



# MIPSfpga SOC

## Advanced Starter Tutorial



These materials produced in association with Imagination.  
Join our University community for more resources.

[community.imgtec.com/university](https://community.imgtec.com/university)

## 1. Introduction

In this tutorial, you will learn how to download and boot Linux on the MIPSfpga SOC system-on-chip platform and write a simple program using memory-mapped I/O to read switches and write LEDs.

This tutorial presents a MIPSfpga system-on-chip (SOC) running Linux on a Nexys 4 DDR board. The processor core communicates with peripherals including DDR memory, a RAM bootloader, a UART, and LEDs and switches. These peripherals are implemented using Xilinx IP Integrator blocks on an AXI bus connecting to MIPSfpga via an AHB-AXI bridge.

Linux is loaded into the DDR memory from a host computer via EJTAG. The user interacts with the Linux system over a COM port on the host computer.

This tutorial assumes that you have already completed the MIPSfpga Getting Started Guide, are comfortable using Linux and Windows, and can read and write C. If you wish to do system-on-chip design with the MIPSfpga platform after this tutorial, it is also helpful to be able to read and write Verilog, VHDL, and MIPS assembly language. If you wish to hack the Linux kernel, low-level Linux experience is essential. The MIPSfpga SOC is a complex system and is intended for sophisticated users with advanced troubleshooting skills.

See the MIPSfpga SOC Documents directory for much more detail about the hardware platform and Linux configuration and boot process.

## 2. Hardware and Software Required

For this tutorial, you will need the same hardware as in the MIPSfpga Getting Started Guide, plus an Ethernet cable to download code:

- 64-bit PC running Windows 7 or later, connected to a network via DHCP
- Digilent Nexys4-DDR FPGA board
- USB cable connected between PC and Nexys4-DDR USB programmer port
  - For providing power, JTAG link for downloading a bitfile to the FPGA, and virtual serial link to communicate with a COM port on the MIPSfpga SOC on the FPGA
- Bus Blaster EJTAG cable between PC and Nexys4 EJTAG port
  - For downloading the Linux kernel to DDR memory
- Ethernet cable between Nexys4 board and a network switch with DHCP

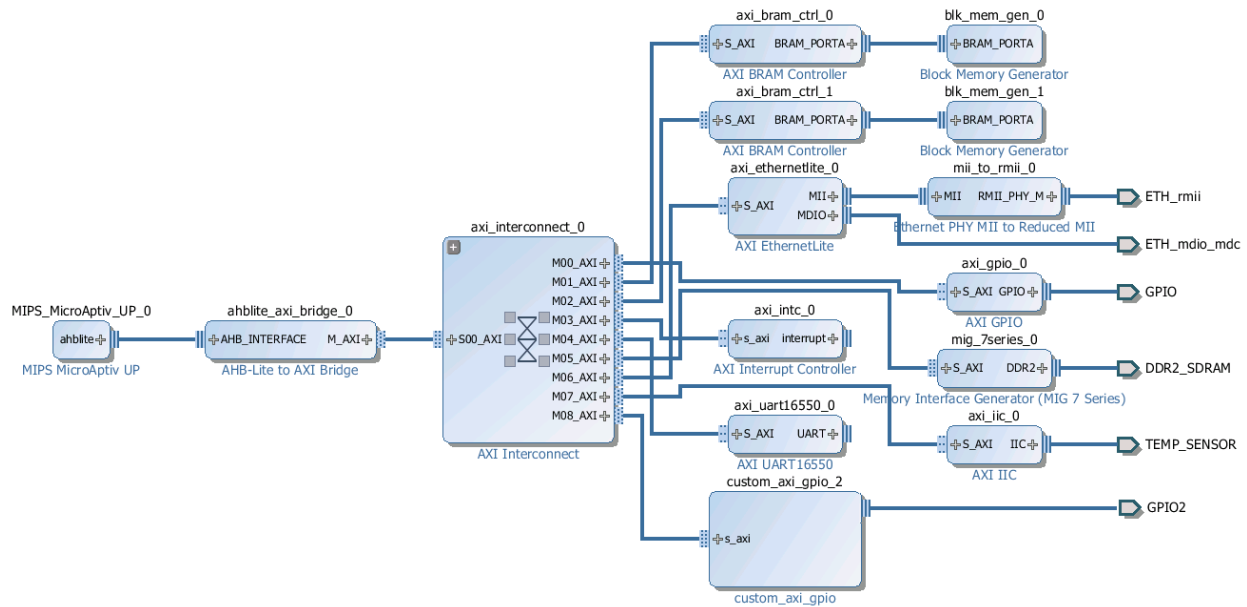
The tutorial has been tested with the following software running on the Windows PC

- Vivado 2014.4
  - For downloading the MIPSfpga bit file to the FPGA
- OpenOCD and gdb
  - For downloading the Linux image to DDR memory over the Bus Blaster
- PuTTY
  - A terminal on the PC for communicating with Linux over a virtual COM port on the USB cable
- CodeScape Linux Toolchain:
  - For the MIPSfpga Getting Started Guide, you were using the CodeScape mips-mti-elf toolchain to compile bare metal programs. You will need to use the mips-mti-linux-gnu toolchain to compile the programs which run under Linux.

### 3. Hardware Platform

The objective of MIPSfpga SOC is to create a System on Chip design that is capable of running Linux on MIPSfpga. The bare minimum requirements for running Linux are a system with 32Mbytes RAM, UART, Interrupts, Timers, MIPS running in cached mode. MIPSfpga SOC provides a system which fulfills these requires and adds IIC and Ethernet as well.

Due to the complexity of the SOC, we use higher level block design tools relying on several Xilinx IP blocks for MIPSfpga SOC. To use Xilinx's block design tools, the MIPS core is wrapped in an interface module called MIPS\_MicroAptiv\_UP exposing clock, reset, EJTAG, and the AHB-Lite bus. Figure 1 shows the MIPSfpga SOC hardware platform from Xilinx IP Integrator.



**Figure 1. Simplified MIPSfpga SOC block diagram in Vivado**

Starting from the left, we have the MIPS core. Then an AHBlite to AXI bridge which allows us to connect to AXI IP blocks from Xilinx to MIPSfpga. Then the AXI interconnect which allows the MIPS master to connect to several slaves depending on addresses. The slave peripherals are block ram controllers, Ethernet MAC, AXI Interrupt controller, an AXI GPIO controller by Xilinx, UART16550, an AXI IIC master, a Custom GPIO Controller by IMG and the Memory Interface Generator which serves as the DDR controller.

The physical memory map for the peripherals is given in Figure 2. Recall that virtual addresses add 0xA0000000 to the physical addresses, so the reset vector virtual address of 0xBFC00000 corresponds to a physical address 0x1FC00000, which is at the beginning of the boot SRAM.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
MIPS_MicroAptiv_UP_0					
ahblite (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x1FC0_0000	8K	0x1FC0_1FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x1000_0000	8K	0x1000_1FFF
mig_7series_0	S_AXI	memaddr	0x0000_0000	128M	0x07FF_FFFF
axi_gpio_0	S_AXI	Reg	0x1060_0000	64K	0x1060_FFFF
axi_intc_0	s_axi	Reg	0x1020_0000	64K	0x1020_FFFF
axi_uart16550_0	S_AXI	Reg	0x1040_0000	64K	0x1040_FFFF
axi_ethernetlite_0	S_AXI	Reg	0x10E0_0000	64K	0x10E0_FFFF
axi_iic_0	S_AXI	Reg	0x10A0_0000	64K	0x10A0_FFFF
custom_axi_gpio_2	s_axi	S00_AXI_reg	0x10C0_0000	64K	0x10C0_FFFF

**Figure 2. Physical memory map for MIPSfpga SOC system**

Most of the peripherals are Xilinx IP Integrator blocks and are rather complex to understand. The custom\_axi\_gpio block is simpler custom-developed Verilog demonstrating how to interface a custom peripheral to the AXI bus.

The Nexys4 DDR board has 16 LEDs and 16 switches. In the MIPSfpga SOC project, half are connected to the axi\_gpio which is Xilinx's GPIO controller IP and half to the custom\_axi\_gpio, as given in Table 1.

**Table 1. Switch and LED connections on the Nexys4 DDR board**

Port[bits]	Peripheral
axi_gpio[7:0]	SW[7:0]
axi_gpio[15:8]	LED[7:0]
custom_axi_gpio[7:0]	SW[15:8]
custom_axi_gpio[15:8]	LED[15:8]

The custom\_axi\_gpio and axi\_gpio memory maps each define two registers at the offsets from the base address as seen in Table 2. The base address can be seen in Figure 2.

**Table 2. Memory-mapped I/O register offsets for the custom\_axi\_gpio and axi\_gpio modules**

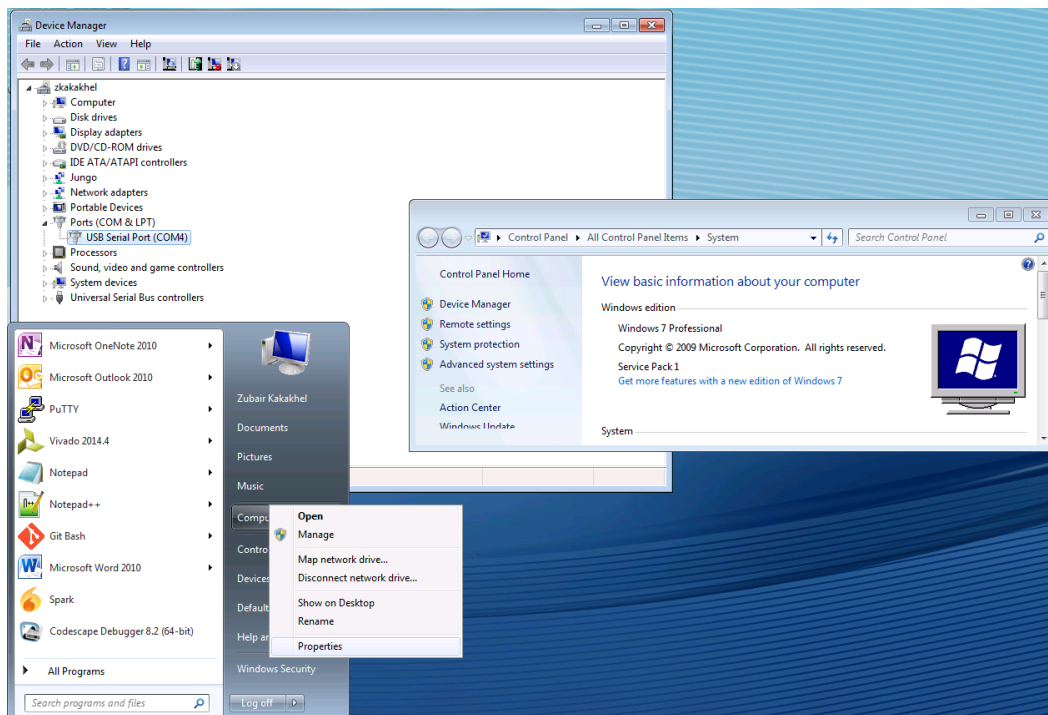
Register	Address offset	Function
DATA	0x00	If pin is input, reads to bit read pin value. Writes are discarded. If pin is output, reads to bit read the current output value on pin, write to bit write the value on the pin.
TRIS	0x04	Selects which bit is input (1) or output (0).

## 4. Download the Bitstream

Part 1 of MIPSfpga SOC documentation in the documents/Part1 folder contains extensive documentation on setting up the hardware with Xilinx IP Integrator. Synthesis, placement, and routing eventually produce the bitfile for the Nexys4 DDR board. This bitfile is provided in the documents/Starter\_Tutorial folder. Download the project\_linux.bit file onto the Nexys4 DDR board using Vivado and steps similar to those described in Section 6.4.1 of the MIPSfpga Getting Started Guide.

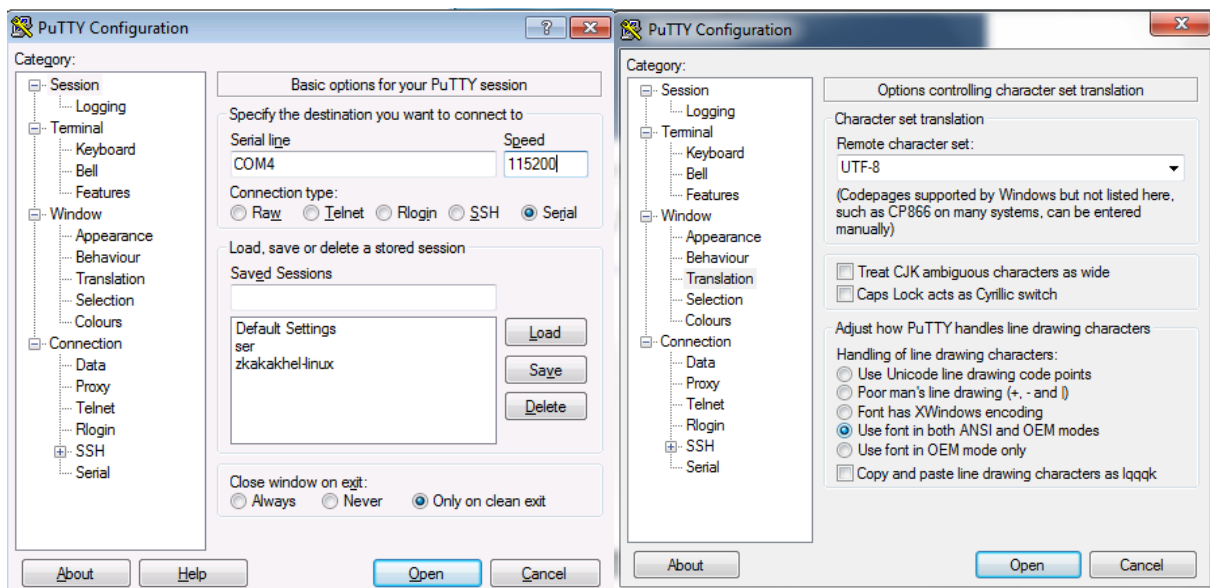
## 5. Terminal Emulation

Now you will set up the terminal for communicating with the Linux running on MIPSfpga SOC system. Linux communicates with a terminal via a UART. You can use your computer as the terminal. The USB cable between the PC and the Nexys4 DDR board doubles as a virtual COM port and connects to the axi\_uart16550 block's transmit and receive pins on the SOC design. The COM port number is enumerated by Windows. You will have to check your device manager as to which COM port you have, as shown in Figure 3.



**Figure 3 Check Device Manager for COM port number (COM4, in this case)**

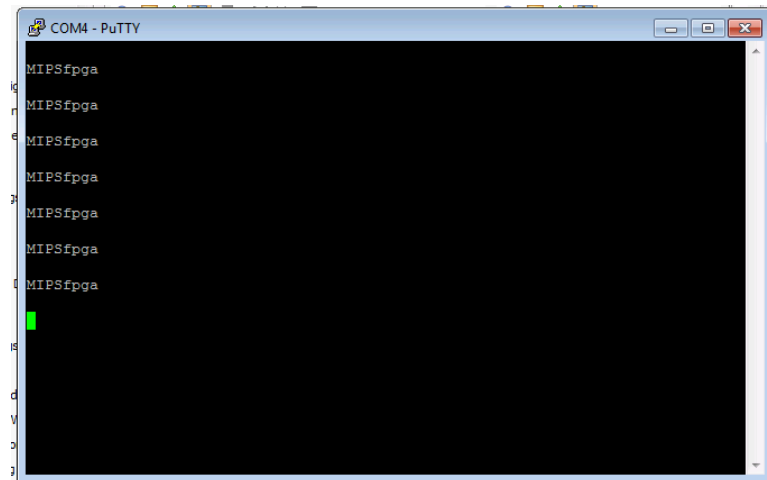
Run PuTTY, select serial, connect it to COM4 (i.e., the COM port you found in the previous step) with a speed of 115200. Also Under category->window-> translation choose the option “Use font in both ANSI and OEM modes”, as shown in Figure 4



**Figure 4 Select Serial COM4 port and 115200 baud rate and font options**

## 6. Run the bootram code

The bootram of the MIPSfpga SOC system is pre-initialized with code that initializes the cache, shifts the LEDs and prints the words "MIPSfpga" repeatedly to the UART, as shown in Figure 5. Press the CPU Reset button on the Nexys4 DDR board. You should see the following in the PuTTY screen and after a while 4 LEDs should shift on the Nexys4 DDR board.



**Figure 5 PuTTY terminal output**

The Linux kernel expects the MIPS core's cache, memory, and UART to be initialized and in a ready state, which has occurred now by running the pre-loaded program. Now we are ready to load Linux.

## 7. Load Linux to memory via EJTAG

It is possible to write bare metal C code to access and use the various peripherals such as Ethernet, IIC etc on the MIPSfpga SOC platform. But a system at this level of complexity necessitates the usage of an operating system.

Linux is a very popular open source operating system that is highly scalable and well suited to this purpose. Part 2 of the documentation in documents/Part2 folder contains further documentation on compiling the Linux kernel and Buildroot to generate a vmlinux ELF file that runs on the MIPSfpga SOC platform. The Linux Kernel is the part of the Linux operating system that interacts directly with the hardware. Buildroot is the part of the Linux operating system that interacts with the user. Vmlinux is a statically linked ELF file that can incorporate both the Kernel and Buildroot in one ELF.

A precompiled vmlinux ELF file is provided and is located in the documents/Starter\_Tutorial folder. Download this ELF file onto the DDR memory over EJTAG using the Bus Blaster probe, OpenOCD, and gdb as described below.

Note that the Linux file system is stored in RAM. Hence, anything you upload will be lost when the Nexys 4 DDR board powers down.

Open a cmd window and run OpenOCD to connect to the MIPS core on the Nexys4 DDR board by typing the following two commands:

```
cd C:\Program Files\Imagination Technologies\OpenOCD
openocd-nexys4.bat
```

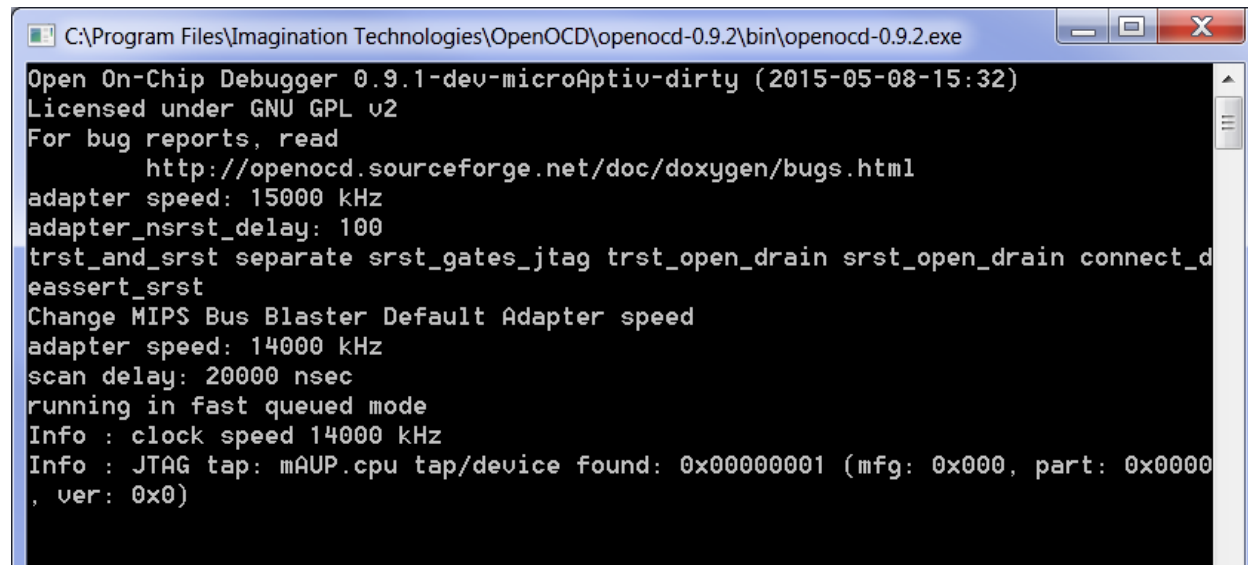


Figure 6 OpenOCD connected to the MIPS core

Open another cmd window, change to the documents\Starter\_Tutorial path, and run gdb by executing the following two commands:

```
cd C:\MIPSfpgaSOC\documents\Starter_Tutorial
"C:\Program Files\Imagination Technologies\Toolchains\mips-mti-elf\2015.06-05\bin\mips-mti-elf-gdb.exe" -x startup.txt
```

This will start gdb, the GNU debugger. Please note that Linux expects the cache and UART to be initialized before being loaded. In traditional systems, this is handled by the bootloader such as u-boot. In MIPSfpga SOC, this is already handled by the bootrom code preinitialized in the block memory in the hardware bitstream. Hence, startup.txt leaves the bootrom in a running state instead of halted as in the Getting Started Guide.

Stop the running code by pressing **CTRL+C**. Now, load the vmlinux ELF file provided in the documents/Starter\_Tutorial folder by typing the following at the gdb prompt (as shown in Figure 7):

```
load vmlinux
```

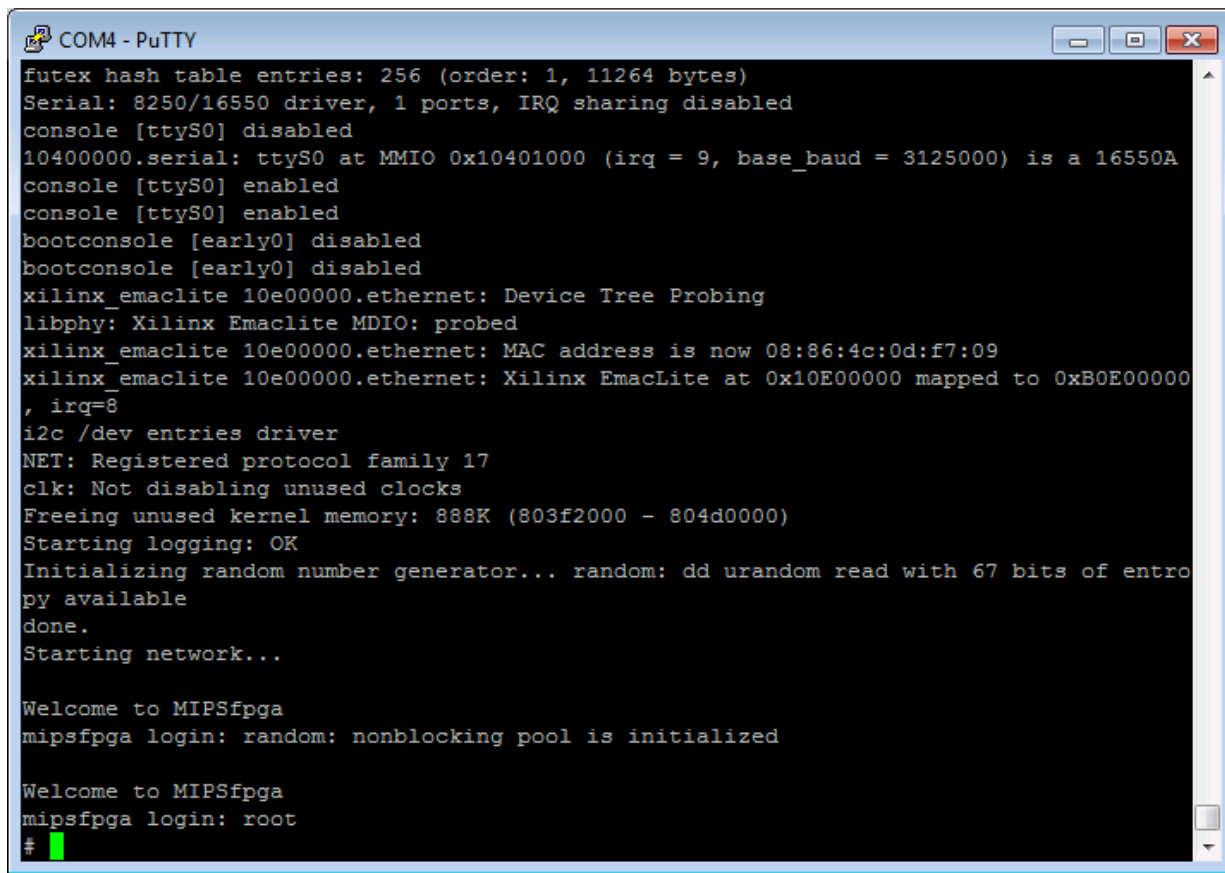


```
(gdb) load vmlinux
Loading section .text, size 0x21bbdc lma 0x80100000
Loading section __ex_table, size 0x1a40 lma 0x8031bbe0
Loading section .notes, size 0x24 lma 0x8031d620
Loading section .rodata, size 0x66cb0 lma 0x8031e000
Loading section .MIPS.abiflags, size 0x18 lma 0x80384cb0
Loading section __param, size 0x280 lma 0x80384cc8
Loading section __modver, size 0xb8 lma 0x80384f48
Loading section .data, size 0x16740 lma 0x80385000
Loading section .init.text, size 0x19554 lma 0x8039c000
Loading section .init.data, size 0x505f0c lma 0x803b5560
Loading section .exit.text, size 0x2ac lma 0x808bb46c
Start address 0x80312850, load size 8103052
Transfer rate: 20 KB/sec, 4082 bytes/write.
warning: GDB can't find the start of the function at 0x80312850.
```

**Figure 7. Loading vmlinux onto MIPSfpga SOC**

## 8. Booting Linux

After loading the ELF file, the PC register should automatically be set to the `kernel_entry` point which is where the Linux Kernel code starts from. Type 'continue' at the gdb prompt and hit enter and Linux will boot. You will see the initialization sequence in the PuTTY terminal shell. After a few minutes, the PuTTY terminal will show the Linux prompt. Log in as root with no password, as shown in Figure 8.



```
COM4 - PuTTY
futex hash table entries: 256 (order: 1, 11264 bytes)
Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
console [ttyS0] disabled
10400000.serial: ttyS0 at MMIO 0x10401000 (irq = 9, base_baud = 3125000) is a 16550A
console [ttyS0] enabled
console [ttyS0] enabled
bootconsole [early0] disabled
bootconsole [early0] disabled
xilinx_emaclite 10e00000.ethernet: Device Tree Probing
libphy: Xilinx EmacLite MDIO: probed
xilinx_emaclite 10e00000.ethernet: MAC address is now 08:86:4c:0d:f7:09
xilinx_emaclite 10e00000.ethernet: Xilinx EmacLite at 0x10E00000 mapped to 0xB0E00000
, irq=8
i2c /dev entries driver
NET: Registered protocol family 17
clk: Not disabling unused clocks
Freeing unused kernel memory: 888K (803f2000 - 804d0000)
Starting logging: OK
Initializing random number generator... random: dd urandom read with 67 bits of entro
py available
done.
Starting network...

Welcome to MIPSfpga
mipsfpga login: random: nonblocking pool is initialized

Welcome to MIPSfpga
mipsfpga login: root
#
```

Figure 8 Linux terminal prompt

## 9. Networking

You are now ready to test that Ethernet is working on the MIPSfpga SOC system. Connect the Nexys4 DDR board to a switch or router with an Ethernet cable. Now type the following into the PuTTY command window:

```
udhcpc
ping www.imgtec.com
```

Your command window should look similar to Figure 9.

```
Welcome to MIPSfpga
mipsfpga login: root
# udhcpc
udhcpc (v1.23.2) started
Sending discover...
xilinx_emaclite 10e00000.ethernet eth0: Link is Up -
100Mbps/Full - flow control rx/tx
Sending discover...
```

```
Sending select for 192.168.154.28...
Lease of 192.168.154.28 obtained, lease time 604800
deleting routers
adding dns 192.168.152.60
adding dns 192.168.152.63
# ping www.imgtec.com
PING www.imgtec.com (78.137.103.205): 56 data bytes
64 bytes from 78.137.103.205: seq=0 ttl=51 time=40.504 ms
64 bytes from 78.137.103.205: seq=1 ttl=51 time=35.466 ms
64 bytes from 78.137.103.205: seq=2 ttl=51 time=38.004 ms
64 bytes from 78.137.103.205: seq=3 ttl=51 time=34.699 ms
64 bytes from 78.137.103.205: seq=4 ttl=51 time=34.726 ms
^C
--- www.imgtec.com ping statistics ---
6 packets transmitted, 5 packets received, 16% packet loss
round-trip min/avg/max = 34.699/36.679/40.504 ms
#
```

**Figure 9. MIPSfpga SOC internet test**

udhcpc assigns an IP address to the Ethernet interface using dhcp, and ping checks that MIPSfpga SOC can communicate with a remote system, in this case [www.imgtec.com](http://www.imgtec.com). You can also use Ethernet to transfer large programs into your Linux file system using wget to download from a file server.

## 10. Writing and Compiling a Program

Installing CodeScape SDK should automatically install the Linux toolchain. The default location is C:\Program Files\Imagination Technologies\Toolchains\mips-mti-linux-gnu\

A simple GPIO read/write program is provided in the documents/Starter\_Tutorial directory. You may open the program, found in the file `simple_gpio.c`. The program accesses the eight left-most LEDs and eight left-most switches on the Nexys4 DDR board. Specifically, the program:

1. Writes the value 0xab to the LEDs,
2. Reads the value of the switches, and
3. Prints the value of the LEDs and switches to the output terminal in hexadecimal

Open a command prompt window and change to the documents/Starter\_Tutorial directory. Compile the example by typing the following command at the prompt:

```
"C:\Program Files\Imagination Technologies\Toolchains\mips-mti-linux-gnu\2015.06-5\bin\mips-mti-linux-gnu-gcc.exe" -static simple_gpio.c -EL -o simple_gpio -ffunction-sections -fdata-sections -Wl,--gc-sections -Os -s
```

This will compile the C code in little endian mode, with static libraries, and with symbols and unnecessary information stripped from the executable to reduce the size for download.

Linux on MIPSfpga is running in RAM only and has been loaded via EJTAG. The size of Linux running on MIPSfpga has been optimized enough to remove all libraries from it as well. Hence any executable needs to have library calls statically linked.

The easiest way to load a compiled file onto the MIPSfpga SOC system is to download it via a file server. To do so, first download the free edition of Mongoose from <https://www.cesanta.com/mongoose>. Running mongoose makes a temporary file server in the directory it runs. Copy the mongoose executable into the documents/Starter\_Tutorial directory with simple\_gpio.c and run it, as shown in **Error! Reference source not found.**

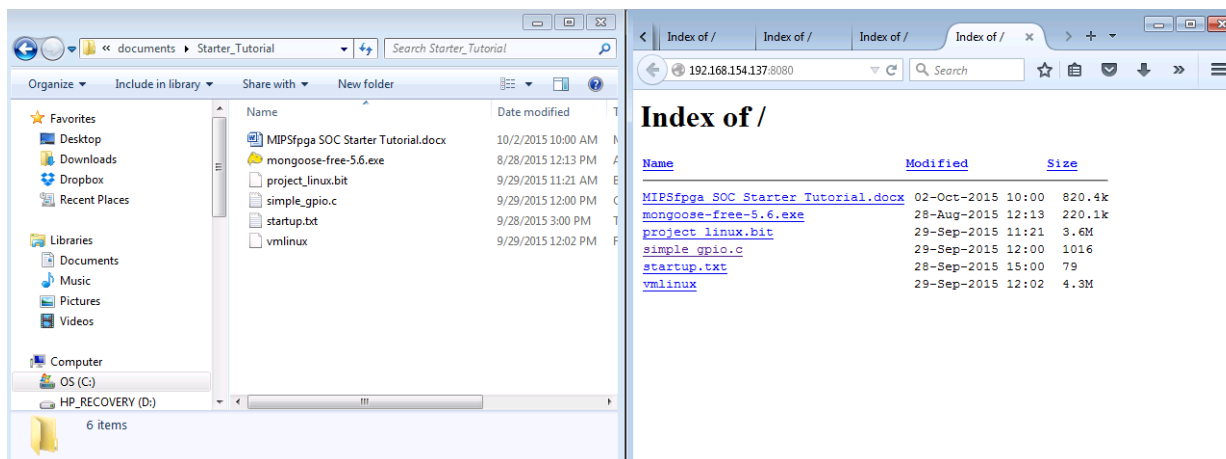


Figure 10 Running mongoose for a simple file server

You can now download the compiled simple\_gpio file to the MIPSfpga platform from Linux using the wget command. wget followed by the IP address shown in the Mongoose browser. For example, from Figure 10, you can see that our computer's IP address is 192.168.154.137. To download the file onto MIPSfpga, we type the following in the PuTTY terminal:

```
watch wget 192.168.154.137:8080/simple_gpio -c -T 3
```

**Warning:** *There is an issue with Ethernet. Ethernet packets are processed packet by packet. Hence, the download can time out, frequently depending on network congestion. We use 'watch' to retry the download every 2 seconds and -T 3 in 'wget' to timeout in 3 seconds. Press CTRL+C multiple times to exit when you see the download is done 100%.*

Note that during this process, Windows might pop up a window asking if you want to allow Mongoose access to some features. You do.

Linux file permissions are restrictive by default to prevent viruses. To enable simple\_gpio to become an executable, at the PuTTY prompt, type:

```
chmod +x simple_gpio
```

Then to run simple\_gpio as an executable use the './' prefix. I.e., at the PuTTY prompt, type:

```
./simple_gpio
```

This process can be seen in Figure 11. Each time you run the program 0xab is written to the eight left-most LEDs and the value of the eight switches is read and printed to the PuTTY terminal (prepended by the value of the switches, i.e., 0xab). You can toggle the switches and re-run the executable, as demonstrated in Figure 11

```
COM4 - PuTTY
Connecting to 192.168.154.137:8080 (192.168.154.137:8080)
simple_gpio      100% |*****|      543k --:--:-- ETA
Every 2s: wget 192.168.154.137:8080/simple_gpio -c -T 3      1970-01-01 00:38:28

^C^C
#
# chmod +x simple_gpio
# ./simple_gpio
Reading 0x10C0_0000 for reading Switches[15:8] 0000ab40
# ./simple_gpio
Reading 0x10C0_0000 for reading Switches[15:8] 0000ab60
# ./simple_gpio
Reading 0x10C0_0000 for reading Switches[15:8] 0000ab70
# ./simple_gpio
Reading 0x10C0_0000 for reading Switches[15:8] 0000ab78
# ./simple_gpio
Reading 0x10C0_0000 for reading Switches[15:8] 0000ab7c
#
```

**Figure 11** Downloading simple\_gpio and running it

## 11. Reading the onboard temperature sensor via sysfs

Sysfs is a virtual file system that allows access to low level hardware from a Linux terminal. The temperature sensor on the Nexys4 DDR board is located in the bottom right corner. The Linux kernel driver for the temperature sensor exports sysfs files to allow the user to read the temperature sensor value from the Linux terminal. The virtual file exporting the temperature sensor data is located in `/sys/class/hwmon/hwmon0/device/temp1_input`. We can read the file by using the built-in Linux `'cat'` command, which opens a target file, displays its contents on the terminal, and then closes the file. Figure 12 shows how to use the `cat` command to read the temperature sensor.

```
Welcome to MIPSfpga
mipsfpga login: root
# cat /sys/class/hwmon/hwmon0/device/temp1_input
31047
# cat /sys/class/hwmon/hwmon0/device/temp1_input
31742
```

**Figure 12. Reading the temperature sensor on the Nexys4 DDR board**

The output is in millidegree Celsius, i.e. about 31 degree Celsius for the example above.

## 12. Accessing general-purpose I/Os via sysfs

The Linux Kernel driver for the `axi_gpio` IP by Xilinx also exports sysfs files to enable the user to easily write/read to/from the general-purpose I/Os (GPIOs). The files are located in `/sys/class/gpio`. If we use `'ls'` to list all files there we see the following:

```
Welcome to MIPSfpga
mipsfpga login: root
# cd /sys/class/gpio/
# ls
export    gpiochip490  unexport
```

Linux assigns virtual GPIO numbers and abstracts GPIO controllers calling them `'chip'` controllers. `gpiochip490` represents the AXI GPIO hardware module. 490 is bit0, 491 is bit1, 492 is bit2 and so on. From Table 1, we see that bit 0 of the `axi_gpio` module is connected to switch 0 (SW[0]). This is virtual GPIO number 490. And bit 8 is connected to LED[0] which is virtual GPIO number 498. To access the GPIOs, we first need to export them by writing 490 and 498 in the export file.

This is done using the 'echo' command in Linux.

```
# echo 490 > export
# echo 498 > export
```

Run 'ls' again and you'll notice more directories.

```
# ls
export    gpio490   gpio498   gpiochip490 unexport
```

Run 'ls' inside the directories.

```
# ls gpio490
active_low device    direction subsystem uevent    value
```

'in' or 'out' can be written in the direction file using 'echo' to configure the GPIO as input or output.

'0' or '1' can be written to the value file if the GPIO is configured as an output. And the value file can be read if the GPIO is configured as an input. Let's configure GPIO 490 (SW[0]) as an input and 498 (LED[0]) as output. Then we will read/write their values. Type the following at the PuTTY command prompt.

```
# echo in > gpio490/direction
# cat gpio490/value
0
```

Now physically toggle Switch 0 on the Nexys4 DDR board. When you read the value of the switch again, as shown below, its value has changed – in our case from 0 to 1.

```
# cat gpio490/value
1
```

Now we can output a value to LED[0]. To turn on LED[0], type the following at the PuTTY command prompt:

```
# echo out > gpio498/direction
# echo 1 > gpio498/value
```

Notice that the right-most LED (LED[0]) on the Nexys4 DDR board is now on.

### 13. What to do next?

Go through the rest of the documentation in the documents folder.

Part 1 has several tutorials that build up the hardware RTL for MIPSfpga SOC from a simple LED blink example to the complete SOC design.

Part 2 has several tutorials that go through the Linux kernel and Buildroot port for the MIPSfpga SOC platform.

You could consider many aspects of MIPSfpga SOC, including the Verilog, Linux, assembly, bare-metal compiler, bootloader, debugger etc. You can explore and/or modify any of the pieces of interest. For example, some modifications include:

- Adding a custom peripheral IP block.
- Adding a crypto accelerator
- Poking in the cache to speed/slow things.
- Adding axi\_spi and the Nexys 4DDR accelerometer
- Porting u-boot or barebox or any other bootloader to the platform
- Porting the bsd plan9 kernel
- Pinging using bare metal code
- Making a debug core

Or, if you are really ambitious, you could connect a VGA monitor and keyboard to the Nexys4 DDR board, write the hardware RTL and software driver for it, and then try playing free doom.

