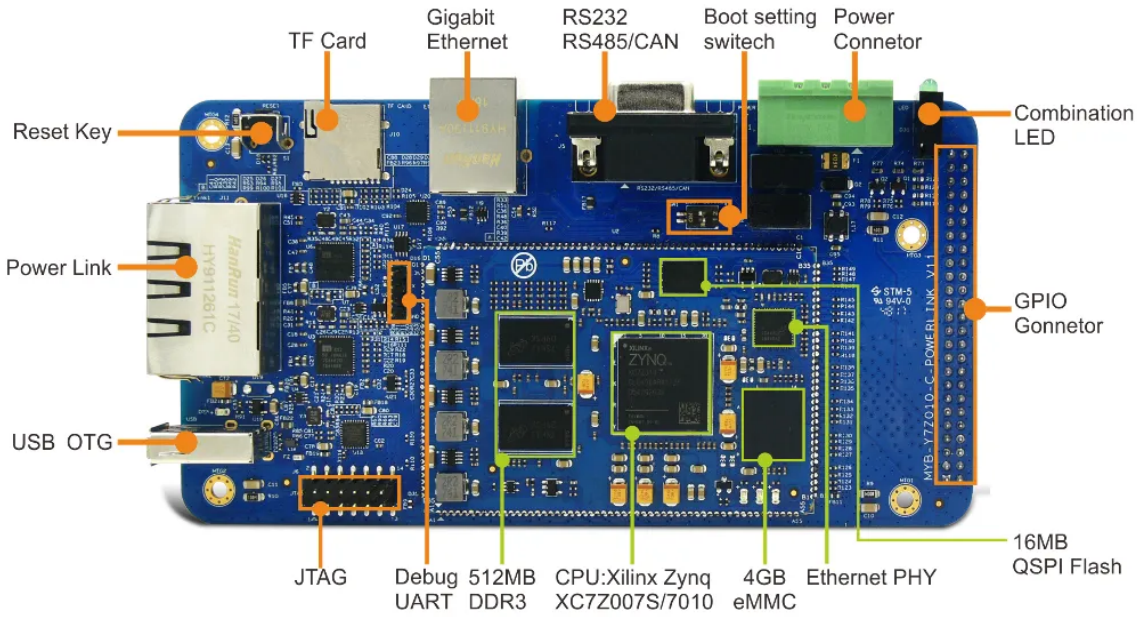
FPGAdeveloper

Board bring-up: MYIR MYD-Y7Z010 Dev board

by Jeff Johnson | May 3, 2018 | Board bring-up, Boards, Development Boards, Ethernet, MYD-Y7Z010, Topics | 2 comments

2 Votes

In this tutorial video, I bring-up the 3x Gigabit Ethernet ports on the MYD-Y7Z010 Development board from MYIR. Firstly, I create a Vivado design for this board, then I export it into the SDK and generate the echo server application for each of the 3 ports (note that the echo server application only supports one port at a time). At the end of the video, I test each of these designs on hardware and ensure that the ports are given an IP address via DHCP and that I can ping the port. I did this on the MYIR dev board but I hope that the tutorial can be of help to people bringing up Ethernet ports on other platforms or their own custom boards.



Requirements

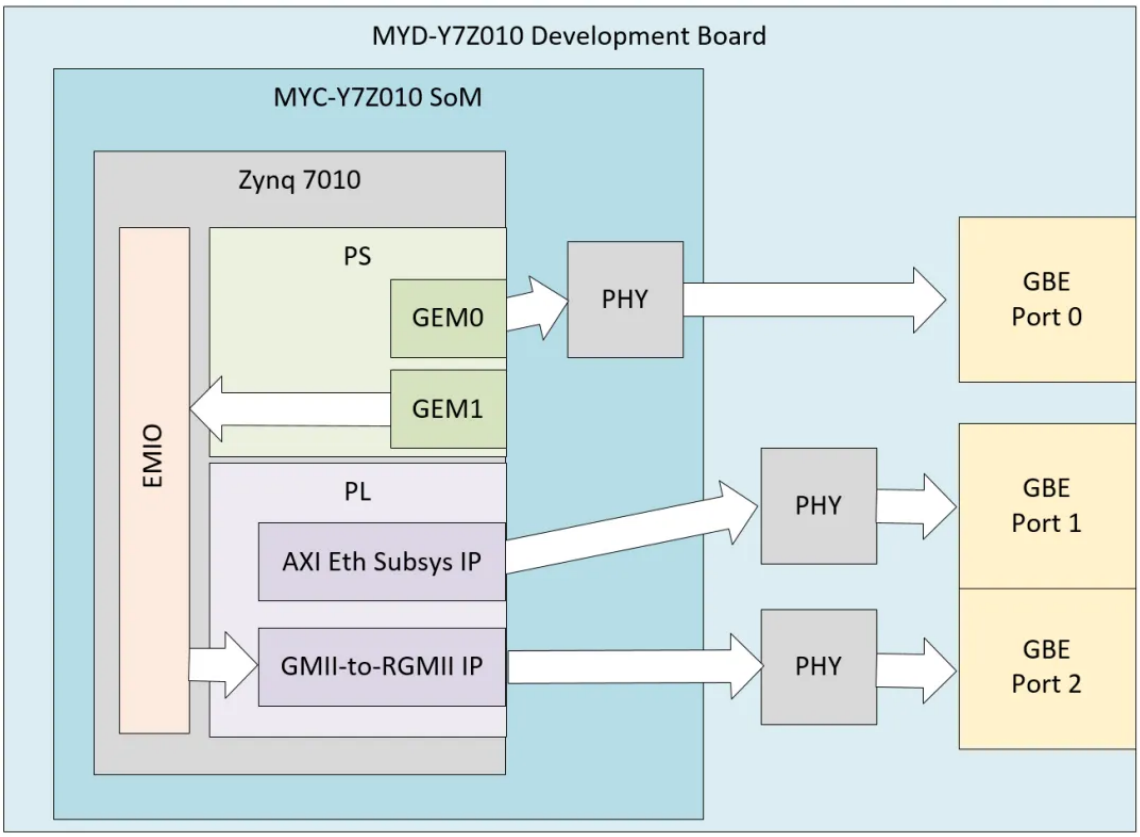
To go through this tutorial yourself, you'll need:

- the board files that I've placed in this Github repository: Projects for the MYD-Y7Z010 Development board
- Vivado 2018.1 – I've done the tutorial with Vivado 2018.1, but you should be able to do it in future versions without too much trouble
- an RS232 to USB converter
- a network running DHCP
- a CAT-5e/6 Ethernet cable

Description

The MYIR board is based on the Zynq 7010 device, so we'll make use of the two build-in GEMs of the Zynq PS and we'll use AXI Ethernet Subsystem IP for the third port. The image below shows how the ports are connected through the Ethernet

PHYs to the RJ45 connectors. All of the PHYs have an RGMII interface to the MACs.

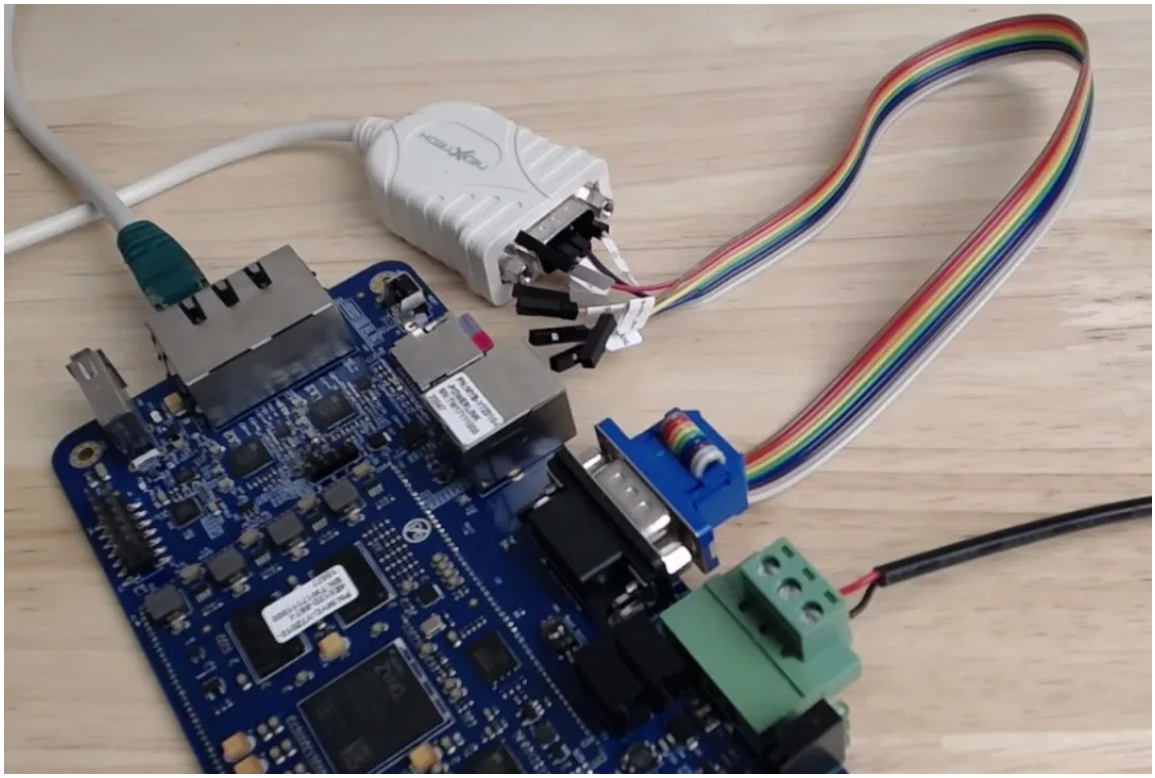


As you can see in the block diagram, one of the PHYs is on the module (SoM) and this PHY is directly connected (via MIO pins) to GEM0 of the Zynq PS. The other two PHYs are on the carrier board and they connect to the FPGA (PL) of the Zynq. For one of these PHYs, we'll route GEM1 to the PL via EMIO, and we'll use a GMII-to-RGMII IP to convert the GMII interface to the RGMII for the PHY connection. For the last PHY we will use the AXI Ethernet Subsystem IP.

UART for debug

When working with any board, a UART for debug is handy. This MYIR board has a few UART options but unfortunately none of them is a USB-UART:

- 3.3V TTL UART to a pin header (connects to UART1 of the Zynq PS)
- RS232 UART to the DB9 connector (connects to PL)
- RS485 UART to the DB9 connector (connects to PL)



If you're going through this yourself, I suggest you use the UART option that is most convenient for you. In my case, I've got a RS232 to USB converter handy, so I'm using the RS232 option. The MYIR board comes with a rainbow ribbon cable to breakout the DB9 connector, so I've just wired up the RS232 signals to the DB9 connector of the converter:

- White wire (RS232 GND) to pin 5 (GND) of the DB9 of the converter
- Grey wire (RS232 TX) to pin 2 (RX) of the DB9 of the converter
- Purple wire (RS232 RX) to pin 5 (TX) of the DB9 of the converter

Board files

Before creating the Vivado project for this board, you should copy the board files into your Vivado installation. Find the board files in the Github repo here:

https://github.com/fpgadeveloper/myd-y7z010-projects/tree/master/Vivado/boards/board_files/MYD-Y7Z010/1.1

Respecting that directory structure, copy the board files into your Vivado installation here:

```
\Xilinx\Vivado\VERSION\data\boards\board_files
```

The next time you run Vivado, the MYIR board will be on the list of boards when creating a new project.

lwIP Library modifications

When we get to the SDK, we only have to generate the echo server application/template for each of the 3 ports. However, as is typical when working with the echo server application, we have to modify some of the code that deals with configuration of the PHY. The code already handles some Marvell and TI PHYs, but not the Microchip PHY that this MYIR board uses (KSZ9031). I've detailed the code modifications below, but I've also included the modified code in the Github repo.

The best way to deal with library modifications is to create a local copy of the original library and bump up the version number. You modify this local copy and then add it's location as a repository to your SDK workspace. Then when generating the echo server application, the SDK will use your modified library instead of the original library. In the Github repo, I've already created this local copy of the library, but only containing the modified sources. To move the rest of the sources into this local copy, I've written a Tcl script that you can run. It's a very simple script and in fact, if you prefer to just copy the files over manually, you can – just make sure not to overwrite the files that are already in the repo, as they contain the required modifications.

For those who prefer to make the modifications themselves, or who want to better understand the modifications, I've described them below.

File to modify

Filename: `xemacpsif_physpeed.c`

Location:

`\EmbeddedSw\ThirdParty\sw_services\lwip202_v1_09\src\contrib\ports\xilinx\netif`

This file contains the code for configuration of the PHY connected to the GEMs. There are two things we need to change in this file. Firstly, we need to add a function to configure the Microchip PHY; for the most part, our function is the

same as the one for the Marvell PHY, except that the detection of the link speed is done through a different register. Secondly, we need to create a define for the PHY address of the GMII-to-RGMII converter, so that the code makes the necessary register changes to the core after link-up.

Add these defines for the Microchip PHY identifier and to specify the PHY address of the GMII-to-RGMII converter:

```
#define PHY_MICROCHIP_IDENTIFIER 0x0022
#define XPAR_GMII2RGMIICON_0N_ETH1_ADDR 8
```

Add this function for the configuration of the Microchip PHY (you can place it below the `get_Marvell_phy_speed` function):

```
static u32_t get_Microchip_phy_speed(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t temp;
    u16_t control;
    u16_t status;
    u16_t status_speed;
    u32_t timeout_counter = 0;

    xil_printf("Start PHY autonegotiation \r\n");

    XEmacPs_PhyWrite(xemacpsp,phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
    control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, control);

    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    &control);
    control |= ADVERTISE_1000;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    control);

    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,&control);
    control |= (7 << 12); /* max number of gigabit attempts */
    control |= (1 << 11); /* enable downshift */
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,control);
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
    control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);
```

```
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_RESET_MASK;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

    while (1) {
        XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
        if (control & IEEE_CTRL_RESET_MASK)
            continue;
        else
            break;
    }

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);

    xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

    while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
        sleep(1);
        XEmacPs_PhyRead(xemacpsp, phy_addr,
                        IEEE_COPPER_SPECIFIC_STATUS_REG_2,  &temp);
        timeout_counter++;

        if (timeout_counter == 30) {
            xil_printf("Auto negotiation error \r\n");
            return XST_FAILURE;
        }
        XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
    }
    xil_printf("autonegotiation complete \r\n");

  // Read from Microchip page 0, register 0x1F (PHY Control)
  // http://ww1.microchip.com/downloads/en/DeviceDoc/00002117F.pdf
    XEmacPs_PhyRead(xemacpsp, phy_addr,0x1F,&status_speed);
    if (status_speed & 0x040)
        return 1000;
    else if(status_speed & 0x020)
        return 100;
    else if(status_speed & 0x010)
        return 10;

    return XST_SUCCESS;
}
```

Modify the `get_IEEE_phy_speed` function as below so that it calls the `get_Microchip_phy_speed` function (above) to configure the Microchip PHY:

```
static u32_t get_IEEE_phy_speed(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t phy_identity;
    u32_t RetStatus;

    XEmacPs_PhyRead(xemacpsp, phy_addr, PHY_IDENTIFIER_1_REG,
                    &phy_identity);
    if (phy_identity == PHY_TI_IDENTIFIER) {
        RetStatus = get_TI_phy_speed(xemacpsp, phy_addr);
    } else if (phy_identity == PHY_REALTEK_IDENTIFIER) {
        RetStatus = get_Realtek_phy_speed(xemacpsp, phy_addr);
    } else if (phy_identity == PHY_MICROCHIP_IDENTIFIER) {
        RetStatus = get_Microchip_phy_speed(xemacpsp, phy_addr);
    } else {
```

```
        RetStatus = get_Marvell_phy_speed(xemacpsp, phy_addr);
    }

    return RetStatus;
}
```

File to modify

Filename: `xaxiemacif_physpeed.c`

Location:

`\EmbeddedSw\ThirdParty\sw_services\lwip202_v1_09\src\contrib\ports\xilinx\netif`

This file contains the code for configuration of the PHYs connected to AXI Ethernet Subsystem IP. There are two things we need to change in this file. Firstly, we need to add a function to configure the Microchip PHY; as described earlier, our function is mostly the same as the one for the Marvell PHY, except that the detection of the link speed is done through a different register. Secondly, because we are using AXI Ethernet Subsystem IP, we need to disable the RGMII TX clock delay that is internal to the PHY (for more information on this topic, read RGMII Timing Considerations).

Add these defines to specify the Microchip identifier and the register masks for TX and RX clock delay settings:

```
#define PHY_MICROCHIP_IDENTIFIER 0x0022
#define IEEE_RGMII_TX_CLOCK_DELAYED_MASK 0x0010
#define IEEE_RGMII_RX_CLOCK_DELAYED_MASK 0x0020
```

Add this function to configure the Microchip PHY, you can add it below the `get_phy_speed_88E1116R` function:

```
unsigned int get_phy_speed_Microchip(XAxiEthernet *xaxiemacp, u32 phy_addr)
{
    u16 phy_val;
    u16 control;
    u16 status;
    u16 partner_capabilities;

    xil_printf("Start PHY autonegotiation \r\n");

  /* RGMII with only RX internal delay enabled */
    XAxiEthernet_PhyWrite(xaxiemacp,phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
```

```
control &= ~IEEE_RGMII_TX_CLOCK_DELAYED_MASK;
control |= IEEE_RGMII_RX_CLOCK_DELAYED_MASK;
  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, control);

  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

  XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
  control |= IEEE_ASYMMETRIC_PAUSE_MASK;
  control |= IEEE_PAUSE_MASK;
  control |= ADVERTISE_100;
  control |= ADVERTISE_10;
  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

  XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
              &control);
  control |= ADVERTISE_1000;
  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
              control);

  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
  XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
              &control);
  control |= (7 << 12); /* max number of gigabit atphy_valts */
  control |= (1 << 11); /* enable downshift */
  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
              control);

  XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
  control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
  control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

  XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
  control |= IEEE_CTRL_RESET_MASK;
  XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);
  while (1) {
      XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
      if (control & IEEE_CTRL_RESET_MASK)
          continue;
      else
          break;
  }

  xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

  XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
  while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
      AxiEthernetUtilPhyDelay(1);
      XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_STATUS_REG_2,
                           &phy_val);
      if (phy_val & IEEE_AUTONEG_ERROR_MASK) {
          xil_printf("Auto negotiation error \r\n");
      }
      XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET,
                  &status);
  }

  xil_printf("autonegotiation complete \r\n");

// Read from Microchip page 0, register 0xA (PHY Control)
// http://ww1.microchip.com/downloads/en/DeviceDoc/00002117F.pdf
  XAxiEthernet_PhyRead(xaxiemacp, phy_addr,0x1F,&status);
  if (status & 0x040)
      return 1000;
  else if(status & 0x020)
```

```
        return 100;
    else
        return 10;
}
```

Modify the `get_IEEE_phy_speed` function as below so that it calls the `get_phy_speed_Microchip` function (above) to configure the Microchip PHY:

```
unsigned get_IEEE_phy_speed(XAxiEthernet *xaxiemacp)
{
    u16 phy_identifier;
    u16 phy_model;
    u8 phytype;

#ifdef XPAR_AXIETHERNET_0_BASEADDR
    u32 phy_addr = detect_phy(xaxiemacp);

    /* Get the PHY Identifier and Model number */
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, PHY_IDENTIFIER_1_REG, &phy_identifier);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, PHY_IDENTIFIER_2_REG, &phy_model);

/* Depending upon what manufacturer PHY is connected, a different mask is
 * needed to determine the specific model number of the PHY. */
    if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
        phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

        if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
            return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
        } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
            return get_phy_speed_88E1111(xaxiemacp, phy_addr);
        }
    } else if (phy_identifier == PHY_MICROCHIP_IDENTIFIER) {
        return get_phy_speed_Microchip(xaxiemacp, phy_addr);
    } else if (phy_identifier == TI_PHY_IDENTIFIER) {
        phy_model = phy_model & TI_PHY_DP83867_MODEL;
        phytype = XAxiEthernet_GetPhysicalInterface(xaxiemacp);

        if (phy_model == TI_PHY_DP83867_MODEL && phytype == XAE_PHY_TYPE_SGMII) {
            return get_phy_speed_TI_DP83867_SGMII(xaxiemacp, phy_addr);
        }

        if (phy_model == TI_PHY_DP83867_MODEL) {
            return get_phy_speed_TI_DP83867(xaxiemacp, phy_addr);
        }
    }
    else {
        LWIP_DEBUGF(NETIF_DEBUG, ("XAxiEthernet get_IEEE_phy_speed: Detected PHY with
    }
#endif
#ifdef PCM_PMA_CORE_PRESENT
    return get_phy_negotiated_speed(xaxiemacp, phy_addr);
#endif
}
```

Booting from SD card

When testing on hardware, unfortunately the MYIR board has the legacy 100mil pitch JTAG header, and I don't have an appropriate adapter for this. So instead of programming the board via JTAG, I generate a BOOT.bin file for each of the echo server applications, then I boot the board from the SD card. To boot the board from SD card, you will have to set the SW1 boot setting to 1-OFF, 2-ON.

What next?

Now that we've validated the hardware of this board, we could make it useful by getting Linux to run on it. When I find some time in the coming weeks, I'll generate PetaLinux for this board and test it out.

By the way, board bring-up is one of the services that we offer our customers. If your company or start-up has a custom board that you would like to bring-up, please get in touch. We are particularly good with Ethernet and PCIe interfaces.

Jeff Johnson

Jeff is passionate about FPGAs, SoCs and high-performance computing, and has been writing the FPGA Developer blog since 2008. As the owner of Opsero, he leads a small team of FPGA all-stars providing start-ups and tech companies with FPGA design capability that they can call on when needed.


Related

Using AXI Ethernet Subsystem and GMII-to-RGMII in a Multi-port Ethernet design
December 8, 2015
In "Ethernet"

Running a lwIP Echo Server on a Multi-port Ethernet design
January 5, 2016
In "Ethernet"

Multi-port Ethernet in PetaLinux
May 3, 2016
In "Ethernet"

2 Comments

Jose on February 6, 2019 at 3:51 am

0 0 Rate This

Hi Jeff,

Great tutorial like always, thanks for sharing.

I wonder when will be the second part booting linux?…..

It's being a while since your last post, are you continuing the blog? hope yes, because I'm following your posts since long time and I found them always inspiring and helpful.

Regards

Reply

sheree subahan on April 19, 2019 at 2:08 am

0 0 Rate This

Can I get schematic for the module MYD-Y7Z010 Dev board

Reply

Most popular posts

Posts

All | Today | This Week | This Month

- Connecting an SSD to an FPGA running PetaLinux

  5/5 (7 votes)

- Getting Started with the MYIR Z-turn

  5/5 (7 votes)

- Zynq and the trend towards ARM-FPGA architectures

  5/5 (5 votes)

- Creating a custom AXI-Streaming IP in Vivado

  5/5 (5 votes)

- Create a custom PYNQ overlay for PYNQ-Z1

  5/5 (5 votes)

Recent Posts

- NVMe SSD Speed test on the ZCU106 Zynq Ultrascale+ in PetaLinux
- Measuring the maximum throughput of Gigabit Ethernet on the Ultra96
- Ethernet Mezzanine for Ultra96
- Introducing 96B Quad Ethernet Mezzanine
- Board bring-up: MYIR MYD-Y7Z010 Dev board

Topics

96boards AC701 Aurora custom ip dma Ethernet finance FMC fpga drive hardware acceleration high frequency trading impact jtag KC705 lwip MicroZed ML505/XUPV5 ML605 multigigabit transceiver myir ncd nvme PCIe peripheral petalinux picozed rocketio root complex sdk som ssd svn tutorial ultra96 VC707 Virtex-5 Virtex-6 Virtex-II Pro vivado XUPV2P ZC702 ZC706 ZedBoard Zynq zynq ultrascale+

Products Services

- Ethernet FMC
- FPGA Drive
- 96B Quad Ethernet Mezzanine
- FPGA Design Services