1541 Ultimate-II Documentation

Gideon Zweijtzer
August 19, 2010
Version 0.3

TABLE OF CONTENTS

1. LICENSING

The 1541 Ultimate-II project is released to the public under the GPLv3 license. A full copy of the license text can be obtained here: http://www.gnu.org/licenses/gpl-3.0-standalone.html

1.1. The Foundations of the GPL

Nobody should be restricted by the software they use. There are four freedoms that every user should have:

- the freedom to use the software for any purpose,
- the freedom to change the software to suit your needs,
- the freedom to share the software with your friends and neighbors, and
- the freedom to share the changes you make.

When a program offers users all of these freedoms, we call it free software. Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft: the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom.

1.2. My interpretation

The GPL license gives you the freedom to use and modify the source code as provided, as long as the copyright notices and references to the author are left in tact. Basically, releasing the project's sources under GPL means that it becomes free software, and is required to stay free software. When parts of the 1541 Ultimate project is used in other projects, the GPL license will apply to the derivates and new projects as well.

2. Setting up the build environment

This chapter explains the structure of the 1541 Ultimate-II, and describes how to set up the build environment on Linux and Windows.

2.1. Background

The firmware of the 1541 Ultimate-II consists of a few different parts that work together. First of all, there is the FPGA, the Field Programmable Gate Array. This chip is the main component of the hardware board and is manufactured by Xilinx. This programmable chip consists of a lot of 'logic-gates' and some memory blocks, which are configured each time at power-on. The configuration thus contains all logic circuits that define the 1541 Ultimate platform. It contains the logic of the 1541 drive, including the 6502 processor, but also another processor that runs the software part of the firmware of the 1541 Ultimate. Attached to this processor, there are a lot of I/O modules to 'talk' to the external world.

In FPGA world, there are two serious description languages; VHDL and Verilog. Verilog has a greater share in the USA, while Europe is –traditionally– more VHDL-oriented. I am not going to bias towards either language here; each of them has its benefits. But as I am a European guy, you can guess which language was used in the 1541 Ultimate; VHDL! The VHDL code is synthesized and mapped onto the core elements of the FPGA by the Xilinx tool-chain.

The processor that is currently running the Ultimate firmware is a 'ZPU' from Zylin; or in fact a derivate that I created from the sources publically available. The ZPU is a 32-bit stack processor. Its instructions do not have operands, nor does this processor have any registers. All values that it needs are found on the stack, or placed on the stack from memory. This makes it a very small (in terms of logic cells), but relatively slow processor. The reasons for choosing this processor are its size, the 32-bit flat memory model and the availability of an open source compiler, based on the GNU C/C++ compiler suite.

Then, there is a small amount of 6502 code in the project that will run on the 6510/8500 processor in your Commodore computer. This code is assembled using 'tass64'.

Thus, in order to build the firmware completely, the FPGA needs to be built and the software needs to be built. You will need two complete tool-chains and the tass64 executable to build the project.

2.2. *Linux*

All the instructions that you will find in this chapter are based on the Ubuntu 10.04 LTS distribution. Of course, other distributions will work, but you might need different commands to set up your environment.

2.2.1. Subversion (SVN)

First, let's set up SVN and download the source code of the project. We will do this, so we can make some changes to the configuration files as we install other tools.

Open a Terminal and type the following command:

```
sudo apt-get install subversion
```

Then, from your home directory:

```
mkdir ultimate
cd ultimate
svn co http://svn2.xp-dev.com/svn/1541UltimateII/trunk .
```

2.2.2. Xilinx toolchain

In order to install the Xilinx tool-chain, you would need to register with the Xilinx website, and download the Xilinx WebPack edition. http://www.xilinx.com/support/download/index.htm
At the time of writing, the newest version is 12.2. The build has been tested with 12.1, but I assume 12.2 will work as well.

Xilinx will install its tools in /usr/local/Xilinx

You will need to tell the make-file based environment that Xilinx has been installed there.

```
cd global_makefiles
gedit ise_locations.inc &
```

and edit, if necessary, the location of the Xilinx toolchain, in the IF clause where version is compared with 12.

It's better to copy this file to your home directory, and set an environment variable pointing to this file, such that if ever you make a clean checkout from the archive, you don't need to make this modification again:

```
cp ise_locations.inc ~
```

Add to your ~/.bashrc:

```
export ISE_LOCATIONS_FILE_PATH=~/ise_locations.inc
export PATH=$PATH:/usr/local/Xilinx/12.1/ISE_DS/ISE/bin/lin
```

2.2.3. ZPU compiler

Download the package 'zpu_linux.tgz' from the 1541 Ultimate website, and unpack it:

```
sudo mkdir /usr/local/zpu
sudo tar –xf  zpu_linux.tgz /usr/local/zpu
```

Now, add the following to your ~/.bashrc:

```
export ZPU_TOOLCHAIN=/usr/local/zpu
export PATH=$PATH:$ZPU_TOOLCHAIN/install/bin
```

Restart your shell, and type

```
cd ultimate
make
```

and watch it happen!

2.3. Windows

Development of the 1541 Ultimate-II has taken place under Windows for a large part, under both Windows XP as well as Windows 7.

2.3.1. Cygwin

Under windows, you will need Cygwin to build the 1541 Ultimate-II using the makefiles, as provided in the archive. To install Cygwin, go to http://www.cygwin.org/cygwin and run the setup.exe from there.

When you are requested to select which packages to install, I strongly recommend you to install the whole developer (Devel) branch of the tree, so you will for sure have tools like 'make' and 'awk'.

2.3.2. Subversion (SVN)

There are two options, basically. You can use the command-line svn tools, that is installed with Cygwin when you install all "Devel" tools, and follow the instructions given in the Linux chapter. You can also choose to install TortoiseSVN from http://tortoisesvn.tigris.org. TortoiseSVN will integrate nicely with the Windows shell, and will show you which files have been modified, etc.

In order to check out the 1541 Ultimate-II archive with TortoiseSVN, create a directory somewhere in a path without spaces, such as: C:\proj\ultimate. Then, use the windows explorer to go to this directory, and right-click with the mouse and select "SVN Checkout". Enter http://svn2.xp-dev.com/svn/1541UltimateII/trunk in the edit field and press "OK".

2.3.3. Xilinx toolchain

In order to install the Xilinx tool-chain, you would need to register with the Xilinx website, and download the Xilinx WebPack edition. http://www.xilinx.com/support/download/index.htm
At the time of writing, the newest version is 12.2. The build has been tested with 12.1, but I assume 12.2 will work as well.

Xilinx will install its tools by default in C:\Xilinx

You will need to tell the make-file based environment that Xilinx has been installed there, by editing the *ise_locations.inc* file in the directory 'global_makefiles' of the project.

You will see some IF-clauses, checking on the version to be used. The easiest way is to change the version check from "12" to "12lin" and "12w" to "12", such that by default the windows installation will be taken. Please also verify that the path is actually correct. Mine looks like this:

It's better to copy this file to your project directory or cygwin home directory (outside of the ultimate checkout) and set an environment variable pointing to this file, such that if ever you make a clean checkout from the archive, you don't need to make this modification again. In cygwin:

```
cd /cygdrive/c/proj/ultimate/global_makefiles
cp ise_locations.inc ~
```

Add to your ~/.bash_profile:

```
export ISE_LOCATIONS_FILE_PATH=~/ise_locations.inc
export PATH=$PATH:/cygdrive/c/Xilinx/12.1/ISE_DS/ISE/bin/nt
```

The latter, adding it to the path directly, is for invoking the 'data2mem' command in one of the makefiles. It should actually use the ISE_LOCATIONS_FILE… Oh well. ☺

2.3.4. ZPU compiler

Download the package **'zpu_cygwin.zip'** from the 1541 Ultimate website, and unpack it somewhere in a path without spaces. You can use any free program like 7-Zip, or you can do it in the cygwin shell as follows (directory is just an example, of course):

```
mkdir ~/zpu
unzip zpu_cygwin.zip ~/zpu
```

Now, add the following to your ~/.bash_profile:

```
export ZPU_TOOLCHAIN=~/zpu
export PATH=$PATH:$ZPU_TOOLCHAIN/install/bin
```

2.3.5. Summary

To give you an idea of what my .bash_profile and ise_locations.inc look like:

```
export ZPU_TOOLCHAIN=/cygdrive/d/zpu
export PATH=$PATH:$ZPU_TOOLCHAIN/install/bin
export ISE_LOCATIONS_FILE_PATH=~/ise_locations.inc
export PATH=$PATH:/cygdrive/c/Xilinx/12.1/ISE_DS/ISE/bin/nt64
cd /cygdrive/d/proj
```

ise_locations:

```
ISE_VERSION?= 12     # when not defined use highest version
available


export ISE_LOCATIONS_FILE_INCLUDED=yes


ifeq ($(ISE_VERSION),12)
    export MYXILINX=/cygdrive/c/Xilinx/12.1/ISE_DS/ISE
    export XILINX=C:\\Xilinx\\12.1\\ISE_DS\\ISE
    PLATFORM=nt64
endif


ifeq ($(ISE_VERSION),12lin)
    export MYXILINX=/usr/local/Xilinx/12.1/ISE_DS/ISE
    export XILINX=$(MYXILINX)
    PLATFORM=lin
endif
```

After everything is set, you are ready to go. Type:

```
cd /cygdrive/c/proj/ultimate
make
```

and watch it happen!

3. SOFTWARE

3.1. *Language*

The 1541 Ultimate-II application and utilities are written in C/C++. Some might see C++ as a 'dirty' language, but in my opinion any language can be both clean and dirty. I tried to keep the code reasonably clean, and limit the use of exotic C++ features.

3.2. *Directories*

In the ultimate archive, you will find various directories. Most of them will be clear to you what you will find inside of them. However, what you have to keep in mind that results of the tools will never appear in the source directories. I created a target directory for this purpose, which holds the make files and settings for the build, and the results and intermediate files that are being generated during the build process.

3.3. *Boot*

It is important to fully understand and realize how the 1541 Ultimate-II cartridge starts. The first step after applying power is the configuration of the FPGA. This is an autonomous process; the FPGA loads its configuration from the attached Flash (ROM) device.

From a software point of view, the 1541 Ultimate-II uses the two-level bootloader concept. This means that the first bootloader will load the second bootloader, which in turn will load the application.

3.3.1. 1st_boot

The first bootloader is actually embedded in the internal memory of the FPGA, and is thus loaded automatically when the FPGA configures itself from Flash. Since I kept the allocated amount of memory for the bootloader inside the FPGA small (4 KiB), this bootloader can't do a lot. As a matter of fact, it can only read more data from the Flash device and copy it into RAM. And this is what it does: it will attempt to copy the second level bootloader to RAM and run it.

3.3.2. 2nd_boot

The second level bootloader can be much larger. In fact, the only limitation on the size of the bootloader is the arbitrarily chosen area in Flash that is allocated for it. This area is 60K in size, from which about 25K is currently in use. This second level bootloader has the SD-card driver, as well as a rudimentary form of the USB driver, and of course the knowledge of FAT file-systems.

This "2nd_boot" application checks the presence of the files "recover.bin" and "update.bin" (in that order) in the root of the SD card, and will load it if it is present. If not, it will attempt to copy the application from Flash into memory and run it, and if that fails, it will fall back to the X-modem protocol. With the X-modem protocol you can just upload the application from a terminal program over the UART, at 115.200 bps (8,N,1). In FPGA version $95, this UART port is only available inside the black case, but the idea is to route the UART to the tape extension connector instead and switch dynamically between tape emulation and UART; to make software debugging easier with the plastic case closed.

The 2nd bootloader is in some way just an application. It is copied to memory location 0x10000 and runs from there. The linker script included in the target directory will force the linker to create a binary that is loaded from this address.

3.4. Applications

Applications are the binary objects that can be loaded by the boot loader system. They load and run from address 0x20000, and will therefore use the linker script for applications. These can be found in the target directories for each application.

In fact, because all applications use the same linker script and the same load and run address, the bootloader will be able to run it. Any application can be loaded from SD-card, given that it uses the name 'update.bin' or 'recover.bin'.

3.4.1. Ultimate application

The 1541 Ultimate application is the main application that manages the 1541 drive, the user interface, and contains in fact all functionality that you see as user. You will find more about the structure of the main application later, in chapter 5.

NOTE: Just by renaming the ultimate.bin output to 'update.bin' and copying it into the root of the SD will allow you to test the newly compiled application without performing an update. In this case, nothing will be programmed into the Flash rom!

3.4.2. Update application

The update application is in fact a simple flash-tool. It just links all items that need to be stored in Flash memory as binary arrays to the C/C++ code. Therefore, the update application itself does not need to access the file-system, as it loads all necessary data as part of the application.

3.5. Overview

The following figure shows the dependencies and build-flow of the 1541 Ultimate firmware. This summarizes the above.

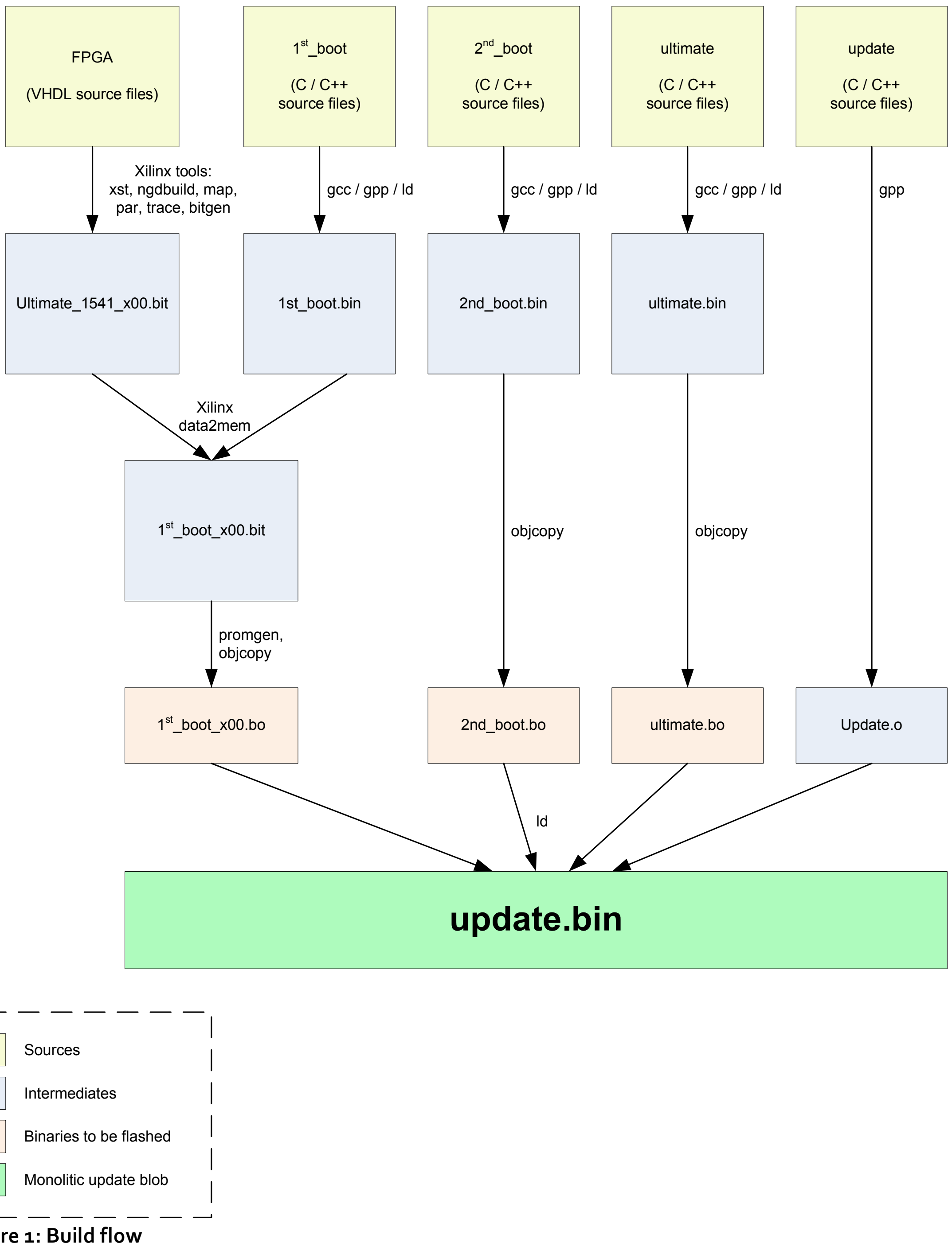

Figure 1: Build flow

4. FPGA

In this chapter the structure of the FPGA design and its functional blocks is described.

4.1. Structure

For the sake of modularity, testability and maintainability, the FPGA design has been built up using some proprietary, yet consistent interfaces. These interfaces have been chosen such that the interconnect at the logic top-level is clean and consistent. The concept is that each module has clock, reset, I/O pins (real outside world connections), and one or more standardized busses and absolutely nothing else. All registers with settings, status and control are placed in the functional block and not concentrated on one place.

Another concept applied to the 1541 Ultimate design is the separation of *logic* from FPGA *I/O*. The logic top-level is basically the whole design, but excludes some FPGA specifics, such as the way clocks and resets are generated. Clock managers (PLLs, DCMs and such) are usually very slow to simulate and are FPGA vendor specific. It therefore makes sense to leave them out during simulation. An additional benefit is that the logic top-level may have some extra ports that facilitate simulation. Tying off these ports during synthesis (in the FPGA top-level) will cause all logic needed to support these extra ports to be optimized away.

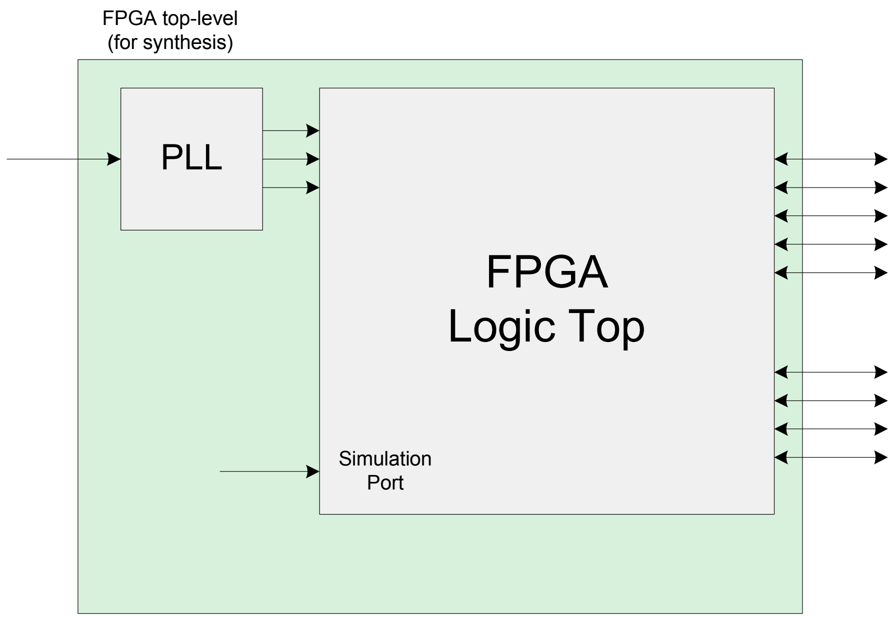

Figure 2: Separation of synthesis and simulation top-levels

4.2. Bus concept

4.2.1. Rationale

To reduce the overhead and logic cell count, the system is based on 8-bit busses. Three busses have been defined; the memory bus, the I/O bus, and the DMA bus. The memory bus gives access to the external SDRAM memory; the I/O bus is meant for internal attachment of functional blocks, and the DMA bus is a local bus for accessing the C64 memory through DMA on the cartridge port.

The busses are separate, to enforce that the data flow is clear and that no bottlenecks are introduced just because of the standardization and use of one bus for everything. Even more important, no complexity is created by defining just one bus that can do all. Creating separate busses allows differentiation in and a tradeoff of complexity and performance. Another reason is that it allows the use of different processor architectures that reach the different busses in different ways (memory mapped I/O vs. I/O mapped I/O).

4.2.2. Bus Interconnect

All busses consist of a 'request' and 'response' structure. A bus-master sends out a request, and receives the acknowledge, and in case of a read also the data, through the response port from the slave. When there is a single master and a single slave, this is easy.

However, on any sensible bus, there are usually more slaves (devices / modules) that are connected. In FPGA world, there is no such thing as tri-state busses.[1] Therefore, a concept has been defined to make each connection from any master to any slave a point-to-point connection. Increasing the number of ports on a master or slave, as the number of slaves and masters increase will result in a 'colliflower'-design; not maintainable. The definition of bus-expanders eliminates this problem. In order to connect more slaves to one master, a *splitter* is defined, and to connect more masters to one slave, an *arbiter* is defined.

4.2.3. Splitters

A splitter divides the available addressing space in separately addressable chunks; or in other words: performs the address decoding. For each slave in the addressable range, a new bus is generated. This results in the desired point-to-point connections. It might sound as if a whole new bus for each slave will result in a lot of logic, but in fact, only the command pulses (read/write) need to be separate; as the address and data can be sent to any slave. After synthesis, all these wires are combined, and each slave will connect only to these wires from the bus it really needs.

[1] Well, designers could use 'Z' to emulate tri-state and let the synthesizer generate the necessary logic to resolve multiple responses on one wire, but this makes debugging hard and turns out to be very expensive in terms of logic cells.

The second function of the splitter is creating one response back to the master from all the responses from each individual slave. A simple and effective way to do this is just 'or-ing' busses together. This will work when each slave drives all zeros on the response port if it is not addressed.

4.2.4. Arbiters

Arbiters do the opposite of what splitters do. They have multiple slave ports and combine these back to one master. An arbiter can favor one port over the other, or perform a so called 'round robin' algorithm to give each port equal rights to access the target.

The way the responses are created to each master depends on the timing model of the bus. When the bus is defined to have a state (idle, busy with access) and will just process one access at the time, the arbiter simply remembers from which port it is accessing the slave. The response is sent back to that port, and the arbiter goes back to idle, becoming ready for the next request. This is perfect and simple for low-performance busses, like simple I/O.

However, a higher performance bus splits the requests (commands) from the data transfer. This allows queuing of commands and out-of-order responses. In this case, the arbiter simply adds (or passes) a tag along with the request and the slave echoes the tag along with the data. The arbiter uses this tag to send the acknowledge/data to the correct port. Another way is to send the response to all masters and let the master filter the response by itself. The latter is the method used in the 1541 Ultimate memory bus.

4.3. 1541 Ultimate-II FPGA Overview

The figure below shows how the functional diagram of the FPGA logic top-level. For the sake of clarity, the bus splitters and arbiters have been omitted, as well as the connections to the outside world (I/O).

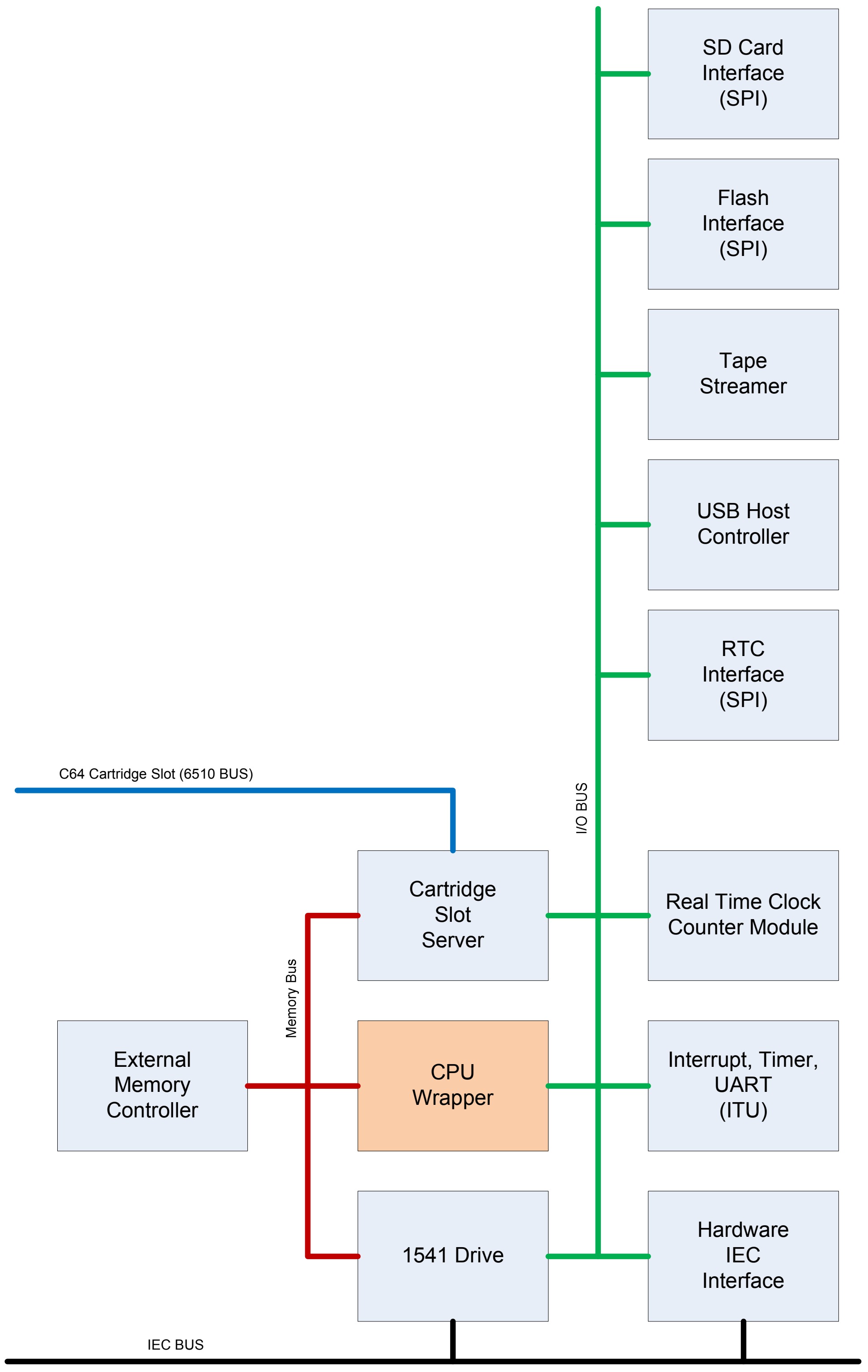

Figure 3: 1541 Ultimate-II FPGA Logic

4.4. *1541 Ultimate-II FPGA Functional Blocks*

This paragraph will give a description of each of the functional blocks inside the 1541 Ultimate-II FPGA design.

4.4.1. CPU wrapper

The CPU-wrapper is in fact an abstraction layer around any CPU. At the moment of writing, the 1541 Ultimate-II uses the ZPU processor. The CPU wrapper component splits up the external memory bus of the ZPU, becoming a master on the memory bus, and a master on the I/O bus. This decoding is done based on address-bit 26; '0' => memory space, '1' = I/O space. Thus address 000.0000 – 3FF.FFFF is memory space (64MB, of which 32MB used as SDRAM), and 400.0000 – 7FF.FFFF is I/O space.

This wrapper also contains a small memory that overlaps with the SDRAM memory. This is the boot-memory, and is pre-loaded with the first-level bootloader. See paragraph 3.3.1.

4.4.2. 1541 Drive

The 1541 drive component is probably the primary reason of many 1541 Ultimate-II owners to buy this cartridge. It implements the Commodore 1541 disk-drive. It is therefore not a surprise that this paragraph will show 'familiar' diagrams. :-)

The 1541 drive is divided into five functional sub-blocks:

- The timing generator;
- the 1541 drive logic, called 'cpu_part' (6502 CPU/6522 VIAs);
- the floppy emulation logic;
- the floppy sound generator;
- the register block with settings and control.

Timing generation

The timing generator simply outputs clock enable pulses at the system clock, providing 1 MHz timing pulses and 4 MHz timing pulses. The 1 MHz is used for the CPU, the 4 MHz is the main clock enable for the floppy emulation logic.

1541 Drive Logic

The 1541 drive logic (cpu_part.vhd) contains the 6502 CPU and the 6522 VIAs, and all address decoding that is required. Conceptually, the 6502 address space is fully mapped (the whole 64K) in the Ultimate-II's SDRAM, with the exception of the VIAs. The address of the 6502 is extended with a hard-coded prefix, and placed on the internal memory bus of the FPGA. The software simply copies the ROM images at the right memory location, such that the 6502 will see them as its own ROMs. Some register bits define which areas of the 64K are writable by the 6502. This yields an emulation of various RAM expansions for the drive for 'free'!

Floppy Emulation Logic
The floppy emulation logic is also very straight forward. The emulation uses a base address and offset counter to generate an address in memory to which the 'read/write head' is currently pointing. The offset counter wraps around when it reaches the value in the track-length parameter, such that it appears like a cyclic storage. The base address and track-length parameter for each half-track are stored in a small local FPGA memory (BRAM). When the software encodes a floppy image, it also fills this BRAM with pointers to each (half-) track. The bytes in memory are serialized and de-serialized again such that even images that use sync-patterns that are not a multiple of 8 bits will work as intended.

Sound generator
The sound generator is a very arcane state-machine that simply uses the track pulses and motor on/off signal to generate a floppy-alike sound. It has two 'voices'; the first voice implements the motor-hum or floppy drag sound (depending on the floppy-inserted signal), and the second voice plays the events like "track_in" and "track_out". The samples are taken from memory, scaled, added and converted to a DS (some sort of PWM) pulse-train to be output on the audio-jack.

Register block
The register block provides an interface on the I/O bus, for the software to control the drive. Settings include register bits that define the level of the VIA pins that define the IEC bus address, the reset, power on/off and such.

The register block also has a 'dirty-track' bitmap. This bitmap reveals to the software which tracks are 'dirty' (have been written to). The software uses this to write modified tracks back to the D64/G64 images.

4.4.3. GCR Codec

The GCR codec was included in the FPGA to speed up the conversion from binary to GCR; the format the 1541 drive uses for its floppy data. The ZPU turned out to be terribly slow on its bit-shift operations, as many of these operations are emulated by a whole bunch of other instructions rather than implemented in the hardware of the ZPU processor.

The GCR codec simply consists of a 40-bit shift register, which is loaded with eight bits at a time, each time the software writes a value to it. The shift-register is re-coded from binary to GCR, and vice versa with some simple lookup tables. The register map of this I/O module provides access to the recoded values. The software simply reads from the correct register addresses to obtain the converted data.

4.4.4. Cartridge Slot Server

TODO (Important block!)

4.4.5. ITU

The ITU is the interrupt, timer and UART block, which provides the simplest, low level functions that are needed around the CPU. It contains a fixed-rate UART (at 115.2 kbps, 8 bits, 1 stop-bit and no parity), which is used for debugging. The timer is there for some simple timed operations. The software call 'wait_ms(int)' is based on it. The interrupt controller is currently not used at all; aside from reading the external buttons. However, the status of the buttons is polled; interrupts are not used (nor tested on the ZPU).

4.4.6. Hardware IEC Interface

The hardware IEC interface is a state-machine that implements the basic IEC protocol. It is used to implement the secondary drive on the bus that gives direct access to the SD-card. This state-machine implements the transport of complete bytes and messages, such that the CPU does not need to perform this timing-critical task. However, it is not very flexible and needs to be replaced with something else. Therefore, support for this hardware block has not yet been incorporated in the software.

4.4.7. USB Host Controller

Slow, bulky, yucky implementation of a USB host controller. This was my first attempt, based on a complete absence of any knowledge of USB at the moment I started with it. Therefore it is on the list to be rewritten. A complete description of this design component here does not make sense.

4.4.8. Tape Streamer

The tape streamer is a very simple component that implements the decoding of the TAP 1.0 and TAP 1.1 data stream into tape pulses. It implements a 2KB FIFO to which the software can write its data, and a status register that shows whether the component can receive at least another 512 bytes of data. The software parses the header of the TAP file and decides whether this unit implements the V1.0 or V1.1 format.

4.4.9. SPI interfaces

The SPI interface is used for access to the MicroSD card, the RTC chip, the Flash chip. To reduce the overhead of the software transfer loop, an access to the SPI device has been made blocking. This means that an I/O read will stall the bus, output "$FF" to the SPI device and sample the data that comes back. The bus is released when all of the bits have been transferred. Similarly for writes, but then the byte written is shifted out and the input value is discarded. Some extra registers (which can be turned off with a generic) are used to set the transfer speed. For MicroSD, the SPI interface calculates the checksum on the fly. The software simply copies the checksum register into the data register to append the correct checksum to the data stream.

4.4.10. Real Time Clock Counter

The real-time clock counter is there to overcome superfluous accesses to the RTC chip whenever the time is needed for an operation. The RTC chip is fed ONLY by the battery that is inside of the plastic case and many accesses will drain the battery quicker than necessary.

Basically, this counter just counts the time. It's possible to write its counter values through the I/O bus and read them back. An additional feature of this block is that it already provides a 32-bit FAT-time formatted DWORD. This is simply the same counter bits, but then packed together differently in just 4 bytes.

4.4.11. External Memory Controller

The memory controller in the 1541 Ultimate-II is, at the time of writing this document, a very simple one, as the memory-bus definition is also very simple and does not support bursting. The SDRAM controller is a simple state-machine that opens a row, reads or writes one byte and closes the row again. This is far from optimal, but provides just enough bandwidth for the current application.

The memory controller uses an inhibit-pin that prevents any operation to take place. When inhibit is true, no actions take place and refreshes are delayed as well. This has been done to guarantee a fixed delay between the request coming from the slot-server and the response. The inhibit, which comes from the timing block in the slot-server, is 'true' just long enough for any current operation to finish. The first cycle after inhibit becomes 'false' the memory bus request will be the one from the slot server. The priority on the arbiter enforces this. Only in this way, the timing requirement for the 6510 can be met.

5. Ultimate Application

In this chapter, the 'ultimate' main application will be discussed, together with the building blocks that are used to build the application.

5.1. Operating system

The 1541 ultimate application was designed to be OS-less. Some people might argue that this is not a wise choice, but at the time the application was designed, the limitations in the hardware (only one fixed processor stack, for example) have guided me into this direction. Not having an OS does not impose that many limitations, however. Providing some building blocks already make the creation of a higher-level application possible.

5.2. Exceptions

Although exceptions in C++ could make error-handling a lot easier than using return values, it showed that just enabling exceptions added about 48 KiB to the binary executable! At the scale of the 1541 Ultimate application, this is relatively large. So, I tried to build the application without using exceptions anywhere, and so far it didn't really cause a lot of trouble.

When I started developing, I used X-modem to upload the application to the FPGA for testing, so I tried to keep the executable relatively small.

Disabling exceptions does have a consequence, though. Most C++-libaries cannot be used, since they do throw exceptions, and will always cause the exception handler to be linked. I did some tests, and even using a simple List template created an executable of 120 KiB or so. So.. back to the basics!

5.3. Components

5.3.1. Main loop

The application is now built around an event-based main loop. Compiled units can register themselves in the main loop list. Being registered causes these units to be 'polled' in every iteration cycle. Each polled function can issue new events that will be processed by itself or other modules in the next iteration cycle. This principle allows inter-module communication and a strictly ordered execution of code.

5.3.2. Events

Events are defined as an event-code, a pointer to an object and a parameter. Not in all cases are both the object pointer and the parameter used. Some event codes already embed a parameter. This is not very clean, but originates from the time events didn't have a parameter. ;-)

5.3.3. Indexed List

The indexed list template class provides a way to create linear lists of any arbitrary length, and implements the [ ] operator to simply index the contents of the list. The storage of the list is internally allocated at the start-size, which could even be zero. Each time more items are pushed onto the list than the allocated size can handle, the size of the list is doubled. The implementation is simple; when this happens, the contents of the small list are copied into the larger list and the smaller list is discarded. This is done to maintain a contiguously allocated array and keeping indexing very simple. Since the indexed list is actually always used for pointers to objects, the overhead is still limited.

5.3.4. Fifo

The FIFO (First In First Out) is a software implementation of a simple queue. The class template provides methods to push and pop elements from the queue. It is used only for the event queue at this moment.

5.3.5. MyString

'MyString' is a simple implementation of a string-container and provides some simple operators like + and += and methods like length() and c_str(). It is not so clever as the implementation from the C++-library, but is by far smaller and simpler. It is not used consistently throughout the code, but provides the storage for named objects.

5.3.6. Pattern

The pattern component handles pattern matching for file names, searches and such. It is capable of handling '?' and '*' in any arbitrary order and number. It uses a recursive invocation to handle characters after '*'. Therefore match strings such as '*er' and 'b*t*n' will work as expected.

5.4. *Input Output System (I/O)*

The concept applied in this project, is that every hardware component will have a software class to accompany it, regardless whether it is a directly accessible chip, or a module in the FPGA. This gives structure to it all. The interface of these classes is made logical (at least for me!); the provided methods implement operations on the hardware components on a functional level. These classes therefore implement a layer of abstraction of the hardware components.

5.4.1. C1541

This class implements operations on the C1541 drive. It uses a few helper classes, like BinImage and GcrImage, which in turn provide the conversion necessary functions for the data. The C1541 class itself has an interface to mount images, set the drive address, ram/rom configuration, power, reset and so on.

5.4.2. C64

The C64 class is the software abstraction of the C64 as I/O unit of the Ultimate. It provides functions to enable cartridges and their ROMs, freeze and unfreeze the machine and so on.

5.4.3. Flash

The Flash class is an abstraction of the Flash storage on the board. This class is inherited to AT45_Flash and W25Q_Flash to implement different access methods of different Flash chips. A similar child class could be created to support the AT49 flash that is found on the Ultimate-I, to port this firmware to the 'old' hardware.

5.4.4. ICAP

The ICAP I/O module is nothing more than a connection to the configuration logic of the FPGA. This allows software to dynamically change the FPGA content (not recommended!), or request for a reconfigure of the FPGA. The latter can be used to start another FPGA image. The code to do this can be found in the Flash classes; because the addresses and how the addresses are encoded can differ.

5.4.5. Real Time Clock (RTC)

The real-time clock block encapsulates the access to the real time clock chip and the real time clock counter in the FPGA. Methods provided allow the user to retrieve the time from the chip, from the counter, convert it to readable text, etc.

5.4.6. SD Card

Provides access to the SD-card. The class "SdCard" is a derivate from the class "BlockDevice". Therefore, the helper-class "SdCardManager" has been created, which registers itself in the main loop of the application, and checks for insert and removal of the card. The file "sdio.cc" implements the raw I/O functions to the SPI interface and is thus the software layer closest to the hardware.

5.4.7. Tape

The Tape I/O module provides the software interface to the tape streamer block in the FPGA. This module registers itself in the main event loop, such that data can be streamed while the Ultimate-II program is running. The class provides a way to set the file to be streamed, starting and stopping the tape, etc.

5.4.8. USB

Important and relatively large block. Maybe best to describe this in a new chapter.

5.5. File System

The file system layer is the software layer that implements access to files and directories on a medium, in most cases a block-based medium. It is important to understand how the different classes of the file system layer work together.

5.5.1. BlockDevice

The lowest level is the block device class. This class provides a standard interface to communicate with a block-based medium. In fact, the BlockDevice class is an empty base-class[2]. The simplest inherited class from this base class is the "SdCard"-class, which implements access to the MicroSD.

The block device uses a very commonly used interface: read, write and ioctl (I/O control).

5.5.2. Disk

The Disk class is in fact a representation of a physical disk. This class is capable of reading the master boot record and finding the partitions available on the disk, using the block-device as its lower-level interface. The partitions created and attached to this class. When the partitions are created, the disk-class is in fact no longer needed.

5.5.3. Partition

The partition class provides a clean way to access data that is within the partition. It simply adds the offset to any read/write request from the file-system, and forwards the request to the block device.

At the time of writing this document, the partition class does not look for file-systems on the partition. This code is now part of the File Manager structure. Conceptually, the code should be here, because the disk also looks for partitions.

5.5.4. FileSystem

The FileSystem class attempts to create an abstract interface layer for any file system, by providing functions like dir_open, dir_read, dir_create, file_open, file_rename, file_read, file_write, etc... For this purpose, the FileSytem class makes use of three helper-classes: FileInfo, File and Directory.

5.5.5. FileInfo

FileInfo is nothing more than just a struct with data, describing the file. It contains its name, its location on the disk, the size, time, date and attributes. Because FileInfo is very handy for

[2] The base class is not purely virtual, because a pure virtual function requires the exception handler to be linked.

displaying directories (which could be seen as a collection of FileInfo structs), the FileSystem has also been made such that it can open files directly from FileInfo. This eliminates the need for searching through the whole directory tree from the root of the disk, as the sector where the file resides is already known; as well as a reference to the directory it resides in. You could therefore see it as a shortcut (in the 1541U-II's memory) to the file.

5.5.6. Directory

The Directory class is a base-class that wraps the FileSystem call 'dir_read', for file systems that need extra processing or data storage when reading directories. The FAT file system for example needs a DIR structure to remember access parameters of a directory that is opened in the user application.

5.5.7. File

The File class provides a handle to a file to the application. It is a wrapper to hide the actual implementation of a stream. Right now, the diversification is handled by derivates of the FileSystem class, but future specials could handle direct file implementations as well. However, one thing needs to be noted: the 'node' field should always refer to a valid object. If this variable is set to NULL, the file is invalid. This can be caused by a tree cleanup, at the removal of a media. When the file is opened using a Path class, a reference to this path is available as well. More about paths later.

5.5.8. FAT File System

The FAT file system implementation is based on the open source code from Chen. However, Chen's code is not strictly layered as it should be. So I attempted to remove all references to partitions, paths and stuff, but the final result is not as clean as I would like it to be. I think I went a little too far in forcing the FATFS from Chen into my design, and would have been better off to just create wrappers for most functions. In the end, this is what happened; as you can find many wrapper functions to be able to call Chen's functions by means of a call from the FileSytem interface.

5.6. File Manager (Path Manager)

5.6.1. Rationale

- Why is this a different layer than the file system layer?

5.6.2. Paths

- What do paths do and why were they introduced?

5.6.3. Structural hierarchy overview

- How are path components stacked in the tree?

5.6.4. File Types

- How does file type recognition and promotion work?

5.6.5. Creating your own file type

- What are the minimum requirements for creating a file type?

5.7. User Interface

5.7.1. UserInterface

5.7.2. TreeBrowser

5.7.3. ConfigMenu

5.7.4. ContextMenu

5.7.5. TaskMenu