A-Z80 CPU User's Guide

An FPGA project recreating the Z80

12/14/2014

Revision History

Date	Revision	Change
2014-12-14	1.0	Initial revision
2014-12-21	1.1	Added ZX Spectrum Turbo mode, speaker blink etc.

Table of Contents

Revision History 1
Introduction 3
Project Directory Structure 4
Environment 5
Simulation 6
  Module simulations 6
  Top-level simulations 9
Verification 11
  Fuse tests 11
  Selected functional tests 12
  Z80 Assembly level tests 13
Tools 14
  PLA Checker Tool 14
  Arduino Tools 17
Integration 18
Interface 19
Sample Implementations 20
  Simple host 20
  Sinclair ZX Spectrum 22
Advanced Topics 24
  Modifying the A-Z80 CPU 24

Introduction

A-Z80 is a conceptual implementation of the venerable Zilog® Z80 processor targeted to synthesize and run on a modern FPGA device. It differs from the existing Z80 implementations in that it is designed from the ground-up through the schematics and low-level gates.

This design is capable of mimicking the actual Z80 CPU and it illustrates its inner workings.

The A-Z80 implementation strives to be internally structurally identical to the original Z80. Using this approach the model achieves a full cycle accuracy and has identical behavior for all documented and undocumented features (*) not by explicitly hard-coding them but by mimicking their actual design.

Various *Zilog Z80* references are widely available so the CPU, its instructions and behavior will not be covered in this document.

This document focuses on the structure and mechanics of working with the A-Z80 project; it should help you understand it and incorporate it into your designs.

You can read more about the conception and implementation of the A-Z80 on its home website: www.baltazarstudios.com .

Project Directory Structure

A-Z80 project can be downloaded at OPENCORES as a SVN repo http://opencores.org/project,a-z80 and also on Bitbucket: https://bitbucket.org/gdevic/a-z80 .

The following table describes its hierarchical directory structure:

Directory	Sub-directory	Description
cpu		Contains all core files of the A-Z80 CPU
	alu	Arithmetical and Logical Unit files
	bus	Various bus-related files
	control	Control unit files
	registers	Register block files
	toplevel	A-Z80 top level interfaces and projects
docs		Documentation and schematic images
host		Two implementations using the A-Z80 on Altera DE1 FPGA
	basic	Basic computer containing UART mainly for testing and verification
	zxspectrum	Sinclair ZX Spectrum implementation
resources		General project resources and scripts
tools		Building and testing utilities and misc. files
	Arduino	Software for Arduino Mega dongle to interface with a Z80
	dongle	Dongle and simulation scripts and golden files
	z80_pla_checker	Windows utility to test and create A-Z80 PLA tables
	zmac	Z80 test and verification assembler files

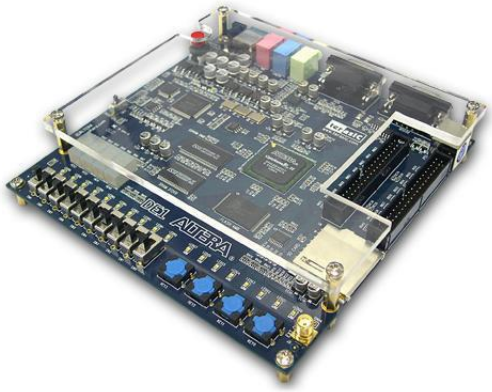
Environment

A minimal set of tools needed to compile various parts of the project is:

- Altera Quartus II Web Edition (Free)
- ModelSim (Altera edition) – needed only for module simulation (Free)
- Python 2.7 – needed only to change and compile CPU modules. All necessary files needed to include A-Z80 sources in your own project are included (Free)
- Microsoft Visual Studio 2010 SP1 – needed only to recompile the z80_pla_checker tool yourself. This is normally not needed since the sources and precompiled executable are checked in with the project.

This project is developed and tested on a Windows 7 OS. Your mileage may vary on Linux.

All designs are tested on an Altera FPGA DE1 board:
http://www.altera.com/education/univ/materials/boards/de1/unv-de1-board.html



This particular board has a **Cyclone II EP2C20F484C7** FPGA alongside a number of useful peripherals including a 512 KB SRAM bank, PS/2 keyboard, UART and a VGA connector. Project can easily be ported to similar boards since Verilog (and SystemVerilog) files that comprise A-Z80 and other add-on designs in this package are synthesizable for other vendors (such as Xilinx) and their tool chains.

Simulation

Module simulations

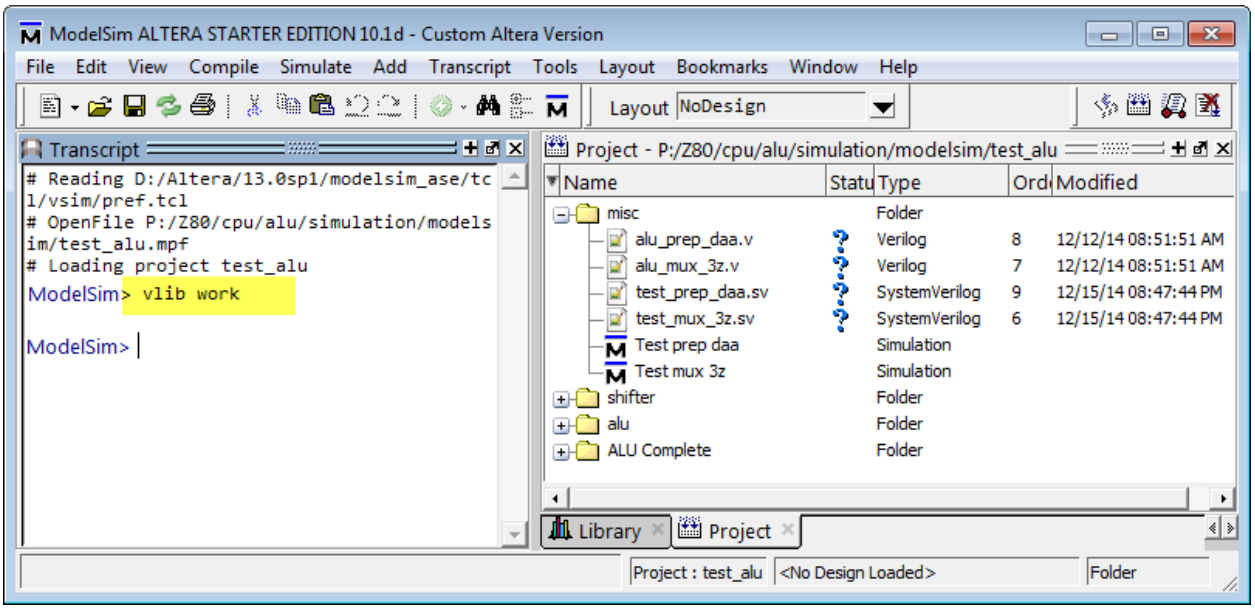
If you start making any changes to the core A-Z80 files, you should run one or more simulations to verify the correctness of your modifications.

Each module in the "**cpu**" directory contains a ModelSim simulation project that verifies the functionality of one or more of its blocks. Before opening any project in ModelSim, run "**modelsim_setup.py**" script located in the project root directory. That script will set up *relative file mappings* to enable project to reside anywhere on your drive.

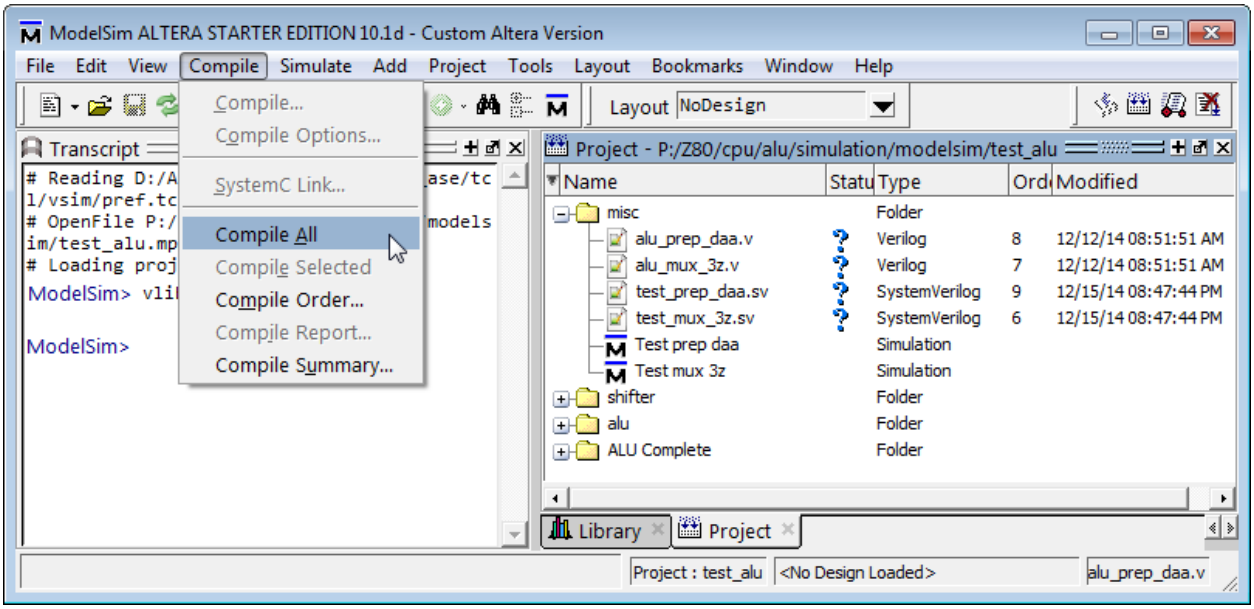
If you have installed and configured ModelSim correctly, double-clicking on any ***.mpf** file will open a project in the ModelSim GUI.

This particular example will illustrate setting up and starting a simulation of a specific logic block in the **alu** module.

Important: Before you can compile any simulation test bench, you need to create a library by typing "**vlib work**" as shown:



Next, select "**Compile->Compile All**" to compile all files that are part of a module simulation.



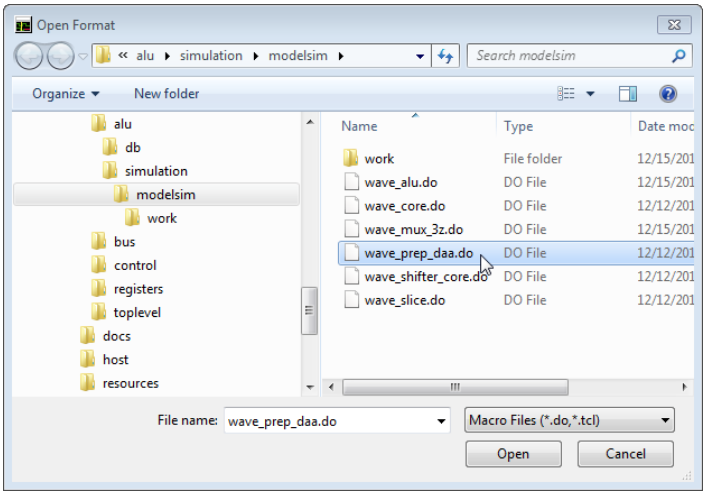
Each project has one or more *simulation configurations*; each configuration tests a specific block of logic. In addition, each configuration has its own wave file which you can load before you run a simulation. Wave files are customized for a specific test and a handy way to quickly see all relevant signals.

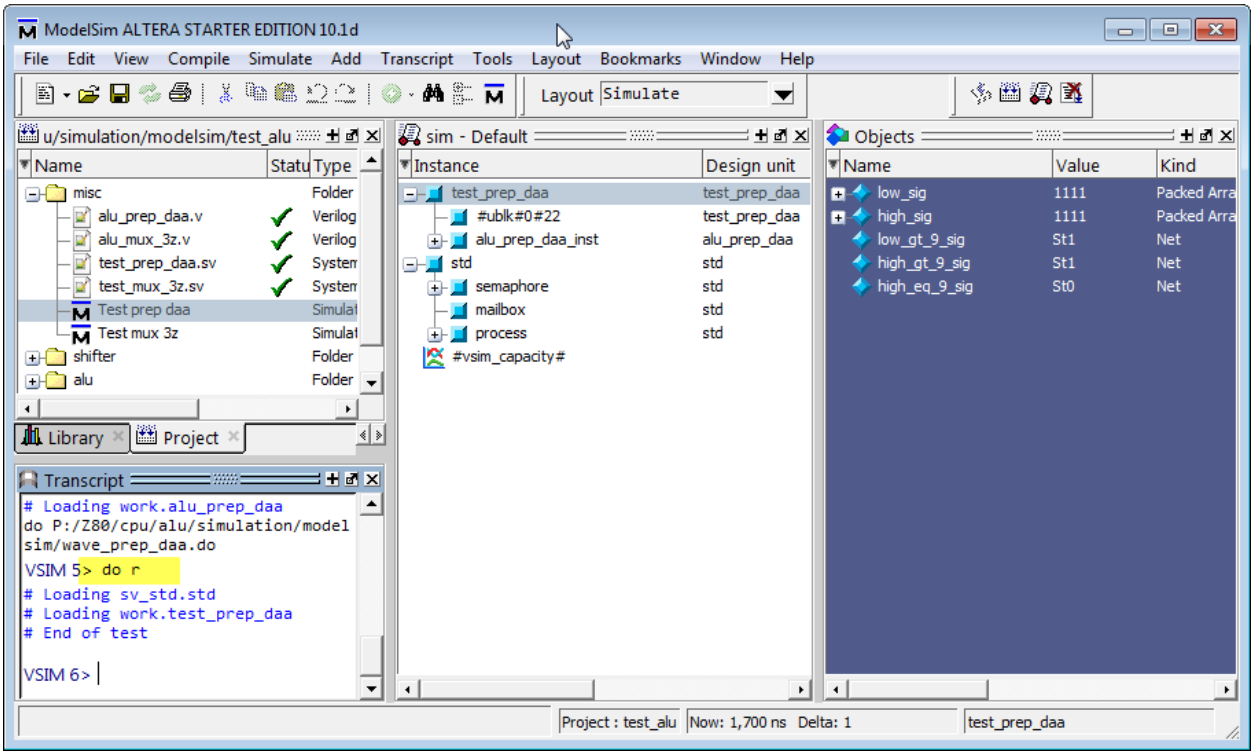
In this example, we will run "Test prep daa" configuration. DAA is a Z80 instruction that adjusts accumulator for a decimal operation. It requires calculating the adjustment addend based on the result of a previous operation. Hence, this test is written to verify the correctness of that calculation.

Each test configuration is run by a main test bench file that is always written in a *System Verilog* language with the extension ***.sv**. A file that runs the "Test prep daa" configuration is "**test_prep_daa.sv**".

Double-click on the "Test prep daa" configuration and your simulation should be loaded.

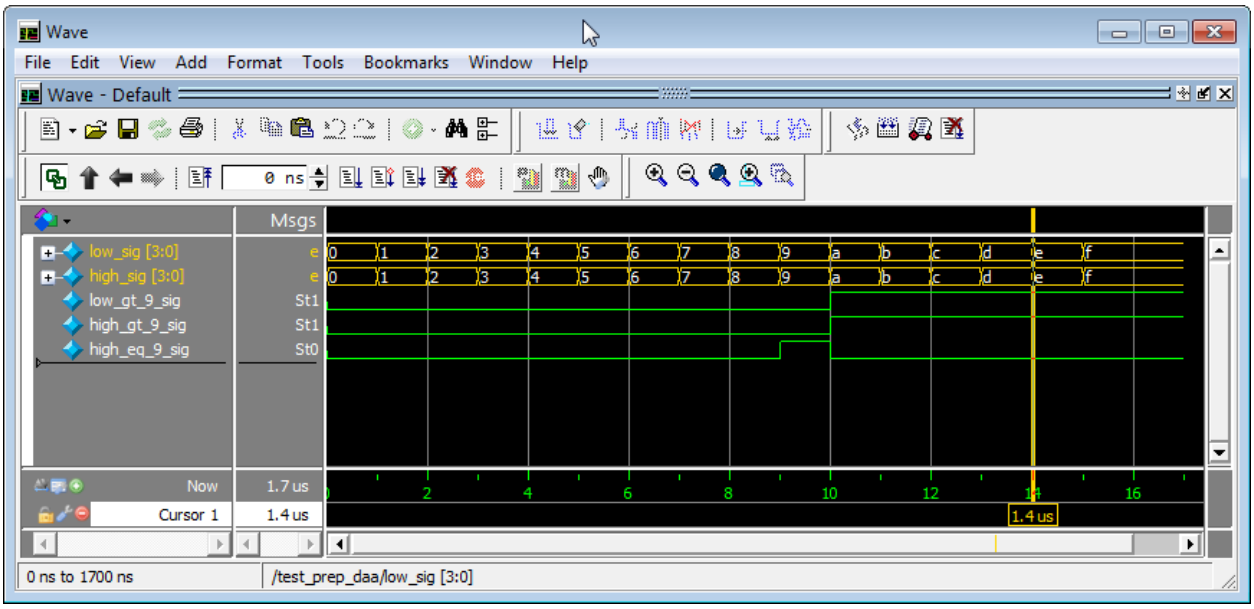
Open the wave window if it is not already visible and select File->Open to load a wave file as shown:





A handy shortcut is provided to run a simulation: each ModelSim directory contains a small text file with the name "r" that contains command "**restart -f ; run -all**". Run, or rerun, a test simply by typing "**do r**" as shown above.

After running this particular example, you should see a waveform of the DAA preparation block:



Although this was a very simple example, it illustrated a method of running a simulation and that can be repeated on other configurations and modules. The pattern of configurations, files and waveform names is the same.

Each main test bench file (like the "**test_prep_daa.sv**") contains a set of **assert()** statements to verify the signal correctness. These assert()s will fail and your simulation will stop if the signals take unexpected values.

Most simulations run for the predetermined number of clocks. The exceptions are top-level simulations (in the directory "**cpu\toplevel\simulation\modelsim**") and a basic host simulation (in the directory "**host\basic\simulation\modelsim**"). These simulations need to be stopped manually since they simply continue to execute given Z80 executable code.

Top-level simulations

The two top-level simulations are designed to load an arbitrary Z80 assembly code and execute it. A simple unidirectional UART model is provided for the Z80 software to write to the ModelSim console. The UART model will simulate the behavior of a synthesized serial port. When the same design is synthesized for the FPGA, the same Z80 code will write messages through a physical serial port.

Module	Simulation project
Toplevel	cpu\toplevel\simulation\modelsim\test_top.mpf
Basic host	host\basic\simulation\modelsim\test_host.mpf

Those two simulation configurations can run any Z80 code, and several sample test sources can be found in the directory "**tools\zmac**" along with the ZMAC assembler and a few batch scripts that simplify compilation and the test setup. Z80 test files are roughly based on CP/M and have a BDOS style text print interface.

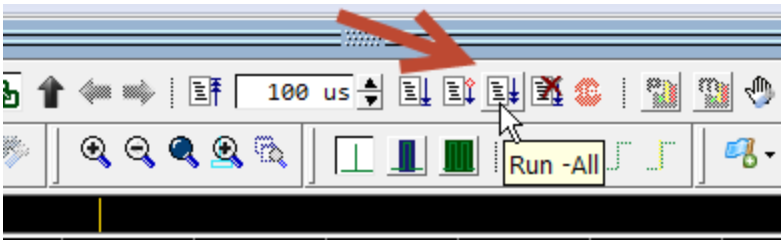
Two MS DOS batch files are used to compile and run a test (you can also create and run your own tests as well):

Batch file	Description
tools\zmac\make_modelsim.bat	Compiles and generates executable code for a ModelSim test at "cpu\toplevel\simulation\modelsim\test_top.mpf", for "test_top" configuration.
tools\zmac\make_fpga.bat	1. Compiles and generates executable code in Intel HEX file format to be included into the target FPGA data file for basic host "host\basic\ host_board.qpf" 2. Also generates executable code for the basic host ModelSim test at "host\basic\simulation\modelsim\ test_host.mpf"

You can simply drag and drop an assembly source file (*.asm) onto one of those batch files and a batch file will compile and copy the results into proper directories after which you only need to recompile a relevant project.

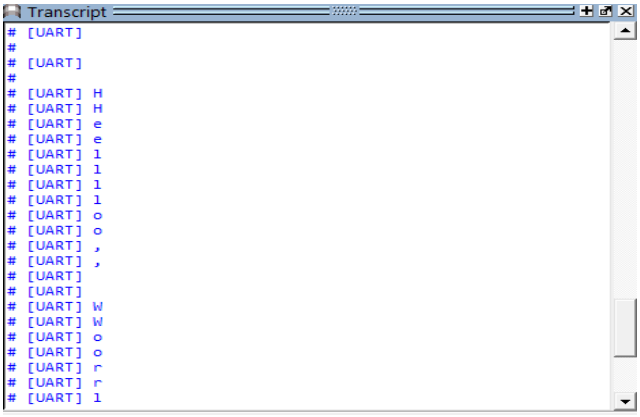
For this example, we will compile and run a "Hello, world" test ("**tools\zmac\hello_world.asm**").

Drag and drop "**hello_world.asm**" onto the "**make_modelsim.bat**" and start a top level simulation ("**test_top**" configuration) in the ModelSim.



Shortly, you should see the output in the ModelSim console window.

After you see the text being written to the virtual UART device, you can stop the simulation.



Verification

Fuse tests

Fuse is a set of tests to verify Z80 at the individual instruction level. Written for software emulator designers, it contains a fairly complete set of input and output states for each instruction.

Files that are used in this verification are subset of the Fuse emulator source package: http://fuse-emulator.sourceforge.net . You can find them in the "**cpu\toplevel\fuse**" directory.

The files describe individual instruction's tests and need to be processed into a format that we can run – which is Verilog. A Python script "**cpu\toplevel\genfuse.py**" generates Verilog test code for a selected number of Fuse tests.

See that script file for more details on how to configure it before running.

When run, it creates "**cpu\toplevel\test_fuse.i**" include file.

```
// Automatically generated by genfuse.py
force dut.reg_file_.reg_gp_we=0;
force dut.reg_control_.ctl_reg_sys_we=0;
force dut.z80_top_ifc_n.fpga_reset=1;
#2 //--------------------------------------------------------------------------------
   force dut.instruction_reg_.ctl_ir_we=1;
   force dut.instruction_reg_.db=0;
#2 release dut.instruction_reg_.ctl_ir_we;
   release dut.instruction_reg_.db;
$fdisplay(f,"Testing opcode 00      NOP");
...
```

Once generated, this include file needs to be compiled with a ModelSim project file "**cpu\toplevel\simulation\modelsim\test_top.mpf**" to run a set of tests. The test output will show in the ModelSim window and the test will also create and write a file "**fuse.result.txt**".

Hint: You can speed up Fuse simulation if you disable output to the wave window by typing:

```
VSIM 10> nolog –all
```

The following command re-enables the output:

```
VSIM 10> nolog –reset
```

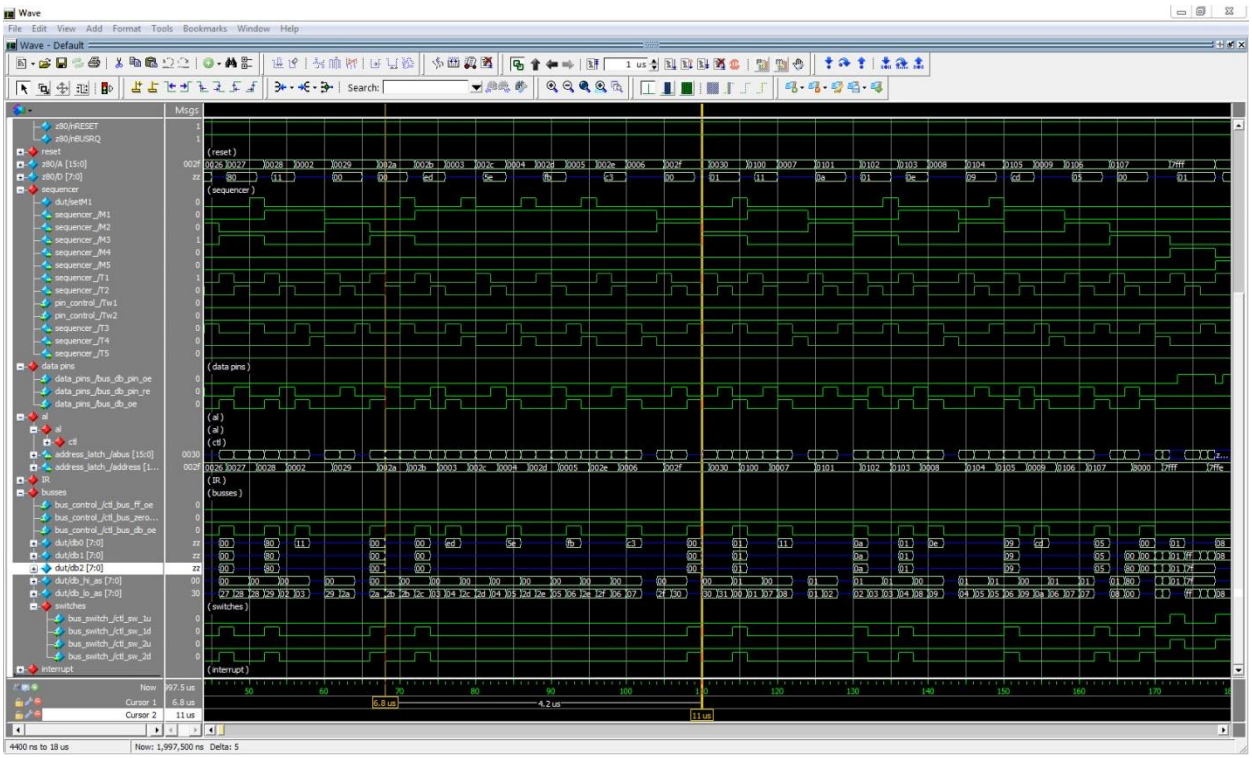


Image 1 : Fuse tests in ModelSim

Results of Fuse tests are written in the file "**fuse.result.txt**", one instruction per line:

```
Testing opcode 00      NOP
Testing opcode ed67    RRD
Testing opcode ed6f    RLD
Testing opcode 81      ADD A,C
Testing opcode cb41    BIT 0,C
Testing opcode cb93    RES 2,E
...
```

Selected functional tests

There are 3 tests that verify specific ALU operations by cross-checking the results run on a real Z80 with the algorithm written in Python:

Test directory	Z80 test file	Description
tools\dongle\daa	tools\zmac\test.daa.asm	Execute DAA instruction for all values 0-255
tools\dongle\neg	tools\zmac\test.neg.asm	Execute NEG instruction for all values 0-255
tools\dongle\sbc	n/a	Simulate SUB and SBC instructions

Python scripts run the Arduino Z80 dongle (described in the Tools section) and generate output files. Those files are then compared with the output produced by another set of Python scripts (they implement corresponding algorithms). Lastly, the same text files are compared with ModelSim

simulation of those instructions and also by running the same executable on the Simple Host FPGA implementation and capturing the UART output.

The "golden" files include values of flags and accumulator going into the instruction and the result after the instruction has completed:

```
F:00 A:00 -> 00 F:44
F:00 A:01 -> 01 F:00
F:00 A:02 -> 02 F:00
F:00 A:03 -> 03 F:04
F:00 A:04 -> 04 F:00
F:00 A:05 -> 05 F:04
F:00 A:06 -> 06 F:04
F:00 A:07 -> 07 F:00
...
```

Z80 Assembly level tests

Folder "**tools/zmac**" contains several Z80 assembly level tests.

Test source file	Description
tools\zmac\hello_world.asm	A mandatory "Hello, World"
tools\zmac\zexdoc.asm	Tests documented Z80 instructions and flags
tools\zmac\zexall.asm	Tests ALL Z80 instructions and flags (documented and undocumented)

While all of them can run in ModelSim, the last two are very comprehensive tests and should normally be run only in the FPGA hardware in full speed mode.

"**hello_world.asm**" source is written to allow the test bench "**cpu\toplevel\test_top.sv**" to exercise various interrupt modes. It contains interrupt handlers and logging for the test bench to run the following cases:

- Inject a single or periodic NMI
- Inject a single or periodic INT
- Test response to the nWAIT signal
- Test response to the nBUSRQ signal
- Test resets

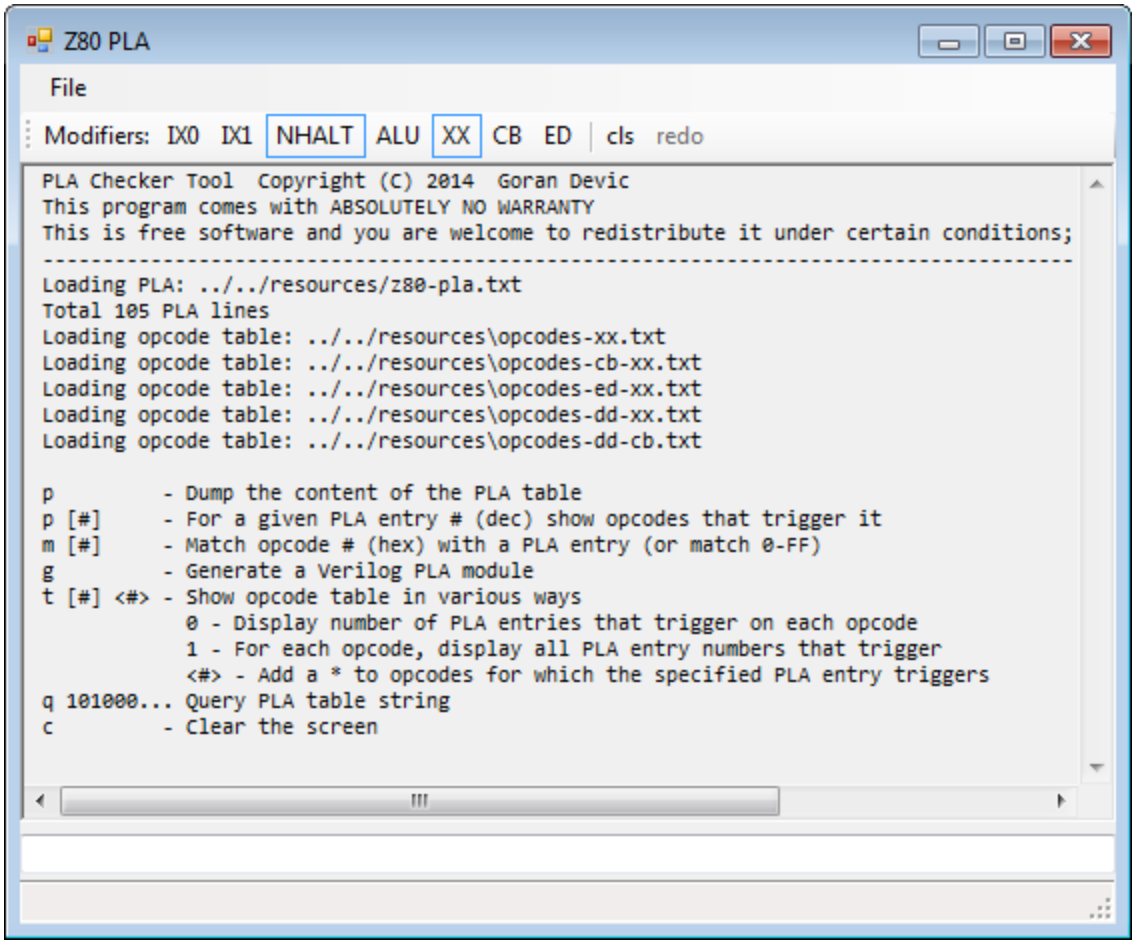
Tools

PLA Checker Tool

PLA checker tool in "**tools\z80_pla_checker**" directory is a test utility to verify and create PLA code used to statically decode Z80 instruction groups.

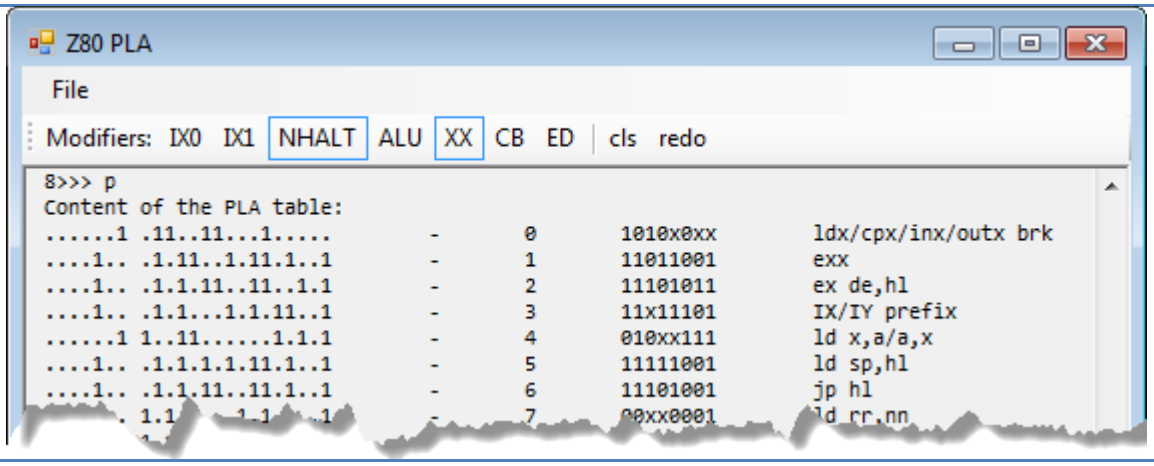
In addition to the C# source code, the Windows executable is also checked in so you don't have to have Microsoft Visual Studio IDE installed to use the tool.

Upon start, the PLA checker tool loads a number of files from the "**resources**" directory. That includes the raw PLA table definition as reverse-engineered from an image of a Z80 die.



The tool was invaluable in the development phase of the A-Z80 and maintain its value as a cross-checker for the PLA code. Available commands are:

Cmd	Description
h or ?	Help, list all commands.
p	PLA table contains a set of modifiers and a gate-level logic array that 'filters' various instruction opcode groups. This command shows you those groups.



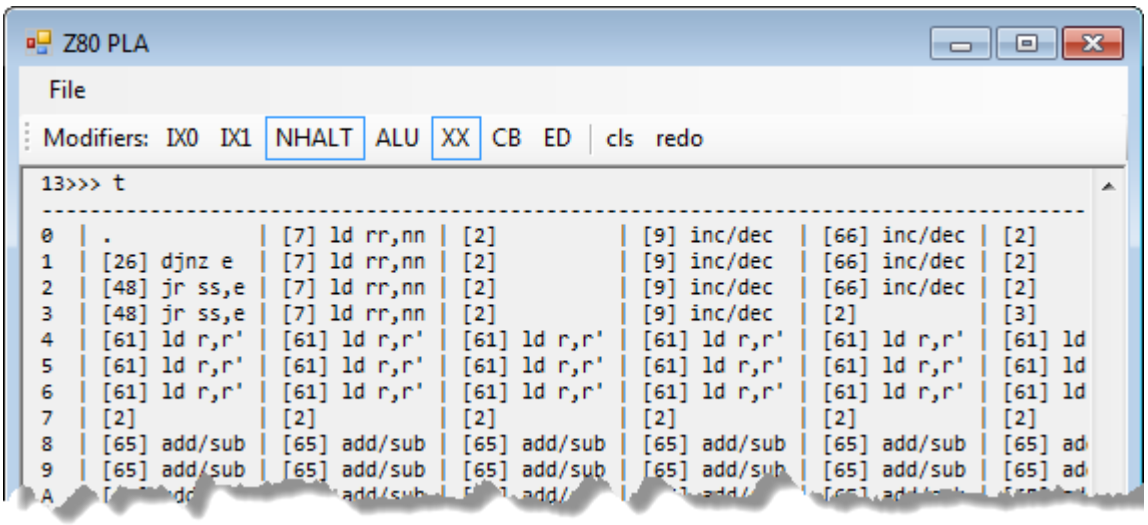
p # Given a PLA entry number (decimal), show opcodes that are activated by it

```
10>>> p 3
PLA Entry: 3  Modifier: XX, NHALT
DD => [3] IX/IY prefix
FD => [3] IX/IY prefix
```

m # This is a reverse-lookup that shows all PLA table entries that would activate a specific opcode given as a hex number:

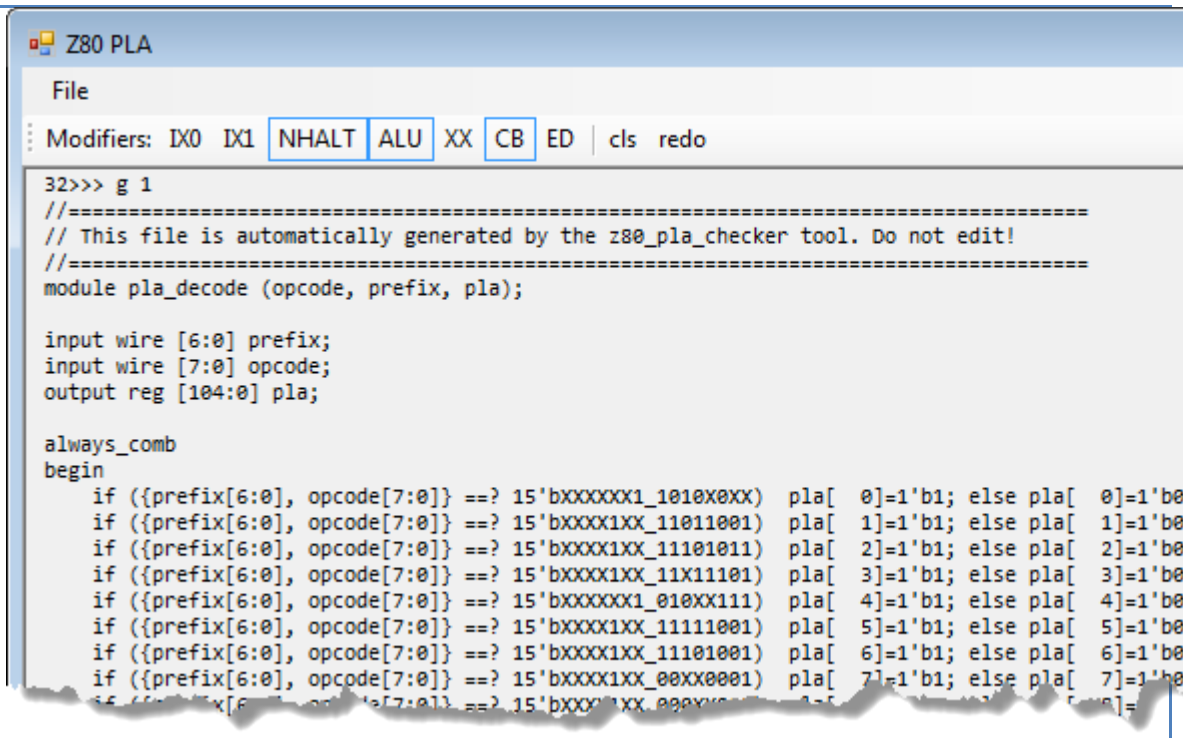
```
12>>> m 76
Opcode: 76
[58] ld r,(hl)
[59] ld (hl),r
[61] ld r,r'
[95] halt
```

t Dumps the opcode table in several ways. One or two optional arguments are given which restrict the table or show extra information including the number of PLA entries that trigger for each opcode etc.



q # Useful only while simulating the CPU design, this command decodes the actual PLA table string which is a long sequence of binary digits (105 bits in total)

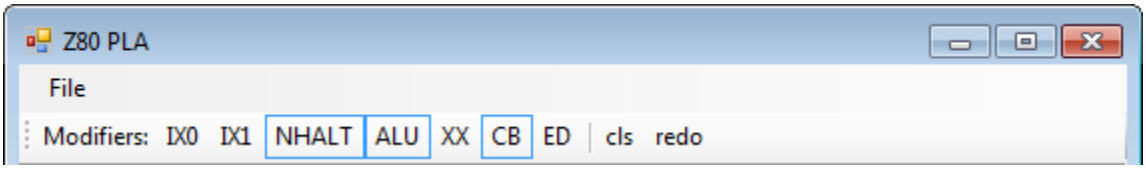
g Generates Verilog code that implements the PLA decode. The output of this command is used to create "**cpu\control\pla_decode.sv**" source file which is at the core of the design.



The image shows the start of the PLA decode module implemented in Verilog.

Z80 has several opcode tables and addressing modes selected either by a combination of instruction prefix bytes (0xCB, 0xED and IX/IY) or by the internal state (HALT, ALU,…)

PLA checker tool lets you set or unset any of these modifiers:



The modifier buttons directly correspond to modifiers in the PLA table and let you simulate the exact PLA logic behavior as you are executing various tool dumps.

The tool keeps a history of commands that are typed in; a number displayed at the front of a *prompt* ">>>" is a location in the history buffer. Pressing **PgUp** and **PgDown** selects a command from the history buffer; **ESC** clears the command line.

Arduino Tools

Directory "**tools\Arduino\Z80_dongle**" contains firmware for the *Arduino Mega* connected to a Zilog Z80 through a custom dongle. This setup can be used to pace Z80 in a controlled way and to execute individual instructions and monitor bus activity. You can read more about that dongle at www.baltazarstudios.com.

It was heavily used to generate tables for the correct bus behavior. These tables and Python scripts to create them are checked in the directory "**tools\dongle**".

Integration

This section describes how to integrate the A-Z80 CPU into your own project.

The method is tested with Altera design tools (Quartus), but it should be relatively easy for someone skilled in the art to use any other vendor (for example Xilinx).

The process of integration involves adding all relevant source files, and those are:

```
cpu/alu/alu_slice.v
cpu/alu/alu_shifter_core.v
cpu/alu/alu_select.v
cpu/alu/alu_prep_daa.v
cpu/alu/alu_mux_8.v
cpu/alu/alu_mux_4.v
cpu/alu/alu_mux_3z.v
cpu/alu/alu_mux_2z.v
cpu/alu/alu_mux_2.v
cpu/alu/alu_flags.v
cpu/alu/alu_core.v
cpu/alu/alu_control.v
cpu/alu/alu_bit_select.v
cpu/alu/alu.v

cpu/bus/bus_switch.sv
cpu/bus/inc_dec_2bit.v
cpu/bus/inc_dec.v
cpu/bus/data_switch_mask.v
cpu/bus/data_switch.v
cpu/bus/data_pins.v
cpu/bus/control_pins_n.v
cpu/bus/bus_control.v
cpu/bus/address_pins.v
cpu/bus/address_latch.v
cpu/bus/address_mux.v

cpu/control/sequencer.v
cpu/control/resets.v
cpu/control/ir.v
cpu/control/interrupts.v
cpu/control/decode_state.v
cpu/control/clk_delay.v
cpu/control/pin_control.v
cpu/control/pla_decode.sv
cpu/control/memory_ifc.v
cpu/control/execute.sv

cpu/registers/reg_latch.v
cpu/registers/reg_file.v
cpu/registers/reg_control.v

cpu/toplevel/z80_top_direct_n.sv
```

In addition, two fully working sample implementations (*basic host* and *zxspectrum*) provide good starting points.

Interface

The top-level file "**cpu\toplevel\z80_top_direct_n.sv**" exports the following interface:

```
module z80_top_direct_n(
    output wire nM1,
    output wire nMREQ,
    output wire nIORQ,
    output wire nRD,
    output wire nWR,
    output wire nRFSH,
    output wire nHALT,
    output wire nBUSACK,

    input wire nWAIT,
    input wire nINT,
    input wire nNMI,
    input wire nRESET,
    input wire nBUSRQ,

    input wire CLK,
    output wire [15:0] A,
    inout wire [7:0] D
);
```

This pinout is 100% identical to the Zilog Z80 package. The interface implements Z80 bus timings and features tri state buses. (While this is admittedly not optimal for an FPGA implementation, the goal of the project was to mimic the actual Z80 silicon).

Your design should include all core files listed above and instantiate a "**z80_top_direct_n**" module.

Sample Implementations

Two working implementations are included. They are both located in the "**host**" directory and use the Altera DE1 FPGA development board.

Warning: The synthesis and *fMax* numbers as shown might vary depending on your tool version, applied timing constraints and the exact configuration.

Simple host

A "*basic host*" board contains A-Z80 CPU, 16 KB RAM configured as single port Cyclone RAM cells and a unidirectional implementation of the UART for the text output. Since the board's architecture is so simple, there is also a corresponding ModelSim configuration used in verification.

File	Description
host\basic\host_board.qpf	Quartus project file for FPGA
host\...\simulation\modelsim\ test_host.mpf	ModelSim project file for simulation
host\basic\host_board_fpga.sv	Top-level board source file for FPGA implementation
host\basic\ host_board_ModelSim.sv	Top-level board source file for ModelSim board model
host\basic\test_host.sv	ModelSim test bench for the simulation model

This host board can load and run any Z80 executable (for example, one of those in "**tools\zmac**" directory). Programs can print to UART and, on a physical DE1 board, the text is seen through the attached serial terminal. In the simulation environment the text is written in a ModelSim output window. **Error! Reference source not found.** shows the output of the "**tools\zmac\hello_world.asm**" eing captured through the serial port.

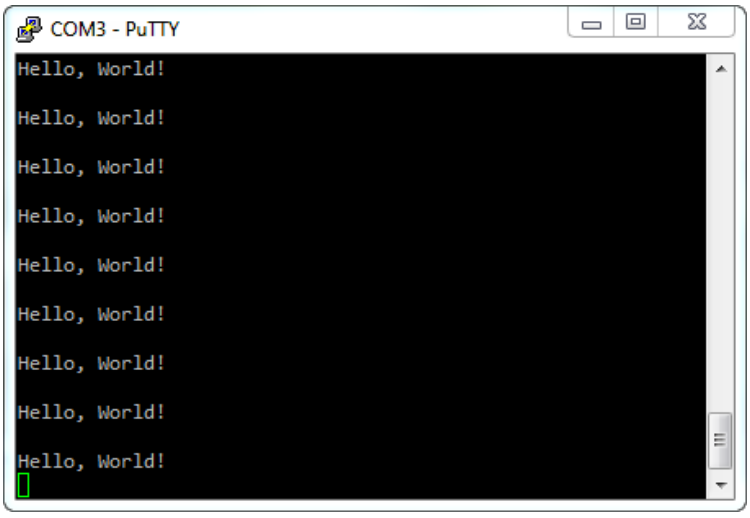


Image 2: "Hello, World"

This is a synthesis result of the simple host design on an Altera DE1 board:

Flow Summary	
Flow Status	Successful - Tue Dec 16 00:50:47 2014
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	host_de1
Top-level Entity Name	host
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	3,341 / 18,752 (18 %)
Total combinational functions	3,248 / 18,752 (17 %)
Dedicated logic registers	468 / 18,752 (2 %)
Total registers	468
Total pins	11 / 315 (3 %)
Total virtual pins	0
Total memory bits	131,072 / 239,616 (55 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	1 / 4 (25 %)

Since the CPU CLK is derived from the *pll_clk*, the effective A-Z80 *fMax* for this compilation is 19.86 MHz.

Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	19.86 MHz	19.86 MHz	pll_clk	
2	58.82 MHz	58.82 MHz	CLOCK_50	
3	90.43 MHz	90.43 MHz	pll_|altpll_component|pll|clk[0]	

Sinclair ZX Spectrum

This project fully implements a *Sinclair ZX Spectrum 48K* computer from the year 1982.

The model codes in Verilog all parts that make up that computer. Directory "**host\zxspectrum\ula**" contains blocks (drivers) for the keyboard, video signal using the VGA, sound, RAM memory, clocks etc.

There are 2 system ROM images included in the "**host\zxspectrum\rom**" directory – the original ZX Spectrum ROM and an improved, so-called "Gosh Wonderful" ROM – merged into a single image which is to be flashed into the DE1's flash memory starting at the address 0. Use a flash tool that came with your DE1 board software to flash this data.

The following table shows the function of buttons and switches; when a switch is activated, a red LED above it glows.

Button and Switch	Description
KEY0	Reset
KEY1	Issues NMI
SW0	Selects "Gosh Wonderful ROM" image versus the original ROM image
SW1	Disables interrupts
SW2	Turbo mode (3.5 MHz x 2 = 7.0 MHz)

Function of **green LED**s is to show:

GREEN LED	Description
LEDG0-LEDG4	Kempston joystick UP, DOWN, LEFT, RIGHT, FIRE is pressed
LEDG5	A key is pressed
LEDG6	When blinking, a speaker or line-in is active

Image 3 shows a game "*Manic Miner*" being loaded through the audio line-in connector into the FPGA board visible in the middle and a Kempston compatible joystick in the foreground.

Image 3 : Sinclair ZX Spectrum on Altera DE1

This is a synthesis result of the ZX Spectrum host design on Altera DE1 board:

Flow Summary	
Flow Status	Successful - Tue Dec 16 00:59:29 2014
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	zxspectrum_board
Top-level Entity Name	zxspectrum_board
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	3,692 / 18,752 (20 %)
Total combinational functions	3,621 / 18,752 (19 %)
Dedicated logic registers	584 / 18,752 (3 %)
Total registers	584
Total pins	148 / 315 (47 %)
Total virtual pins	0
Total memory bits	131,072 / 239,616 (55 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	1 / 4 (25 %)

Although the computer runs at 3.5 MHz, the *clk_cpu fMax* for this compilation is 10.65 MHz.

Fmax Summary

	Fmax	Restricted Fmax	Clock Name	
1	10.65 MHz	10.65 MHz	clk_cpu	
2	69.29 MHz	69.29 MHz	ula_|pll_|altpll_component|pll|clk[0]	
3	76.88 MHz	76.88 MHz	CLOCK_24	
4	1157.41 MHz	450.05 MHz	ula_|pll_|altpll_component|pll|clk[1]	limit due

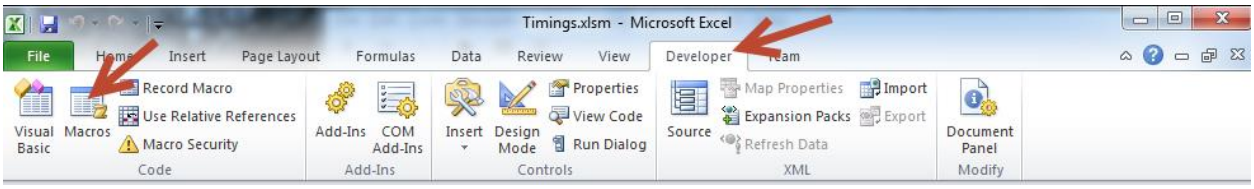
Advanced Topics

Modifying the A-Z80 CPU

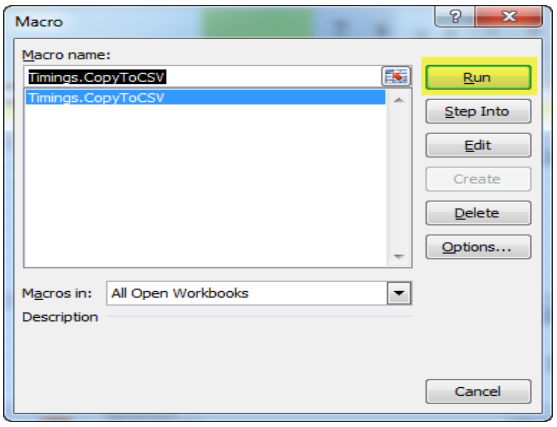
If you want to make a change to any instruction's timing or a sequence of micro-operations, do it in the file "**cpu\control\Timings.xlsm**". This is a Microsoft Excel spreadsheet file that contains timing tables for each instruction group. Vertical columns are operations on specific blocks. Instruction groups are listed by the **M** and **T**-clocks providing the exact timing for each set of operations.

Micro-operations are represented by short tokens (for example, "PC" or "mr", etc.) and defined in the file "**cpu\control\timing_macros.i**". In that file, every token is translated into one or more concrete control signals or operations.

If you change the timing spreadsheet, export it into a TAB-delimited file. The spreadsheet contains a macro to do that for you: click on the "Developer" menu and run Macros:



Running "CopyToCSV" macro will replace the existing CSV file which is ok: both are checked in although one is generated from another.



Next step is to create a Verilog file from those timings by running a python script "**cpu\control\genmatrix.py**". That script reads in the CSV file containing timing tables and generates "**exec_matrix.i**" file that implements actual Verilog code to control the timings.

All Python scripts in this project can be run in-place without the need to specify any arguments.

If you change any *schematic file* and your change adds or removes global input or output signals, you need to run two Python scripts to recreate global includes:

"**cpu\control\genref.py**" – generates global include files using all exported module signals:

- "exec_module.i" contains input/output definitions to be included in the module def.
- "exec_zero.i" contains Verilog code to set all input wires to zero.

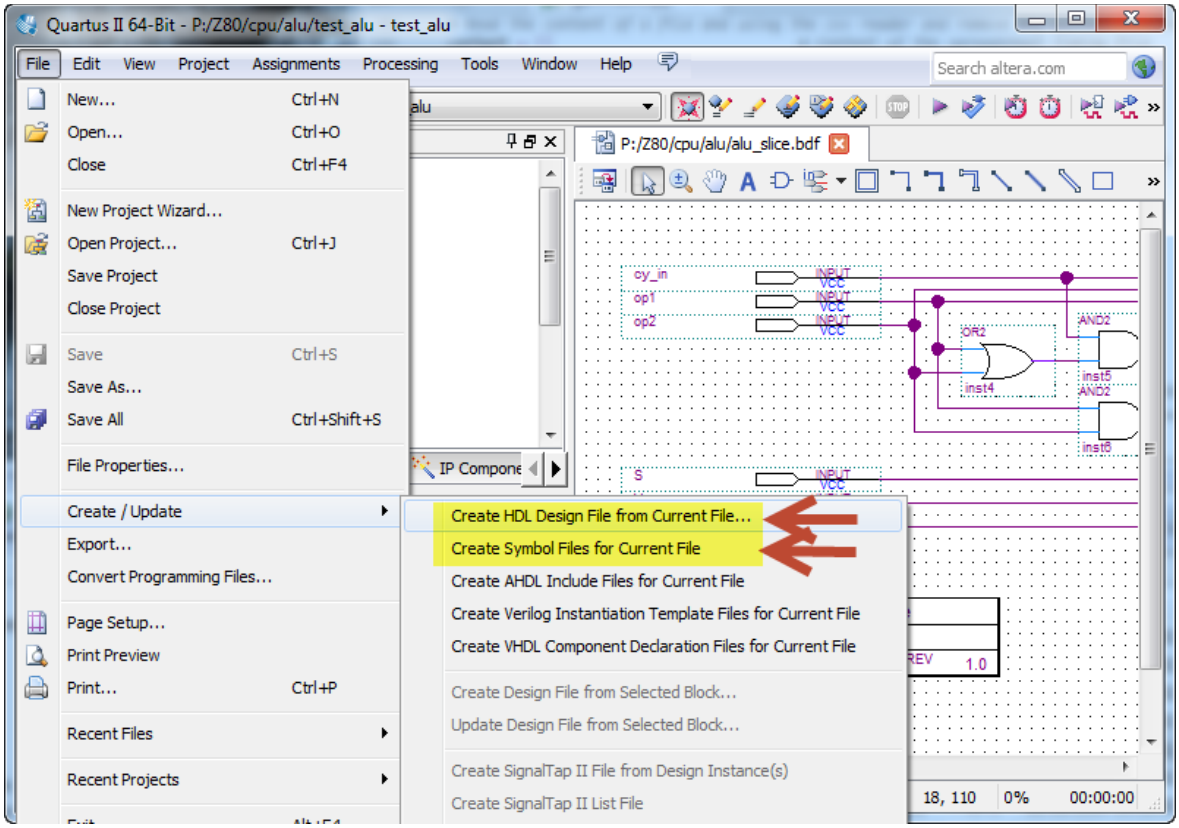
"**cpu\toplevel\genglobals.py**" – generates a list of global *wire* defines:

- "globals.i" contains Verilog code that defines all global signal wires.

Quartus project files (*.qpf, *.qsf) in "**cpu\alu**", "**cpu\bus**", "**cpu\control**" and "**cpu\registers**" directories are non-functional and just conveniently hold sets of module files together. *Quartus* project in the "**cpu\toplevel**" directory only contains a top-level schematic diagram and is also not functional. They are only containers to hold files.

In order to compile a project, look in a sample project such is "**host\basic\host_board.qpf**".

When modifying a schematic (most of the A-Z80 blocks are designed at the schematic level), open a corresponding *Quartus* container project (for example, when modifying a schematic in the ALU block, open "**cpu\alu\test_alu.qpf**"), change the schematics, compile it (to make sure it has no errors) and then export it to both the Verilog equivalent and a symbol file, as shown below:



Verilog code is used to compile with the rest of A-Z80 core files while symbol files are (at the moment) optional but could be used in the future to create a schematic top-level.