Toprammer - Developers guide

February 27, 2010

1 Definitions

DUT Device Under Test. The device put into the ZIF socket of the programmer

VPP Programming voltage for the DUT (usually 12V)

VCCX Supply voltage for the DUT

GND Ground for the DUT

ZIF Zero Insert Force socket of the programmer.

2 TOP2049 device hardware

The TOP2049 consists of four basic hardware parts

- USB interface (PDIUSBD12 chip)
- Microcontroller (Megawin MPC89E52A)
- FPGA (Xilinx Spartan2 XC2S15)
- VCC/GND/VPP supply circuitry

The microcontroller's job is to initialize and communicate to the FPGA and set up the VCC/GND/VPP supply circuitry. The microcontroller can receive commands via USB interface to do these things.

3 Communicating with the programmer via USB

In the "toprammer_main" module there is the "class TOP" which is used for communication with the programmer device. The class has various methods for hardware access:

3.1 cmdRequestVersion()

Reads the programmer identification and versioning string and returns it.

3.2 cmdReadStatusReg()

Reads the "status register" from the microcontroller. That register is used for buffering of status information and for buffering of data fetched from the FPGA.

3.3 cmdReadStatusReg32()

Same as cmdReadStatusReg(), but just returns a 32bit int which was formed by the first 4 bytes of the register (little endian).

3.4 cmdReadStatusReg48()

Same as cmdReadStatusReg(), but just returns a 48bit int which was formed by the first 6 bytes of the register (little endian).

3.5 cmdSetVPPVoltage(voltage)

Set VPP (programming voltage) to the specified voltage. Voltage is a floating point number.

3.6 cmdSetVCCXVoltage(voltage)

Set VCCX (DUT supply voltage) to the specified voltage. Voltage is a floating point number.

3.7 cmdLoadGNDLayout(layoutID)

Load a ZIF-socket GND-layout. You usually don't want to call this directly. Use an autogenerated layout instead.

3.8 cmdLoadVPPLayout(layoutID)

Load a ZIF-socket VPP-layout. You usually don't want to call this directly. Use an autogenerated layout instead.

3.9 cmdLoadVCCXLayout(layoutID)

Load a ZIF-socket VCC-layout. You usually don't want to call this directly. Use an autogenerated layout instead.

3.10 cmdFPGAWrite(address, byte)

Writes a byte to the FPGA using "address" for address latching and "byte" as payload data.

3.11 cmdFPGAReadRaw(address)

Reads a byte from the FPGA and puts it into the microcontroller's status register. "address" is used for address latching on the FPGA. The microcontroller's status register has an automagically incrementing pointer. So issueing several cmdFPGAReadRaw() in a row will result in all the bytes being put one after each other into the status register. The status register can hold up to 64 bytes. Reading the status register (cmdReadStatusReg()) will reset the automagic pointer to zero.

3.12 cmdFPGAReadByte()

This is a shortcut to cmdFPGAReadRaw(0x10). In general it is used for payload data reading. Using cmdFPGAReadByte() instead of cmdFPGAReadRaw(0x10) is more efficient for payload reading, because on the USB bus the command is only one byte wide instead of two. So it uses up less space in the transmission queue and can thus speed up operation.

3.13 cmdFlush()

Is believed to be kind of a flush command. However, there also are suspections that it has certain delay semantics, too. You usually don't have to call this directly.

3.14 delay(seconds)

Sends all queued commands to the device and waits for "seconds". "seconds" is a floating point number.

4 TX command queueing

All commands transmitted to the device are not sent immediately, but queued in software and sent later. This is done to speed up device access significantly. The command transmission queue has several flushing conditions:

- Commands can be flushed explicitely using the "flushCommands()" method of "class TOP". Note that this is not to be confused with cmdFlush(), which does some flushing tasks inside of the microcontroller.
- Commands are automatically flushed on cmdReadStatusReg() before reading the data from the device. This is to ensure sequential consistency of the commands.
- Commands are flushed on various voltage-layout operations.

You usually do not need to flush commands explicitely.

5 Implementing a new chip (DUT) algorithm

The reading and programming algorithms for the chips (DUTs) are separated into two parts:

- Low level FPGA bottom-half
- High level Python code top-half

The FPGA bottom-half implements the basic operations (fetching data from DUT. Writing data to DUT. etc...). It may also implement timingcritical parts of the algorithm. Everything else is implemented in the high level Python code, that lives on the other end of the USB line.

5.1 Python top-half implementation

The DUT specific top-half lives in the "chip_xxx.py" files, where xxx is the ID of the DUT. This file contains a class derived from the "Chip" class. The "Chip" class provides some basic helper methods for algorithm implementation. It also defines the interface that is to be re-implemented in the derived subclass. This interface consists of the following methods:

initializeChip() Called once on chip initialization. Reimplement this, if required.

shutdownChip() Called once on chip shutdown. Reimplement this, if required

readSignature() Read the DUT signature and return it. Reimplement this, if your DUT supports signature reading.

erase() Erase the DUT. Reimplement this, if your DUT supports electrical erasing.

readProgmem() Read the program memory and return it. Reimplement this, if your DUT has program memory and supports reading it.

writeProgmem(image) Write the program memory. Reimplement this, if your DUT has program memory and supports writing it.

readEEPROM() Read the (E)EPROM memory and return it. Reimplement this, if your DUT has (E)EPROM memory and supports reading it.

writeEEPROM() Write the (E)EPROM memory. Reimplement this, if your DUT has (E)EPROM memory and supports writing it.

readFuse() Read the Fuse memory and return it. Reimplement this, if your DUT has Fuses and supports reading them.

writeFuse() Write the Fuse memory. Reimplement this, if your DUT has Fuses and supports writing them.

readLockbits() Read the Lockbit memory and return it. Reimplement this, if your DUT has Lockbits and supports reading them.

writeLockbits() Write the Lockbit memory. Reimplement this, if your DUT has Lockbits and supports writing them.

After defining your "Chip"-derived class you need to add it to the "supportedChips" array.

```
supportedChips.append(Chip_MyDevice())
```

6 Automatic layout generator

TODO