# Why new protocolks were added not to system but to user layer

Original project implements all protocols at the system layer. This is best solution from architecture point of view because user no need think about any system purposes. Everything is hidden in the low level modules. Also that modules are located in the system folders. When any new board is added, it will use that files.

But of course, we must pay for this architect. The payment is using of multiplexers. There are many AXI Multiplexers are present in the project for choose the way of data from layer to layer. That multiplexers are producing many wires. And wires are worst item for Lattice FPGA.

Lattice FPGA has no good solutions for inter-block communication. Most connections are going through trace resources. That is why router has much problems to optimize FMax. AS a result, any change of design can significantly changes FMax. Strange, but sometimes adding of extra wires improves FMax to 10..20 MHz. Sometimes remove of any modules drops FMax. That is wwhy we can say that FMax depends not only from Netlist, but also from router’s quality. And less wires => easier processing for router.

That is why if we can skip multiplexer for Lattice FPGA, better to do it. And that is wy we added all protocols to User Layer.

For now , all support files are also located in the Top Level Folder.Maybe later will be better to move them to any common place because now we must synchronize that files when any changes were added. From other hand, now we are providing many experiments with different boards and sometimes that differences are important. But in the future, all versions of the protocol support modules will be the same.

# Which buses we have at the User Layer

The User Layer has two global buses (x2 directions = 4 totally) for provide the functionality.

**rx\_udp\_.**.. This bus will be activated when detected UDP Datagram is came from the network. The receiver is processing datagrams in the promiscuous mode but it contains all addressing information such as Target IP, Target MAC and Target Port. User layer’s logic can use or ignore it.

**tx\_udp\_**.. This bus can be used for send UDP Datagram from User Layer to the network.

**rx\_ip\_**... All Ethernet Frames which were not served at Low Level and not detected as UDP Datagrams will be routed to this bus. We are using it for provide ICMP Protocol

**tx\_ip\_**... If we need send any non-UDP Packet, we must do it, using this bus

# Project’s items

Now project contains 4 subprojects.

1. Overclocked version for ColorLight\_5A-75B board. Uses Ethernet in Gigabit Mode.
2. 100M version for ColorLight\_5A-75B board. System Bus has CLK= 104.167 MHz. Why not 100M? This value is got from the PLL Generator. We ordered primary clock 125 MHz because this is most important for RGMII. Of course, we ordered Secondary Frequency 100 MHz but got this value. Also Chinese Oscilloscope and Chinese Logic Analyzer shows that duty cycle of PLL Output is not 50%. Maybe this is problem of Chinese equipment at high frequency, maybe not.
3. Overclocked version for ColorLight\_i5 Board (DIMM Version). Uses Ethernet in Gigabit Mode.
4. 100M version for ColorLight\_i5 Board (DIMM Version) with System Clock 104.167 MHz

Why we still using two boards? The most problem of current DIMM solution is motherboard. We don’t see any input pins. Also we don’t know which FPGA pin is connected to the button on that motherboard. But we need Input pins at least for force DHCP or AutoIP processing.

For ColorLight\_5A-75B, we know which pin is connected to the button. That button is single. We cannot test DHCP and AutoIP together. But one is more than nothing! That is why we must use ColorLight\_5A-75B too.

# Project’s top hierarchy



The Top Level Module has name **fpga**. It is located in file **fpga.v**. This tradition is came from the original project. This file contains all FPGA Based items such as PLL, and some non-standard pins, if needed. That pins are needed for debug purposes, but not for real production. Also traditionally, this module contains LED Blinker because it also depends of hardware depended frequency divider. This is last module in ther hierarchy, which contains any physically depended information.

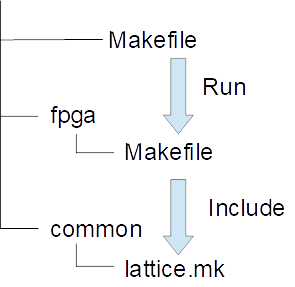
Next layer has name **fpga\_core**. This module is also came from the original project. Exacxtly this module is used for automatic tests in the original project. It has unified interface. This interface is same for Xilinx, Altera, Lattice etc.

The module **fpga\_logic** is added at this project. We separated it from fpga core for simplify simulation process in the ModelSim. This module receives Ethernet data already packed into AXI-Stream format. That is why we can use files, captured by WireShark without translate them into any more low level fromat and without thinking about any low level CRS. The mail FSM is located in this file.

# Project’s metafiles

## Makefile

The project comprises a structure of Make files (see structure below). Main Make file located in a main folder of every example. This Make file comprises general project information and runs more specific Makefile located in **fpga** folder.



fpga/Makefile specifies:

* list of files to be processed by Yosys
* Paramters for Nextpnr
* Trellis file location
* Command downloading firmware to FPGA

The most important settings:

FPGA\_TOP – name of target file.

SYN\_FILES – list of files to be processed by Yosys.

NEXTPNR\_PARAMS – Parameters for Nextpnr, including:

* type of package,
* lpf-file describing the FPGA’s pinout
* log-file, where information on place and route process
* times – this is not Nextpnr parameter, it is intended to be provided to common/place\_and\_route.py script (described below) and defines the number of Nextpnr passes with different seeds.

fpga/Makefile includes common/lattice.mk, which comprises rules for targets all, and clean. Also by target $(FPGA\_TOP).ys it specifies process of creation of script for Yosys. This script comprises commands and list of files to be processed by Yosys.

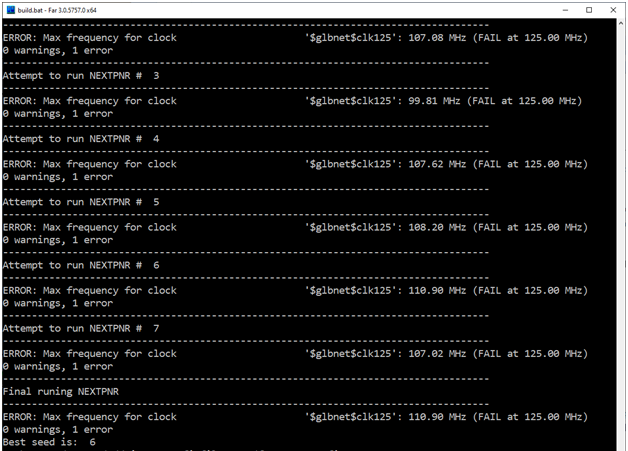
Among other things, lattice.mk runs common/place\_and\_route.py script, which is described below.

## NextPNR Script

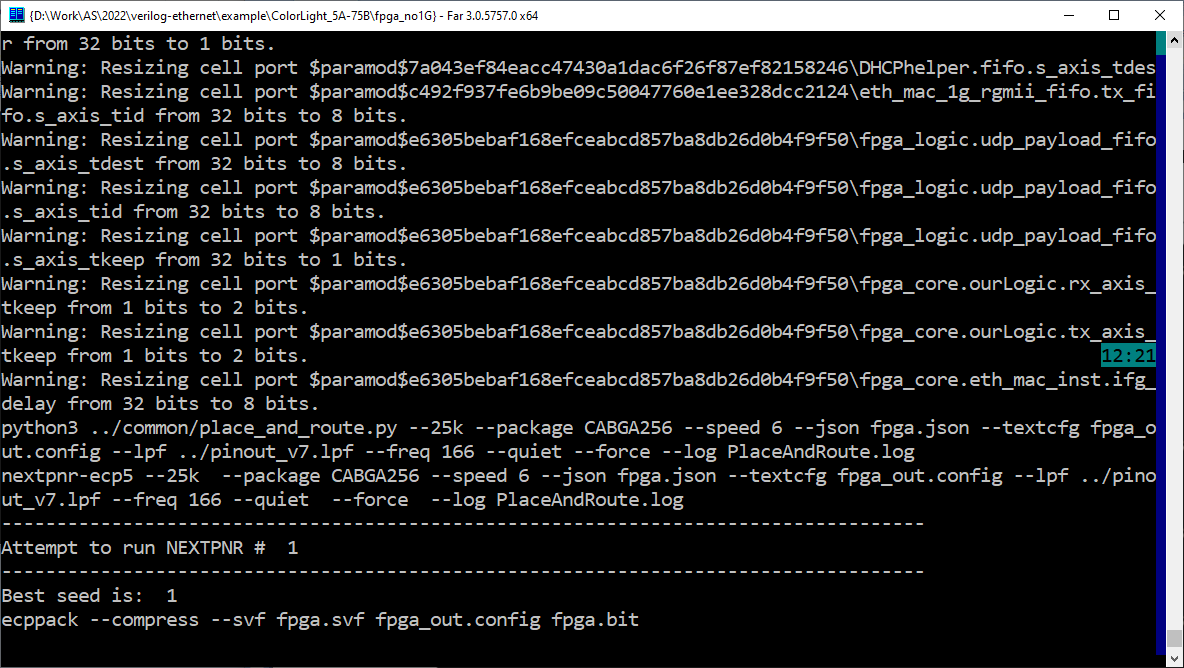
According to our investigations, FMax may differ for different seed parameter of Nextpnr. This difference may be significant (10 to 20 MHz) that is why we have implemented a python script, which runs Nextpnr several times with different seed values and chooses the best one.

The lattice.mk runs common/place\_and\_route.py script and provides it with parameters for Nextpnr and **–times** parameter, which defines number of times Nextpnr will be run with different seeds (7 times by default).

The script chooses the best seed by analyses of Nextpnr’s log-file (can be set by NEXTPNR\_PARAMS in fpga/Makefile or is equal to Logfile.log by default) based on parsing errors related to FMax values.



If FMax is reached, then process will be automatically stopped



# Protocol Based Modules

## Module SpiFlashReader

This module is used for give access to SPI Flash. The logic of this module is unusual for the project. Most modules are waiting in idle state most time. This module drops to reset state when all strobes are inactive. It is made for release SPI Bus immediately.

if (rst || ((read\_strobe == 0) && (write\_strobe==0)&& (erase\_strobe==0)))

begin

state <= idle;

spi\_cs <= 1;

spi\_sck <= 0;

spi\_mosi <= 0;

m\_tvalid <= 0;

s\_tready <= 0;

finished <= 0;

end else

User must set up 23 bit **start\_addr** parameter then activate read, write or erase strobe. Tata will be read into Master AXIS, written from Slave AXIS or erased (4K block) without transferring data. For erase case, **finished** signal will be activated at the end of the process.

## Icmp

This module contains FSM for processing ICMP Functionality. For now ip\_read\_... and ip\_write\_... buses are simply routed into this module and cannot be used at the top level.

## Module DHCPhelper

This module is just helper. It is designed for simplify main module and contains specific functionality for provide DHCP protocol. There two State Machines are present. Forst FSM is using for Generate DHCP Traffic, the second – for analyze incoming DHCP packets.

When we designed second FSM, the FMAx was dropped to value about 80 MHz. Probably it happened because too many compares of wide values are providing at the same clock period. We don’t see any good way for put them to the pipeline that is why we added one more clock domain with Frequency = 52.0833 MHz (also result of auto generated PLL, we ordered for 50 MHZ).

That is why DHCPHepler contains not only two State Machines, but also Dual Clock FIFO. This FIFO is filling at the System CLK (125 MHz for overclocked case or 104.167 MHz for 100M Case). Read speed is 52.0833 MHz. And the second FSM is clocked from the same slow speed.

Also module contains some shift registers fro provide cross clock domain for control and status signals.

## Module mDNShelper

Very simple module which contains some logic for remove it from the main module. It contains comparator for detect Unicast MAC Address 01 00 5E 00 00 FB which is used for mDNS.

All other functionality was easier keep in the main FSM. Any separate could create too much extra wires which could drop FMax.

## Module prng

AutoIP Protocol requires Random Generator. This module is providing this functionality. DHCP also uses it for generate XID Value.

## Module lfsr

This module replaces basic CRC Calculate method because original CRC Generator is using very specific Verilog Language constructions and Yosys producing wrong RTL code for it. We got more traditional version of CRC Generator. Need use it in the project instead original file

# Main FSM Logic

**We are not sure which precision is needed to use for details in this chapter. More details – more hours for description. Let’s we make small part and you will tell, need continue the same way or more detailed or more coarse.**

## Some details of idle state

Will be done when we will understand how detailed need to do it…

## Reflect Functionality

This part of the FSM is adeed for very preliminary tests and for speed measure. The port 1234 is used for reflect UDP datagrams.

In the **idle** state:

case (rx\_udp\_dest\_port)

…

16'd1234: begin

tx\_udp\_ip\_source\_ip <= local\_ip;

tx\_udp\_ip\_dest\_ip <= rx\_udp\_ip\_source\_ip;

tx\_udp\_source\_port <= 1234;

tx\_udp\_dest\_port <= rx\_udp\_source\_port;

tx\_udp\_length <= rx\_udp\_length;

rx\_udp\_hdr\_ready <= 0;

answerState <= pre\_reflect\_1;

end

First of all, we must keep in mind that we must fill Target IP As Soon As Possilble. This value must be compared in low layers. This compare process is pipelined. The pipeline adds latency of processing this parameters. That is why IP must be set some CLK periods (as I remember, 3 but as more as possibly because sometimes multiplexers are adding more latency but we still cannot understand how those multiplexers working). That is why we are setting Target IP not when need but As Soon As Possible. All other parameters – just together with IP. Why not? Then we are going to the reflect branch. This branch contains next states:

**pre\_reflect\_1,pre\_reflect\_2,reflect\_1,reflect\_2,**

**pre\_reflect\_1** is used as delay for compensate the latency of the ip comparator. But we must copy data from input FIFO to output FIFO at this state. But, of coursem if they are really present only. That is why we are copying either tdata or tvalid.

pre\_reflect\_1: begin

ourData\_tdata <= rx\_udp\_payload\_axis\_tdata;

ourData\_tvalid <= rx\_udp\_payload\_axis\_tvalid;

answerState <= pre\_reflect\_2;

end

**pre\_reflect\_2** is the second delay step

pre\_reflect\_2: begin

ourData\_tdata <= rx\_udp\_payload\_axis\_tdata;

ourData\_tvalid <= rx\_udp\_payload\_axis\_tvalid;

answerState <= reflect\_1;

end

**reflect\_1** In this state we are waiting for ready for UDP Write Engine Ready to transmit data. If ready then we are initiating the Write Datagram Process. The data from Input FIFO is still not fully copied to Transmit FIFO. That is why we are still copying it if it is present.

reflect\_1: begin

ourData\_tdata <= rx\_udp\_payload\_axis\_tdata;

ourData\_tvalid <= rx\_udp\_payload\_axis\_tvalid;

if (tx\_udp\_hdr\_ready)

begin

tx\_udp\_hdr\_valid <= 1;

answerState <= reflect\_2;

end

end

**reflect\_2** Continue copy data from Read FIFO to WRITE FIFO until tlast signal.

reflect\_2: begin

tx\_udp\_hdr\_valid <= 0;

ourData\_tdata <= rx\_udp\_payload\_axis\_tdata;

ourData\_tvalid <= rx\_udp\_payload\_axis\_tvalid;

ourData\_tlast <= rx\_udp\_payload\_axis\_tlast;

if (rx\_udp\_payload\_axis\_tvalid & rx\_udp\_payload\_axis\_tlast)

begin

answerState <= idle;

end

end

## Read EEPROM

Will be done if need because this is not very important branch

## Write EEPROM

Will be done if need because this is not very important branch

## Erase EEPROM

Will be done if need because this is not very important branch

## DHCP Processing

DCHP process has two stages.

|  |  |
| --- | --- |
| **Stage** | **Initiator** |
| DISCOVER/OFFER | Hardware Input |
| REQUEST/ACK | UDP Datagram with “OFFER” packet to port 68 |

### Start of DHCP Process

In **idle** state:

end else if (!rxd) // Negative logic! DHCP trigger activated

begin

answerState <= dhcp\_start;

end /\*else if (!rxd) // Negative logic! DHCP trigger activated

As we see, this branch is located outside of Received Datagrams processing in **idle** state. Hardware contains single input only. That is why we are using “rxd” input for now. In real production, dedicated input must be used.

### dhcp\_start state

First of all, we must wait until return of “rxd” to passive state for prevent multi processing. Today simple button is used for start DHCP process.

When signal is returned to the passive level, as usually, we are setting Target IP and other parameters because we must do it ASAP for compensate Pipeline’s latency. Let’s look to code and then I will continue to describe it.

dhcp\_start: begin

// negative logic! Wait for release trigger

if (rxd)

begin

// We are planning to send DHCP\_DISCOVERY

// This is a first packet in DHCP processing

m\_dhcp\_discover\_step\_request <= 1; // This step is discover

tx\_udp\_ip\_source\_ip <= 32'h00000000; // Broadcasting...

tx\_udp\_ip\_dest\_ip <= 32'hffffffff; // ...

tx\_udp\_source\_port <= 16'd68; // DHCP ports

tx\_udp\_dest\_port <= 16'd67; // ...

dhcp\_m\_dhcp\_discover\_start <= 1; // Ask for helper to produce data

rx\_udp\_hdr\_ready <= 0; // Now we are not ready to receive any data from UDP

tx\_udp\_length <= 8; // Extra length

answerState <= dhcp\_fill\_0;

end

end

Some functionality of DISCOVER/OFFER and REQUEST/ACK branches are processing in the same states. But details of processing will be different. That is why we must set **m\_dhcp\_discover\_step\_request** signal for mark current step as DISCOVER before leave this state (this state is 100% used for DISCOVER step only).

Now we don’t know which size Datagram will be filled by DHCP Helper, that is why we must set current size as 0. But Original firmware needs set up Length increased to 8. That is why here we are setting **tx\_udp\_length** register to 8.

We are setting dhcp\_m\_dhcp\_discover\_start signal for ask DHCP Helper start to generate OFFER Request for us.

How this Request is generating, we will describe in dedicated chapter if need. Maybe source code is enough.

### dhcp\_fill\_0 state

For copy generated data into Output FIFO, we are coming to the **dhcp\_fill\_0** state. This is mixed state. AS we saw at reflect states, here we are copying data from dhcp\_m\_dhcp\_discover\_... bus to ourData\_... bus (this is Write FIFO input). Let’s look to Source Code and then describe extra functionality of this state

dhcp\_fill\_0: begin

dhcp\_m\_dhcp\_discover\_tready <= 1; // We are ready to receive the data from helper

ourData\_tdata <= dhcp\_m\_dhcp\_discover\_tdata; // Receive data from DHCP Helper to FIFO if it is present

ourData\_tvalid <= dhcp\_m\_dhcp\_discover\_tvalid; // ...

ourData\_tlast <= dhcp\_m\_dhcp\_discover\_tlast; // ...

if (dhcp\_m\_dhcp\_discover\_tvalid) // If really present

begin

tx\_udp\_length <= tx\_udp\_length + 1; // Then increase length

end

if (dhcp\_m\_dhcp\_discover\_finished) // If helper has no any more data

begin

spi\_start\_addr <= 24'h1ff040;

spi\_read\_strobe <= 1;

spi\_m\_tready <= 1;

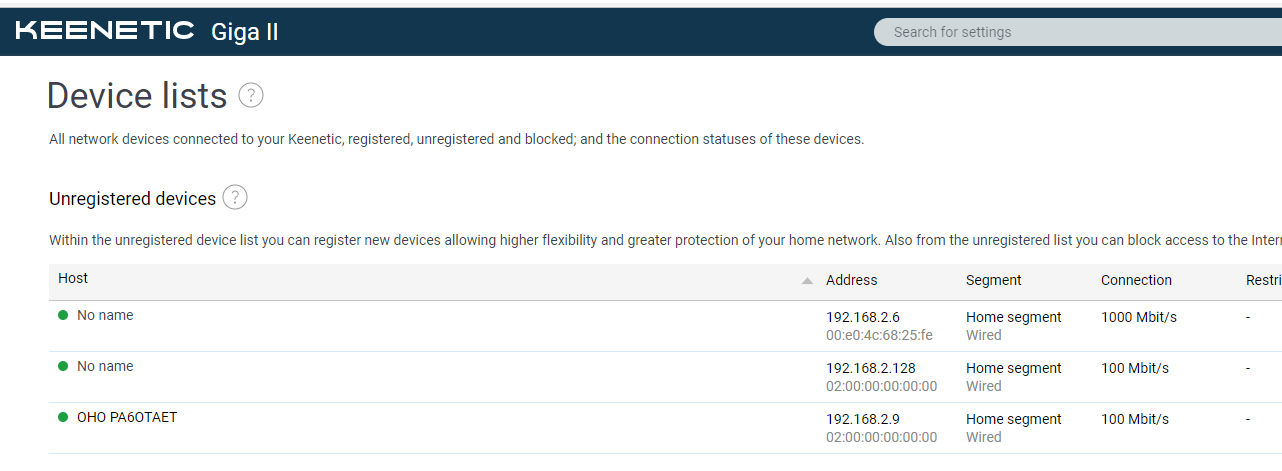
answerState <= dbg\_string\_1;

end

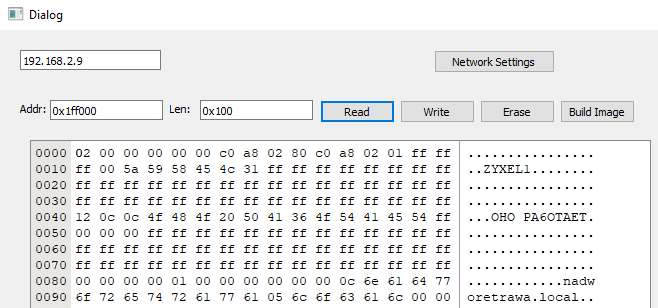
end

In reflect state we just copied data. Here we also need check, is data really present on the bus. If yes, then we are increasing Length register.

DHCP Client may have name. This name is very helpful for choose devices at the router. For example:



The name of device is stored in the EEPROM (use Setup Tool GUI for change it, the format is not simple ASCII String, this is a raw data for finalization of DHCP Request).



As a result, if Helper says that data is fully finished, our FSM sets up hardcoded EEPROM Address **24'h1ff040** and goes to the branch, which is very similar with Read EEPROM Branch. The start of this branch is located at state **dbg\_string\_1**. This name is subject to be changed later. We planned more complicated processing, but rejected that idea when FMax dropped too much.

### dbg\_string\_1 state

Ath this state we are receiving Data Size from EEPROM (first byte in EEPROM contains actual length of stored data). That is why when data is present (EEPROM is very slow device), we will store this value and go **to dbg\_string\_2** state.

dbg\_string\_1: begin

if (spi\_m\_tvalid)

begin

eeprom\_cnt <= spi\_m\_tdata - 1;

answerState <= dbg\_string\_2;

end

end

### dbg\_string\_2 state

Here we are receiving data from EEPROM **(spi\_m\_...** bus) to FIFO (**ourData\_**... bus) as usually. If data is present at the bus (very eventual situation because SPI is very slow) we will increment packed length and decrement remaining counter. If this counter is already 0, we will mark FIFO transaction as finished (tlast signal), cancel EEPROM operation (clear **spi\_read\_strobe**) and jump to state dhcp\_fill\_1 for transmit totally built data.

dbg\_string\_2: begin

ourData\_tdata <= spi\_m\_tdata;

ourData\_tvalid <= spi\_m\_tvalid;

if (spi\_m\_tvalid)

begin

tx\_udp\_length <= tx\_udp\_length + 1;

if (eeprom\_cnt == 0)

begin

spi\_read\_strobe <= 0;

spi\_m\_tready <= 0;

answerState <= dhcp\_fill\_1;

ourData\_tlast <= 1;

end else

begin

eeprom\_cnt <= eeprom\_cnt - 1;

ourData\_tlast <= 0;

end

end

end

### dhcp\_fill\_1 state

We are enabling **tx\_udp\_hdr\_valid** request and waiting for **tx\_udp\_hdr\_ready** acknowledge. When it is received, we will return to **idle** state. Request will be cleared in that state. Well. DISCOVER Datagram is sent. Now we must be ready to react to OFFER Datagram from Server. But this is another branch of the State Machine.

dhcp\_fill\_1: begin

tx\_udp\_hdr\_valid <= 1; // Start send to UDP

ourData\_tvalid <= 0;

if (tx\_udp\_hdr\_ready)

begin

answerState <= idle;

end

end

### Reaction to OFFER Datagram in idle state

When Datagram to port 68 (DEC) is received, we are enabling copying process from UDP Receive FIFO to internal Cross Domain FIFO of DHCP Helper by setting s\_dhcp\_offer\_start signal. Second State Machine of DCHP Helper starts to work (will be descriped if need in dedicated chapter) and we are jumping to state dhcp\_offer\_processing1 for be ready start processes when DHCP Helper will finish it’s job.

16'd68: begin

rx\_udp\_hdr\_ready <= 0; // We are not ready to receive any other UDP packets

s\_dhcp\_offer\_start <= 1; // Start transmitting data from UDP FIFO to slow helepr's FIFO

answerState <= dhcp\_offer\_processing1;

end

### dhcp\_offer\_processing1 state

Here we are providing copy process data from UDP RX FIFO (**rx\_udp\_payload\_axis\_...** bus) to helper’s FIFO (**s\_dhcp\_offer\_axis\_**... bus). If rx\_udp\_payload\_axis\_tlast is detected, we will jump to next state **dhcp\_offer\_processing2**.

### dhcp\_offer\_processing2 state

First of all , we are awining for finish of working DHCP Helper. Port 68 could be called either for OFFER or for ACK requests. Of course, any unexpected types of packets could be came. If ACK was received, Helper already updated IP Parameters inside it’s body. If packet is unexpected, we must not process it. That is why, in all cases, except OFFER, we are jumping to the idle state

if (!dhcp\_offerIsReceived)

begin // No! This is answer for DHCPACK. No need send anything

answerState <= idle;

end else // Yes! And we must send DHCPREQUEST

Else we are coming to already well known state **dhcp\_fill\_0** with well known preparations except setting **m\_dhcp\_discover\_step\_request** to 0 because now we are sendiong not DISCOVER but REQUEST packet. Next steps will be the same as we describet above.

dhcp\_offer\_processing2: begin

s\_dhcp\_offer\_axis\_tvalid <= 0;

if (s\_dhcp\_offer\_finished)

begin

// Let's stop slow State Machine in DHCP helper

s\_dhcp\_offer\_start <= 0;

// Are we serving OFFER packet?

if (!dhcp\_offerIsReceived)

begin // No! This is answer for DHCPACK. No need senb anything

answerState <= idle;

end else // Yes! And we must send DHCPREQUEST

begin

m\_dhcp\_discover\_step\_request <= 0;

tx\_udp\_ip\_source\_ip <= 32'h00000000;

tx\_udp\_ip\_dest\_ip <= 32'hffffffff;

tx\_udp\_source\_port <= 16'd68;

tx\_udp\_dest\_port <= 16'd67;

dhcp\_m\_dhcp\_discover\_start <= 1;

rx\_udp\_hdr\_ready <= 0;

tx\_udp\_length <= 8; // Extra length

answerState <= dhcp\_fill\_0;

end

end

end

## AutoIP Processing

Will be described if need

## ICMP Processing

Will be described if need

## mDNS Processing

Will be described if need

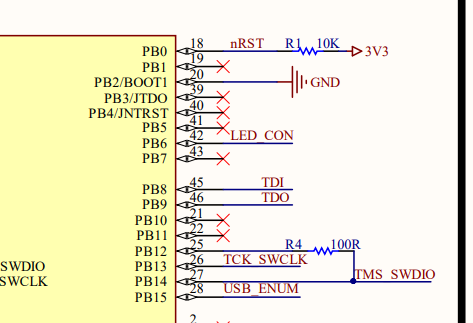
# How to unprotect EEPROM

When Board is came from Ali Express, the EEPROM Area is protected. We need unprotect it. **NB!** I don’t know why, but my ColorLight\_5A-75B board self protects EEPROM some days later after unlocking. Need unlock it again before wirting. “DIMM” version has no this problem. I unlocked it once.

For provide unlock operation, First of all, need use CMSIS DAP adapter. We made this adapter from BluePill Board, using the project

[wuxx/Colorlight-FPGA-Projects: current focus on Colorlight i5 series module (github.com)](https://github.com/wuxx/Colorlight-FPGA-Projects)

The pinout can be got from Schematic folder (TDI – PB8, TDO – PB9, TCK – PB13, TMS – PB14).



The firmware file is: **\firmware\flash\_image\_20210824.bin**

When this adapter is used, we can use **ecpdap** project

Where to get:

[adamgreig/ecpdap: ECPDAP allows you to program ECP5 FPGAs and attached SPI flash using CMSIS-DAP probes in JTAG mode. (github.com)](https://github.com/adamgreig/ecpdap)

How to use:

**D:\LATICE\ecpdap\_windows\_x86\_64.exe flash unprotect**

# How to adjust Network Parameters

Describe Qt program

# How to flash firmware with

Describe how to build Flash Image with parameters and flash it