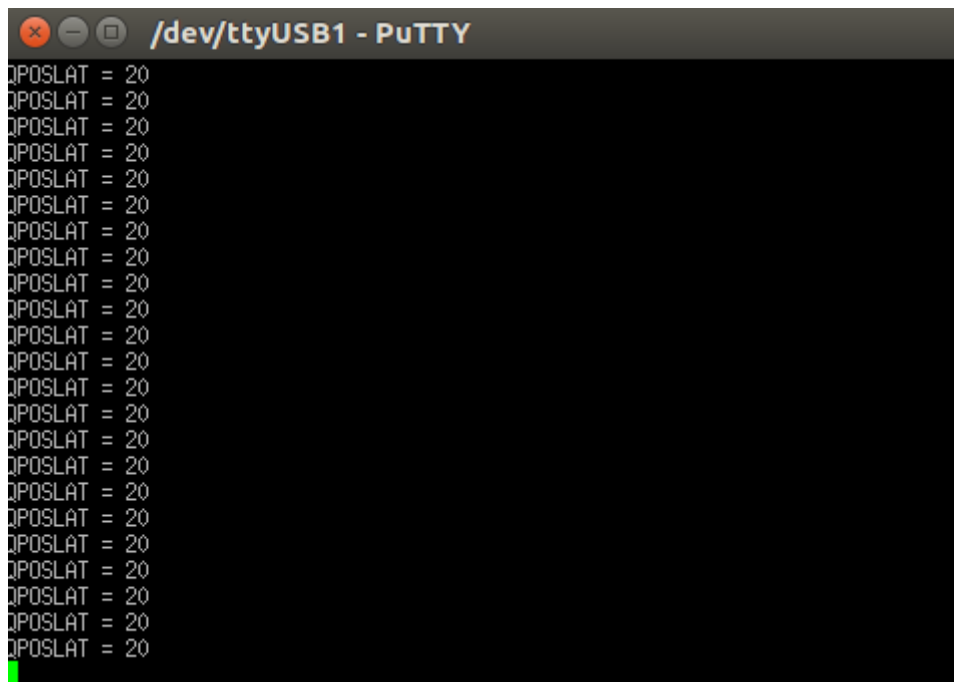
Generate Bitstream을 통하여 진행을 하고 Export Hardware을 하여 하드웨어 파일을 생성해준다.
그리고 SDK를 실행한다.

사용된 SDK의 실험 소스는 다음과 같다.

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "xparameters.h"

#define XPAR_MY_PWM_CORE_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_MY_PWM_CORE_1_S00_AXI_BASEADDR 0x43C10000
#define XPAR_MY_EQEP_CORE_0_S00_AXI_BASEADDR 0x43C20000
#define QUPRD 20000000

int main(){
      int num1 = 0;
      int i = 0;
      //int QPOSLAT=0;
      while(1){
            num1 = 1000000;
            Xil_Out32(XPAR_MY_PWM_CORE_0_S00_AXI_BASEADDR, num1);
            Xil_Out32(XPAR_MY_PWM_CORE_1_S00_AXI_BASEADDR, num1);
            Xil_Out32(XPAR_MY_EQEP_CORE_0_S00_AXI_BASEADDR, QUPRD);
            //QPOSLAT = Xil_In32(XPAR_MY_EQEP_CORE_0_S00_AXI_BASEADDR + 0x04);
            xil_printf("QPOSLAT = %d\n\r",
Xil_In32(XPAR_MY_EQEP_CORE_0_S00_AXI_BASEADDR + 0x04));
            for(i=0; i<2000000; i++);
      }
  return 0;
}
```

코어의 QUPRD를 지정하여 원하는 값을 지정된 시간으로 고정하고,
QPOSLAT의 값을 받아와서 이동한 거리의 데이터를 수신한다. TI Coretex R-5F 의 eQEP 패리페럴과 같은
동작을 수행한다.

 Putty를 이용하여 확인한 데이터의 값이다. 20ms의 주기를 가진 PWM 신호 A, B상을 eQEP 패리패럴에 넣었다. 0.2 초 동안에 확인한 결과 값을 출력하고 있다.

200ms 동안 20ms 주기는 10번 반복되고, 한 개의 신호당 2개의 edge를 검출이 가능하다. 따라서 두 신호가 물리면 40번, 한 신호만 측정 되면 20번이라는 값이 정확한 결과이다.

다음 실험 과제는 저속모드를 개발하고 페타 리눅스에 올릴 생각입니다.