Using SD Card

OBJECTIVES

- Read and write text and binary files from/to an SD Card.
- Use the project in Unit 4 (custom peripheral) to test the SD card.
- Test this in a ZYBO board.

ZYBO BOARD SETUP FOR HARDWARE/SOFTWARE CO-DESIGN

- The current Zynq Book Tutorials (Aug 15th, 2015) includes a procedure that requires copying definition files into the Vivado installation directory. This helps when setting up the ZYBO Board.
- In this tutorial, we will manually indicate the Zynq device and the Processing System (PS) definition file.

TEST PROJECT

- The test project is the AXI-4 Full Pixel Processor peripheral (Unit 4). It has only been tested on Vivado 2016.2 version.

PROCEDURE

- Open the SDK project of the AXI-4 Full Pixel Processor peripheral.
- Create a new SDK application.
  - ✓ Go to New → Application Project. On Project Name, you can use: `pixSDtst`.
  - ✓ In Board Support Package (*bsp*): You can create a new one or use a previously generated one (recommended).
  - ✓ Go to 'Board Support Package Settings'. Select the *bsp* linked to the `pixSDtst` project.
  - ✓ 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.
- Once the SDK application has been created, do:
  - ✓ Go to `/libsrc/xilffs_v3_3/src/include/ffconf.h` (in the bsp) and modify: `#define _USE_STRFUNC 1` (it is 0 by default). This enables functions that deal with strings.
  - ✓ Copy the following two files in the `/src` folder: `test_sd.c`, `xtra_func.h`. The .h file has top-level functions that read/write binary and text files.
- Make sure data is allocated in the heap and stack: When dealing with large data files in memory, use dynamic memory allocation. This requires modifying the heap/stack in the SDK application (File → Generate Linker Script). Make sure to allocate enough space in the heap/stack for the input and output data. Also, make sure to place code/heap/stack sections in DDR memory (the largest one).
- Copy the following files on the SD card:
  - ✓ `droid.bif`: Input binary file that can be generated in MATLAB®. An .m file is provided that generates a binary file form an image file.
  - ✓ `test_in.txt`: Input text file.
- Once the software code is run, there should 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 compare with a model of the Pixel Processor function.

TIPS:

- 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.

Memory contents	32-bit word
Byte 0: 0x44 Byte 1: 0x00 Byte 2: 0x22 Byte 3: 0x11	0x1122044

- 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)