2010 CASPER Workshop

General ROACH Instructions

Author:

SKA SOUTH AFRICA
SQUARE KILOMETRE ARRAY

1 Transferring your bitstream to the ROACH board

Your compliation would have generated 2 bitstream files at *.bit and a *.bof file. You will find your bitstreams inside the folder called *bit_files*. There are two files, with the same name but different extensions. The filenames <design_name>_<start_compile_time>.XXX. Appending the start time guarantees that you will not overwrite existing bitstreams when recompiling your same design.

The *.bit* is the raw FPGA bitstream that you could load manually using a Xilinx JTAG programmer. Of interest to us now is the *.bof* file. This is a BORPH executable. We will transfer this to your ROACH board now.

To transfer files to ROACH, you can use any standard Linux scheme. We will use *WinSCP* which is a GUI program capable of doing *SCP* transfers (Secure Copy over SSH Protocol).

Use the scp command to transfer your file to the correct server/folder or under Windows you can use winscp.

You will then login to the Linux server ***wsserv*** which boots the ROACH boards and hosts their filesystem. The ROACH filesystem lives in */srv/roach_boot/etch/.*

Username: guest
Password: c45p3r!

In the root folder of this filesystem is a directory called *boffiles.* ROACH boards look in here for executable BORPH files. You will need to transfer your bof file into this directory and make it executable using chmo +x <bof_file_name>.

ie. scp somename.bof tutee@otto:/home/nfs/current/boffiles

2 Connecting to your ROACH board

2.1 Booting your ROACH

To watch ROACH boot, we can connect to it using a serial port. Double-click on *roach_serial.ht* on your desktop. This will open HyperTerminal (a serial terminal program) and set it to connect to *com1* at 115200 baud, 8N1 with no flow control.

Turn the ROACH on.

You will be greeted with a printout similar to this one:

```
U-Boot 2008.10-svn2205 (Aug  3 2009 - 09:31:45)



CPU:   AMCC PowerPC 440EPx Rev. A at 495 MHz (PLB=165, OPB=82, EBC=82 MHz)
       No Security/Kasumi support
       Bootstrap Option H - Boot ROM Location I2C (Addr 0x52)
       32 kB I-Cache 32 kB D-Cache
Board: Roach
I2C:   ready
DTT:   1 is 28 C
DRAM:  (spd v1.2) 512 MB
FLASH: 64 MB
USB:   Host(int phy) Device(ext phy)
Net:   ppc_4xx_eth0

Roach Information
Serial Number:    020112
Monitor Revision: 7.3.0
CPLD Revision:    7.5.1556

type run netboot to boot via dhcp+tftp+nfs
type run soloboot to run from flash without network
type run usbboot to boot using filesystem on usb
type run mmcboot to boot using filesystem on MMC or SD card
type run bit to run tests

Hit any key to stop autoboot: 10
```

This is ROACH's version of a BIOS along with some basic PPC health printouts. Press any key here to stop the automatic boot process. This is the state of the ROACH boards when shipping. You can configure it differently should you please.

You have four boot options: network, onboard flash, USB or MMC/SD. See the ROACH getting started guide on the CASPER wiki for details of these options. If you do not interrupt it, the default is to boot off FLASH. Our filesystem is on a network computer, so type *run netboot* to initiate a network boot.

It will acquire an IP address along with a bunch of DHCP options which define where to find the filesystem. The lab's server is pre-configured to provide this info.

After Linux has completed its boot process, you will be greeted with a Debian prompt.

```
Debian GNU/Linux 4
.0 roach020112 ttyS0

roach020112 login:
```

You can connect to your ROACH board by SSH'ing directly into it. Windows does not include an SSH client natively, but you can use Putty. From linux just run:

ssh root@roach<SerialNumber>

You will be greeted with a dialog box which will ask you for the hostname. Our server assigns roaches IP addresses and hostnames based on the serial number. Complete ***roach<SerialNumber>***. There is a sticker on the front of your ROACH with its 6-digit serial number. For example, if it is 020112, then complete *roach020112.* Leave everything else default. Click *Open*.

There is no root password required on the roach.

```
root@roach020112:~#
```

We are now ready to start working with the ROACH board.

3 Communicating directly using BORPH

3.1 Programming the FPGA from BORPH

To program the FPGA, simply execute it as you would any other linux program. Recall that the filesystem is mapped from the server and the BORPH files were in */boffiles/*.

Let's change to that directory now and see which files are available:

```
root@roach020112:~# cd /boffiles/
root@roach020112:/boffiles# ls -al
total 8196
drwxrwxrwx  2 root root    4096 Aug 20  2009 .
drwxr-xr-x 23 root root    4096 Feb  2  2009 ..
-rwxr-xr-x  1 1000 1000 3894183 Aug 20  2009 das_blinken_lichte_2009_Feb_04_1837.bof
-rw-r--r--  1 1001 1001 4468934 Aug 14  2009 tut1_2009_Aug_14_1140.bof
root@roach020112:/boffiles#
```

We need to ensure that the file is marked as an executable, so that Linux is able to run it. You will notice that our tut1 file is not executable (look at the leftmost columns and see that there are no 'x's). Let's make it executable now:

```
root@roach020112:/boffiles# chmod a+x tut1_2009_Aug_14_1140.bof
root@roach020112:/boffiles#
```

This will change permissions on the file, by adding executable permissions for everybody. Rechecking:

```
root@roach020112:/boffiles# ls -al
total 8196
drwxrwxrwx  2 root root    4096 Aug 20  2009 .
drwxr-xr-x 23 root root    4096 Feb  2  2009 ..
-rwxr-xr-x  1 1000 1000 3894183 Aug 20  2009 das_blinken_lichte_2009_Feb_04_1837.bof
-rwxr-xr-x  1 1001 1001 4468934 Aug 14  2009 tut1_2009_Aug_14_1140.bof
root@roach020112:/boffiles#
```

Now we are ready to execute it.

```
root@roach020112:/boffiles# ./tut1_2009_Aug_14_1140.bof
```

This will go'n program the FPGA, and you should notice that LED0 on the ROACH hardware is now blinking. However, we've lost our terminal, because the process has captured it. To keep the process running, but return the prompt to us, we can now choose to background the task (ctrl-z), or we can kill it (ctrl-c) and restart it with an amperstand, which will start it backgrounded.

```
root@roach020112:/boffiles# ./tut1_2009_Aug_14_1140.bof &
[1] 323
root@roach020112:/boffiles#
```

Our FPGA is now programmed (check for the flashing LED) and we have our prompt back!

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

```
root@roach020112:/boffiles# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND

<snip>

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@roach020112:/boffiles#
```

Notice that PID 323 (yours may be different) is our process. We can now navigate to the *proc* directory which contains our software registers.

```
root@roach020112:/boffiles# cd /proc/323/hw/ioreg/
root@roach020112:/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_ctrl
-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@roach020112:/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 expect. 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_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.

3.2 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).

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

```
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  00 00 00 00                                       |....|
00000004
root@roach020112:/proc/323/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.

```
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\001" > counter_ctrl
```

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_ctrl.* Now let's relook at the counter value...

```
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  da 12 42 de                                       |..B.|
```

```
00000004
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  dd 32 7c 3c                                       |.2|<|
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...

```
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\002" > counter_ctrl
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  00 00 00 00                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg#
```

For you to try: what happens if you try to reset and enable the counter at the same time? Try it in simulation and on the hardware. Do they do the same thing? Is this what you expect?

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

```
root@roach020112:/proc/323/hw/ioreg# hd a
00000000  00 00 00 00                                       |....|
00000004
root@roach020112:/proc/323/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
```

```
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\005" > a
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\014" > b
root@roach020112:/proc/323/hw/ioreg# hd a
00000000  00 00 00 05                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg# hd b
00000000  00 00 00 0c                                       |....|
```

```
00000004
root@roach020112:/proc/323/hw/ioreg# hd sum_a_b
00000000  00 00 00 11                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg#
```

Great! Exactly as expected.

For you to try: What happens if you try to add two very large numbers (try to make the 32-bit register overflow).

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.

4 Interacting 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, whilst still being easy to program clients and servers.

4.1 Connecting to your ROACH

Telnet allows you to connect directly to an open TCP port. Let's connect to ROACH's KATCP port now.

Telnet roach030181 7147

You will be greeted with the KATCP welcome header.

```
#version tcpborphserver-1.0
#build-state tcpborphserver-0.
#log info 1250851713365 tcpborphserver
new\_connection\_192.168.14.1:53088\_to\_192.168.14.112:7147
```

From here you can type KATCP commands. A full list, reference guide and specification sheet can be found on the CASPER wiki at http://casper.berkeley.edu/wiki/KATCP.

To summarize:
•All commands must be preceded by a *?,* and submitted with a *CR* or *LF.*
•All commands will be acknowledged with a response, preceded by a *!.*
•Multi-line responses will be preceded by a # on each line, followed by the standard acknowledge response.
•Spaces are considered argument delimiters. Arguments with spaces are escaped using _.

Let's look at some examples:

To get online help and see a list of commands available, type *?help* and press *enter*.

```
?help
#help tap-stop stop\_the\_tap\_server\_(?tap-stop)
#help tap-start start\_the\_tap\_server\_(?tap-start\_10gbe-register-name...
#help echotest basic\_network\_echo\_tester\_(?echotest\_ip-address\_ip-p...
#help wordwrite writes\_data\_words\_to\_a\_named\_register
#help indexwrite writes\_arbitrary\_data\_lengths\_to\_a\_numbered\_register
#help write writes\_arbitrary\_data\_lengths\_to\_a\_named\_regi...
#help wordread reads\_data\_words\_from\_named\_a\_register
#help indexread reads\_arbitrary\_data\_lengths\_from\_a\_numbered\_register
#help read reads\_arg3\_bytes\_starting\_at\_arg2\_offset\_from\_register\_arg1\...
#help listdev displays\_available\_device\_registers
#help listcmd displays\_available\_shell\_commands
#help listbof displays\_available\_images
#help status displays\_image\_status\_information
#help progdev programs\_an\_image\_(?progdev\_[boffile])
#help sensor-sampling sets\_the\_sensor\_reporting\_mode
#help sensor-list lists\_available\_sensors
#help watchdog pings\_the\_system
#help log-level sets\_the\_minimum\_reported\_log\_priority
#help help displays\_this\_help
#help restart restarts\_the\_system
#help halt shuts\_the\_system\_down
!help ok 21
```

Here you can see what a multiline response looks like. Although a little hard to read by us humans, this is easy to parse by a machine. We have clients available for decoding this text and constructing commands automatically. They will be demonstrated in Tutorial 2. For now, let's look at constructing some commands by hand.

To get a list of bof files available, type *?listbof.*

```
?listbof
#listbof das_blinken_lichte_2009_Feb_04_1837.bof
#listbof tut1_2009_Aug_14_1140.bof
!listbof ok
```

To program the FPGA with one of these bitstreams, type *?progdev <my_boffile>.*

```
?progdev tut1_2009_Aug_14_1140.bof
!progdev ok
```

You will notice that commands are ignored if you don't include the ? in the front.

To list the registers available to you, type *?listdev*.

```
!progdev ok
?listdev
#listdev sum_a_b
#listdev counter_value
#listdev counter_ctrl
#listdev b
#listdev a
#listdev sys_clkfreq
#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.

```
?wordwrite a 0 0x02
!wordwrite ok
?wordwrite b 0 0x07
!wordwrite ok
?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.

5 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.