aijuboard Manual

Julius Schmidt

1. Taking care of your board

There are a few things that you should beware of as they may cause serious damage to the board.

Like all circuit boards aijuboard is sensitive to electrostatic discharge (ESD). While circuit boards are generally less sensitive than individual chips, you probably should not touch aijuboard directly after stroking your cat.

Applying too much voltage to any pin will cause potentially serious damage where 'too much voltage' can be as little as 1.1 V for pins nominally 1.0 V. In particular the absolute maximum for the 3.3 V GPIO pins is 3.85 V: Do not connect them to 5 V logic!

Take care not to short 5 V to either logic pins or 3.3 V.

Do not attempt to operate aijuboard with a power supply of the wrong voltage or wrong polarity. Do not attempt to use an unregulated power supply.

There is a 10 A surface mount fuse on the board. To avoid blowing this fuse, do not operate aijuboard with a power supply capable of supplying 10 A or more (e.g. PC power supplies) without an extra fuse.

Do not operate aijuboard on a conductive surface.

Do not place jumpers in any position not specifically mentioned in this manual.

The heatsink is glued to the system-on-chip. Do not try to remove the heatsink and take care never to apply too much force to it, even accidentally.

2. Quick start guide

1. Verify that both the TX and RX jumpers on P3 are in place. Verify that the boot jumper on P1 is set to 'ROM' and not 'JTAG'.
2. Insert an SD card with a bootable image; alternatively, connect Ethernet and set up another Plan 9 computer to DHCP boot, using the the zynq kernel (the default MAC address is `0ea7deadbeef`).
3. Connect the USB-B connector to a host. Linux and 9front should work out of the box, you may need to install FT2232H drivers on Windows.
4. Connect the board to the power supply. The 'power' LED should turn on.
5. Open a terminal program on the host set to 115200 baud 8N1. You want to use the *second* serial port provided by the FT2232H, e.g. on 9front run

```
echo b115200 >/dev/eiaU*.1ctl
con -r /dev/eiaU*.1
```

whereas on Linux you should run

```
screen /dev/ttyUSB1 115200
```

3. Introduction

aijuboard is a single board computer built around the Xilinx Zynq XC7Z015-1CLG485C system-on-chip (SoC). It has two ARM Cortex-A9 MPCore CPUs that together with supporting devices form the 'Processing System' (PS). In addition it has a 46,200 LUT field-programmable gate array (FPGA) using Xilinx 7 Series (Artix-7) Architecture that forms the 'Programmable Logic' (PL). We will usually be sloppy with terminology and refer to the PS and PL as the CPU and the FPGA, respectively. Almost all devices are connected to one of the CPU or the FPGA and cannot directly be accessed by the other (an exception to this is 1 GB of DDR3 memory that is shared between the CPU and FPGA). This means in particular that during FPGA development (i.e. frequent reprogramming of the FPGA with your own bitstreams), you will not have access to the FPGA connected devices such as Displayport or SATA.

The SoC is very well documented in the Xilinx UG585 Zynq Technical Reference Manual (TRM). You are referred to this document for details on register interfaces etc. If you want to work on the 9front kernel or write your own kernel, you should definitely peruse this document.

4. CPU devices

For the most part the CPU devices should 'just work' with the 9front kernel. The devices accessible by the CPU subsystem are

10/100/1000 Ethernet using the Zynq's MAC and the Micrel KSZ9031RNX PHY. The bootloader sets the MAC address to `0ea7deadbeef`, but you could equally well set it to any other valid MAC address.

2× USB 1.1/2.0 host ports using the Zynq's EHCI controller and Texas Instruments TUSB1210 PHY.

A serial line. On boot it is setup as 115200 baud, 8 bits, no parity, one stop bit. This is by default connected to the FT2232H USB-to-serial (and USB-to-JTAG) converter. By removing the jumpers on P3 it is possible to get direct access to the 3.3 V level signals which could be connected to, e.g., a RS232 level converter.

16 MB of flash memory (N25Q128). Currently this is used only to store the bootloader. It could also store the kernel but the code is not there for it yet. The 64 KB sector size limits its applicability.

an SD card slot. None of the UHS modes are supported and the maximum speed is hence 25 MB/s.

5. Introduction to using the FPGA

This section is meant to be a 'tutorial' that describes how to take a simple design from Verilog to running it on the FPGA. First of all you will, of course, need a design to run. As a first example we will use the program `blink.v` reproduced below that shows a single light 'rotating' around the LEDs.

First of all, line 1 disables implicit declaration of wires, a very annoying Verilog 'feature' that is well worth turning off. Now, to do anything non-trivial you will need a clock from somewhere. The Zynq has a clock generator on the chip. To access it, you need to instantiate the `PS7` module, done here in l. 19 21. Note the use of a `DONT_TOUCH` attribute to prevent Vivado from optimizing `PS7` away which it sometimes does. The four clocks are supplied as a four bit vector; since we only care about the first one we define an alias `clk` in l. 6. By default this clock is set to 100 MHz. Accessing the LEDs is a bit more straightforward, for now just declare it as a port in l. 3.

The actual logic is l. 7 18. We use a 6 bit register to hold the current LED state (defined in l. 3) and a 32 bit counter to keep track of time. The counter is incremented until it reaches 49,999,999; after that it rolls over to zero. When the counter rolls over, the output register is rotated one bit to the left, using the `{}` operator for bit concatenation, in l. 16.

PS and PL are the terms used in Xilinx documentation but they are not as evocative as the more common terms CPU and FPGA. It should be pointed out that they are technically more correct, as the PS and PL contain more than just a CPU (never mind that there are two cores!) and an FPGA.

```
 1      `default_nettype none

 2      module blink(
 3              output reg [5:0] led
 4      );

 5              wire [3:0] fclkclk;
 6              wire clk = fclkclk[0];

 7              reg [31:0] ctr;

 8              localparam PERIOD = 50000000;

 9              initial begin
10                      ctr = 0;
11                      led = 6'b000001;
12              end

13              always @(posedge clk)
14                      if(ctr == PERIOD - 1) begin
15                              ctr <= 0;
16                              led <= {led[4:0], led[5]};
17                      end else
18                              ctr <= ctr + 1;

19              (* DONT_TOUCH="YES" *) PS7 PS7_i(
20                      .FCLKCLK(fclkclk)
21              );

22      endmodule
```

You also need a constraints file that defines various 'implementation details', most importantly what pins correspond to which Verilog ports. Since this file is rather tedious to maintain, a script is provided in the appendix that autogenerates it as follows.

```
$ aijupins11 > blink.xdc
LED0    led[0]
LED1    led[1]
LED2    led[2]
LED3    led[3]
LED4    led[4]
LED5    led[5]
EOF
```

The first column defines the hardware signal name (see the script for a list) and the name of the corresponding Verilog port.

To build this system we now need to use Vivado. Vivado has two basic modes called project mode and non-project mode. In project mode Vivado maintains a 'project' on disk that contains all the metadata about your program. In non project mode projects are created only temporarily in RAM; each build creates a new project. The main advantage of project mode is that, when used with the GUI, it leads to a workflow similar to IDEs such as Visual Studio or Eclipse. Which is why we are not going to use it. With a scripted workflow non-project mode is more pleasant to use as it leaves around fewer pointless files on disk. It's also faster because it keeps things in RAM. And it uses less RAM, too (please don't ask me how that makes sense).

A basic script for Vivado's non-project mode is shown below. It closes any open projects and creates a new 'project' in RAM for the part we are using. It reads in our Verilog file and constraints file (\ lets you split one command over multiple lines, used here so you can expand the script by adding lines for all files in your project). It then synthesizes, places and routes the design, creating 'checkpoints' named `*.dcp` as

it goes through these steps. These checkpoints are useful if you want to examine the output of the synthesis tool, which can be useful for debugging. It writes a timing summary and utilization report into `*.rpt` files. You will probably get awfully familiar with these reports once you start doing more advanced designs, but for now don't worry too much. Line 21 23 prints out a warning if timing fails which means you don't have to check the timing summary every time to compile to find out whether your design would actually work. The script also converts the design into a bitstream that can be loaded into an FPGA using the `write_bitstream` command.

You can run scripts either from the command line by simply doing

```
vivado -mode batch -source build.tcl
```

or you can alternatively run them in the GUI by typing `source build.tcl` in the TCL console. Running them in the GUI has the benefit that the design remains open for you to examine (F4 is a particularly useful key that brings up the schematic view).

The `write_bitstream` command creates output in a format the FPGA actually likes. It creates both a `.bit` and a `.bin` file, the former is the format used by Xilinx tools, whereas the latter is the 'flat binary' format that gets loaded into the FPGA. Except that for some reason it is in the wrong endianness. We can abuse *pcmconv*(1) to convert it into the proper format as follows

```
audio/pcmconv -i u32r1c1 -o U32r1c1 <build/out.bin >/whatever/out.bin
```

The byte swapped `out.bin` file thus generated can now be loaded into the `/dev/pl` device and the LEDs should now be 'rotating'. You can ignore the warning

```
temperature sensor reads 0, shouldn't happen
```

that may get printed on the console, it's a kernel bug that will be fixed eventually.

```
 1      set thepart xc7z015clg485-1
 2      close_project -quiet
 3      create_project -in_memory -part $thepart

 4      set top blink
 5      read_verilog n
 6              blink.v n

 7      read_xdc blink.xdc

 8      set outputDir ./build
 9      file mkdir $outputDir

10      synth_design -top $top -part $thepart -flatten_hierarchy none
11      write_checkpoint -force $outputDir/post_synth
12      opt_design
13      place_design
14      write_checkpoint -force $outputDir/post_place
15      phys_opt_design
16      route_design
17      write_checkpoint -force $outputDir/post_route

18      report_timing_summary -file $outputDir/post_route_timing.rpt
19      report_utilization -file $outputDir/post_route_util.rpt

20      write_bitstream -force -bin_file -file $outputDir/out.bit

21      if {! [string match -nocase {*timing constraints are met*} n
22      [report_timing_summary -no_header -no_detailed_paths -return_string]]} n
23      {puts "timing constraints not met"}
```

6. The CPU-FPGA interface

The CPU can program the FPGA using the internal configuration access port (ICAP). In 9front this is achieved by simply copying the bitstream into `/dev/pl`. To get a suitable bitstream you need to set Vivado to generate a `.bin` file and then reverse the byte order, which is achieved most simply by piping it through the 9front command

```
audio/pcmconv -i u32r1c1 -o U32r1c1
```

The FPGA is allocated two 1 GB regions, named MAXIGP0 and MAXIGP1, in the physical memory map at `0x40000000`. Part of this region can be imported into a user program using the `axi` segment and *segattach*(2), e.g. as follows

```
ulong *reg = segattach(0, "axi", 0x40000000, 0x40000000);
```

(you can also use *seg*(1) to access it from *rc*(1)). On the FPGA side you have to implement an AXI3 slave interface connected to the MAXIGP0 / MAXIGP1 ports of the PS7 macro block. Since there is quite a lot of complexity in AXI3, I wrote a module `axi3.v` that does most of the legwork and that is described below.

The interfaces just described were ‘slaves’ interfaces, i.e. the transfer has to be performed by the CPU. They are great for memory-mapped registers and small memories, but for large transfers this may be unacceptably slow and wastes CPU time. Alternatively, the FPGA can take a more active role in memory transfers using the ‘master’ interfaces. With them the FPGA can access memory completely independent from the CPU and you can burst memory as fast as you want, subject of course to the system’s total memory bandwidth (approx. 2 GB/s).

There is a total of seven master interfaces, to wit `SAXIGP[01]`, `SAXIHP[0123]` and `SAXIACP` (using *rc*(1) notation; nomenclature gets confusing because, from the CPU side, the `MAXI*` interfaces are the master interfaces and `SAXI*` are the slave ones). The general purpose (GP) interfaces are meant for low bandwidth application and can access the device registers. The high performance (HP) interfaces are optimized for high throughput and can only access memory. The accelerator coherence port (ACP) interface is coherent with L1 cache. Note that the other interfaces access DRAM directly and it is the user program’s responsibility to flush caches as needed.

To implement master interfaces you will have to deal with AXI3 yourself, however there is less work to be done as you can just not use the features you don’t need. See my document on the PS7 interfaces and ARM’s AXI specification [AXI].

7. FPGA devices

As previously mentioned to access these devices from the CPU you will have to load a bitstream that provides an interface using, e.g., memory mapped registers. At some point in the future we may provide a ‘canonical’ bitstream that provides a standard CPU interface to all of these devices.

DisplayPort. Connected to two high speed serial (GTP) transmitters. It supports two lane at 1.62 Gbit/s and 2.7 Gbit/s each. We currently have code that implements basic DisplayPort 1.1 functionality but it’s still relatively unpolished, see [DP]. It is possible to remove the DMA code from this code and use it for your own designs.

Serial ATA. Connected to a high speed serial (GTP) transmitter/receiver pair. No code yet. The connector could very well be abused for a generic high speed serial link (up to around 3 Gbit/s) to other boards. There is a possibility that, with a small adapter board, it could be used for single lane PCI express, but this has not been tried yet.

GPIO. There are a total of 30 general purpose input/output (GPIO) pins available on a standard 2.54 mm connector. Only 3.3 V standards are supported, i.e. `LVCMOS33`, `LVTTL`, `PCI33_3`, and are the only supported I/O standards (if you don’t know what I/O standard to use, you probably want

---

The DisplayPort specifications starting with version 1.2 is a closely held secret (encrypted PDFs with hardware dongles) and will probably never be implemented unless the specification leaks.

LVCMOS33).

Do not connect these pins to any voltage higher than 3.85 V; in particular they may be damaged by 5 V outputs (5 V TTL, but not CMOS, inputs can be driven from 3.3 V).

It is worth pointing out that the FPGA has an analog-to-digital converter called the XADC on some of these pins. You are referred to Xilinx UG480 for further details.

GPIO connector P4 pinout			
5V	1	2	GND
GND	3	4	3V3
GP0	5	6	GP1
GP2	7	8	GP3
GP4	9	10	GP5
GP6	11	12	GP7
GP8	13	14	GP9
GP10	15	16	GP11
GP12	17	18	GP13
GP14	19	20	GP15
GP16	21	22	GP17
GP18	23	24	GP19
GP20	25	26	GP21
GP22	27	28	GP23
GP24	29	30	GP25
GP26	31	32	GP27
GP28	33	34	GP29

Audio. Two FPGA pins (LSOUND and RSOUND) are connected, through a low-pass filter and amplifier to a headphone jack. To produce sound you have to toggle the pins at the right rate; a delta-sigma converter will do the job very nicely.

LEDs. There are six LEDs available as active-high signals (LED[0-5]). Not much else to say.

I2C. There is an I2C bus on the board that has an accelerometer (0x1D) and a GPIO expander (0x20) on it. Additionally the bus is wired to connector P2 to allow for small expansion boards.

I2C connector P2 pinout			
SDA	1	2	3V3
GND	3	4	SCL
INT	5	6	NC

Accelerometer. A Freescale MMA8652FC 3-axis, 12-bit accelerometer is wired to the I2C bus at address (0x1D).

GPIO expander for system management. A MCP23009 expander at address 0x20 provides various miscelleanous signals. Since the kernel does not like when you randomly reset its devices, you should be careful with this register.

Bit	Name	Dir.	Function
0	ETHINT	In	Ethernet PHY interrupt
1	ETHRESET	Out	Ethernet PHY reset
2	USB0RESET	Out	USB 0 PHY reset
3	USB1RESET	Out	USB 1 PHY reset
4	ETHLED3	Out	Ethernet Jack LED
5	USB0OC	In	USB 0 Overcurrent
6	USB1OC	In	USB 1 Overcurrent
7	GPIORESET	Out	Available at P7 (see Jumper Settings)

Acknowledgements

Many thanks to all the people who contributed to the aijuboard Indiegogo campaign and the people who bought the board. A special thanks to Ian Sutton (kremlin), rlevi and spew. And of course a special thanks to cinap_lenrek who wrote a large amount of the kernel code.

Appendix: Script to map signal names to FPGA pins

```
#!/bin/rc
echo 'set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.UNUSEDPIN PULLUP [current_design]
create_clock -name FCLK -period 10.000 [get_pins {PS7_i/FCLKCLK[0]}]'
sed 's/#.*//g
/^[     ]*$/d' $1 | sort | join -a1 /fd/0 /fd/3 <<[3]END | awk ' { if(NF == 2)
print "unknown pin", $1 > "/fd/2"; else printf("set_property IOSTANDARD %s " n
"[get_ports {%s}]0et_property PACKAGE_PIN %s [get_ports {%s}]0, $4, $2, $3, $2); } '
CLK     T2      LVCMOS33
GP0     C6      LVCMOS33
GP1     E4      LVCMOS33
GP10    F1      LVCMOS33
GP11    G2      LVCMOS33
GP12    G1      LVCMOS33
GP13    H3      LVCMOS33
GP14    H1      LVCMOS33
GP15    J2      LVCMOS33
GP16    J1      LVCMOS33
GP17    K3      LVCMOS33
GP18    K2      LVCMOS33
GP19    L2      LVCMOS33
GP2     E3      LVCMOS33
GP20    L1      LVCMOS33
GP21    L4      LVCMOS33
GP22    M4      LVCMOS33
GP23    M3      LVCMOS33
GP24    M2      LVCMOS33
GP25    M1      LVCMOS33
GP26    N3      LVCMOS33
GP27    N1      LVCMOS33
GP28    P3      LVCMOS33
GP29    P2      LVCMOS33
GP3     A1      LVCMOS33
GP4     D3      LVCMOS33
GP5     C1      LVCMOS33
GP6     D2      LVCMOS33
GP7     D1      LVCMOS33
GP8     E2      LVCMOS33
GP9     F2      LVCMOS33
HOTPLUG R2      LVCMOS33
INT     A7      LVCMOS33
LED0    C5      LVCMOS33
LED1    C3      LVCMOS33
LED2    B2      LVCMOS33
LED3    B1      LVCMOS33
LED4    B3      LVCMOS33
LED5    A2      LVCMOS33
LSOUND  A4      LVCMOS33
RSOUND  B4      LVCMOS33
SCL     A6      LVCMOS33
SDA     A5      LVCMOS33
END
```