PROGRAMMING THE FPGA using BORPH.

Written By : Shri Irappa M. Halagali & Date : 16/06/2011.
M. Wagner, J. Manley and W. New (https://casper.berkeley.edu/wiki/Tutorials)

Note :	This colour	: used for Main titles.
	This colour	: used for sub/sub-sub titles.
	This colour	: used for command lines.
	This colour	: used for the response from the ROACH in minicom or from the PC.
	This colour	: used for procedure writeup/information/note.

1. Copy the bof file to be programed in the /srv/roachboot/etch_devel/boffiles area of the pc

eg.
```
gmrt@rchpc4:~/IRU/WORKSHOP/PROJECTs/tutorial1/bit_files$
    cp tutorial1_2011_Jun_09_1628.bof /srv/roachboot/etch_devel/boffiles
```

Note : All these cable connections and entries in the /etc/* files of the workshop PCs done.

2. Connect the Serial port cable between the ROACH board's P2 connector and serial port of the PC (on which minicom program exists).

3. Connect the Ethernet cable to J25 port of the ROACH board from the PCs eth1 port. /etc/ethers file should have mac address and corresponding ip address. In the /etc/network/interfaces file , eth1 should be configured. And in the file /etc/hosts , ip address and corresponding roach board(host) name entry to be done.

4. Start the minicom program. It is kept in the /usr/bin/.

eg.
```
[irappa@corrdevel3 ~]$minicom
```

Initializing Modem

```
Welcome to minicom 2.2

OPTIONS: I18n
Compiled on Mar  9 2007, 07:21:40.
Port /dev/ttyS0
        Press CTRL-A Z for help on special keys.
```

NOTE : Press CTRL + A and then Z ; to get Minicom command summary.
Then press W for Line Wrap ON/OFF. This enables us to see the messages from the ROACH board during BOOT-UP.

5. Switch ON the ROACH board.. Refer the the file "Roach_BOOT_proc1_V3.pdf" for complete boot procedure of the ROACH BOARDs. For this purpose it is not required to refer.

6. roach030172 login: root   # login as root w/o any password.

7. root@roach030172:~# cd /

8. root@roach030172:/# cd boffiles/

9. root@roach030172:/boffiles# ls

```
        a10fft7.bof
        a10fft8.bof
          ..........
          ..........
        tutorial1_2011_Jun_09_1628.bof
root@roach030172:/boffiles#
```

10. root@roach030172:/boffiles# ./tutorial1_2011_Jun_09_1628.bof &

```
[1] 230
root@roach030172:/boffiles#
```

Our FPGA is now programmed and we have our prompt back!

Look at the output ie LED on the ROACH board is blinking !!!

We can now see that a process in Linux has started...

11. root@roach030172:/boffiles# ps aux

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 284 0.1 0.0 0 0 ? SN 07:36 0:00 [jffs2_gcd_mtd3]
root 301 0.0 0.2 6700 1160 ? Ss 07:36 0:00 /usr/sbin/sshd
root 311 0.0 0.0 784 192 ? S 07:36 0:00 tcpborphserver
root 318 0.1 0.2 3772 1188 ttyS0 Ss 07:36 0:00 /bin/login -root
319 0.1 0.3 3516 1796 ttyS0 S+ 07:36 0:00 -bash
root 323 0.0 0.0 1632 304 ttyS0 S 07:36 0:00 ./tut1_2009_Aug_14_1140.bof
root 325 4.4 0.5 10000 2672 ? Ss 07:36 0:00 sshd: root@pts/0
root 328 0.8 0.3 3524 1800 pts/0 Ss 07:36 0:00 -bash
root 332 0.0 0.1 2780 996 pts/0 R+ 07:37 0:00 ps aux
root@roach030172:/boffiles#
```

Notice that PID 230 (yours may be different) is our process. We can now navigate to the proc directory which contains our software registers.

```
12. root@roach030172:/boffiles# cd /proc/230/hw/ioreg/ #Pl chage the PID nuber
13. root@roach030172:/proc/323/hw/ioreg# ls -al
total 0
dr-xr-xr-x 2 root root 0 Aug 21 08:00 .
drwxr-xr-x 2 root root 0 Aug 21 08:00 ..
-rw-rw-rw-1 root root 4 Aug 21 08:00 a
-rw-rw-rw-1 root root 4 Aug 21 08:00 b
-rw-rw-rw-1 root root 4 Aug 21 08:00 Counter_ctrl1
-r--r--r--1 root root 4 Aug 21 08:00 Counter_value
-r--r--r--1 root root 4 Aug 21 08:00 sum_a_b
-rw-rw-rw-1 root root 4 Aug 21 08:00 sys_board_id
-rw-rw-rw-1 root root 4 Aug 21 08:00 sys_clkcounter
-rw-rw-rw-1 root root 4 Aug 21 08:00 sys_rev
-rw-rw-rw-1 root root 4 Aug 21 08:00 sys_rev_rcs
-rw-rw-rw-1 root root 4 Aug 21 08:00 sys_scratchpad
root@roach030172:/proc/323/hw/ioreg#
```

Now you can see all our software registers. We have a, b, counter_ctrl, counter_value and sum_a_b as expected. However, in addition, the toolflow has automatically added a few other registers.

sys_board_id is simply a constant which allows software to identify what hardware platform is running. For ROACH, this is a constant 0xb00b001.

sys_clkcounter is a 32-bit counter that increments automatically on every FPGA clock tick. This allows software to estimate the FPGA's clock rate. Useful for debugging boards with bad clock inputs.

sys_rev is not yet implemented, but will eventually indicate the revision of the software toolchain that was used to compile the design

sys_rev_rcs is also not yet implemented, but will eventually indicate the SVN revision of the CASPER library that was used to compile your design.

sys_scratchpad is simply a read/write software register where you can write a register and read it back again as a sanity check.

Communicating with your FPGA process in BORPH

The registers contain binary data. To read and represent these on our text-based terminal, we will use a Linux utility called hexdump, which simply prints out the ASCII representation of hex values (base-16) in binary files. To write into these files, we'll use Linux's echo utility. Echo does not support writing hex values, but does support octal (base-8).

COUNTER

Let's start by having a look at our counter value. Since we haven't started it yet, we expect it to be zero.

```
1. root@roach030172:/proc/230/hw/ioreg# hd Counter_value
00000000 00 00 00 00 |....|
00000004
root@roach030172:/proc/230/hw/ioreg#
```

We notice that the register is indeed 32 bits long and that it contains the value zero. The column on the left tells us the memory addresses, while the four space separated values on the right give us the 4 byte (32 bit) value of the software register in hexadecimal. Right now, both registers report all zeros. According to our Simulink design, we can disable or enable the counter by setting cnt_en to 0 or 1 respectively.

Let's now start the counter and watch it increment.

```
2. root@roach030172:/proc/230/hw/ioreg# echo -n -e "\000\000\000\001" > Counter_ctrl1
```

Here we have told echo not to append a newline character to the end of the line (-n), and told it to interpret the incoming string's escape characters (\0 specifies octal values). Then we pipe it's output into counter_ctrl1. Now let's relook at the counter value...

```
3. root@roach030172:/proc/230/hw/ioreg# hd Counter_value
00000000 da 12 42 de       |..B.|
00000004

4. root@roach030172:/proc/230/hw/ioreg# hd Counter_value
00000000 dd 32 7c 3c       |..w.<|
00000004
```

" " " " " " " " "

You can see that the counter is indeed incrementing, and that it is happening very quickly (remember that it's incrementing by 100 million every second, since the FPGA is running at 100MHz).

0xda1242de = 14291522 in decimal.

0xdd327c3c = 3711073340 in decimal.

You should see the register values increasing until they reach 2^32-1 and then repeat. Resetting the counter has the desired effect...

```
5. root@roach030172:/proc/230/hw/ioreg# echo -n -e "\000\000\000\002" > Counter_ctrl1
```

```
6. root@roach030172:/proc/230/hw/ioreg# hd Counter_value
00000000 00 00 00 00 |....|
00000004
root@roach030172:/proc/230/hw/ioreg#
```

ADDER :

Let's now consider our adder: All registers initialise to zero upon startup, so we'd expect a and b to be zero now.

```
7. root@roach030172:/proc/230/hw/ioreg# hd a
00000000 00 00 00 00 |....|
00000004
```

```
8. root@roach030172:/proc/230/hw/ioreg# hd b
00000000 00 00 00 00 |....|
00000004
```

Indeed, this is the case. Let's write something in there now and have a look at the output. Let's add 5 and 12, so we expect 17. We use 0x to indicate hex, 0o for octal and 0b for binary. No prefix indicates decimal.

05 = 0o05 = 0x05

12 = 0o14 = 0x0c

17 = 0o21 = 0x11

Note : Inpu the data as OCTAL by using \0 eg \005 for decimal 5 & \014 for 12.

```
9. root@roach030172:/proc/230/hw/ioreg# echo -n -e "\000\000\000\005" > a

10. root@roach030172:/proc/230/hw/ioreg# echo -n -e "\000\000\000\014" > b

11. root@roach030172:/proc/323/hw/ioreg# hd a
00000000 00 00 00 05 |....|
00000004

12. root@roach030172:/proc/323/hw/ioreg# hd b
00000000 00 00 00 0c |....|
00000004

13. root@roach030172:/proc/323/hw/ioreg# hd sum_a_b
00000000 00 00 00 11 |....|
00000004
```

Note : Inpu the data as HEXADECIMAL by using \x. eg \005 for decimal 5 & \x0c for 12.

```
14. root@roach030172:/proc/230/hw/ioreg# echo -n -e "\000\000\000\x05" > a
15. root@roach030172:/proc/230/hw/ioreg# echo -n -e "\000\000\000\x0c" > b
16. root@roach030172:/proc/323/hw/ioreg# hd a
00000000 00 00 00 05 |....|
00000004
17. root@roach030172:/proc/323/hw/ioreg# hd b
00000000 00 00 00 0c |....|
00000004
18. root@roach030172:/proc/323/hw/ioreg# hd sum_a_b
00000000 00 00 00 11 |....|
00000004
root@roach030172:/proc/323/hw/ioreg#
```

Great! Exactly as expected.

This shows you a basic view of BORPH and interfacing to the proc files directly from the ROACH using Linux utilities. This method of accessing the shared memory and registers is good for quick verification that your design is running and loaded correctly, but for more advanced command, control and data acquisition, we recommend using the tcpborphserver and KATCP that starts up automatically when booting ROACH.

PROGRAMMING THE FPGA using KATCP.

KATCP is a process running on the ROACH boards which listens for TCP connections on port 7147 (tcpborphserver). It talks using machine-parseable ASCII text strings. It was designed this way so that it is easy to debug by watching the exchange of network traffic, while still being easy to program clients and servers.

1. Copy the bof file to be programed in the /srv/roachboot/etch_devel/boffiles area of the pc

eg.
```
gmrt@rchpc4:~/IRU/WORKSHOP/PROJECTs/tutorial1/bit_files$
    cp tutorial1_2011_Jun_09_1628.bof /srv/roachboot/etch_devel/boffiles
```

Note : All these cable connections and entries in the /etc/* files of the workshop PCs done.

2. Connect the Ethernet cable to J25 port of the ROACH board from the PCs eth1 port. /etc/ethers file should have mac address and corresponding ip address. In the /etc/network/interfaces file eth1 should be configured. And in the file /etc/hosts ip address and corresponding roach board(host) name entry to be done.

3. Switch ON the ROACH BOARD. Refer the file "Roach_BOOT_proc1_V3.pdf" for the complete boot procedure of the ROACH BOARDs.

4. From the PC connect to the ROACH via net using the command

`telnet <ROACH IP assigned> <port #>` note : port # is 7147 decided by protocol.

eg.
```
[irappa@corrdevel3 ~]$ telnet 192.168.100.72 7147
Trying 192.168.5.251...
Connected to 192.168.5.251.
Escape character is '^]'.
```
: To quit from this Press "CTRL"+ "]" (bracket]) then enter. And then quit from it.

```
#version poco-0.1
#build-state poco-0.0a271150300
```

5. `?listbof` : To see the BORPH (*.bof) files on the ROACH for programming FPGA. Pl. notice that commands are starting with ? in the front.

```
    a10fft7.bof
    a10fft8.bof
      ...........
    tutorial1_2011_Jun_09_1628.bof
!listbof ok 161
```

```
6.?progdev <borph file>           : This puts the bof program file in the FPGA.
        eg.  ?progdev tutorial1_2011_Jun_09_1628.bof
!progdev ok 231

7. ?listdev
#listdev sum_a_b
#listdev b
#listdev a
#listdev Counter_value
#listdev Counter_ctrl1
#listdev sys_clkcounter
#listdev sys_scratchpad
#listdev sys_rev_rcs
#listdev sys_rev
#listdev sys_board_id
!listdev ok
```

Here you can see we have the same list as we had before in BORPH.

Normally, machines using this interface would read and write to these registers using raw binary numbers using the read or write commands. For manual interaction, there are wordwrite and wordread commands which do the same with ASCII hex representations of 32-bit values. Let's try and add two numbers together now.

```
8. ?wordwrite a 0 0x02
!wordwrite ok
9. ?wordwrite b 0 0x07
!wordwrite ok
10. ?wordread sum_a_b 0
!wordread ok 0x9
```

You may be wondering what the extra zero in the arguments is for. This is the index offset. It is used when writing to blocks of memory, rather than software registers. For example, if you wanted to write a single 32 bit number into a 1GB DRAM memory chunk, at address 0x12808, you would say ? wordwrite <my_dram> 0x12808 <my_value>.

As you can see, the same FPGA functions that are available in BORPH are accessible through KATCP, with the difference that it can be configured remotely over a TCP network stream.

Conclusion

This concludes Tutorial 1. You have learnt how to constuct a simple Simulink design, transfer the files to a ROACH board and interact with it using BORPH and KATCP.

In Tutorial 2, you will learn how to use the 10GbE network interfaces and interact with your design using the KATCP Python client.