Using SD Card

OBJECTIVES

- Create a software application in SDK 2019.1 to read and write both text and binary files from/to an SD Card.
- Use a previously developed custom AXI peripheral to test the software application.
- Test the software application on a ZYBO or ZYBO Z7-10 board.

SETUP

- HW/SW Test Project: AXI-4 Full Pixel Processor peripheral (Tutorial # 4).
- Software files:
  - ✓ `test_sd.c`: File that reads a binary file from and SD card, streams the data thru the AXI4-Full Pixel Processor peripheral, retrieves output data and writes it back on the SD card. A simple test for writing/reading a text file is also included.
  - ✓ `xtra_func.h`: This library includes two functions:
    - load_sd_to_memory: Reads data (binary/text) from SD card and places it in the memory space of the microprocessor.
    - write_data_to_sd: Writes data (binary/text) to SD card.
- Other files:
  - ✓ `droif.bif`: Input binary file generated in MATLAB® from an image.
  - ✓ `test_in.txt`: Input text file.
  - ✓ `create_binaryfile.m`: MATLAB® script that generates `droid.bif` from an image file (`droid.png`). It also reads an output binary file `droid.bof` and displays the processed image.
- **Note**: You need an adapter to access the micro-SD card on your computer. If your computer has a SD slot, you need a micro SD adapter; otherwise, you need a micro SD to USB adapter.

PROCEDURE

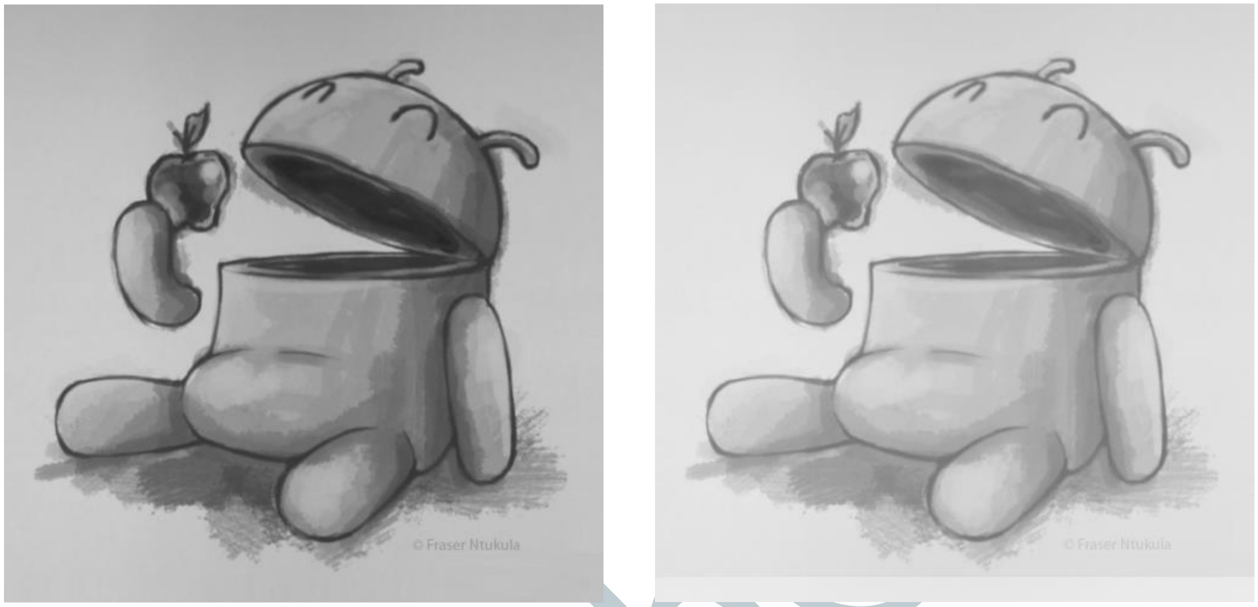
- Open the SDK project of the AXI-4 Full Pixel Processor peripheral.
- Create a new SDK application.
  - ✓ Go to File → New → Application Project. On Project Name, you can use: `pixSDtst`.
    - In Board Support Package (*bsp*): You can create a new one (suggested) or use a previously generated one.
  - ✓ Go to Xilinx → Board Support Package Settings. Select the *bsp* associated to the `pixSDtst` project.
    - On Supported Libraries, check the xilffs library (Generic Fat File System Library): This library provides all the functions to deal with writing/reading to/from the SD card.
    - On Overview → standalone, click on xilffs: A configuration table will open. By default, the string manipulation functions in the xilffs library are disabled. For this tutorial, you need to enable the string manipulation functions:
      - Go to use_strfunc and on the 'Value' Column, change the value from '0' to '1'. This will replace the parameter FILE_SYSTEM_USE_STRFUNC in `/bsp/ps7_cortexa9_0/include/xparameters.h`.
      - Alternative (old) method: Go to `bsp/ps7_cortexa9_0/libsrc/xilffs_v3_3/src/include/ffconf.h` and manually modify: `#define _USE_STRFUNC 1` (it is 0 by default).

- Copy the following two files in the `/…/pixSDtst/src` folder: `test_sd.c, xtra_func.h`.
- You might want to deactivate the (default) option Build All. This way, you compile (Build Project) only when needed.

- Assign Heap and Stack space for input, intermediate, and output data:
  - ✓ Right-click on `pixSDtst` application. Click on Generate Linker Script. You MUST assign enough space in the heap and stack for the input, intermediate, and output data. By default, 1KB is assigned to the heap and stack.
  - ✓ Place the code/heap/stack section in DDR memory ps7_ddr0 (the largest one with 1023 MB in ZYBO Z7-10). There are other memories (like the limited BRAMs inside the FPGA) where you can place your code and heap/stack, but it is better to place everything on the largest memory.
  - ✓ In this test, we dynamically allocate memory (heap) for the input and output data files and define some constants and variables (stack). While the Stack might be fine with 1 KB, the heap will need far more than 1 KB.
    - Use 700 KB for Heap and 4 KB for Stack.
  - ✓ Note that the compiler does not tell you that there is not enough memory when allocating dynamically. This is usually slightly more than the size of the input and output data files.

- Copy these files on the SD card: `droid.bif, test_in.txt`. Place the SD card on the ZYBO or ZYBO Z7-10 micro-SD slot.
- Execute the code. The 500×500 input image (`droid.bif`) amounts to 62500 32-bit words. All those words are streamed to the FIFO-based AXI4-Full Pixel processor peripheral in batches of 512 32-bit words (since the depth of the FIFOs is 512 32-bit words). After each 512 32-bit word batch is sent, we retrieve a 512 32-bit word output batch. We do this $\lceil 62500/512 \rceil = 122$ times, the last time we only send 36 32-bit words.

- If the software application runs successfully, there will be a `droid.bof` (output binary file) and `test_out.txt` (output text file). The contents of `droid.bof` can be verified by reading the file in MATLAB and comparing it with a model of the Pixel Processor function.
  - ✓ Pixel Processor: it applies pixel-to-pixel operations. For the parameter `F=1`, the function is gamma correction with γ=0.5. For a 8-bit grayscale pixel IM, the resulting pixel OM is given by: $OM = round\left(\left(\frac{IM}{256}\right)^{0.5} \times 256\right)$, where IM, OM $\in$ [0,255].



(a) (b)

Figure 1. (a) Input Image. (b) Output image where gamma correction (γ=0.5) was applied

TO KEEP IN MIND:

- Little endianness: A binary file is read/written in a byte-wide buffer. Printing the bytes is straightforward with `xil_printf`. However, when displaying 32-bit words, the system uses the little-endian convention. For example:
  - ✓ Typecasting (from 8 to 32-bit arrays, and viceversa): The table shows the byte arrangement in a 32-bit word.
  - ✓ Reading/writing binary files: A binary file is read/written in a byte-wide buffer.
    - When reading data (byte per byte) as per the table, the resulting 32-bit word (when displaying) is `0x11220044`.
    - When writing data, if we have an array of 32-bit words, a 32-bit word such as `0x11220044` will be stored in memory (byte per byte) as per table.
    - <u>Keep this in mind</u> when you are dealing with 8-bit data that is transformed into 32-bit data.

Memory contents	32-bit word

`Byte 0: 0x44`
`Byte 1: 0x00`
`Byte 2: 0x22`
`Byte 3: 0x11`
	`0x11220044`

- (OLD, not available anymore) Using XMD (Xilinx Tools → XMD Console, we can debug software issues with the following commands:
  - ✓ Always first connect to the ARM:
    ```
    XMD% connect arm hw
    ```
  - ✓ Print memory positions:
    `XMD% mrd 0x00400000 16` → 16 32-bit words printed in little endian format, starting from address `0x0040000`
  - ✓ Dump contents into text file:
    `XMD% set fp [open testi.log w]` // `testi.log` -> text file name
    `XMD% puts $fp [mrd 0x00110CA8 14818]` // `0x00110CA8`: starting address, `14818`: # of words (little endian)
    `XMD% close $fp`
  - ✓ Load a file (binary) into memory:
    `XMD% dow -data file.bin 0x00400000` // `0x0040000`: location where we want our data (variable)