Reference Manual for the FPGA LatticeMico32 Based System Implementation

Simon Southwell

August 2017

Copyright

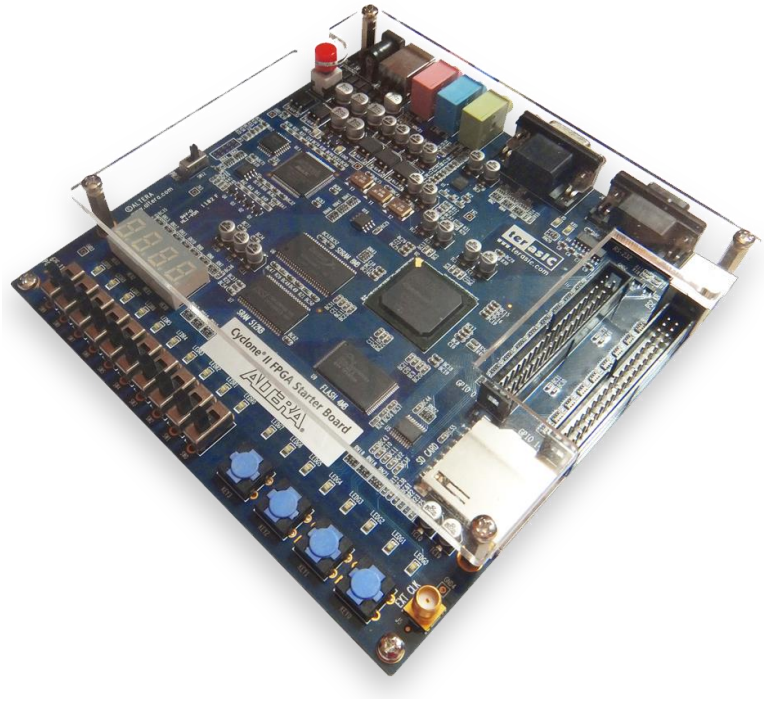
Disclaimers

No warranties: the information provided in this document is “as is” without any express or implied warranty of any kind including warranties of accuracy, completeness, merchantability, noninfringement of intellectual property, or fitness for any particular purpose. In no event will the author be liable for any damages whatsoever (whether direct, indirect, special, incidental, or consequential, including, without limitation, damages for loss of profits, business interruption, or loss of information) arising out of the use of or inability to use the information provided in this document, even if the author has been advised of the possibility of such damages.

Introduction

This project is a complementary project with the LatticeMico32 soft CPU Instruction Set Simulator project. In that project and ISS model was constructed based on the Lattice Semiconductor LM32, with UART and Timer peripherals and a test suite. This project is a hardware implementation of that model, with simulation environment, and the ability to run a subset of the test suite of the ISS.

The project is targetted at a specific platform, the terasIC DE1 development board, based on a Cyclone II Intel/Altera FPGA. The board is show below, and more details can be found on the terasIC website (http://www.terasic.com.tw/en).



This particular board was chosen based on cost (kept to a minimum), and desired requirements, which are not high speed, or within a very large design. The design files are specific to the board and the particular FPGA, but could easily be re-targetted to another platform or FPGA. The project serves as a case study of moving from a modelling environment to hardware implementation, simulation and finally running on a platform.

Specification

The basic specification for the initial design is based on the need to run the test suite that was run on the ISS C/C++ model (minus any tests specific to ISS features). Speed was not of primary concern, but certain limitation on clock rate are enforced due to exctrnal component constraints. The minimum basic specification is thus as shown.

- System clock to run at 40MHz
- Processor and all peripherals to run from system clock
- External asynchrous interfaces to be synchronised to systrem clock before data processed by logic
- Processor Harvard archtecture to have busses combined externally
    - Allows upgrade path back to Harvard without changing processor
- Processor memory to be from SRAM
- A system reset (active low) controlled from push-button key

- A separate CPU reset that can, additionally, be controlled from the host PC
- A single UART with register interface to LatticeMico specifications
    - Interrupt to external interrupt line 0
- A single Timer with register interface to LatticeMico specifications
    - Interrupt to external interrupt line 1
- A means to download program from host PC and control execution
- A means to inspect processor memory from host
- An upgrade path to running uCLinux as on the C/C++ model
    - Move to using SDRAM
    - Additional memory via SDIO/SPI card
    - Total requirement of 16MB (preferably 32MB), with at least 8MB writable.

Features

The design is implemented to meet the above basic specification, but also has some additional features to take advantage of the hardware on the DE1 platform.

Included hardware features:

- Implementation of LM32 with MMU as provided by M-Labs
- UART as available from Intel/Altera
- Timer as available from Intel/Altera
- Interface for s/w download and memory inspection from host to SRAM via USB-JTAG
- Host access to SDRAM
- Processor control of GPIO, with read and write
- Processor read and write to I^2C for audio configuration
- Processor read of PS2 input.
- Processor read of switches
- Processor read of keys 1 to 3 (key 0 is reserved for reset)
- Processor write to seven segment display (memory mapped)
- Status display on LEDs

Features not yet included, but hardware external to the FPGA available on DE1 board:

- Use of SDRAM from processor.
- Use of Flash from processor
- SDI/SPI interface
- ADC/DAC interface
- VGA interface

Along with the hardware features comes some driver code that can load a program to memory, and start the processor running, detecting when the program is complete, if it is such a program. The rest of this document details the FPGA design, the simulation environment, and the building for the development board and running tests and programs on it.

Simon Southwell (simon@anita-simulator.org.uk)
Cambridge
August 2017

Installation

The package for the lm32 system harware implementation is a complimentary package to the lm32 ISS project, and it is meant to be install on top of this, though it is not *strictly* necssary, and the package can stand alone. The ISS is available at:

> https://github.com/wyvernSemi/mico32

If not done so already, get this package and install it somewhere convenient. The hardware package is almost orthoganal to the ISS, with the vast majority of files in a directory called `HDL` that should sit in the install directory of the ISS package. There are a couple of shared files at the top level---`makefile` and `runtests.py`---which are common to both projects, so it does not matter which is installed. The package also has three new files that reside in the ISS's python directory.

After this some third party code must be installed.

Third Party Dependencies

The design is based on some third party components, so in order for this project to compile and run it is required that, in addition to the code supplied in the package, the M-Labs MMU lm32 code is intalled at `<install dir>\third_party\m-labs\lm32`. Version 1.0 (June 2013) was used in this project. The source is available on github at:

> https://github.com/m-labs/lm32

In addition, the UART and Timer peripherals are the open licenced LatticeSemiconductor blocks, and the HDL source code should be installed at:

```
<install dir>\lscc\micosystem\components\uart_core\rtl
<install dir>\lscc\micosystem\components\timer\rtl
```

These are obtainable from Lattice by downloading and installing the Lattice Diamond Software and then the Mico System software. Versions 3.8.0.115.3 were used in testing this project. See:

> http://www.latticesemi.com/Products/DesignSoftwareAndIP.aspx

The modules are located in `<install dir>\diamond\3.8_x64\micosystem\components` (path name may vary for particular installed version or operating system).

The design is targeted at the terasIC Cyclone II DE1 FPGA board. Synthesis requires the Altera Quartus II software to be installed (tested with 13.0.1, build 232 SP1, 64-Bit web edition), whilst simulation is targetted at Mentor Graphic's Modelsim, with the Altera starter edition optionally bundled with Quartus II. Version 10.1d used to test this project. See:

> https://www.altera.com/downloads/download-center.html

For targetting the DE1 development board, drivers for the FTDI chip are bundled for windows in the starter kit package. To get both Linux and windows drivers, the latest versions can be downloaded from the FTDI website. Versions 2.12.24 (for Windows) and 1.3.6 (for Linux) were used for the testing of this project. See:

> http://www.ftdichip.com/Drivers/D2XX.htm

Python3 must be available on the host with tinker and ttk, and in the path. For windows it may need updating to stop tkinter requiring specific versions if multiple versions are installed. In particular, `tcl\tcl8.x\init.tcl` and `tcl\tk8.x\tk.tcl` installed in the modelsim home directory. Comment out the "`package require -exact <...>`" lines. This project was tested with Python 3.6. See:

> https://www.python.org/downloads

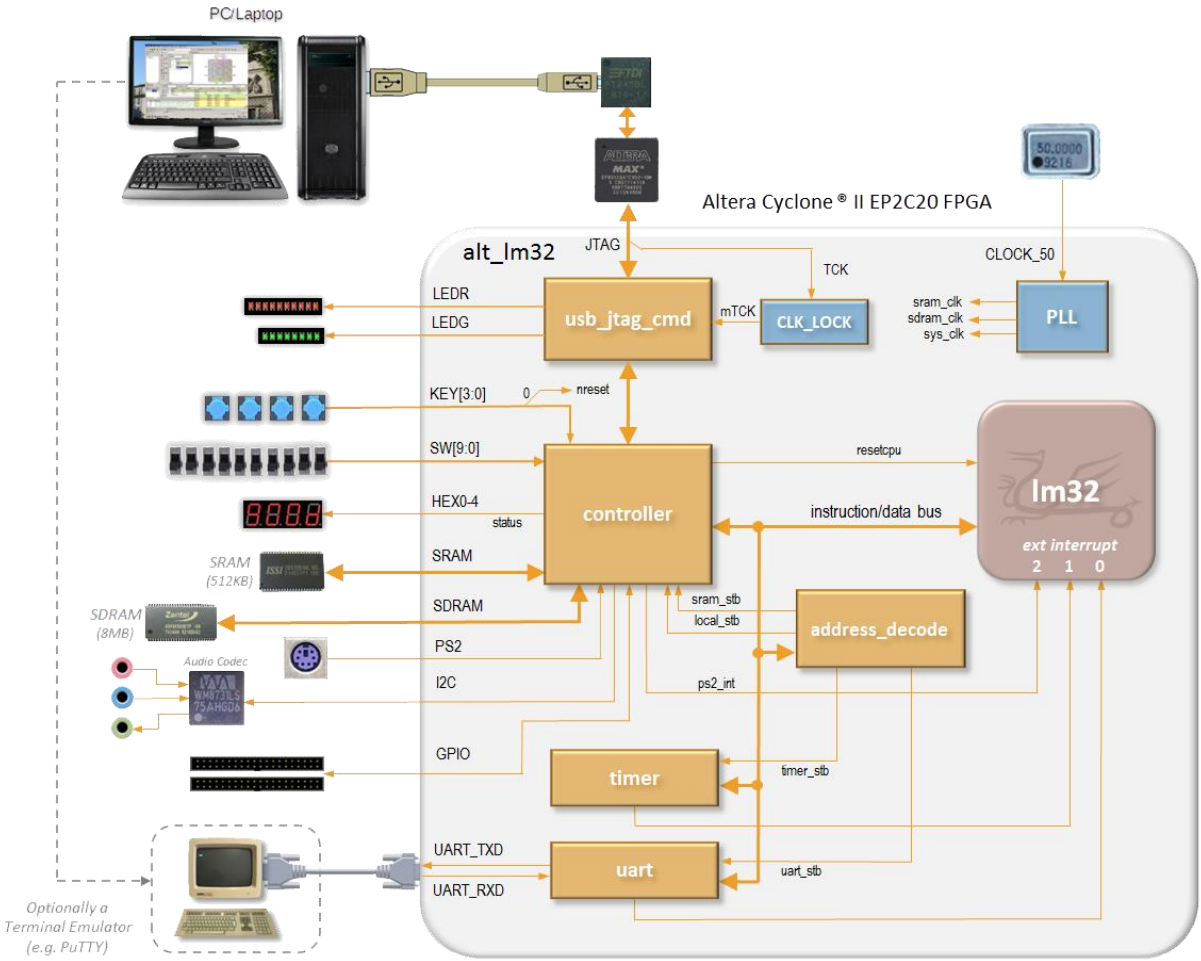
After all the installation, the directory structure of the package will be as shown in Appendix A at the end of this document.

Design

The LM32 based design is targetted at the Cyclone II FPGA on a DE1 development board. The external hardware is fixed by the design of that board, and is connected to the FPGA pins in a unchangeable way. The development board comes with example code, including an empty top level Verilog module that has ports to match all the I/O connected to the FPGA. It is within this enviroment that the LM32 based system is designed, and is discussed in this section.

Top Level Architecture

The following diagram shows the top level of the design, with the major modules within the FPGA, and the connections to the relevant hardware on the board that it interfaces with.



The block diagram within the FPGA looks similar to that shown in the ISS document, and this is not a coincidence. The processor, UART and Timer are all common with the ISS model. The hardware implementation requires some additional blocks to facilitate the functionality, and additional logic is added to interface with other available external hardware to 'future proof' for, as yet, undefined uses.

The processor itself is wrapped up in a hierachy (`lm32_wrap`) to combine the separate instruction and data busses, via an abitrator/mux (`wb_mux`). These busses are to the wishbone specification. Address decoding is done within a separate module, external to the processor, which generates strobe signals to select the appropriate peripheral or memory controller, muxes together any read data and the acknowledges.

The controller houses all the 'other' logic, to interface to most of the external peripherals, including the processor memory. An exception to this is the LED control which is in the last top level block, the USB-JTAG interface. The LED data is for status, and this is centralised in the interface that connects to the host PC. The board has a USB connection which connects to an FTDI 245 USB to FIFO chip. This, in turn, is connected to an Altera MAX chip which, amongst other things, can send data to and from the FPGA via a JTAG style interface. The `usb_jtag_cmd` module decodes the data coming from this interface to allow host access to various parts of the hardware, as the designer chooses.

Clocks and Resets

As well as the functional verilog module blocks, means are required to generate clocks and resets from the features of the FPGA. For the clock, auto-generated files are used (via Quartus II wizards) to generate a PLL module and what is called a clock lock module. This PLL block is generated to take the 50MHZ clock inputs, as generated from an external crystal component and generate the system clock (at 50MHz) the SRAM clock (at 100MHz) and the SDRAM clock at 50MHz. The SRAM clock is used solely to generate the SRAM's write enable with both setup and hold margin against the 50MHz strobes based on the system clock. The SDRAM clock does not drive the internal logic, but is connected directly to the DRAM clock output pin to minimise the latency, and give margin on the strobe signals, generated from the system clock. It also means it is less loaded and thus a cleaner clock signal for use by the memory device. The clock lock module *can* be used to gate a clock. In this case it is used simply to connect an external pin (the `TCK` of the JTAG interface) to an internal global clock wire, as this can't be done directly. A PLL could also have been used, but PLL resources are more scarce, and this would have been overkill.

Registers

The register map, for the system peripherals, as seen from the LM32 CPU, is defined in a spreadsheet, and all the header files, defining addresses and field extraction macros, for the verilog, assembler and C code, are auto-generated from this spreadsheet using a python script that utilises the `xlrd` module. (`<install dir>/python/reg_gen.py`)

This spreadsheet can be found at `<install_dir>/HDL/registers/register.xls`, as can the generated headers (`regs.s`, `regs.h` and regs.vh) and the details of the registers and their fields are defined there. The table below summarises the registers and their addresses.

Address	Peripheral	Register Name	Description
0x80000000	UART	LM32_UART_RBR	Receive buffer register/transmit holding register
0x80000004		LM32_UART_IER	Interrupt enable register
0x80000008		LM32_UART_IIR	Interrupt identification register
0x8000000C		LM32_UART_LCR	Line control register
0x80000010		LM32_UART_MCR	Modem control register
0x80000014		LM32_UART_LSR	Line status register
0x80000018		LM32_UART_MSR	Modem status register
0x8000001C		LM32_UART_DIV	Baud-rate divisor register
0x80002000	Timer	LM32_TIMER_STATUS	Timer staus
0x80002004		LM32_TIMER_CONTROL	Timer control
0x80002008		LM32_TIMER_PERIOD	Count period
0x8000200C		LM32_TIMER_SNAPSHOT	Timer snapshot
0x90000000	GPIO	LM32_GPIO_0_LO	Port0 36 bit view bits 0 to 31
0x90000004		LM32_GPIO_0_HI	Port0 36 bit view bits 32 to 36
0x90000008		LM32_GPIO_0_OE_LO	Port0 36 bit view OE bits 0 to 31
0x9000000C		LM32_GPIO_0_OE_HI	Port0 36 bit view OE bits 32 to 36

0x90000010		LM32_GPIO_0_X_LO	Port0 40 bit view bits 0 to 31
0x90000014		LM32_GPIO_0_X_HI	Port0 40 bit view bits 32 to 39
0x90000018		LM32_GPIO_0_X_OE_LO	Port0 40 bit view OE bits 0 to 31
0x9000001C		LM32_GPIO_0_X_OE_HI	Port0 40 bit view OE bits 32 to 39
0x90000020		LM32_GPIO_1_LO	Port1 36 bit view bits 0 to 31
0x90000024		LM32_GPIO_1_HI	Port1 36 bit view bits 32 to 36
0x90000028		LM32_GPIO_1_OE_LO	Port1 36 bit view OE bits 0 to 31
0x9000002C		LM32_GPIO_1_OE_HI	Port1 36 bit view OE bits 32 to 36
0x90000030		LM32_GPIO_1_X_LO	Port1 40 bit view bits 0 to 31
0x90000034		LM32_GPIO_1_X_HI	Port1 40 bit view bits 32 to 39
0x90000038		LM32_GPIO_1_X_OE_LO	Port1 40 bit view OE bits 0 to 31
0x9000003C		LM32_GPIO_1_X_OE_HI	Port1 40 bit view OE bits 32 to 39
0xA0000000	Switches	LM32_SWITCH_DPDT	Read port of switches
0xA0000004		LM32_SWITCH_KEY	Read port of keys
0xB0000000	PS2	LM32_PS2_RX	Read port of PS2 (pop)
0xB0000004		LM32_PS2_RX_PEEK	Read port or PS2 (no pop)
0xC0000000	I^2C	LM32_I2C_CTL	I^2C Control register

The address decoder module (address_decode) generates the appropriate selects for each of the individual peripherals (using the auto-generated veriog header), whilst the registers and fields within the peripherals themselves are decoded by the periperal's own logic (again with reference to definitions in the header). The UART and Timer are peripherals defined in separate modules, whereas the logic for the GPIO, Switches, PS2 and I^2C resides in the controller module, as does the logic to arbitrate access to the memories between the host and the lm32 CPU.

USB-JTAG Protocol

The logic for decoding this protocol data is based on the Intel/Altera examples supplied with the DE1 board, with some small modifications and additions. The logic is wrapped in a top level module (`usb_jtag_cmd`), which has two sub modules to decode the commands: `USB_JTAG` to deserialise input, and serialise output, and `CMD_Decode` to interpret and respond to the command. From the hardware perspective, data arrives at the JTAG interface as sets of 64 bit commands, the logic shifts this into a 64 bit register (MSb first), and at the and of the transfer, this register contains the following fields:

- `CMD_Action` (bits 63:56)
  - Valid values are `SETUP`, `ERASE`, `WRITE`, `READ`
- `CMD_Target` (bits 55:48)
  - Valid values are `LED`, `SEG7`, `PS2`, `FLASH`, `SDRAM`, `SRAM`, `VGA`, `SDRSEL`, `FLSEL`, `SET_REG`, `SRSEL`
- `CMD_ADDR` (bits 47:24)
- `CMD_DATA` (bits 23:8)
- `CMD_MODE` (bits 7:0)
  - Valid values are `OUTSEL`, `NORMAL`, `DISPLAY`

The action, target and mode fields have 'enumerated' values, which are defined in the file `<install dir>/HDL/rtl/RS232_command.vh`. When a command action is a `READ`, an additional transfer takes place of 16 bits, where 16 bits of data are returned corresponding to the data at the address of the command that was sent. The actions are four simple operations.

Firstly, the `SETUP` command is used exclusively with the `OUTSEL` mode, and this selects which data paths are selected for the Flash, SDRAM or SRAM, as specied by the target value (`SDRSEL`, `FLSEL`, `SRSEL`), with the selected path chosed from the lower two bits of the

CMD_DATA field. Similarly, for the USB-JTAG return read data, the action is SETUP, the mode is OUTSEL, but the target is SET_REG. The read data source is selected from Flash, SDRAM, PS2, SRAM and seven segement display via the CMD_DATA field, using the same enumerations as for CMD_Target. The ERASE action is exclusively used for Flash erasing.

The READ and WRITE actions are used, along with a specified target, to do accesses to the particular peripherals. Some targets are write only (LED, VGA), some targets are read only (PS2). Most reads and write use the NORMAL mode, but the display targets (VGA, SEG7 and LED) have a mode of DISPLAY, presumably as they are not memory or register writes that can be read back, as would be the case for memories.

All data transfers are 16 bits, and it is up to the logic to discard unused write bots, or pad out read data if less than 16 bits.

Driver

Included in the package is some driver code (in <install dir>/HDL/driver). As delivered, its function is to take an compiled program ELF file and download it to SRAM on the board and enable execution by the processor.

The driver utilises the low level FTDI 2xx driver code, a version of which comes with the DE1 board software (for windows). The latest drivers, and ones for Linux, can be down loaded from the FTDI website (http://www.ftdichip.com/Drivers/D2XX.htm). In the driver the FTDI driver is wrapped into a clas USB_JTAG defined in USB_JTAG.h, and the API for this is detailed in this file as the class header. However, macros are defined for this interface (in lm32_driver.h) to hide the specific details. A set of macros are also defined which hide the details of the protocol mentioned in the previous section, as well as some to set the seven segment display to certain predefined values.

The device must be opened before use, and closed when finidhed with. In addition, the device can be reset after opening (though opening performs a reset).

- USB_OPEN_DEVICE(USB_JTAG* p_device)
- USB_CLOSE_DEVICE(USB_JTAG* p_device)
- USB_RESET_DEVICE(int sleep_time, USB_JTAG* p_device)

After the device is opened successfully access to the device can be made through the other macros. The SRAM must be selected for the driver to have access, when read and write becomes available. When host access is finished it can be deselected to allow the CPU access once more.

- USB_SELECT_SRAM(uint8_t char *databuf, USB_JTAG* p_device)
- USB_DESELECT_SRAM(uint8_t *databuf, USB_JTAG* p_device)
- USB_WRITE_SRAM(uint16_t val, uint32_t addr, uint8_t *databuf, USB_JTAG* p_device)
- USB_READ_SRAM(uint32_t addr, uint8_t *databuf, USB_JTAG* p_device)

The databuf byte buffer must be a pointer to a buffer of at least 8 bytes. It is used to construct the command to be sent of the interface, and also return read data. The LEDs can be set and cleared, as can the seven segment display

- USB_SET_LEDS(int val, int curr_state, uint8_t *databuf, USB_JTAG* p_device)
- USB_CLR_LEDS(int val, int curr_state, uint8_t *databuf, USB_JTAG* p_device)
- USB_SELECT_SEG7(int val, uint8_t char *databuf, USB_JTAG* p_device)
- USB_SET_SEG7_DEC(int val, uint8_t char *databuf, USB_JTAG* p_device)
- USB_SET_SEG7_HEX(int val, uint8_t char *databuf, USB_JTAG* p_device)
- USB_SET_SEG7_ALT(int v3, int v2, int v1, int v0, uint8_t *databuf, USB_JTAG* p_device)
- USB_READ_SEG7(uint8_t char *databuf, USB_JTAG* p_device)

The LED macros us the value (`val`) a bit map, with the bit positions set setting those LEDs, and the bit positions clear leaving the LED state unchanged. As the LED state can't be read, the macro requires an `led` argument (intialised to zero before the first call to the LED macros) where the state of the LEDs is maintained.

The seven segment display macros come in various flavours. The LEDs must be selected first with `USB_SELECT_SEG7`, and then the display can updated with a decimal number or a hex number. For the other types of symbols and alternative version of letters and numbers, the `USB_SET_SEG7_ALT` is used. A set of definitions are in the header for the values that are supported for various symbols, alternative cases and other non-hex and non numerical settings. See the header also for so predefined macros that spell out useful words and display on all four segments: e.g. `USB_SEG7_PASS().`

The driver program makes use of these macros to load the ELF program to memory and set offthe processor. It (optionally) waits for a termination signal from the program. When compiled the driver program has the resultant usage:

```
Usage: lm32_driver [-f <filename>][-d <delay ms>][-n|-l][-w]

    -f Specify ELF file to load (default test.elf)
    -d Specify delay (in ms) after program signals termination (default 0)
    -n No load of program, only execute (default false)
    -l Only load program, don't execute (default false)
    -w Don't wait for program termination (default wait)
```

Normally the program only needs the ELF filename specified (and not even that if it is called `test.elf` and is local). Some additional options are available. A delay can be programmed for after the program detects a CPU termination before the device is closed (for any buffer flushing that may be needed---but this is not required with delivered test suit). If a program is loaded in memory already, either previously, or by some other means, than the driver can be instructed to run the program without downloading one first. Similarly, a program can be loaded but not executed. Finally, the driver can be instructed not to wait for a program termination indicator for case where none is programmed in the executing code, or the code is to run indefinitely.

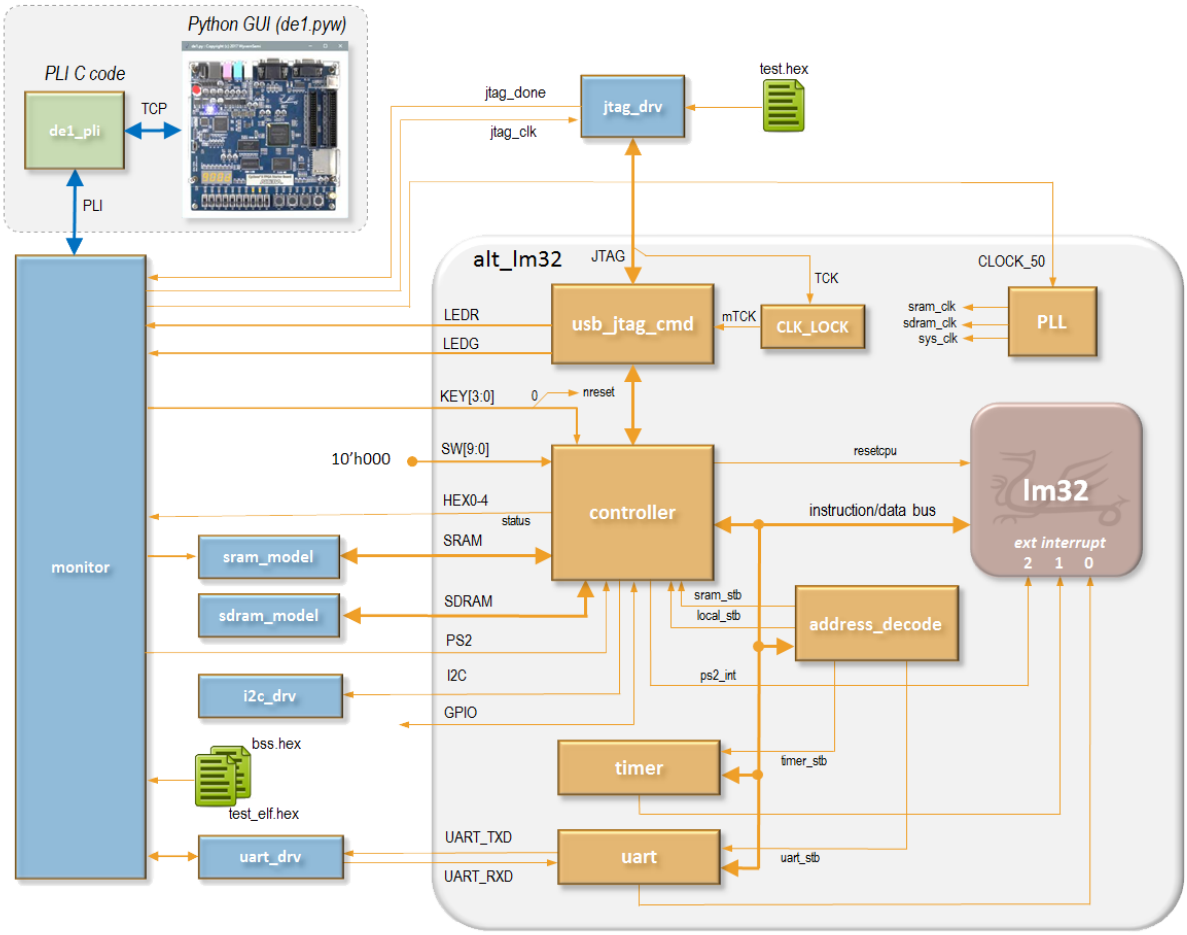
The termination indication comes from reading the seven segment display. When this is non-zero, the program has terminated. The logic is configured to display the contents of address `0xfffc`, nominated as a special test address that has test program pass/fail status written to fro the CPU. When a write to that address occurs, it is latched to the register that is diplayed on the seven segment display. All the suite of tests in the package clear the value at the beginning of the test, and update this value at the end of the test (or on failure). Although the driver can read SRAM locations, the SRAM has been deselected (for host) and is being used by the CPU, so is unavailable during execution. The seven segment display can be read at any time.

Simulation

A simulation test environment is bundled with the package. It is verilog based, and constructed for use on Modelsim, though can be adpated for other simulation targets. As well as drivers for the major interfaces, the harness can load and run a program on the lm32 processor, send commands over the USB-JTAG interface, and has an optional display representing the DE1 board, indicating the state of the LED and seven segment display.

Test Harness

The top level test harness design is shown in the diagram below.



The harness instantiates the `alt_lm32` top level as the UUT and provides clock and reset for the module. Most of the interfaces have a driver of some sort, with the exception of the switches and GPIO (at this time). The command USB-JTAG interface is connected to a driver that read a command file and sneds commands over the interface. The I2C interface has a driver that will display a message when wrtten to, and the UART has a driver module.

There are SRAM and SDRAM models, with the ability to load a program image directly into memory by the harness, negating the need to simulate the loading of this via the USB-JTAG interface (though this can still be done if desired).

The main test module is the `monitor`, which is responsible for generating clocks and resets, loading program data to memory, relaying LED and seven segment status to the DE1 python GUI over the PLI, display UART data and sending input to the UART, and terminating the simulation. As the USB-JTAG commands are not controlled directly by the monitor, but by a

separate module (usb_jtag_cmd), a jtag_done signal is sent to the monitor to hold off termination until the commands are completed.

JTAG Driver

The USB-JTAG driver sits in place of the host driver code within the test harness. It reads a verilog format test hex file (called test.hex by default, that resides in the test directory), which specifies when command to send over the interface, and at what times. A termination command indicates to the driver when there are no more commands, and the drive flags that it is done.

test.hex File Format

The format of the commands in the test,hex file take the following generate format:

- A four byte delta time, counting from the end of the last command
- A one byte length for the command
- The command itself

The time command is fairly self explanitory, and the units are in JTAG clock cycles. It is MSB first format. The length byte indicates the bytes in the following command. Usually this is 8 bytes to send, say, a write command, but might be 2 bytes for the read following a read command. The actual commands themselves must be hex number, so no emumeration is supplied. The order of the bytes is as shown below, starting with the first byte:

0. action
    - Valid values: 61=SETUP, 72=ERASE, 83=WRITE, 94=READ
1. target
    - Valid values: f0=LED, e1=SEG7, d2=PS2, c3=FLASH, b4=SDRAM, a5=SRAM, 87=VGA, 1f=SDRSEL, 2e=FLSEL, 4c=SET_REG, 5b=SRSEL
2. hw_addr (bits 23 down to 16)
3. hw_addr (bits 15 down to 8)
4. hw_addr (bits 7 down to 0)
5. data (bits 15 down to 8)
6. data (bits 7down to 0)
7. mode
    - Valid values: 33=OUTSEL, aa=NORMAL, cc=DISPLAY, ff=BURST

An example fragment of a test fle is shown below:

```
.
.
.
// SDRAM write 0xcafe to byte address 0x0123456
00 00 00 10
08
83 b4 09 1a 2b ca fe aa

// Setup TXD output for SDRAM
00 00 00 10
08
61 4c 12 34 56 00 b4 33

// SDRAM read from byte address 0x00080604
00 00 00 10
08
94 b4 04 03 02 00 00 aa
```

```
// Send two bytes to flush read data
00 00 00 10
02
00 00
.
.
.
// Terminate
ff ff ff ff
```

I²C driver

The I2C driver simply sits on the interface, monitoring for write activity. If a valid write is detected it will log a message showing the address and the write data. It contains the pullup on the data line, and generates the acknowledge signal. No validation of expected results is done within the module, and this is expectd to be done post simulation by parsing the messages generated by the driver.

Memory and PLL Models

The test harness has two behavioural verilog memory models for the SRAM and SDRAM. They are not, at this point, highly configurable. It also provides behavioural models to replace the `PLL1` and `CLOCK_LOCK` modules within the UUT. Models can be generated for these components from within the Quartus II environment, but these models are highly accurate and very slow, so behavioural models are provided with sufficient operations to facilitate test, whilst running the simulation at an acceptable speed

The SRAM model is fixed as a 512KB asynchronous memory, organised as 256K x 16 bits, reflecting that on the DE1 board. The value of the DQ bus being driven by the model is tristate unless the chip enable and output enable are active (low), and the relevant byte enable is active, with separate upper and lower byte controls.

The SDRAM model can be configured, via parameters, for different numbers of banks (`SDRAM_NUM_BANKS`), rows (`SDRAM_NUM_ROWS`) and columns (`SDRAM_NUM_COLS`), but defaults to the DE1 configuration of 4 banks, 4K rows and 256 columns for an 8MB memory. It has a fully implemented state machine, but performs no program compliance or timing checking, so if commands come in that violate the protocol, the behaviours is undetermined and it may, falsely, function on violating commands and/or timing.

The `CLOCK_LOCK` model (`CLOCK_LOCK_sim.v`) is trivial. Since, in this design, none of the clock controls are used on this component, it is essentially a clock buffer, and the model simply wire the input clock to the output.

The `PLL1` model (`PLL1_sim.v`) does not attempt to model the transitional behaviour of a PLL during capture and lock of an input, but it has a set of parameters, matching those of the actual Altera component, to configure it for various multiply and division settings, as well as phase. The model has a single clock input and three clock outputs. The configurable parameters are:

- `inclk0_input_frequency` (default 20000)
- `clk0_multiply_by` (default 4)
- `clk0_divide_by` (default 5)
- `clk0_phase_shift` (default “0”)
- `clk1_multiply_by` (default 4)
- `clk1_divide_by` (default 5)
- `clk1_phase_shift` (default 4)
- `clk2_multiply_by` (default 8)

- clk2_divide_by (default 5)
- clk2_phase_shift (default “0”)

All the parameters are used to generate the clocks, and so different frequencies can be modelled to match the design. Note that, matching the Altera PLL configuration, the phase shift values are strings, not integers like the other parameters, and the units are in picoseconds, not degrees.

Monitor

The monitor is the heart of the test harness, providing the major control for tests, and monitoring of UUT outputs. It is responsible for generating the main 50MHZ clock and the JTAG clock, and main system reset. It will load a program data image to memory, using a hex file that has been converted from an original LM32 ELF file via a command similar to:

```
objcopy -O verilog test.elf test_elf.hex
```

In addition a bss.hex file, full of zeros is loaded by the monitor in lieu of executing BSS initialisation code on the CPU.

The module also monitor the LED and seven segment dispay outputs, and can optionally relay the status to the DE1 python GUI over the PLI. This gives a visual representation of the DE1 board and is useful in developing code to give useful status, debug and diagnostic information before commiting to running on the platform, where the full simulation debugging facilities are not available. The monitor sends and receives data, via the uart_drv module from the system’s UART. It displays all received characters and will send characters sent over the PLI.

The monitor decides when to terminate the simulation. Currently, two things determine when to end the simulation: the CPU is completed, the JTAG driver has completed and the UART is inactive. The former of these is done by reading the LED status, with read LED 2 being a visual CPU done status. When this is ‘lit’ the CPU has finished. The JTAG ‘done’ signal comes in as an input signal from the JTAG driver. Finally, the UART state (internal to the monitor anyway), is inspected to determine that the UART is inactive. If all three conditions are met, the moinitor either stops of finishes the simulation, depending on settings.

PLI and GUI

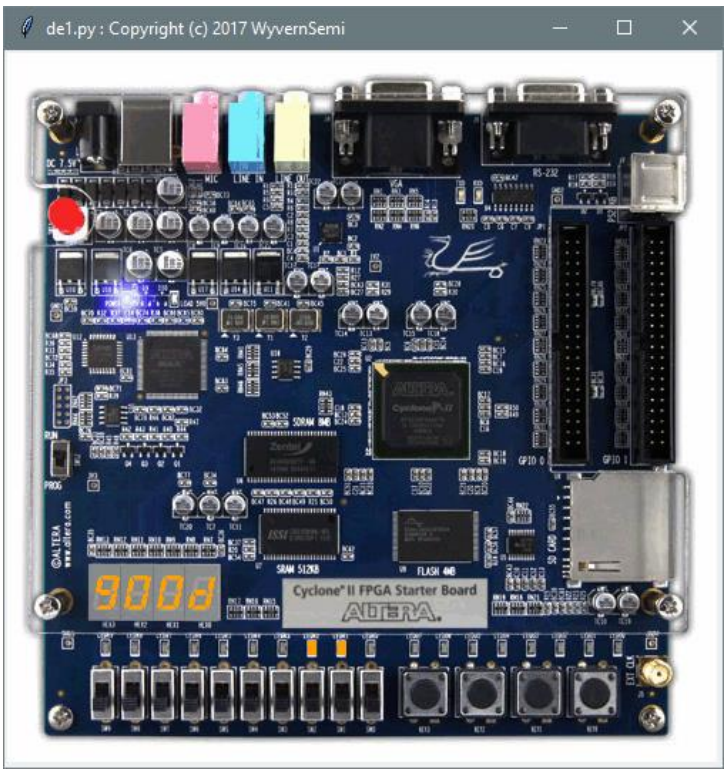
The PLI C code itself resides in the HDL/test/pli directory, and consists of three files:

- de1_pli.c
- de1_pli.h
- veriuser.h

The PLI code connects the simulation to the GUI and also UART input and output. Within the verilog, four functions become avalable:

- $de1getchar()
- $de1putchar(byte)
- $de1init()
- $de1update(cmd, val)

The first two functions are UART get and put charcater functions, and operate as one might expect.The last two are related to the GUI, with $de1init() firing up and initialising the GUI (if enabled), and $de1update() sending display update commands and data to the GUI. The GUI appears in a separate window and looks something like the following diagram:



This is a separate Python script connected to the PLI via a TCP socket on port 49153. It has a particular protocol to update the LEDs and seven segement display. From the perspecitive of the veriog test harness there are three commands:

- `TCP_CMD_HEX` (0) : update the seven segment display
- `TCP_CMD_LEDR` (1): update the read LEDs
- `TCP_CMD_LEDG` (2): update the green LEDs

For example, to update the seven segment display with a value of `0x1234`, the the verilog can call `$de1update(TCP_CMD_Hex, 16'h1234)`. The valid range of values for the `TCP_CMD_HEX` command are 0 to 65535. Similarly for the LEDs the appropriate command is used with `$de1update()`. Since there are 10 red LEDs, valid values range from 0 to 1023. There are 8 green LEDs, so valid values are 0 to 255.

On completion of a simulation the GUI will be closed via the PLI sending a command to the python script. If, however, it is required to terminate the script manually, the on/off button on the GUI can be clicked, and the script will close.

Building Code

Building the simulation is simply a matter of running `make`, in the `HDL/test` directory. For a Windows environment this assumes that GnuWin32 is installed, or Cygwin and run from a Cygwin terminal. This will do an incremental compile, but for a complete compile run `make clean` first.

Compile options

When compiling the simulation some compile options are available to control the build. These can be defined on the make command line vi the `VUSRFLAGS` make variable.---e.g. `make VUSRFLAGS=+define+DISABLE_PLI` The valid definitions are listed below.

- `DISABLE_PLI
- `DISABLE_BSS_INIT
- `SIM

The `DISABLE_PLI definition, when set, will remove building all code associated with the DE1 PLI GUI., should this never be required for the build. There is a means to disable it at runtime, so this is for permenantly deleting it.

The `DISABLE_BSS_INIT definition will compile out code that reads a BSS file (bss.hex by default). If a BSS section is not needed then to avoid having a dummy bss.hex file, the code can be compiled with this option.

The last definition (`SIM) is listed for completeness, but will be set by the makefile. It modifies certain parameters to make the timing of serial interfaces ten times faster to reduce simulation time. It would not normally be changed on the command line, but the makefile can be modified to remove this definition if it is required to simulate at the correct speed.

There should be no reason to build the project without reference to the make method, but for reference, the command for ModelSim looks somethings like the following (for Linux), equvalent to a make:

```
vlog -incr +define+SIM +incdir+../registers –f test.vc
```

Of course, any of the above listed definitions can alos be added to this command line, with a +define argument. The build assumes that the PLI code was made (make –C ./pli), and that the registers headers have been built (make –C ../registers). This is all taken care of by the local makefile, and is far simpler than a manual compilation.

Running Tests

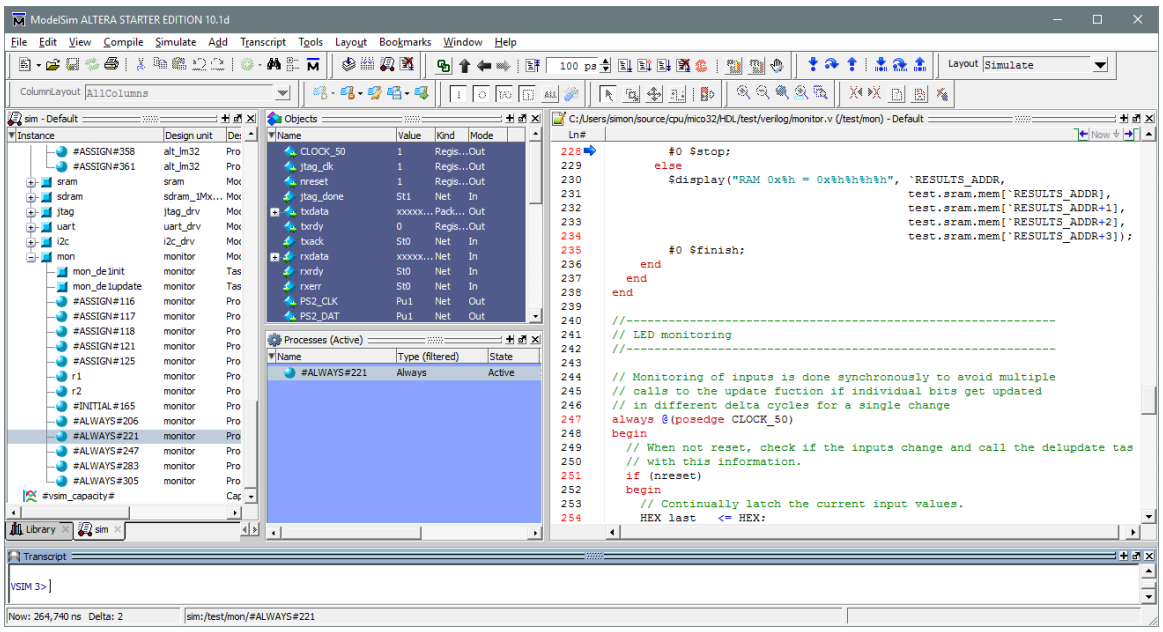
From the HDL/test directory, running a simulation is a simple matter of executing make run. This will (if needed) also build the simulation, and the options described aboe are available. There is an assumption that the directory contains a test.hex, a bss.hex (unless disabled) and a test_elf.hex file. Note that a make clean will remove the test_elf.hex file. The program hex file is generated from an ELF file (e.g. test.elf) using objcopy. E.g.

```
objcopy –O verilog test.elf test_elf.hex.
```

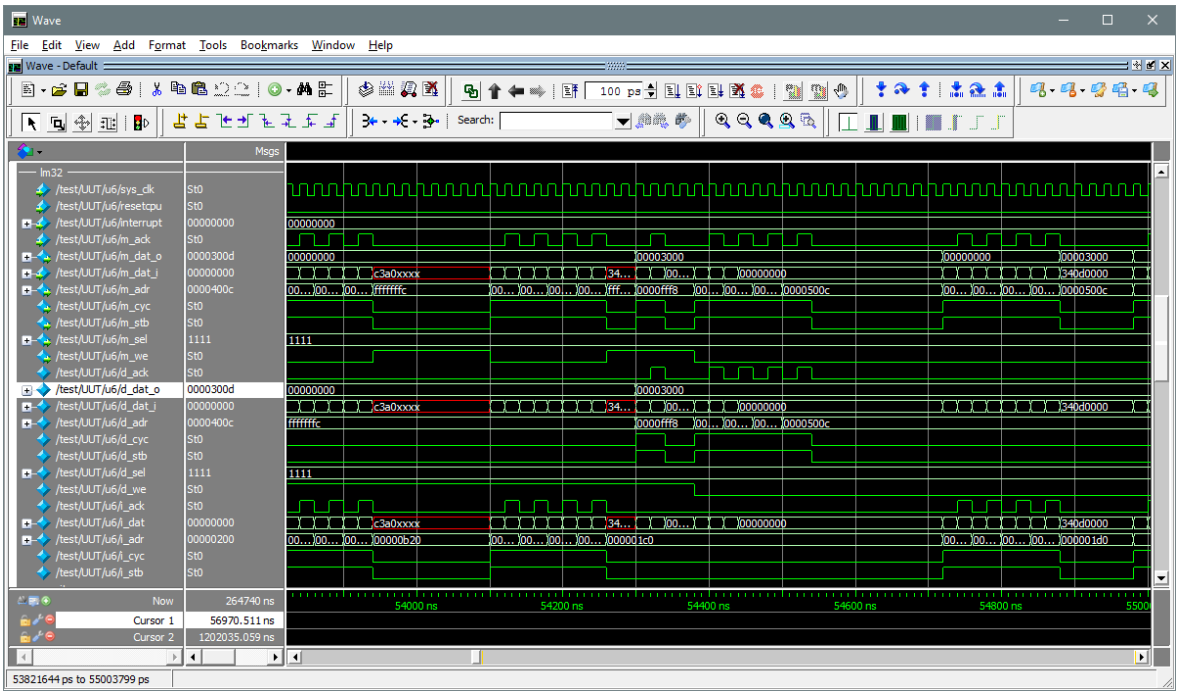
Any of the relevant tests in the <install dir>/test/… can be used as a source of a test hex file. In addition some variations on the basic make run are available.

- make rungui: Run with ModelSim GUI
- make gui: Synonymous with above
- make runnb: Run without building
- make regression: Run regression tests

When running with the GUI, the ModelSim front end will window will open up with the lm32 project already loaded. This looks something like the following diagram:



If a `wave.do` file has previously been created, and exists in the test directory, a waveform view is opened. This will look something like the following diagram:



A `wave.do` file can be created when running a ModelSim GUI session by setting up a wave view with the desired signals selected. When run and a wave view window displayed, the setup can be saved into wave.do from the `file->Save Format...` pulldown menu of the window (or type CTRL-S). The default value for the filename is `wave.do`. If the wave signals are changed, then the `wave.do` file can be updated in the same manner, and at the next GUI run will display with these latest modifications.

Runtime Options

In addition to ocmpile options some ‘plusargs’ runtime options are made available to configure the run. These are set via the `USRFLAGS` makefile variable. E.g. `make USRFLAGS="FINISH=0"`. The valid values are shown below.

- `+JTAG_FILE="<filename>"` (default `test.hex`)
- `+PROG_NAME="<filename>"` (default `test_elf.hex`)
- `+BSS_FILE="<filename>"` (default `bss.hex`)
- `+FINISH=[0|1]` (default 0)
- `+DISABLE_GUI=[0|1]` (default 0)

The first three options allow the user to select the different test files from the defaults. The `FINISH` argument will force the simulation to execute `$finish` when completed instead of `$stop`. This is useful for running regression and batch tests. The default is to run `$stop` so that, when running in the ModelSim GUI environment, the simuation does not end. The `DISABLE_GUI` option allows the simulation to be run without firing up the DE1 python GUI—again, useful for regression and batch runs.

There should be no reason to run any of the tests without reference to the make method, but for reference, the command for ModelSim looks somethings like the following (for Linux), equvalent to a `make run`:

```
vsim -c -pli ./pli/lib/de1_pli.so test -suppress 3017,3722 -quiet -do "run -all"
```

If a GUI is required, then one can add the arguments `-gui -do wave.do`, in place of the `-c` and `-quiet` arguments

Synthesis

Synthesis for the design targets the Intel/Altera 2C20 Cylone II FPGA, as used on the DE1 development board, and uses the Quartus II software (v13.0 SP1). The home directory for synthesis is `<install dir>/HDL/synth/altera`, and the synthesis is run from this location.

For the Quartus II software a project file is defined (`alt_lm32.qpf`), and a setup file (`alt_lm32.qsf`). The project file specifies very little and is fairly standard. The setup file specifies the device type, the top level entity, all the pinout attributes, and would normally kist the design files. In this project these have been separated out into another file (`synth_file_paths.tcl`) that is included into the setup file (or sourced as a TCL file, actually). This is because, though most settings are fixed, the file list is the most likely to change, and can be managed separately as far as revision control is concerned.

Constraints

The design synthesis is constrained with a Synopsys design contrainst file (`alt_lm32.sdc`). In here the mast clock is specified (`CLOCK_50`) and the generated clocks (`sys_clk`, `sdram_clk` and `jtag_clk`), relative to the master clock. The JTAG clock is false pathed with the other clocks, as it is treated as asynchronous in the design, and the inputs and outpts are given some constraints.

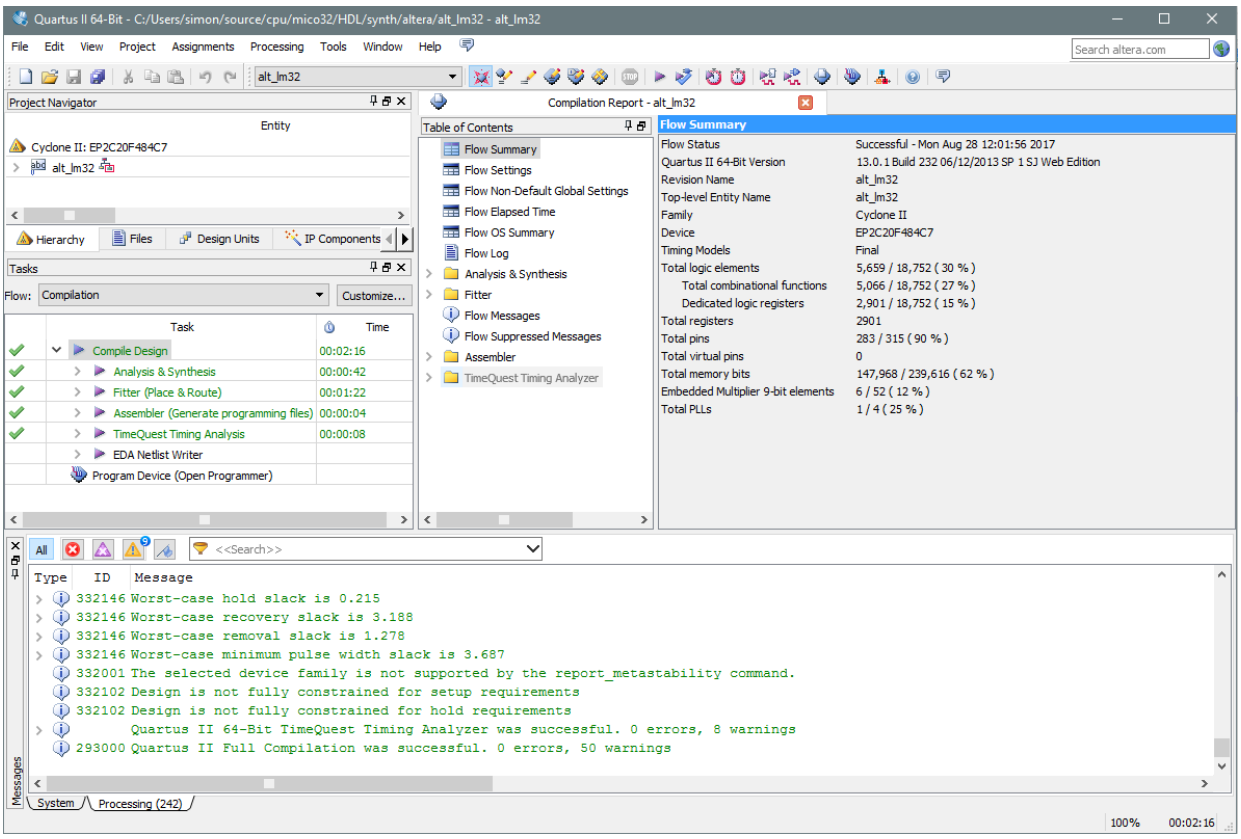
The master clock timing specifications is set for a frequency of 66.67MHz instead of 50MHz (as it actually) to add timing margin to the synthesis, in addition to some specified timing uncertainity on the clock.

Building

As for simulation, the synthesis build is controlled via the make command, and typing `make` will synthesise the design. The synthesis consists of fource stages: map, fit, assemble and static timing analysis. The build can be up to any of these intermediate points with, respectively, `make map`, `make fit`, `make asm` and `make sta`. The make script ensures any pre-stage is built before the requested stage. Again, as for simulation, the register headers must be built (i.e. `make -C ../../registers`), but the script takes care of this.

After synthesis, there will be four report files, one for each of the four stages, in the alt_lm32/ directory created in the synthesis directory. Perhaps the most useful one is that for static timing analysis (STA), whch will detail any timing contraint violations. In addition to the reports files, in the synthesis directory will be the prgramming files `alt_lm32.pof`, and `alt_mem.sof`. The one we are interested in is the `.sof` file, as this is used to program the FPGA directly (rather than an eeprom to program the device at power up).

One can synthesise the design using the Quartus II tool's GUI. From within the synthesis directory, simply type `quartus alt_lm32.qpf`. In the resultant window, in the flow box, right click on the compilation entry and select `Start`. Synthesis will proceed (assuming no errors) until STA is completed. The resultant window looks something like the diagram below.


Quartus II 64-Bit - C:/Users/simon/source/cpu/mico32/HDL/synth/altera/alt_lm32 - alt_lm32
File Edit View Project Assignments Processing Tools Window Help
Search altera.com
alt_lm32
Project Navigator
Entity
Cyclone II: EP2C20F484C7
alt_lm32
Hierarchy Files Design Units IP Components
Tasks
Flow: Compilation
Customize...
Task Time
Compile Design 00:02:16
Analysis & Synthesis 00:00:42
Fitter (Place & Route) 00:01:22
Assembler (Generate programming files) 00:00:04
TimeQuest Timing Analysis 00:00:08
EDA Netlist Writer
Program Device (Open Programmer)
Compilation Report - alt_lm32
Table of Contents
Flow Summary
Flow Settings
Flow Non-Default Global Settings
Flow Elapsed Time
Flow OS Summary
Flow Log
Analysis & Synthesis
Fitter
Flow Messages
Flow Suppressed Messages
Assembler
TimeQuest Timing Analyzer
Flow Summary
Flow Status Successful - Mon Aug 28 12:01:56 2017
Quartus II 64-Bit Version 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name alt_lm32
Top-level Entity Name alt_lm32
Family Cyclone II
Device EP2C20F484C7
Timing Models Final
Total logic elements 5,659 / 18,752 (30 %)
Total combinational functions 5,066 / 18,752 (27 %)
Dedicated logic registers 2,901 / 18,752 (15 %)
Total registers 2901
Total pins 283 / 315 (90 %)
Total virtual pins 0
Total memory bits 147,968 / 239,616 (62 %)
Embedded Multiplier 9-bit elements 6 / 52 (12 %)
Total PLLs 1 / 4 (25 %)
All
<<Search>>
Type ID Message
332146 Worst-case hold slack is 0.215
332146 Worst-case recovery slack is 3.188
332146 Worst-case removal slack is 1.278
332146 Worst-case minimum pulse width slack is 3.687
332001 The selected device family is not supported by the report_metastability command.
332102 Design is not fully constrained for setup requirements
332102 Design is not fully constrained for hold requirements
Quartus II 64-Bit TimeQuest Timing Analyzer was successful. 0 errors, 8 warnings
293000 Quartus II Full Compilation was successful. 0 errors, 50 warnings
Messages
System
Processing (242)
100%
00:02:16

Platform

As mentioned numerous times, the target platform is the terasIC DE1 Cyclone II development board with a Cylone II 2C20 device. Having synthesised the design and obtained a .sof file, the design may be loaded onto the platform and then programed an run.

It is assumed at this point that a DE1 board is powered up and connected to the host computer via the USB cable, such that the USB-Blaster device is visible.

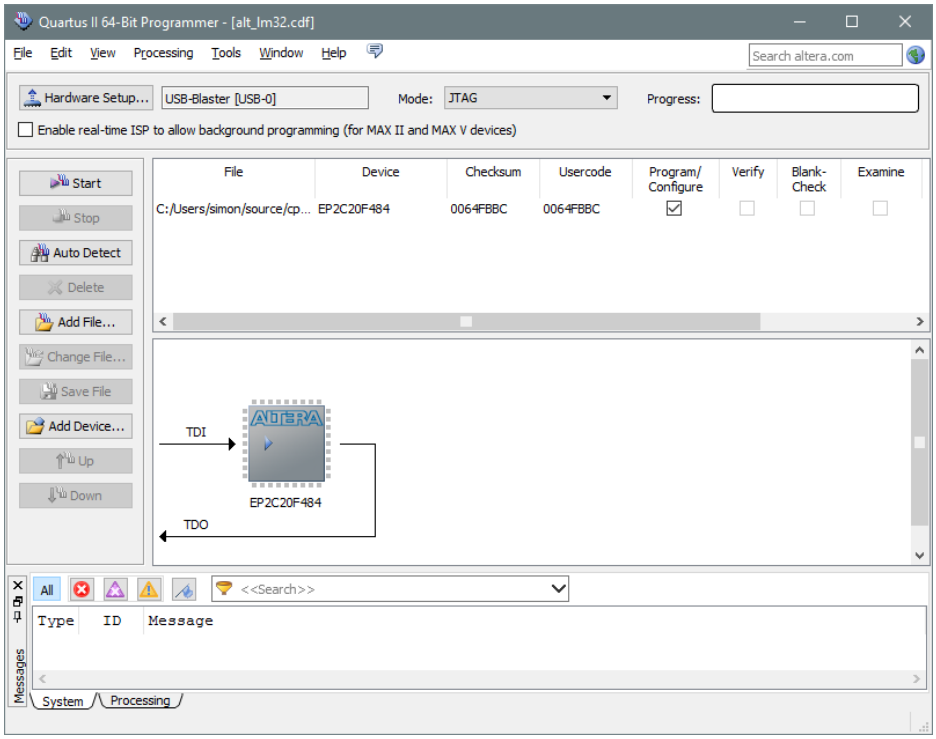
Configuring the FPGA

The design can be down loaded to the board via the make command:

```
make prog
```

This will also build the design first if this hasn't been done already. One can also type the command `make prognb` to download a design without a build if it is known that the file is up to date or to load an archived file, say.

The board can be programmed from the Quartus tool tools gui as well. Either from the main GUI windows (Tools->Programmer), or using the command `quartus_pgmw alt_lm32.sof`. A window will open that looks something like the following:



Clicking Start will configure the board's FPGA. Firing up this GUI is also a good way of checking if the USB blaster is detected if there are issue using make.

Running Tests

To run a single test this would normally be done via the program driver code (see driver section), which will load a program and execute it on the platform, assuming it is already programmed. For example:

```
lm32_driver -f test.elf
```

If the code that's run was one of the test suite, then a message is printed out: E.g.:

```
RAM 0xfffc = 0x0000900d
```

All of the relevant regression test suite can be run in one go from the synthesis directory using the command:

```
make regression
```

If all is well the the output will look somethings like:

```
Running test instructions/add
  PASS
Running test instructions/sub
  PASS
Running test instructions/cmp_e_ne
  PASS
    .
    .
    .
Running test instructions/mul
  PASS
Running test exceptions/instruction
  PASS
Running test mmu/tlb
  PASS

Tests run : 19
Tests pass: 19
Tests fail: 0
```

The regression tests for the simulation and the platform can also all be run from the top level installation directory with `make sim` and `make hwtest`. If the project has been installed on top of the instruction set simulator, than typing `make alltest` will run all tests on all platforms, building all that is necessary to do this.

Appendix A: File Structure

Files

```
<install dir>
 |
 |-- runtest.py                                    -- Common top level test execution script
 |-- makefile                                      -- Common top level build and run make file
 |
 |-- HDL/
 |   |-- driver/
 |   |   |
 |   |   |-- msvc/                                 -- Driver MSVC files (Express 10)
 |   |   |   |
 |   |   |   |-- lm32_driver.sln
 |   |   |   |-- lm32_driver.vcxproj
 |   |   |   |-- lm32_driver.vcxproj.filters
 |   |   |   `-- lm32_driver.vcxproj.user
 |   |   |
 |   |   |-- lib/                                  -- Driver code libraries from FTDI
 |   |   |   |
 |   |   |   |-- ftd2xx.dll
 |   |   |   |-- FTD2XX.lib
 |   |   |   |-- libftd2xx.a
 |   |   |   |-- libftd2xx.so.1.3.6
 |   |   |   |-- libftd2xx32.a
 |   |   |   `-- libftd2xx32.so.1.3.6
 |   |   |
 |   |   `-- src/                                  -- Source code for driver
 |   |       |
 |   |       |-- ftd2xx.h                          -- FTDI headers
 |   |       |-- USB_JTAG.h
 |   |       |-- WinTypes.h
 |   |       |-- lm32_driver.cpp                   -- Driver source code
 |   |       |-- lm32_driver.h
 |   |       |-- lm32_driver_elf.cpp
 |   |       `-- lm32_driver_elf.h
 |   |
 |   |-- registers/                                -- Register definitions
 |   |   |
 |   |   |-- registers.xlsx
 |   |   `-- makefile
 |   |
 |   |-- test/                                     -- Test harness
 |   |   |
 |   |   |-- de1_top.vc
 |   |   |-- lm32.vc
 |   |   |-- test.vc
 |   |   |-- test_harness.vc
 |   |   |-- makefile
 |   |   |-- test.hex
 |   |   |-- radix.do
 |   |   |
 |   |   |-- verilog/                              -- Test harness source code
 |   |   |   |
 |   |   |   |-- test.v
 |   |   |   |-- sram.v
 |   |   |   |-- sdram.v
 |   |   |   |-- monitor.v
 |   |   |   |-- jtag_drv.v
 |   |   |   |-- uart_drv.v
 |   |   |   |-- i2c_drv.v
 |   |   |   |-- PLL1_sim.v
 |   |   |   |-- CLK_LOCK_sim.v
 |   |   |   `-- test_defs.vh
 |   |   |
 |   |   `-- pli/                                  -- PLI routines for comms with GUI and screen IO
 |   |       |
 |   |       |-- de1_pli.c
 |   |       |-- de1_pli.h
 |   |       |-- veriuser.c
 |   |       `-- makefile
 |   |
```

```
|   |-- rtl/
|   |   |
|   |   |-- alt_lm32.vc                              --  Top level RTL
|   |   |-- alt_lm32.v
|   |   |-- lm32_config.v
|   |   |-- address_decode.v
|   |   |-- controller.v
|   |   |-- usb_jtag_cmd.v
|   |   |-- wb_mux.v
|   |   |-- lm32_wrap.v
|   |   |-- CMD_Decode.v                             -- Top level modified files from DE1 examples
|   |   |-- USB_JTAG.v
|   |   |-- Flash_Command.vh
|   |   |-- RS232_Command.vh
|   |   |-- SEG7_LUT.v
|   |   |-- SEG7_LUT_4.v
|   |   |-- I2C_Controller.v
|   |   |-- Sdram_Controller\Sdram_Controller.v
|   |   |-- Sdram_Controller\command.v
|   |   |-- Sdram_Controller\control_interface.v
|   |   |-- Sdram_Controller\sdr_data_path.v
|   |   |-- Sdram_Controller\Sdram_Params.h
|   |   |-- PLL1.v                                   -- Altera Quartus II generated files
|   |   `-- CLK_LOCK.v
|   |
|   `-- synth/
|       `-- altera/                                  -- Altera synthesis files
|           |
|           |-- alt_lm32.qpf
|           |-- alt_lm32.qsf
|           |-- alt_lm32.sdc
|           |-- alt_lm32.srf
|           |-- makefile
|           `-- synth_file_paths.tcl
|
|-- python/                                          -- Python support code
|   |
|   |-- de1.pyw
|   |-- de1.gif
|   `-- reg_gen.py
|
`-- third_party/                                     -- Third party source code
    |
    |-- m-labs/                                      -- MLAB's MMU lm32 core
    |   |
    |   `-- lm32
    |       |
    |       |-- rtl/...
    |       `-- test/
    |          |
    |           `-- lm32_config.v
    |
    `-- lscc                                         -- Lattice's peripheral cores
        |
        `-- micosystem/
           |
            `-- components/
                |
                |-- timer/...
                `-- uart_core/...
```

Appendix B: List of Tools

In this appendix is listed the tools used in this project. Some are essential, whilst others are simply useful and ease dvelopment and support. All (except where noted) are freeware tools, as cost constarinst are a key factor in this design.

Equipment

In order to run the design on hardware, the following equipment was used. These components were the *only* expense (other than a PC, obviously) that was necessary to get a working system. At the time of writing, the board cost £150 ($200) and the cable cost £17 ($22), though I have seen cheaper cables (£2/$2.50).

The design was targetted at the particular development board, but could (somewhat) easily be ported to other platforms, particularly other terasIC Cyclone boards.

Cyclone II DE1 platform:	http://www.terasic.com.tw/en
USB to RS-232 converter:	http://plugable.com/products/pl2303-db9

Other Tools

Other tools of interest, used locally by the author in the development of the project and its publication (all excellent, and all for free).

Recommended for support in running under Windows:

Eclipse (Neon used)	https://www.eclipse.org/downloads
Visual Studio Community:	https://www.visualstudio.com/downloads
Cygwin:	https://www.cygwin.com
GnuWin32	http://gnuwin32.sourceforge.net/
PyCharm:	https://www.jetbrains.com/pycharm
PyDev (for Eclispe):	http://www.pydev.org
PuTTY:	http://www.putty.org
com0com (original):	http://com0com.sourceforge.net
com0com (signed):	https://code.google.com/archive/p/powersdr-iq/downloads

Other useful tools:

Ubuntu Linux (16.04LTS):	https://www.ubuntu.com
gcc/g++:	https://gcc.gnu.org
gdb:	https://www.gnu.org/software/gdb
CVS:	http://savannah.nongnu.org/projects/cvs
TkCVS:	http://www.twobarleycorns.net/tkcvs.html
INNO setup:	http://www.jrsoftware.org/isinfo.php
Notepad++:	https://notepad-plus-plus.org
gVim:	http://www.vim.org
Visual Studio Code:	https://code.visualstudio.com
ConEmu:	https://conemu.github.io
filezilla:	https://filezilla-project.org
firefox:	https://www.mozilla.org/en-GB/firefox/desktop
gimp:	https://www.gimp.org

FreeFileSync: https://www.freefilesync.org
7Zip: http://www.7-zip.org
hexedit: http://www.geoffprewett.com/software/hexedit
Verilator: https://www.veripool.org/wiki/verilator
Icarus verilog: http://iverilog.icarus.com
Windows driver Kit: https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit
JPEGsnoop: http://www.impulseadventure.com/photo/jpeg-snoop.html

Appendix C: Related Documentation

gdb remote debug protocol: https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html
Elf file format: http://refspecs.linuxbase.org/elf/elf.pdf
ext2 format: http://e2fsprogs.sourceforge.net/ext2intro.html
LatticeMico32 Reference: www.latticesemi.com/view_document?document_id=52077
LatticeSemi UART manual: http://www.latticesemi.com/view_document?document_id=32336
LatticeSemi Timer manual: http://www.latticesemi.com/view_document?document_id=51128
Wishbone Interconnect spec: http://cdn.opencores.org/downloads/wbspec_b4.pdf
lm32-mmu documentation: https://github.com/m-labs/lm32/blob/master/doc/mmu.rst
Cyclone II DE1 board ref:
https://www.altera.com/en_US/pdfs/literature/manual/mnl_cii_starter_board_rm.pdf
Cyclone II DE1 user guide:
https://www.altera.com/en_US/pdfs/literature/ug/ug_cii_starter_board.pdf
SDRAM: http://www.issiusa.com/pdf/42S16400.pdf
SRAM: http://www.issi.com/WW/pdf/61-64WV25616.pdf