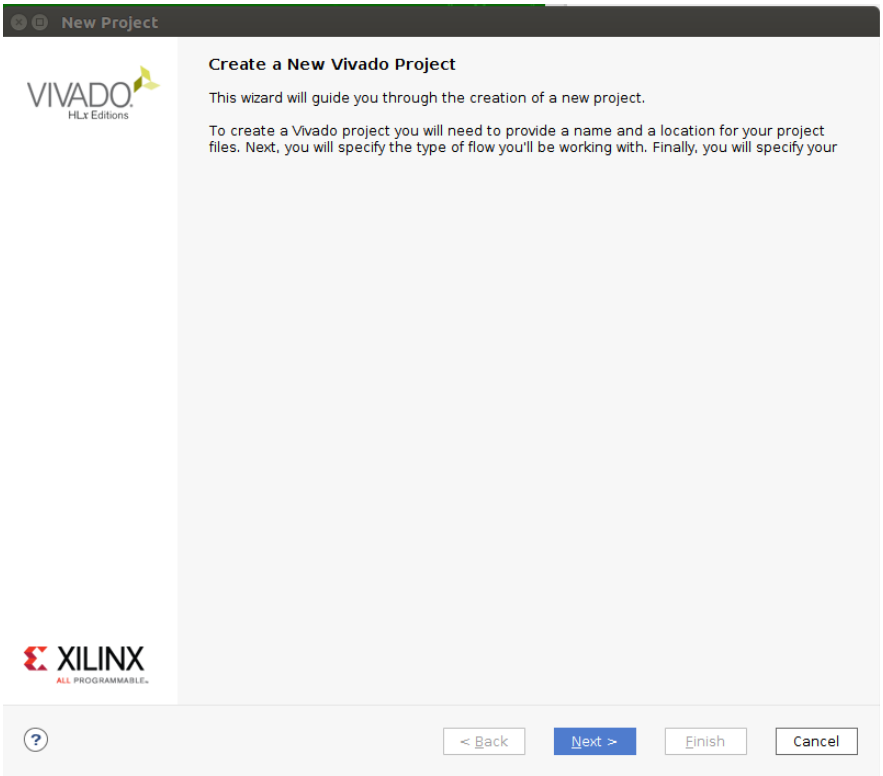
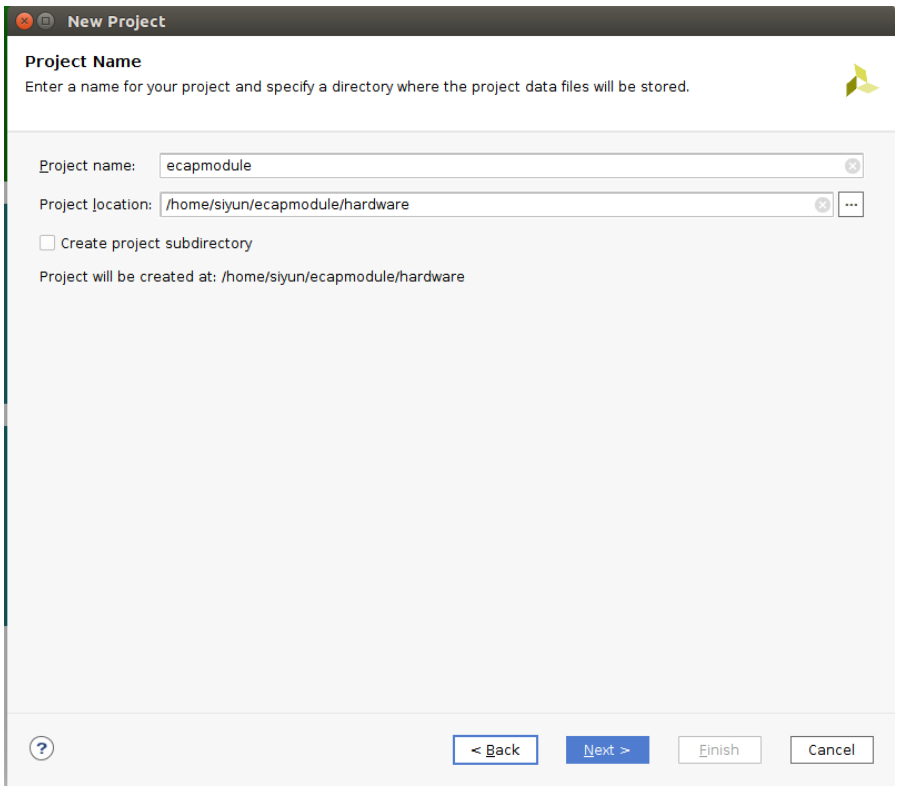
Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

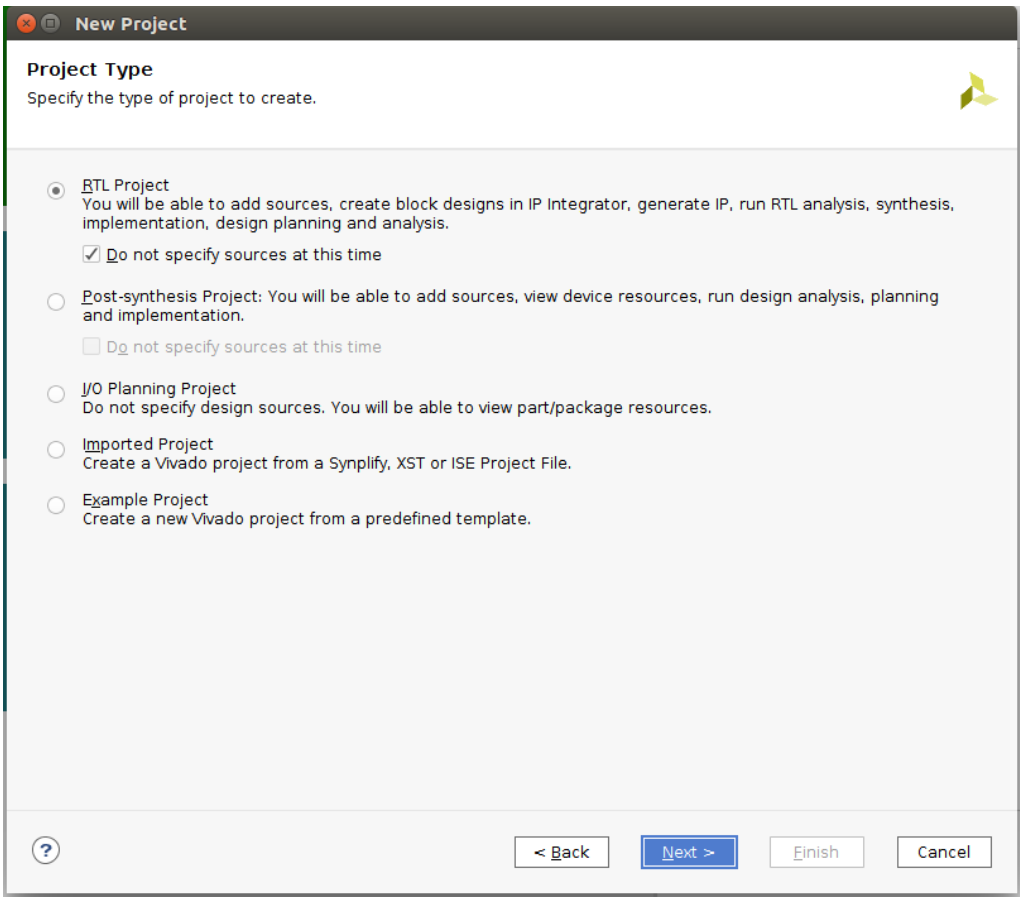
Make eCAP custom IP

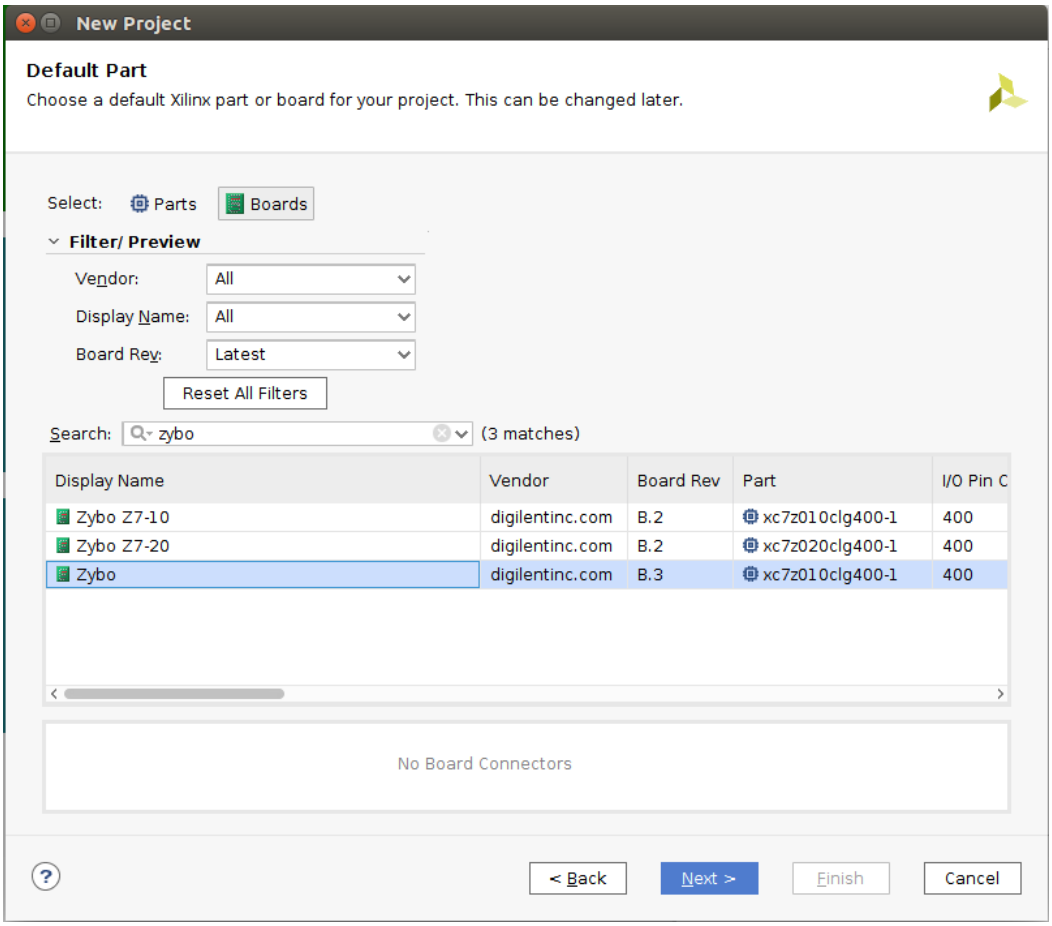
강사 : Innova Lee(이 상훈)
학생 : 김 시윤

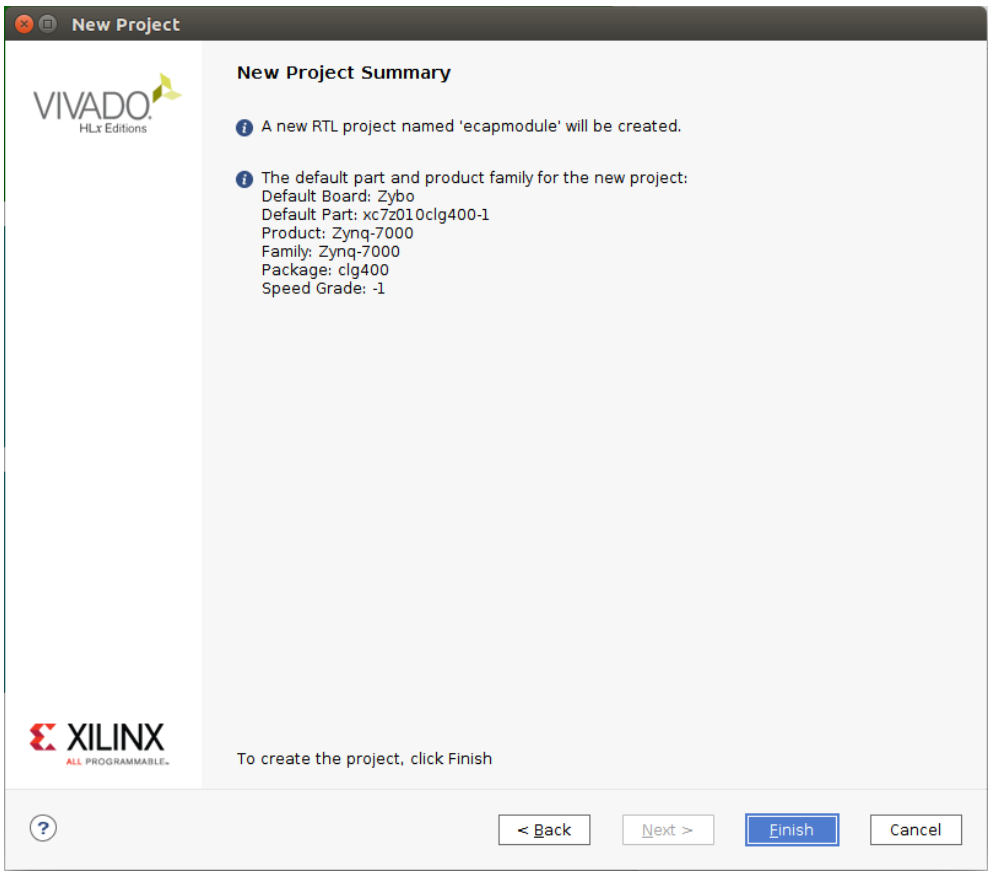
1. Vuvadi 를 실행시켜 프로젝트를 하나 생성한다.





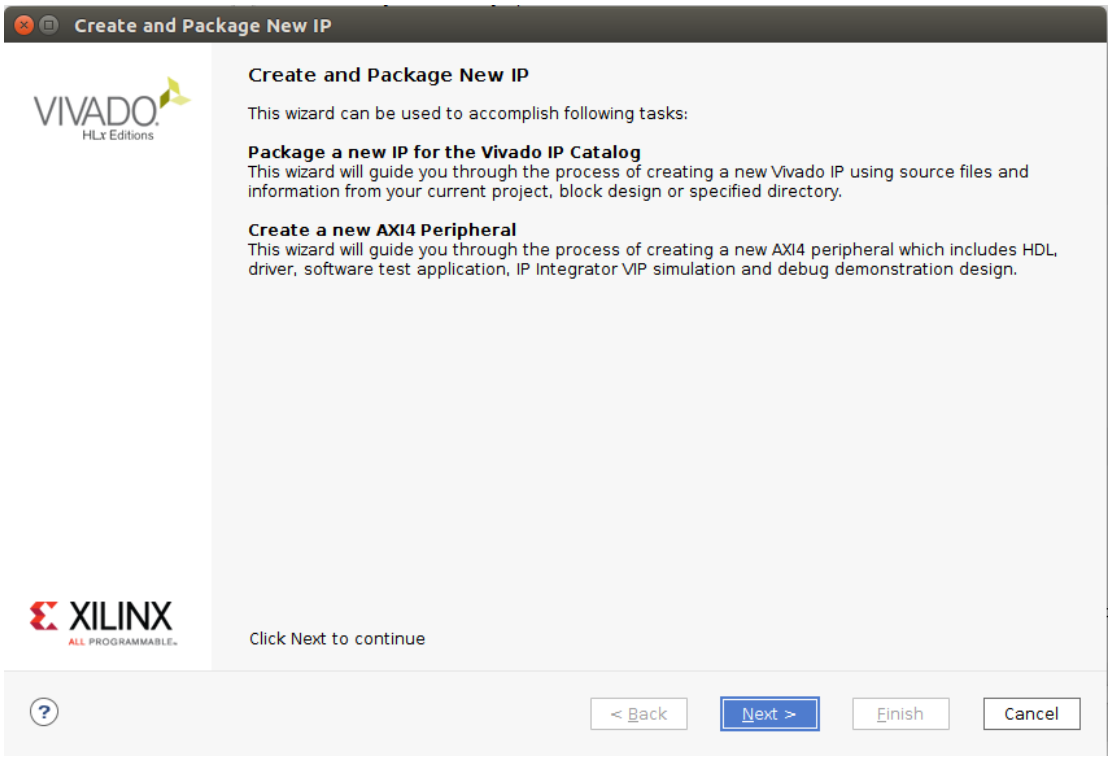

New Project
Project Type
Specify the type of project to create.
RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
Do not specify sources at this time
Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.
Do not specify sources at this time
I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.
Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.
Example Project
Create a new Vivado project from a predefined template.
< Back
Next >
Finish
Cancel


New Project
Default Part
Choose a default Xilinx part or board for your project. This can be changed later.
Select:
Parts
Boards
Filter/ Preview
Vendor:
All
Display Name:
All
Board Rev:
Latest
Reset All Filters
Search:
zybo
(3 matches)
Display Name
Vendor
Board Rev
Part
I/O Pin C
Zybo Z7-10
digilentinc.com
B.2
xc7z010clg400-1
400
Zybo Z7-20
digilentinc.com
B.2
xc7z020clg400-1
400
Zybo
digilentinc.com
B.3
xc7z010clg400-1
400
No Board Connectors
< Back
Next >
Finish
Cancel

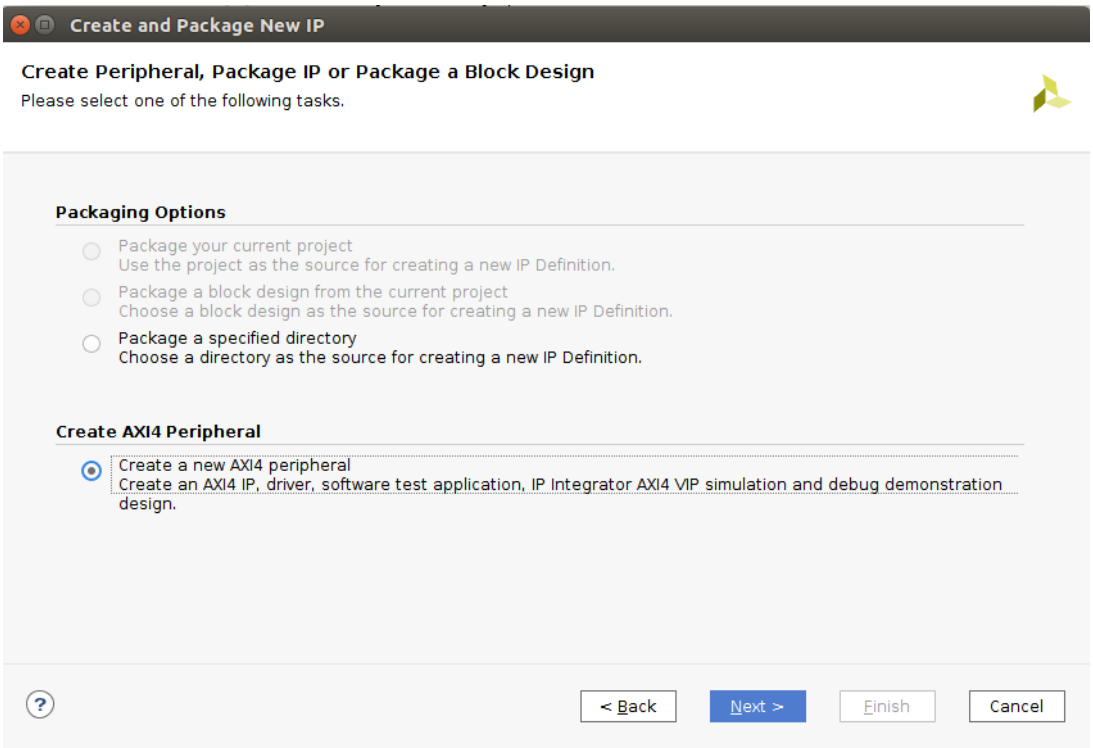


프로젝트 생성이 완료되었으면 Vivado 상단에 Tools → Create and Package New IP

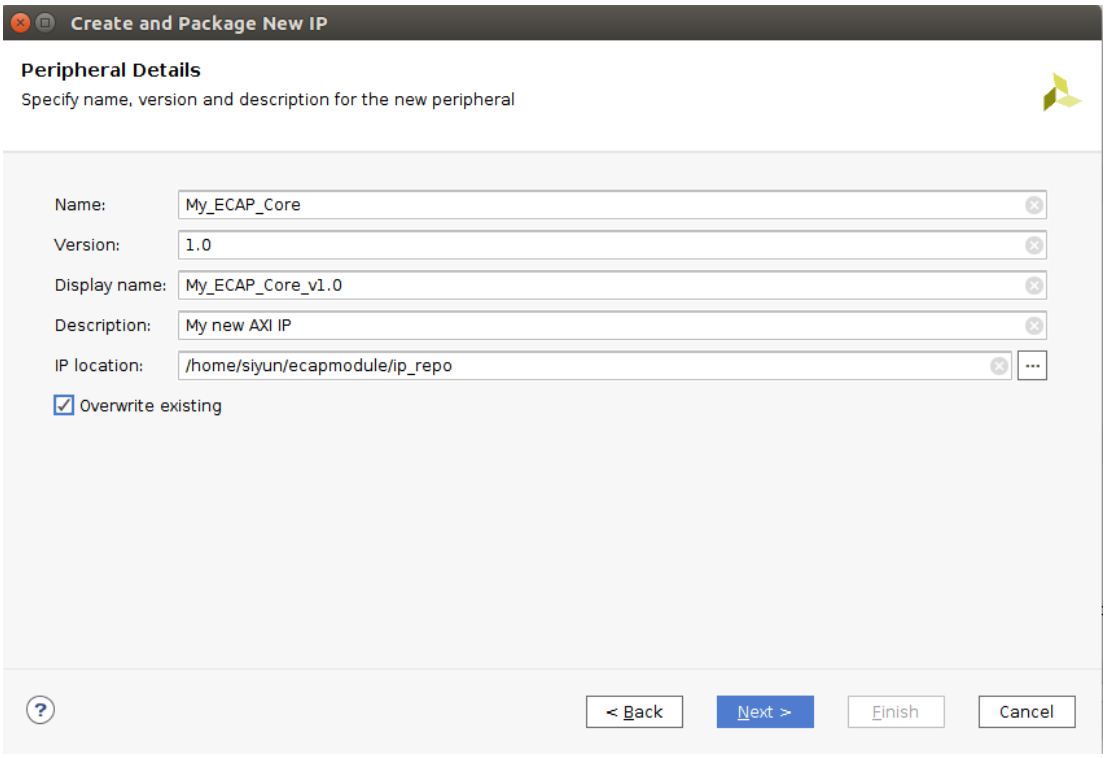
Next



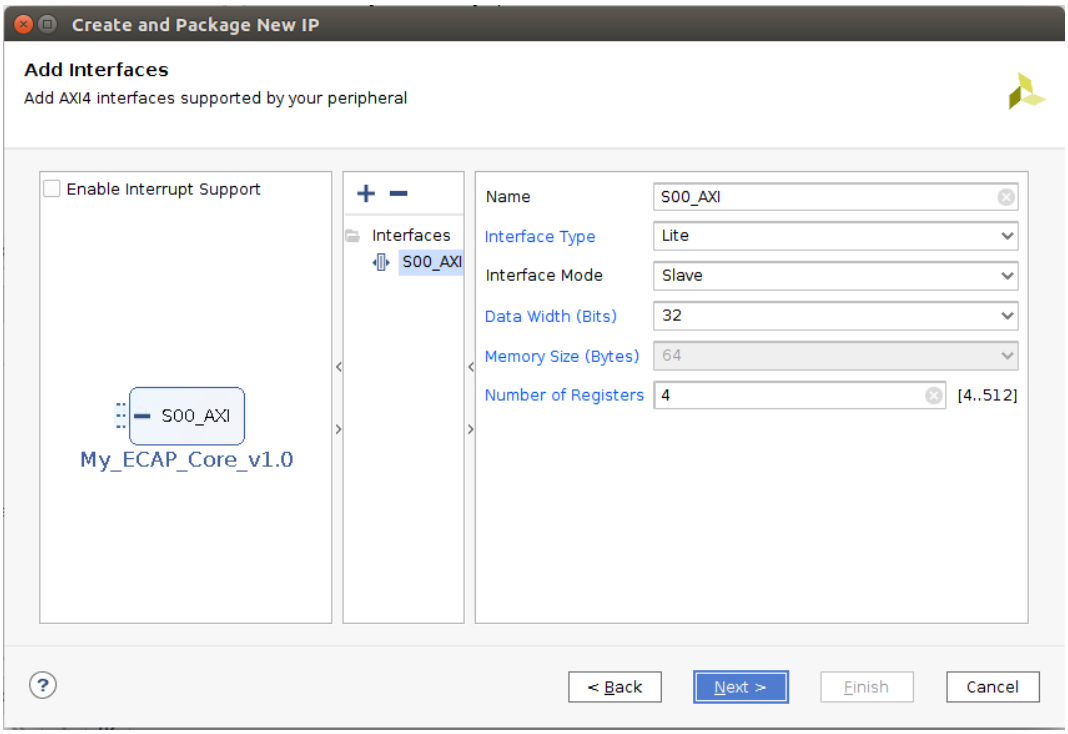
Create a new AXI4 peripheral 선택 후 Next



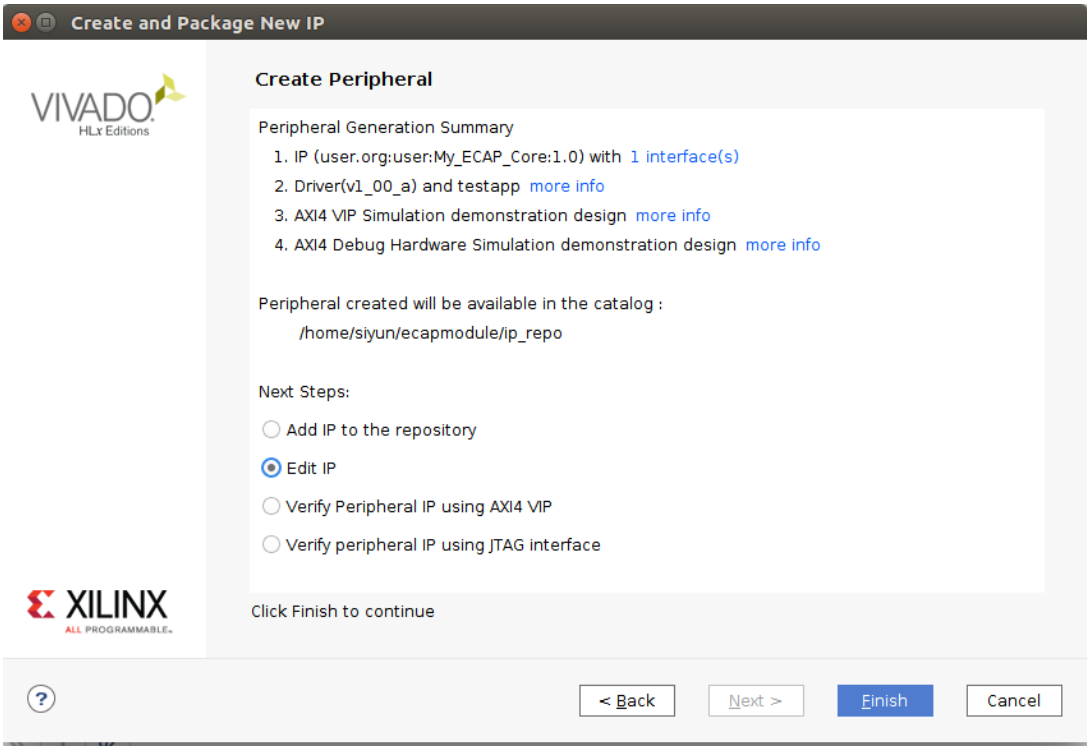
이름설정.
Name 에 My_ECAP_Core 입력 후 Next



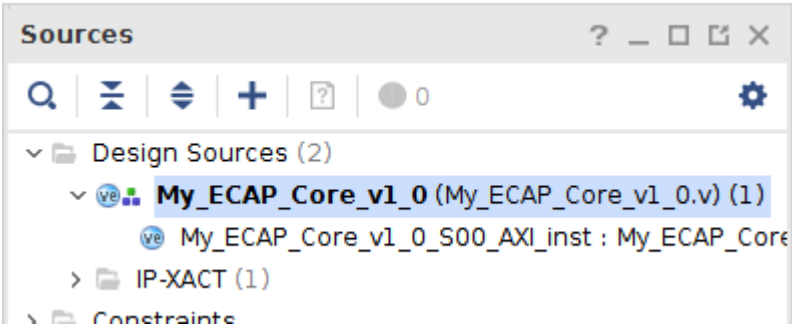
아무 설정 하지 않고 Next



Edit IP 선택 후 Finish



Finish 하고 나면 좌측 상단에 베릴로그 파일이 2 개 생성된다.
매번 말하듯이 위는 탑모듈 아래는 하위모듈이다.
우선 하위모듈부터 수정하도록 한다.



My_ECAP_Core_v1_0_SOO_AXI_inst.v

```
`timescale 1 ns / 1 ps

	module My_ECAP_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,
		// 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
		// 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
	// 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;

	// 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
reg [31:0] duty;
reg [31:0] period;
reg [31:0] duty_e;
reg [31:0] period_e;
reg [31:0] cap1;
reg [31:0] cap2;
reg [31:0] cap3;
reg past_pwm_sig;
reg [3:0]rising_edge_flag;
reg [31:0] counter;
	// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
	// 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
    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;
    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
                slv_reg0[(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 <= slv_reg0;
                      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
// 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
          // indicates that the slave has acceped 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
          case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
            2'h0   : reg_data_out <= period;
            2'h1   : reg_data_out <= duty;
            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
  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

if(pwm_sig == 1 && past_pwm_sig ==0) begin
    rising_edge_flag = rising_edge_flag + 4'd1;

    if(rising_edge_flag == 4'd1) begin
            cap1<=counter;
    end

    else if(rising_edge_flag == 4'd2) begin
          cap3<=counter;



          duty_e <= cap2 - cap1;
          duty <= duty_e;

          if(cap3 > cap1) begin

          period_e <= cap3 - cap1;
          end
          else if(cap1 > cap3) begin

          period_e <= cap1 - cap3;
          end

          period <= period_e;


          counter <= 32'd0;
        rising_edge_flag <= 4'd0;
    end

end

else if(pwm_sig == 0 && past_pwm_sig ==1) begin
            cap2<=counter;
end
    past_pwm_sig <= pwm_sig;
    counter <= counter + 32'd1;

end
          // User logic ends

          endmodule
```

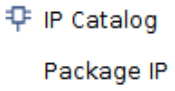
작성이 완료되었으면 탑모듈을 작성하도록 한다.

My_ECAP_Core_v1_0.v

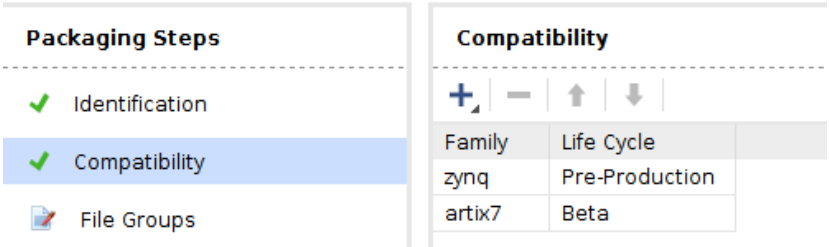
```
`timescale 1 ns / 1 ps

	module My_ECAP_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,
		// 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,
		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_ECAP_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_ECAP_Core_v1_0_S00_AXI_inst (
	   .pwm_sig(pwm_sig),
		.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
```

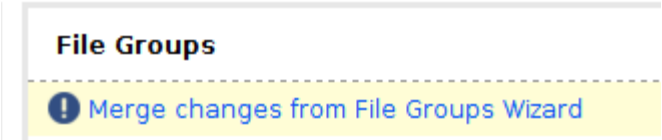
위 코드가 모두 작성되었다면 Package IP 를 클릭한다.



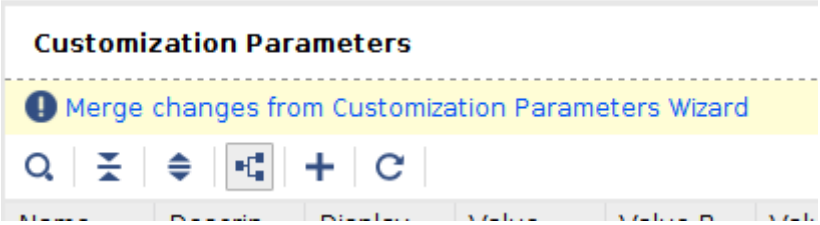
이 전처럼 Compatibility 에서 artix7 을 추가한다.



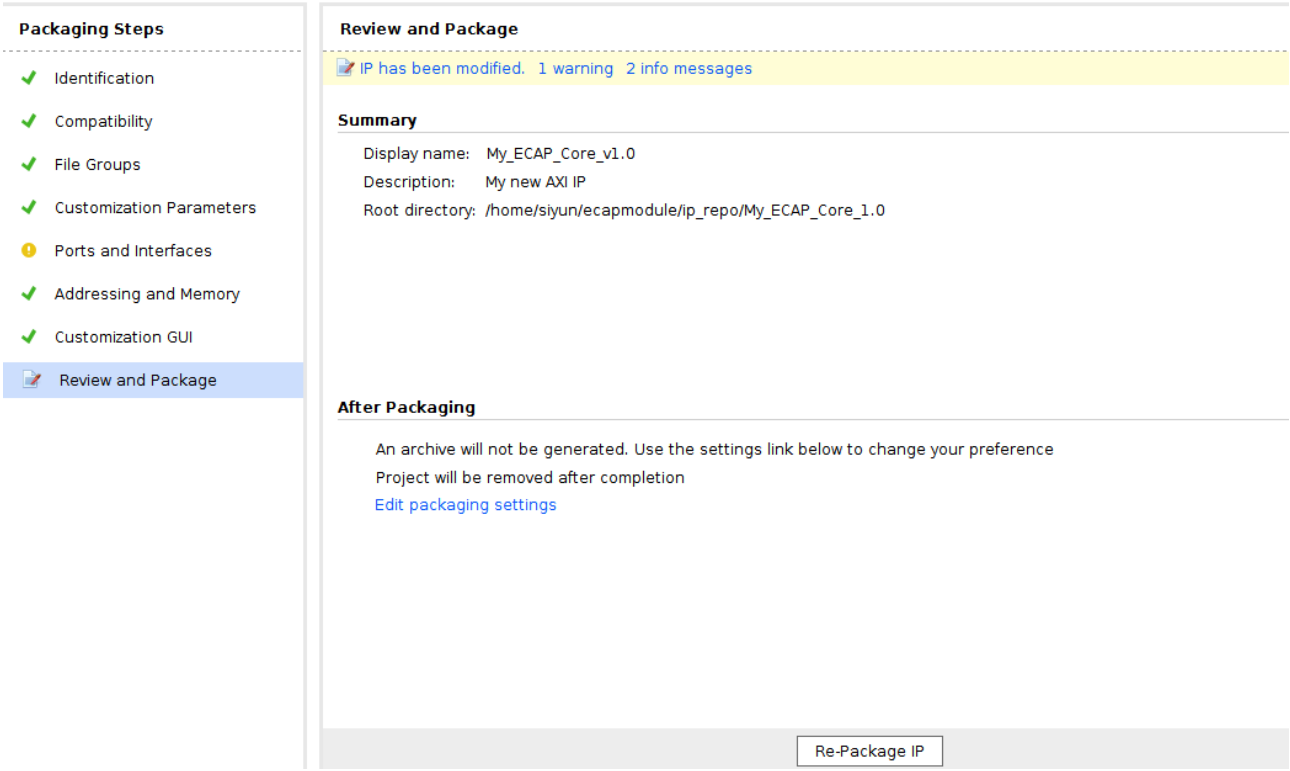
File Groups 단계에서 Merge Chages form File Groups Wizard 를 클릭한다.



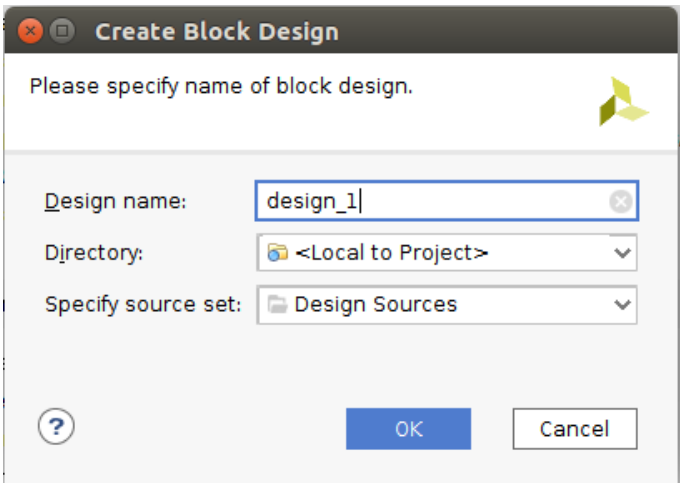
Customization Parameters 도 마찬가지로 Merge changes from Customization Parameters Wizard 를 클릭한다.
우리가 추가한 파라미터가 없기 때문에 따로 설정해 줄건 없다.



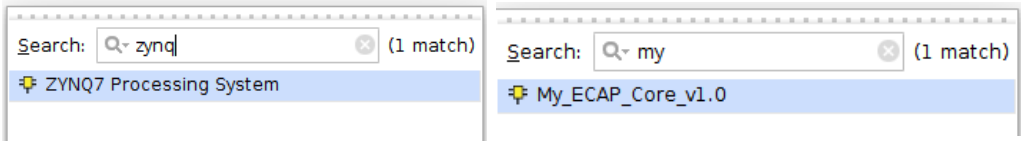
마지막 Review and package 에서 Re-Package IP 를 클릭하여 IP 를 생성해 준다.

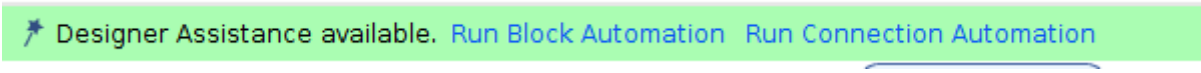


다시 프로젝트로 넘어와 Create Block Design 을 해준다.

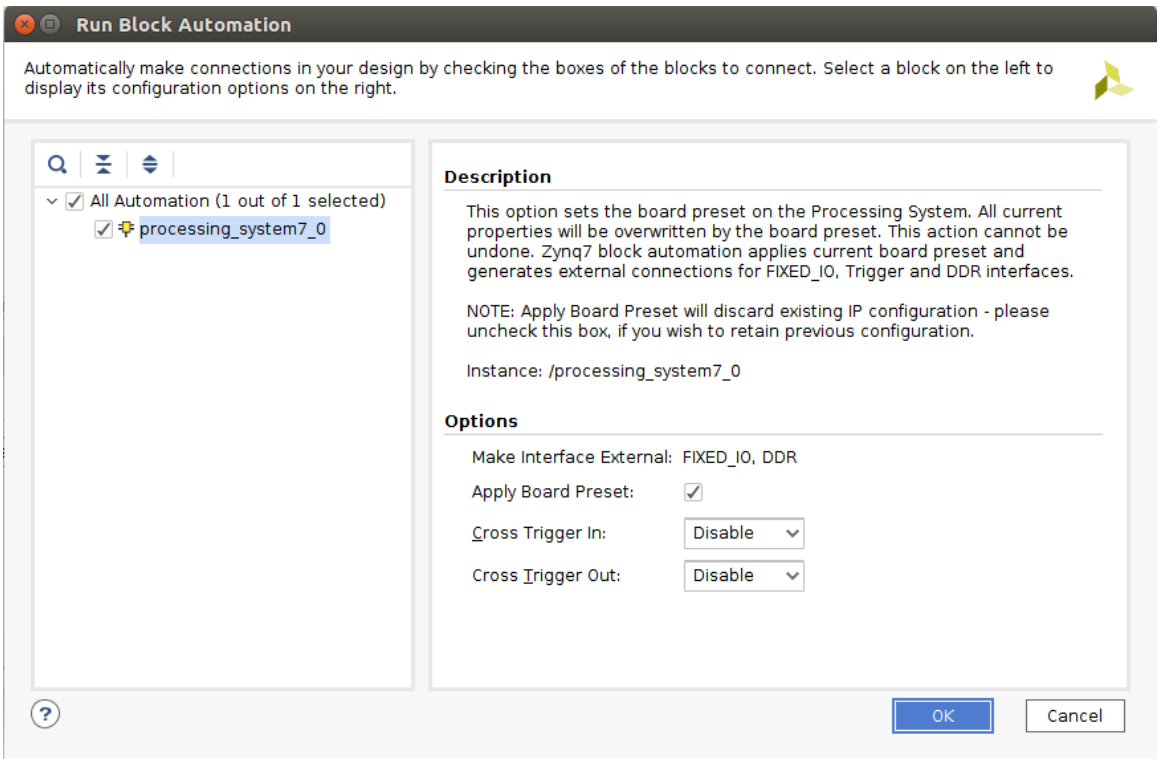


그리고 zynq 프로세서랑 새로만든 Custom IP 를 추가시킨다.

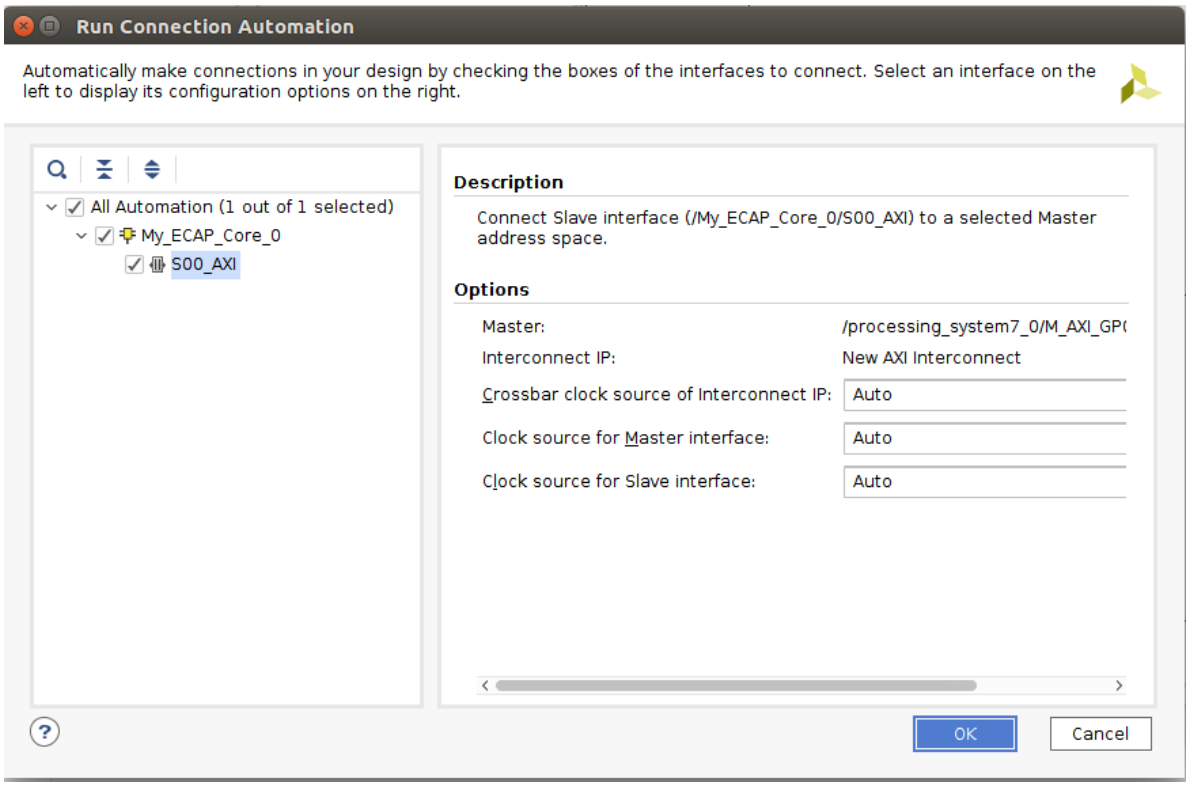




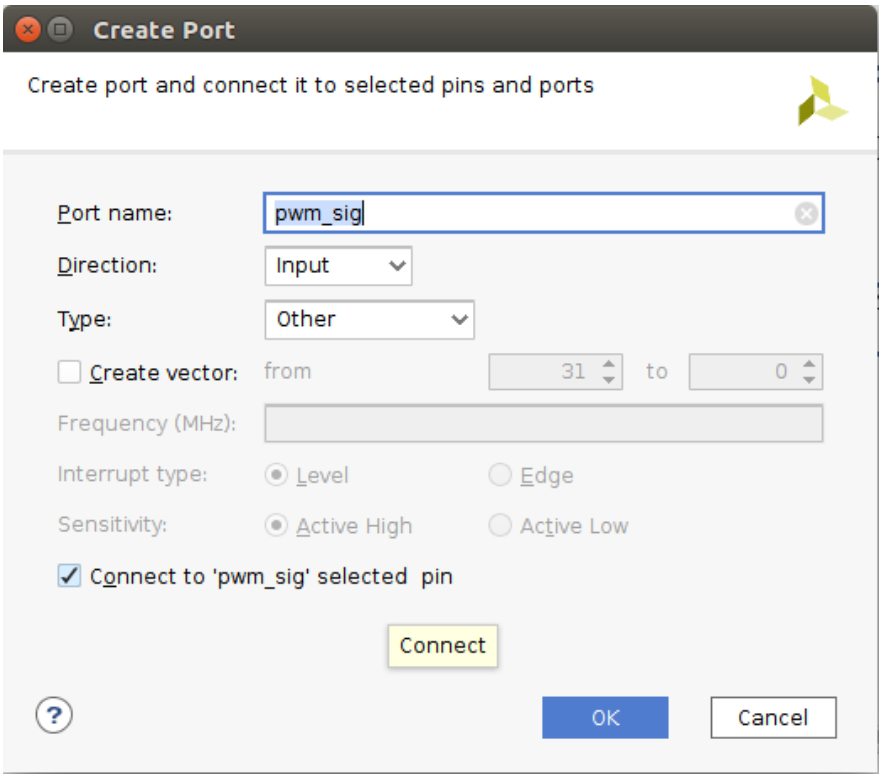
먼저 Run Block Automation 을 아래와 같이 해준다



완료 후 Run Connetion Automation 도 아래와 같이 해준다.



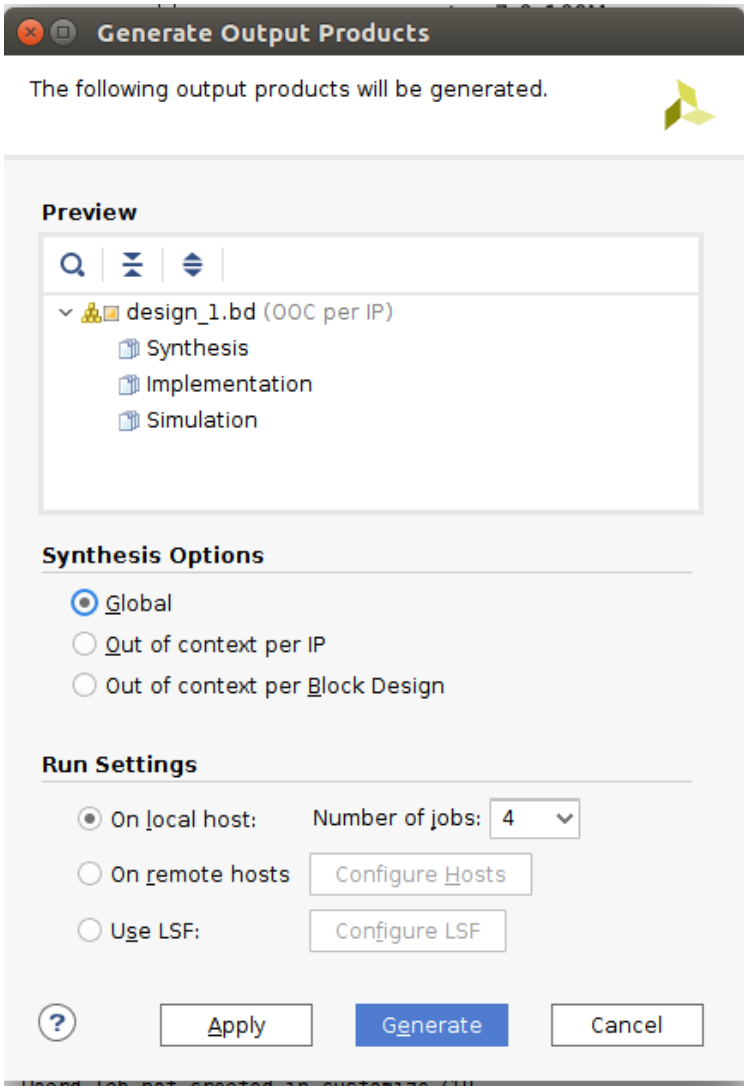
위 과정을 완료하면 우리가 만든 ECAP custom IP 에 있는 pwm_sig 를 마우스 우 클릭 하여 create port 를 해준다.



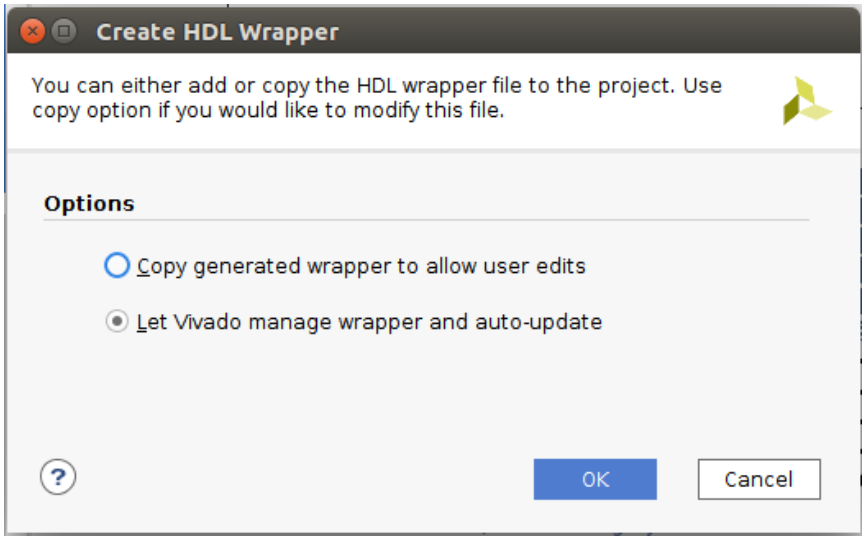
위 과정이 완료되면 회로를 이쁘게 볼수 있게 해줄 새로고침 모양의 버튼 을 눌러준다

완료되면 회로에 오류가 있는지 ← 버튼을 눌러주어 확인한다.

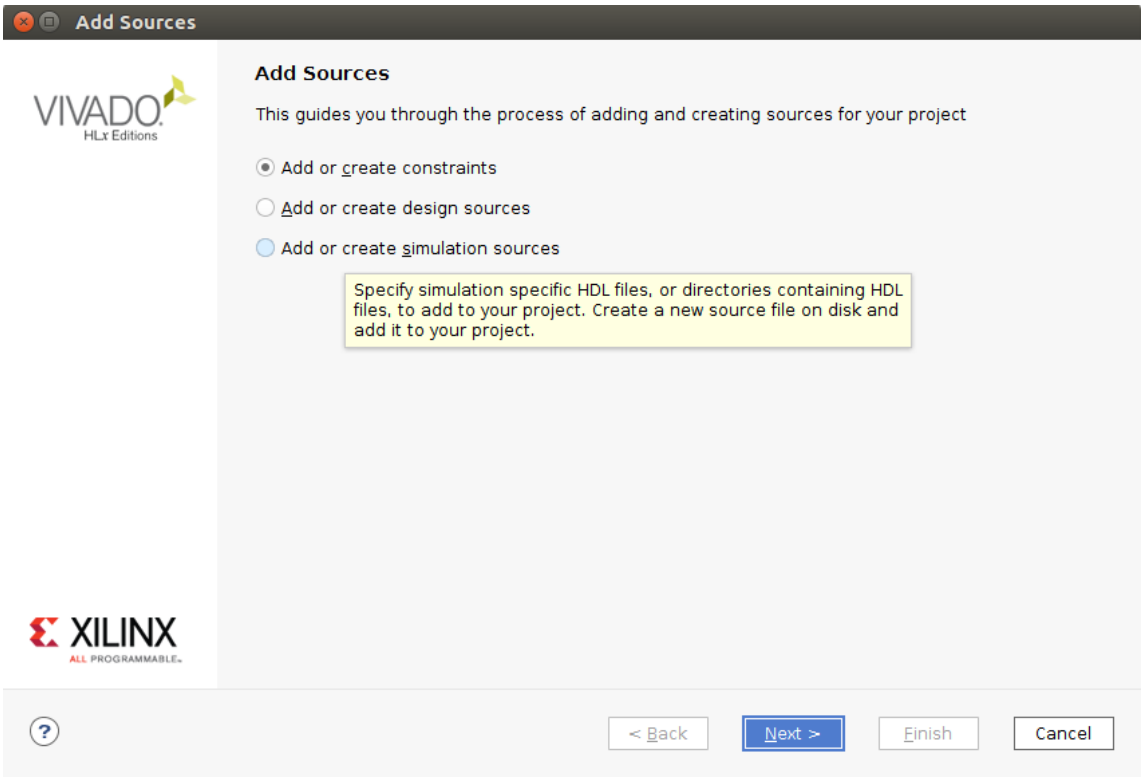
오류가 없는게 확인되면 Source 로 가서 Design_1.bd 를 우클릭 한 후 Generate Output Products 를 아래와 같이 해준다.

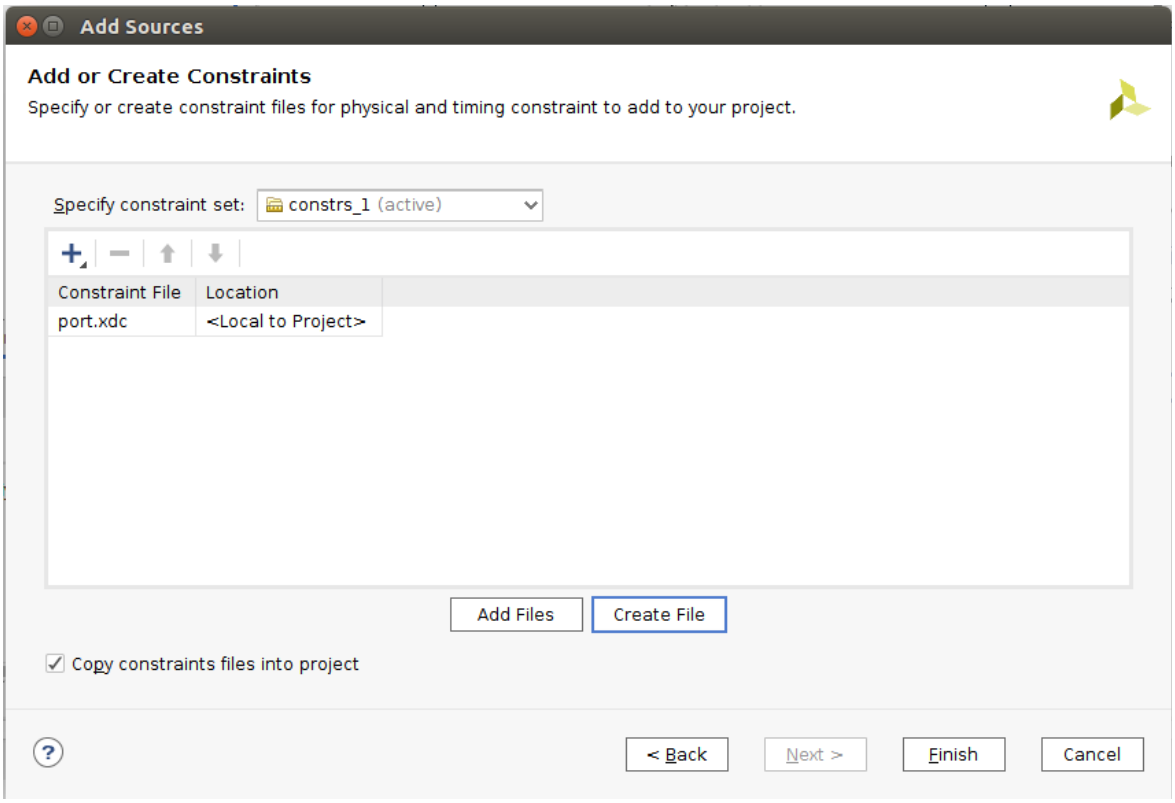


그 후 다시 우클릭하여 Create HDL Wrapper 을 해준다.

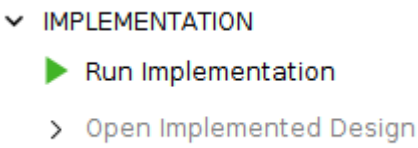


이제 Implement 를 하기전에 Constraints 를 생성한다.

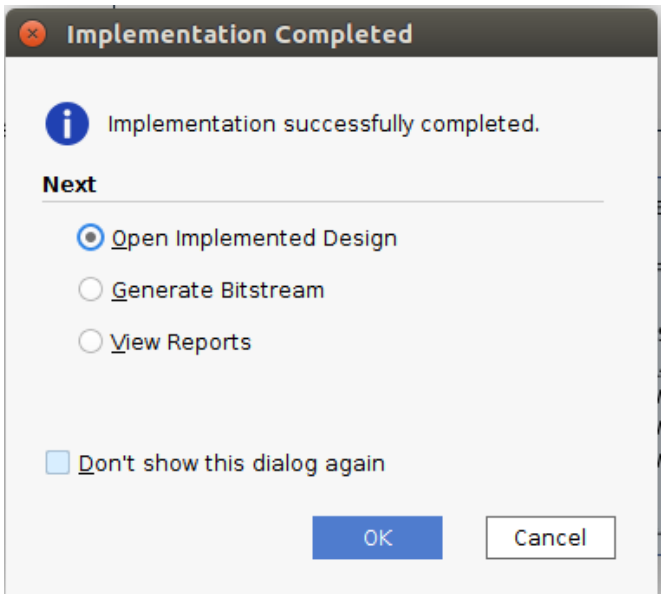


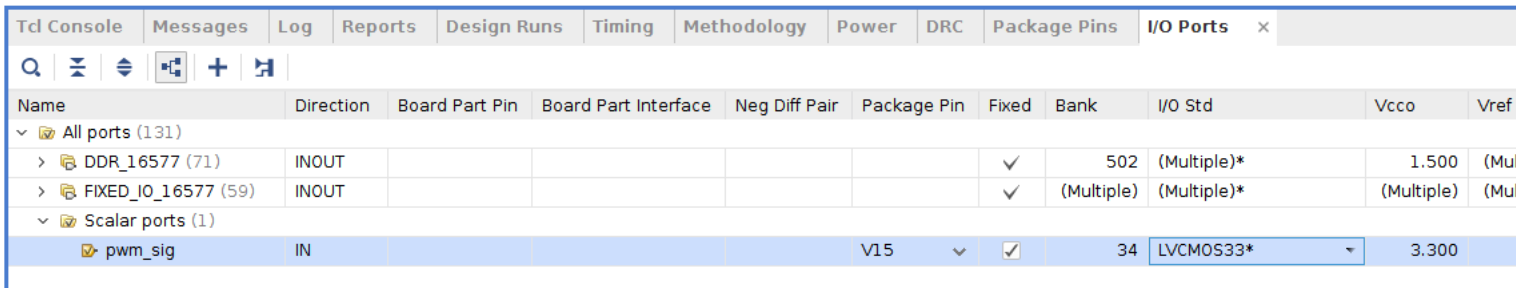


생성 완료 후 Implementation 을 해준다.



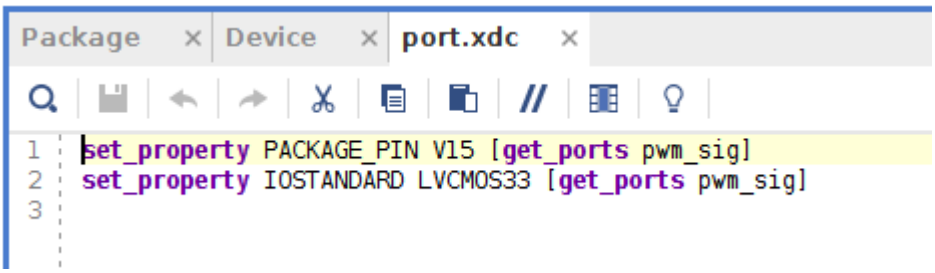
완료 되면 임플리먼트 디자인을 오픈하여 포트 설정을 아래 사진과 같이 해준다.



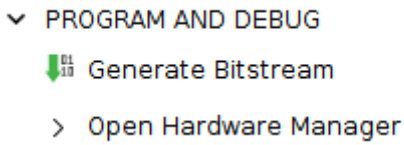


나는 pwm 파형의 입력을 V15 핀에 받을것이다.

저장(ctrl+s)를 해주고 xdc 에 포트정보가 로드가 됐는지 확인한다. 만약 로드 되지 않았다면 아래와 같이 써주면 된다.

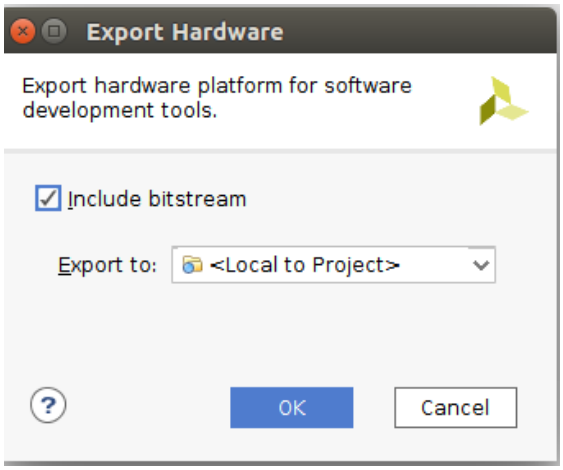


위 과정이 완료되면 Generate Bitstream 을 해준다.

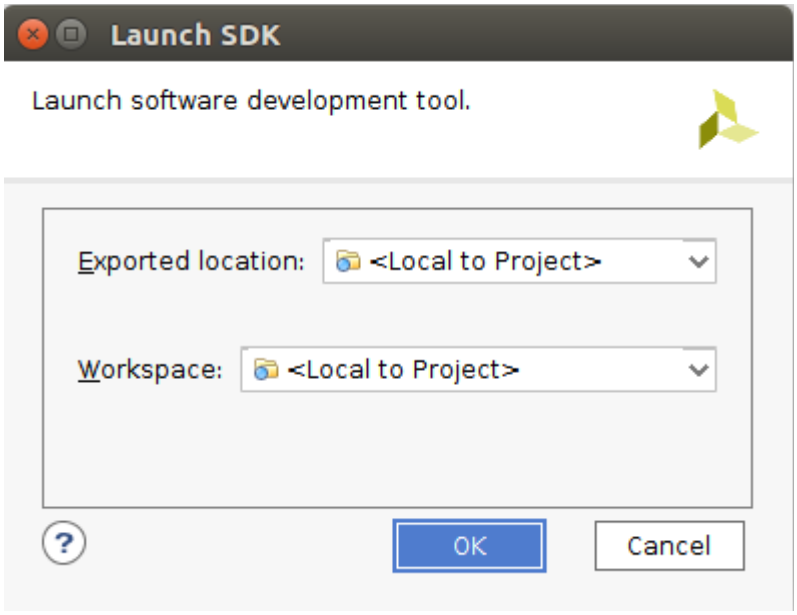


비트스트림이 완료되면 하드웨어 설계가 모두 완료되었다.
하드웨어 정보를 Export 해준다

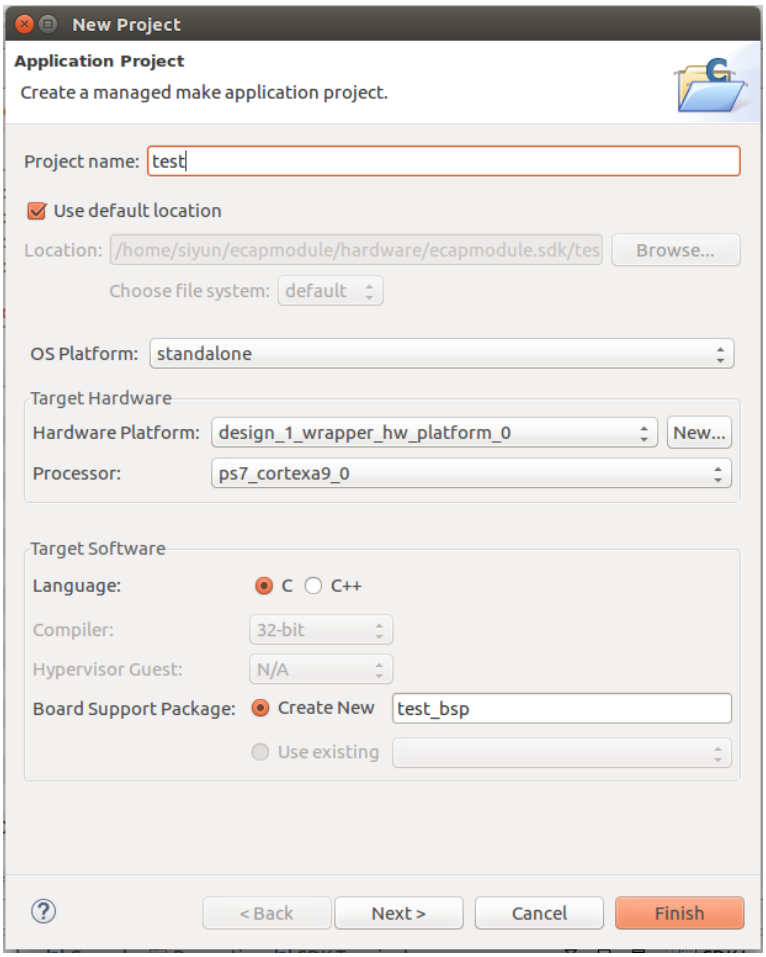
File → Export → Export Hardware



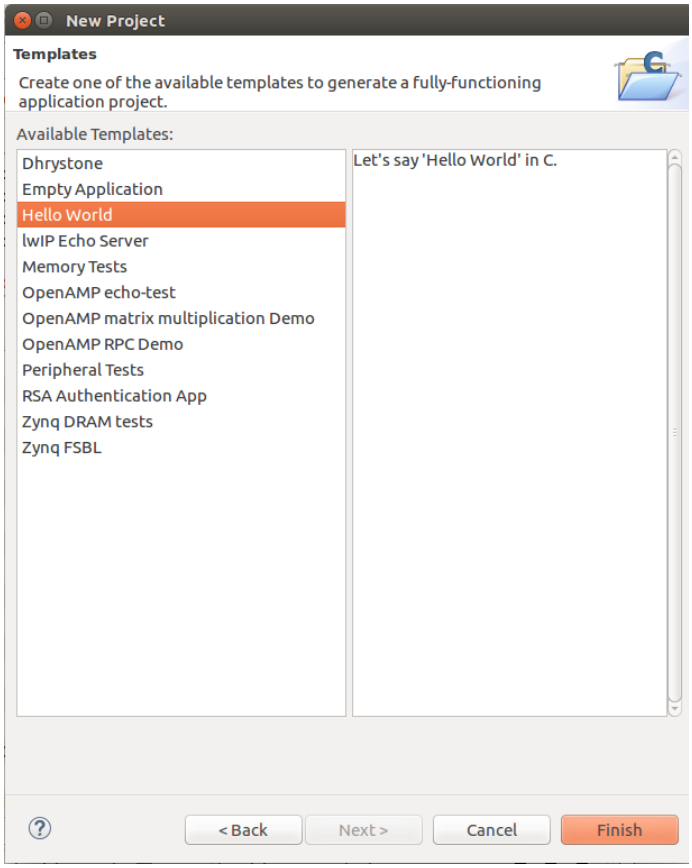
인크루드 박스는 무조건 체크!!!!!!
export 가 완료되었으면 SDK 를 켜주어 설계한 디바이스가 작동하는지 소스코드를 작성하여 확인한다.



New → Aplication Project 를 선택하여 프로젝트 이름을 작성하고 Next 를 클릭한다.



디바이스에 관련한 아무정보를 알 수 없을 때, 프로젝트는 Hellow World 로 만들어준다. 기본적인 플랫폼을 생성해준다.



Src 안에 HelloWorld.c 에 아래와 같은 소스코드를 작성한다.

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

Xuint32 *addr = (Xuint32 *)XPAR_MY_ECAP_CORE_0_S00_AXI_BASEADDR;



int main()
{
    init_platform();

    while(1){
    xil_printf("\n\r ECAP_TEST \n\r");

    xil_printf("period = %d \r\n",*(addr+0));
    xil_printf("duty = %d \r\n",*(addr+1));
    }

    return 0;
}
```

그 후 build 와 program 을 완료 한 후 test 프로젝트 우클릭 Run As → Launch On Hardware (GDB) 클릭하면 실행이 된다.

푸티를 실행시켜 확인하면 다음과 같은 결과물이 나온다..

입력신호는 PWM 주기 : 20ms , 듀티 = 50% 의 MCU PWM 이다

```
duty = 999913

 ECAP_TEST
period = 1999825
duty = 999913

 ECAP_TEST
period = 1999825
duty = 999913

 ECAP_TEST
period = 1999825
duty = 999913

 ECAP_TEST
period = 1999825
duty = 999913

 ECAP_TEST
period = 1999825
duty = 999913

 ECAP_TEST
period = 1999825
duty = 999913
```

Zybo 클럭 주파수가 100MHz 를 상기하고 결과물을 보면 약간의 오차가 있지만 맞는 결과물이 나온다.
(FPGA PWM 신호를 입력하면 되게 정확한 값이 나옴)