Table of contents

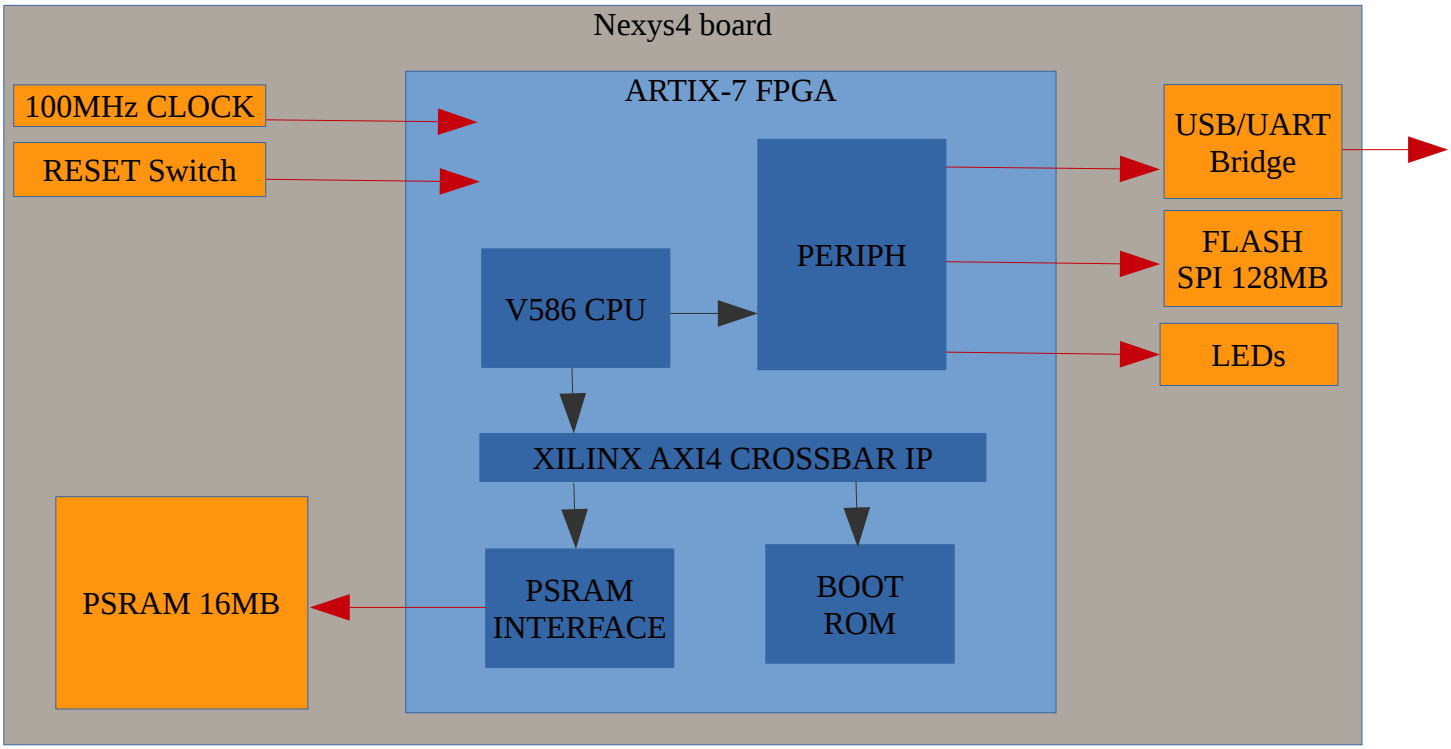
1. OVERVIEW

The v586 is made of a CPU that executes 586 opcodes but also a system around that brings the minimum vital functions to boot Linux in text mode.
This system is meant to be used with DIGILENT NEXYS4 board with ARTIX-7 fpga.

In a nutshell the system has to be seen in 2 parts :
a) is written in VERILOG , it is the CPU plus the interfaces → inside the FPGA
b) the components outside the FPGA that are driven by interfaces : SD connector, SPI flash, UART, CLOCK input, switches and Leds, and many more as described in the NEXYS4 board manual.

1.1 Block diagram of v586 system board with nexys4 board

In grey, the arrows represent an AXI4 protocol interface, in red the connections are not AXI4 but off-FPGA connections to NEXYS4 components like UART, SPI, PSRAM and Leds.



All what is inside ARTIX7 (in blue) is verilog code that has to be compiled with Virtuoso.

All what is in RED arrows are off-fpga connections and they have to be coherent with DIGILENT specification and signal routing in the PCB as it is described in NEXYS4 board manual.

The FLASH SPI 128MB will play an essential role in the system boot and configuration.

The system has 16MByte of RAM, this is an external chip. This chip can be accessed with conventional SRAM signaling or Synchronous SRAM (with a clock) – i.e. this design will not work on NEXYS4 DDR unless the PSRAM interface in changed by a AXI4/DDR interface.

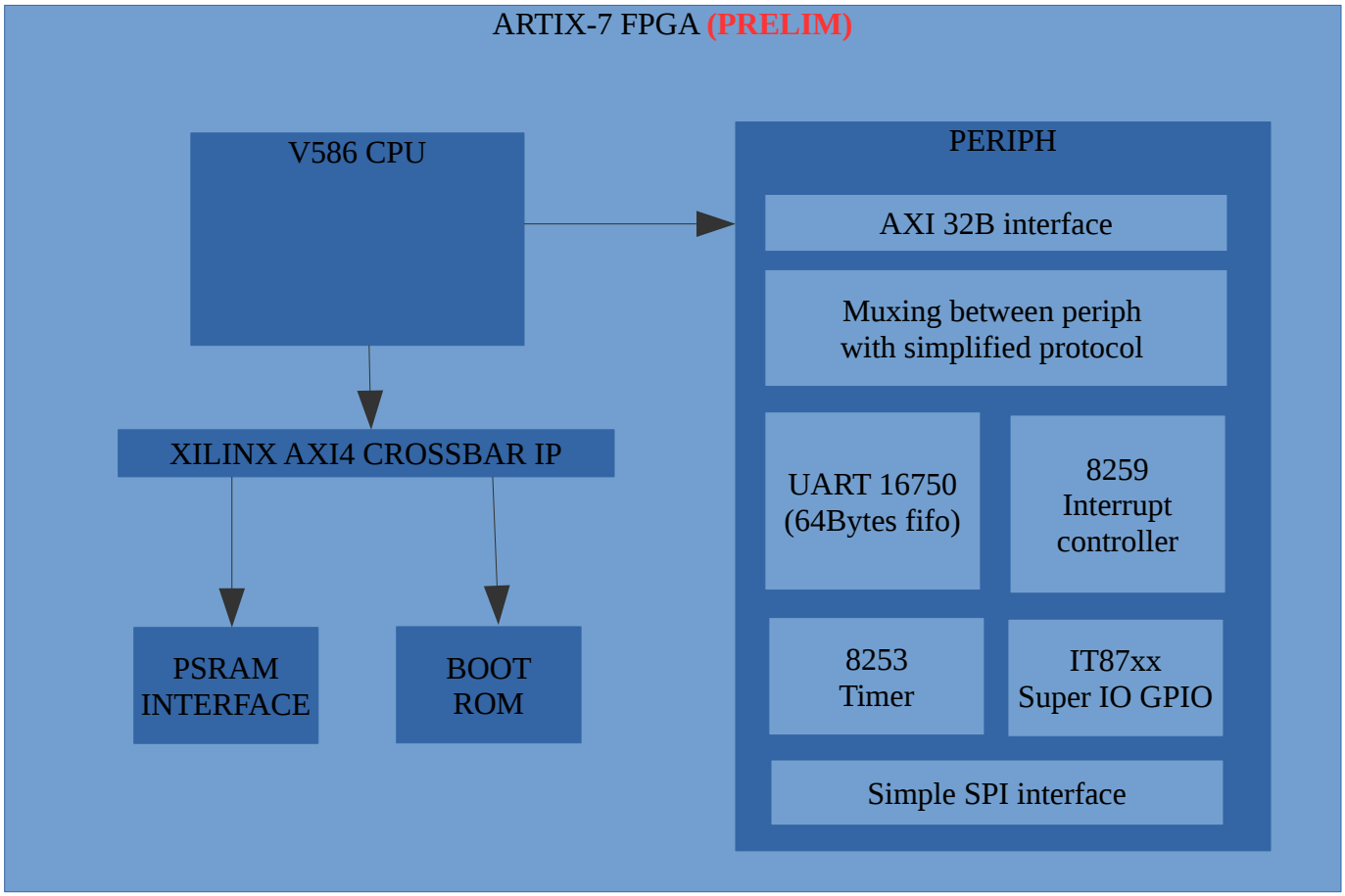
The FPGA contains RAM inside also but it is way to small to run linux by several order. Some micro-controller projects are designed to use very small RTOS (few kBytes) and these projects can be run only with FPGA ressources. Which is not the case here since we target Linux.

1.2 Block diagram inside artix7-100 fpga

The system inside the FPGA has a central processor that communicates with the rest of the world with 4 types of interfaces :
a) clock and reset inputs
b) memory map through an AXI4 32bits R/W with burst for unaligned access and cache refill.
c) io map through an AXI4 32bits , no burst.

Note: x86's unlike RISCs like ARM have a memory space plus an IO space. Nowadays a peripheral for x86 can be memory mapped (like ARM) or io mapped. Most legacy functions like interrupt controller and timers are usually found in the io space for historical reasons.

d) interrupt interface , with acknowledge and vector input.



Inside the Periph block we translate AXI protocol into a simplified chipselect/write enable protocol.

IO RANGE ADRESS	PERIPH NAME	Comments
0x3F8-0x3FF	UART 16750	Uses IRQ4
0xA0-0xA1 0x20-0x21	Primary & Secondary 8259	
0x40-0x43	Timer 8253	
0x2E-0x2F 0x290-0x291	IT87xx (only GPIO part)	see IT87xx datasheet
0x500-0x504	Simplified SPI interface	Only to read SPI flash

Table 1: IO Map with address ranges for the system

The memory region contains 16MB or sram and 512 bytes of ROM.
The ROM is inside the FPGA and the RAM access are going to the PSRAM interface.

MEMORY RANGE ADDRESS	REGION NAME	Comments
0x0-0x090000	RAM LOW LEGACY	640k dos area
0x090000-0x0f0000	RAM LOW NOT LEGACY	Normally old PC wouldn't have RAM there but video controller and other periphs. Not clear if linux by default will use this amount of RAM.
0x0f0000-0x0fffff	Boot ROM	
0x0100000-0x0BFFFFF	HIGH MEM above 1MB	Rest of 15MBytes

Note: old PC have a special region betwenn 640k and 1MB for video buffer and bios but here the “hole” is way smaller to hold the small boot code.

2. Boot up sequence information

In this section we will find explanation of the boot code inside the FPGA , how the SPI is expected to be organised by the boot code, some hardware initialization and finaly the linux configuration.

2.1 boot code inside FPGA

this section gives some hints on the boot code found in the test.s file

```
.code32
/* start protected mode , no more CS/DS prefix */
start:
movl $0x01,%eax
movl %eax , %cr0                    ----------> This section of code sets protected mode as per x86-32 manual
.code32
ljmp $0x0 , $0x0ffc20
.org 0x020

/* select boot type */
movl $0x500, %edx
inb (%dx) , %al
and $3 , %al
cmp $1 , %al
jz boot_test              -----> this section tests of SW1and SW2 of the board to see if it copies first the spi to boot linux or not.
cmp $2 , %al
jz boot_ram
cmp $3 , %al
jz boot_spi
jmp boot_linux

boot_spi:
movl $0x1000, %esp    → initialize the stack pointer to some value and also initialize the UART for 115200 bauds , start/stop bits.
call init_uart

call banner               ----> print something on the uart  to show that everything went ok so far, and the uart is working ok

mov $6,%al
mov $0x500,%edx
out %al,(%dx)
mov $2,%al
mov $0x500,%edx
out %al,(%dx)
movb $0x03,%bl
call send8b_spi       ------> sends some commands to the SPI flash on 0x500 port to initialize SPI flash.
movb $0x3F,%bl
call send8b_spi
movb $0xFF,%bl
call send8b_spi
movb $0xF0,%bl
```

```
call send8b_spi

mov $0x0FFFF0,%edi
mov $0x0c0000,%esi     ----> copy 3MByte from SPI , this is the vmlinux.bin kernel as compiled for x86 target, see more details in the embedded sw
call fill_spi

call banner    ----> print something

mov $6,%al
mov $0x500,%edx
out %al,(%dx)
mov $2,%al
mov $0x500,%edx
out %al,(%dx)
movb $0x03,%bl
call send8b_spi
movb $0x7F,%bl
call send8b_spi                    -----> transfers also 2Mbyte from SPI at for the initial ramdisk as compiled by buildroot , more details in sw section
movb $0xFF,%bl
call send8b_spi
movb $0xF0,%bl
call send8b_spi

mov $0x3FFFF0,%edi
mov $0x080000,%esi
call fill_spi

call banner

jmp boot_linux   -----> jump to configure &prepare linux boot section

fill_spi:
call recv32b_spi
mov %ebx ,%eax
rol $8,%eax
mov %al , (%edi)
inc %edi
rol $8,%eax
mov %al , (%edi)
inc %edi
rol $8,%eax
mov %al , (%edi)
inc %edi
rol $8,%eax
mov %al , (%edi)
mov (%edi), %bl
cmp %al , %bl
jz okpass
push edi
push esi
call banner
pop esi
pop edi
okpass:
inc %edi
dec %esi
jnz fill_spi
ret


// send %bl to spi , msb first
send8b_spi:
movw $0x500,%dx
movb $8,%cl
rol $1,%bl
nextbit:
mov %bl , %al
and $1, %al
outb %al, (%dx)
or $2, %al
outb %al, (%dx)
xor $2, %al
outb %al, (%dx)
rol $1,%bl
dec %cl
jnz nextbit
ret

//init spi
```

```
mov $6,%al
mov $0x500,%edx
out %al,(%dx)
mov $2,%al
mov $0x500,%edx
out %al,(%dx)
mov $0xF0,%bl
call sen8b_spi
mov $6,%al
mov $0x500,%edx
out %al,(%dx)
ret


// receive spi to %ebx
recv32b_spi:
movw $0x504,%dx
movb $32,%al
outb %al, (%dx)
mov $30,%ecx
waitloop:
dec %ecx
jnz waitloop
in (%dx), %eax
in (%dx), %eax
in (%dx), %eax
mov %eax,%ebx
ret
```

```
boot_linux:                    ----------> configure linux by wrtting several value into “magic” ram location in the 0x90000 ram section
movl $0x1000, %esp

call init_uart
/* setup ebda ptr at 0x40e*/
movl $0x0fff00 , %ebx
movl $0x040e , %ecx
mov %ebx , (%ecx)

/* eax = ram size */
/* ebx = ramd size */
/* ecx = ptr to cmdline */

mov $0x90000, %edi
mov $0x400 , %ecx
mov $0 , %eax
rep
stosl

/* command line */
mov $0x90800, %edi
mov %edi , 0x90228
mov $0xfff20, %esi
mov $0x100 , %ecx
rep
movsb

/* loader type */
mov $1, %al
mov  %eax , 0x90210

/* mem size */
movl $0x003c00 , %eax
mov  %eax , 0x901e0

/* initrd start */
mov  $0x00400000 , %eax
/* mov  $0, %eax */
mov  %eax , 0x90218

/* initrd size */
movl $0x00200000 , %eax
/* movl $501047 , %eax */
/* movl $0 , %eax */
mov  %eax , 0x9021c

/* row cols */
mov $80,%al
mov %al,0x90007
mov $25,%al
```

```
mov %al,0x9000e

call banner
```

```
movl $0x00090000, %esi
ljmp $0x10, $0x00100000     ----------> JUMP to LINUX KERNEL as copied from spi flash , and that is the END of boot INIT.
```

```
boot_test:
mov $0x1000,%esp
call init_uart

mov $0 , %bl
loopboot:
call sendchar
incb %bl
jmp loopboot

sendchar:
push %eax
push %edx
/* wait if there is character to be sent */
wait_rdy:
movl $0x3fd, %edx
in (%dx),%al
andb $0x20,%al
jz wait_rdy
movl $0x3f8, %edx
mov %bl, %al
outb %al, (%dx)
pop %edx
pop %eax
ret

init_uart:
/* set 8N1 flow dlab =1*/
movl $0x3fb, %edx
movb $0x83 , %al
outb %al , (%dx)

/* set DLL divisor 1 = 115200 bauds , 2= 57600 bauds , ...*/
movl $0x3f8, %edx
movb $1 ,%al
outb %al , (%dx)
movl $0x3f9, %edx
movb $0 ,%al
outb %al , (%dx)

/* set 8N1 flow dlab=0*/
movl $0x3fb, %edx
movb $0x3 , %al
outb %al , (%dx)

/* disable fifo*/
movl $0x3fa, %edx
movb $0x7 , %al
outb %al , (%dx)

/*  */
movb $0 ,%al
movl $0x3f9, %edx
outb %al , (%dx)
movl $0x3fc, %edx
outb %al , (%dx)
/*  test char */
movl $0x3f8, %edx
ret

boot_ram:
mov $200,%ecx
mov $aabb1122,%ebx
mov %ebx , %eax
mov %ebx , (%ecx)
mov $0 , %ebx
mov (%ecx), %ebx
cmp %ebx , %eax
jz testok
movl $0x3f8, %edx
movb $0x41 , %al
addb %bl , %al
```

```
outb %al, (%dx)
jmp final

testok:
movl $0x3f8, %edx
movb $0x42 , %al
addb %bl , %al
outb %al, (%dx)

banner:
mov $0xfffb0, %esi
banner_loop:
movb (%esi),%bl
mov $0,%al
cmp %al,%bl
jz exit_banner
inc %esi
call sendchar
jmp banner_loop
exit_banner:
ret

final:
jmp final

/*  ebda */
.org 0x0300
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0

/*  cmdline */
.org 0x0320
.asciz "console=ttyS0,115200n8 root=/dev/ram0 rw"

/* banner */
.org 0x03b0
.ascii "Boot copy flash"
.byte 10
.byte 13
.byte 0

/*  init jump bios */
.org 0x3d0
.code16
start2:
jmp start

.org 0x3f0
.code16
jmp start2
```

2.2 external spi flash organisation

ADRESS RANGE INSIDE EXTERNAL SPI	SECTION NAME
0x0	FPGA BIT FILE FROM VIVADO
0x400000 (4MB limit)	vmlinux.bin , the linux kernel uncompressed and stripped off from ELF information
0x800000 (8MB limit)	Initramfs.cpio.gz , the compressed cpio.gz initram file from buildroot.

Note : in order to have the FPGA automatically load the FPGA BIT file from SPI , we need to put jumper on the board on the “QSPI” position for “MODE”

2.3 hardware init
Most is explained inside the boot code description

2.4 linux kernel configuration and launch
Most is explained inside the boot code description.
We can explain here that the Linux kernel is configured with 2 inputs:
a) several value in the 0x90000 region , like ram size information
b) a string (with terminal 0) , also location in the 0x90000 region , that is the BOOT KERNEL CONFIGURATION , it is found inside the boot and is equal to :

console=ttyS0,115200n8 root=/dev/ram0 rw

This line syntax is pure Linux convention , in case of troubleshooting it is useful to add debug to have more verbosity on linux boot messages :

console=ttyS0,115200n8 root=/dev/ram0 rw debug

3. Embedded Software

We have already seen that the boot code will copy some sections of the the external spi flash into ram , the kernel and the ramdisk. The boot romm will then jump to first address of the vmlinux kernel.
Here is how to generate the 2 files : the kernel and initial ramdisk.

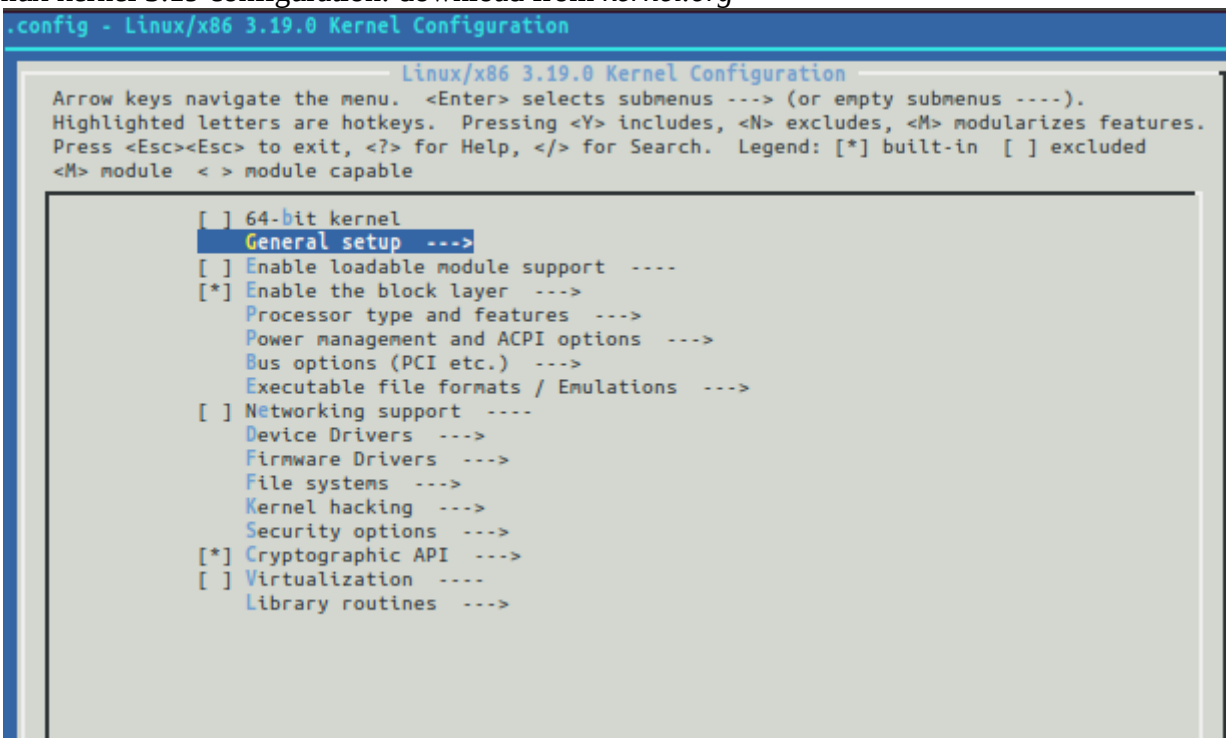
3.1 Linux configuration for general features : cpu & legacy devices

Typically for these project we target the smallest linux kernel , the default configuration produces large kernel of several Mbytes, and we have only 16MByte in the board. The target is to maintain the kernel in the 1.7 to 2.5Mbytes. By experience , with these sizes we can have lot of funtionality.

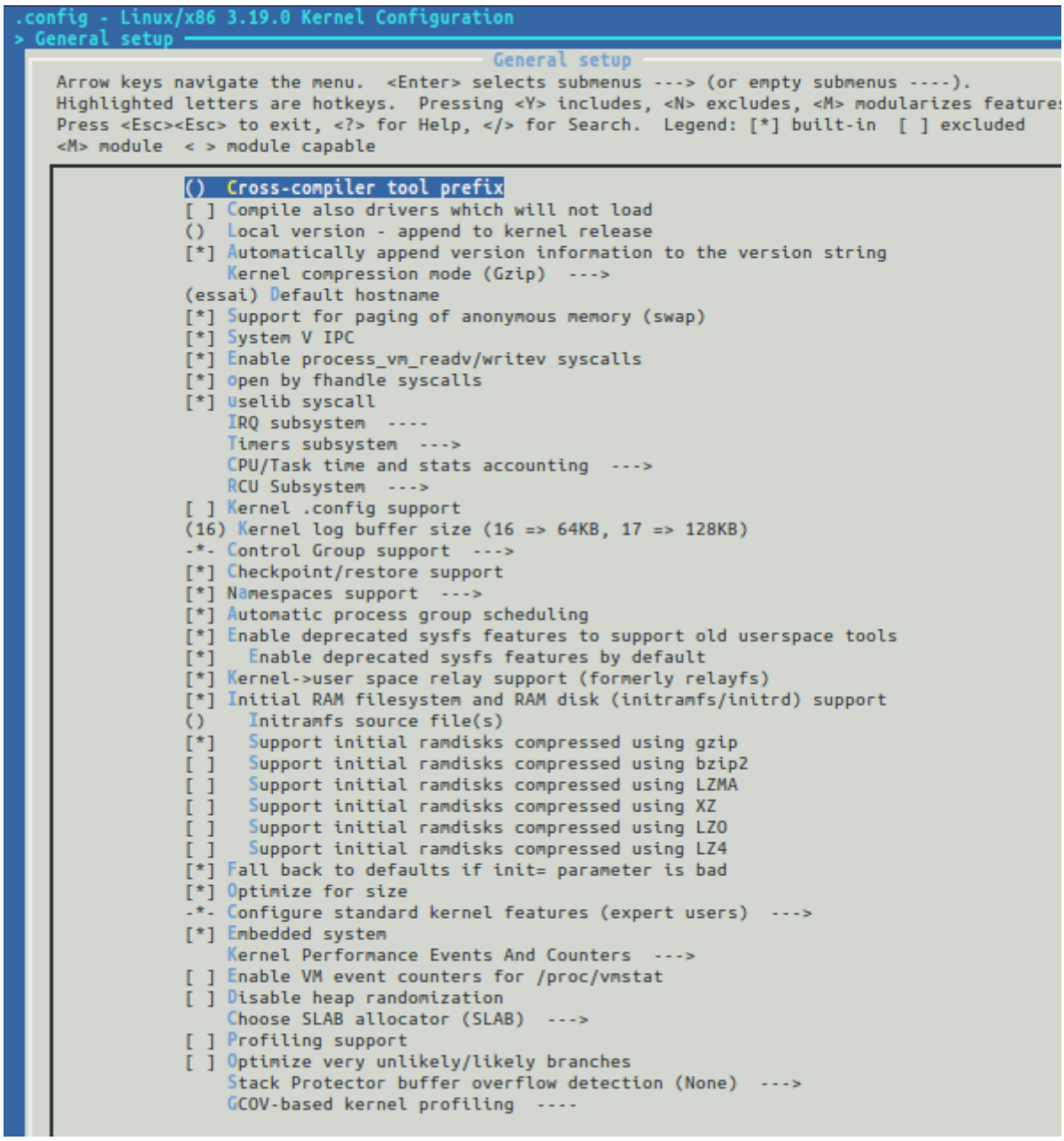
The next task will also to add the right drivers and devices for the board peripherals.

Note: The GPIOs with IT87xx chip is very powerful since we can “BITBANG” several protocol throught GPIOs to access board ressources such as accelerometer and SD card.

Linux kernel 3.19 configuration: download from *kernel.org*

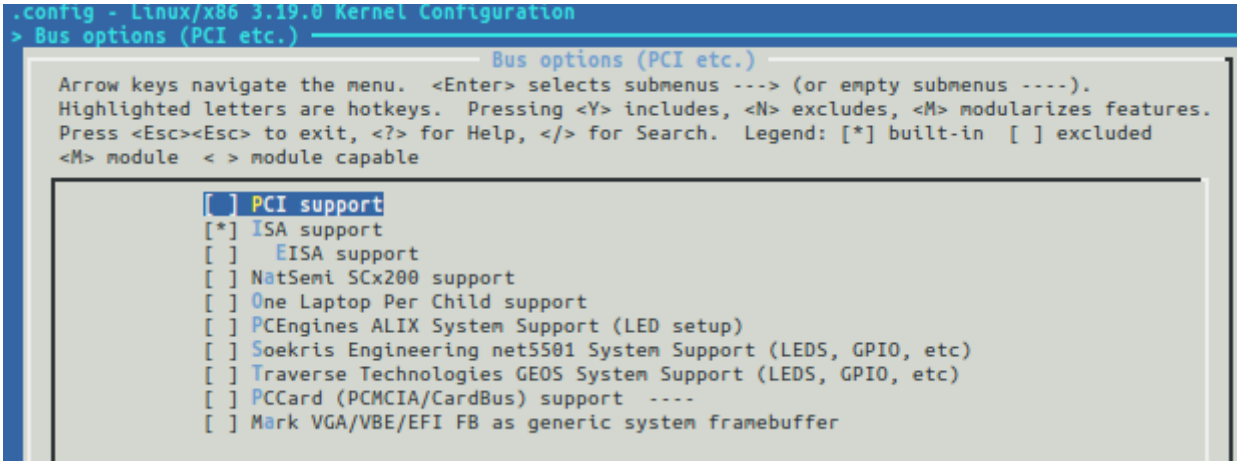


a)We need to select block layer for the SD card support. It is a block device.

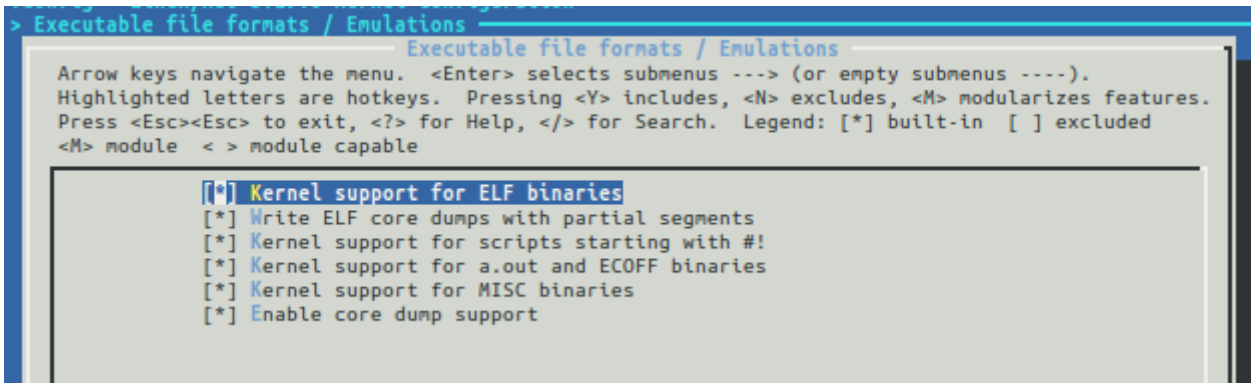


b)we select initial ramdisk support and optimize for size and compressed initramfs with gzip.

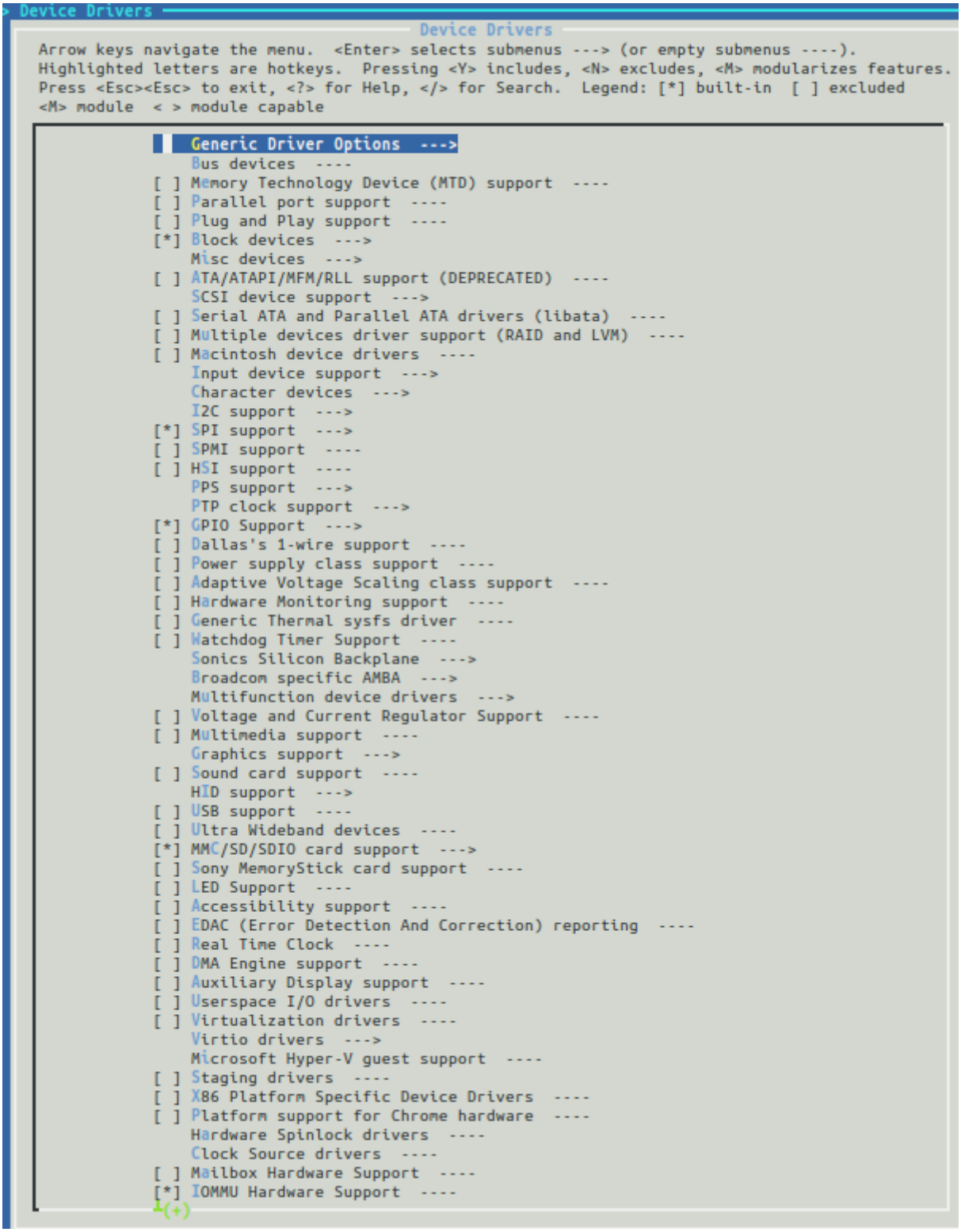
c) in processor type we select a low frequency for timer , 100Hz, and it is essentiam to select “math emulation” and also to set the start address for kernel to 0x100000.
The gpio mmc v586 option is not necessary .



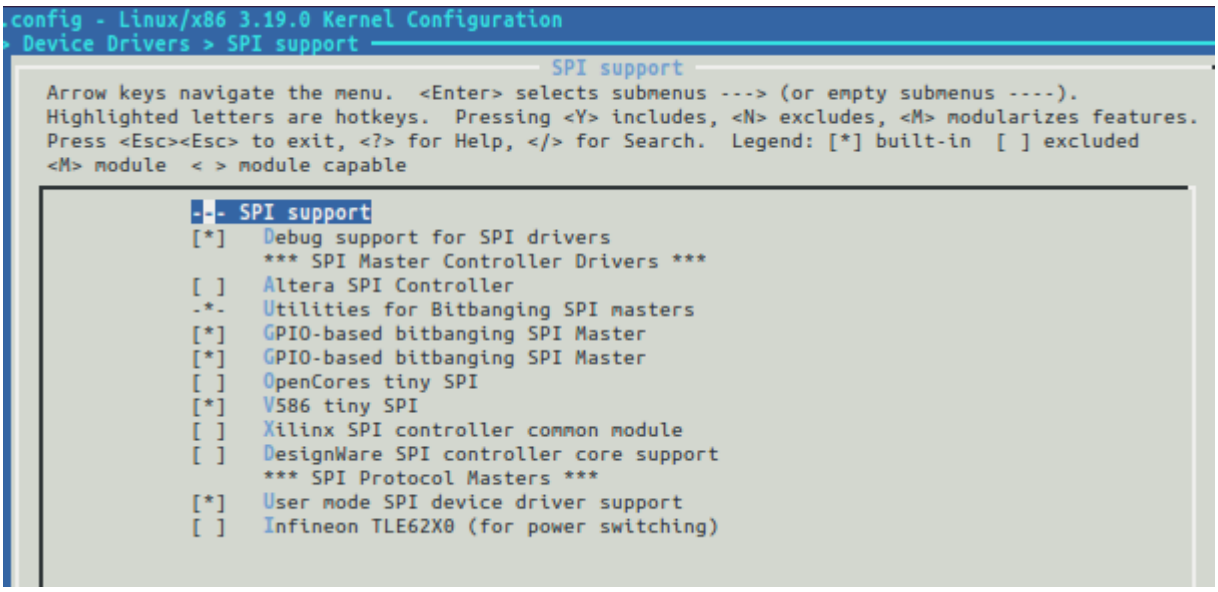
d) for BUS options we only need ISA , we don't have PCI hardware of video card in 0xA000 segment.



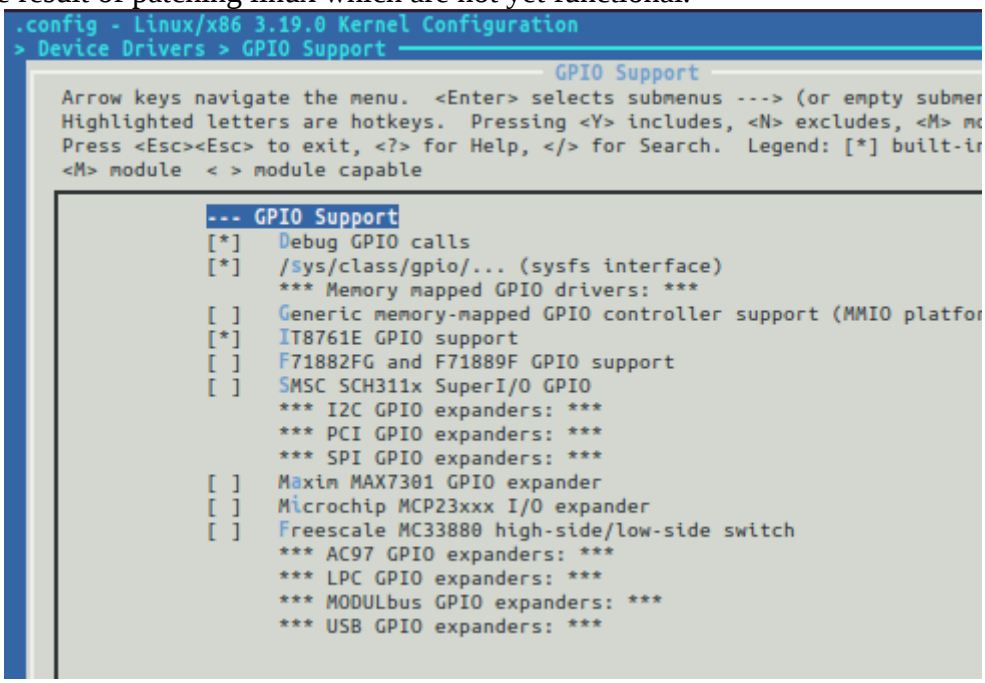
e) support for all type of binaries , especially the scripts starting with magic codes “#!”.



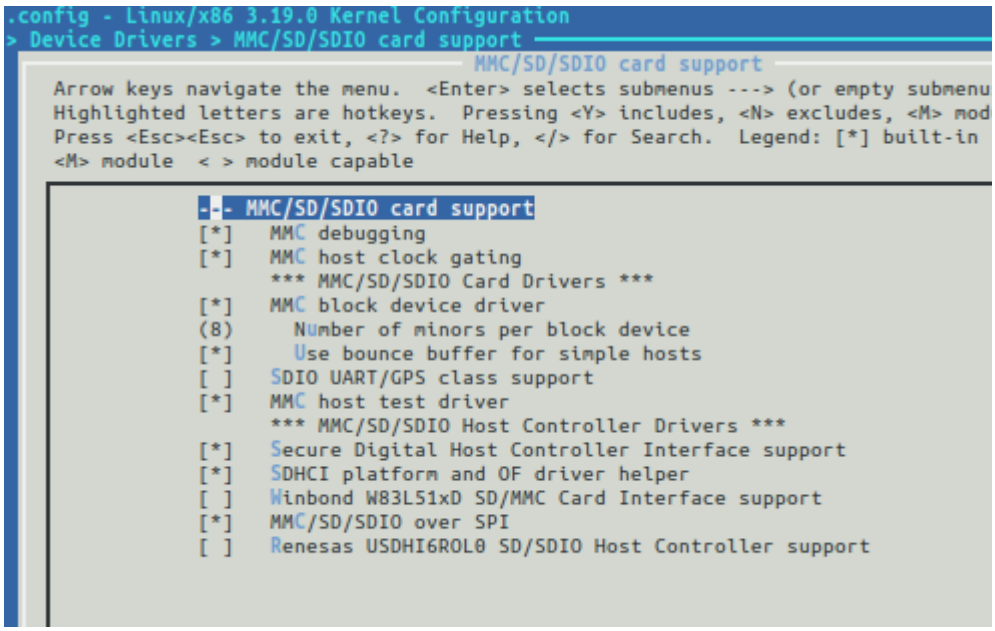
f) device driver is key here , select gpio /SD and block devices as well as SPI. All this to bitbang SD card throught GPIOs.



g) for SPI the choice GPIO_based bitbanging SPI master is the one. The others choices are just for debug or the result of patching linux which are not yet functional.



h) For the GPIO IT87xx support is the one , /sys/class/gpio is neat to control GPIOs from user space with configfs method.
With IT87xx we have 16 GPIOs number under linux from 496 to 496+15.
bank A is 8 GPIOS from 496 to 496+7
bank B is 8 GPIOS from 496+8 to 496+15



i) MMC over SPI is the choice to take. So SD card we need a device driver for IT87xx GPIOS, a SPI – GPIO bitbanging master and a MMC/SD over SPI stack. But it is not enough to get SD working we need also to declare DEVICEs , the stack is only for DRIVERs.

3.2 Linux device description for the board spi components

We have now configured Linux , but it is not enough , we need also to create a “board specific” description file that will be placed inside the linux sources.

Inside ./arch/x86/platform/v586/v586.c

Also we need to declare and include this file as part of the Linux kernel by editing the
./arch/x86/platform/v586/Kconfig
and
./arch/x86/platform/v586/Makefile

This is very common in embedded board world , but not necessary easy to do.

Example of v586.c file for board description , we will register devices :

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/io.h>
#include <linux/string.h>
#include <linux/module.h>
#include <linux/leds.h>
#include <linux/platform_device.h>
#include <linux/gpio.h>
#include <linux/spi/spi.h>
#include <linux/spi/spi_gpio.h>
#include <linux/spi/spi_bitbang.h>
#include <linux/gpio.h>
```

```
static struct spi_gpio_platform_data spi_gpio_data = {
	.sck = 500,  /* spi clock is GPIOA(4) */
	.mosi = 502, /* data output from fpga to sd/spi is GPIOA(6) */
	.miso = 503, /* data input from SD to FPGA is GPIOA(7) */
	.num_chipselect = 1, /* one chip select that will be defined later on */
};

static struct platform_device v586_spi_gpio = {
	.name		= "v586_spi",
	.id = 0,
	.dev.platform_data		= &spi_gpio_data,
};


static struct platform_device *v586_devs[] __initdata = {
		&v586_spi_gpio,
};

static struct spi_board_info board_spi_devices[] = {
	{
			.modalias = "mmc_spi",
			.max_speed_hz = 10000000,
			.chip_select = 0,
			.bus_num = 0,
			.controller_data = (void *) 498, /* the GPIOA(2) is the CARD select for SD*/
	},
};

static int __init v586_init(void)
{
		platform_add_devices(v586_devs, ARRAY_SIZE(v586_devs));

		spi_register_board_info(board_spi_devices, ARRAY_SIZE(board_spi_devices));

		return 0;
}

module_init(v586_init);
MODULE_AUTHOR("Philip Prindeville <philipp@redfish-solutions.com>");
MODULE_DESCRIPTION("Traverse Technologies v586 System Setup");
MODULE_LICENSE("GPL");
```

NOTE ON SD CARD:
So we are using gpioA(2/4/6/7) for the SD card , but we need also to modify the XDC file for virtuoso accordingly to wire those GPIOs to the right SD pins :

```
set_property PACKAGE_PIN E2 [get_ports {sdreset}]
set_property IOSTANDARD LVCMOS33 [get_ports {sdreset}]

set_property PACKAGE_PIN B1 [get_ports {gpioA[4]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {gpioA[4]}]
set_property PACKAGE_PIN C1 [get_ports {gpioA[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpioA[6]}]
set_property PACKAGE_PIN C2 [get_ports {gpioA[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpioA[7]}]
set_property PACKAGE_PIN D2 [get_ports {gpioA[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpioA[2]}]
```

we created an additional output to the virtuoso project , the sdreset, signal it is mandatory as explained by DIGILENT in the NEXY4 manual (see NEXYS4 manual for more information about E2/B1/C1/C2/D2 pin function of the FPGA)

3.3 buildroot compilation and utilities

3.4 micropython