# Revision Log

|  |  |  |
| --- | --- | --- |
| Date | Revision | Change |
| 2014-12-15 | 1.00 | Initial revision |

Table of Contents

[Revision Log 1](#_Toc406481341)

[Introduction 2](#_Toc406481342)

[Project directory structure 3](#_Toc406481343)

[Environment 4](#_Toc406481344)

[ModelSim simulations 5](#_Toc406481345)

[Module simulations 5](#_Toc406481346)

[Top-level simulations 8](#_Toc406481347)

[Verification 10](#_Toc406481348)

[Fuse tests 10](#_Toc406481349)

[Z80 Assembly level tests 10](#_Toc406481350)

[Tools 10](#_Toc406481351)

[Integration 11](#_Toc406481352)

[Pinout 12](#_Toc406481353)

[Sample implementations 13](#_Toc406481354)

[Simple host 13](#_Toc406481355)

[Sinclair ZX Spectrum 13](#_Toc406481356)

[Advanced Topics 15](#_Toc406481357)

[Modifying the A-Z80 CPU 15](#_Toc406481358)

# 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.

Zilog Z80 CPU references are widely available so the CPU itself will not be described here.

You can read more about the conception and implementation of the A-Z80 on its home website: [www.baltazarstudios.com](http://www.baltazarstudios.com) .

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

# Project directory structure

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

The following table describes the hierarchical directory structure.

|  |  |  |
| --- | --- | --- |
| Directory | Sub-directory | Description |
| cpu |  | Contains all core files for 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 |  | Build and testing utilities and misc. files |
|  | Arduino | Software for Arduino Mega dongle to interface with a Z80 |
|  | dongle | Results from running a dongle and simulation golden files |
|  | z80\_pla\_checker | Windows utility to test and create A-Z80 PLA tables |
|  | zmac | Z80 test and verification assembler files |

# Environment

Minimal set of tools needed to compile various parts of the project are:

* Altera Quartus II Web Edition (Free)
* ModelSim (Altera edition) – only if you would like to run module simulation (Free)
* Python 2.7 – only if you will 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 – only if you want 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 with it may vary if you use Linux.

All designs in this project are tested on the Altera FPGA DE1 board: <http://www.altera.com/education/univ/materials/boards/de1/unv-de1-board.html>



This board hosts Cyclone II EP2C20F484C7 and a number of useful peripherals including 512 KB SRAM.

However, Verilog files that comprise A-Z80 should be synthesizable for other vendors such as Xilinx and their tool chains.

# ModelSim simulations

## Module simulations

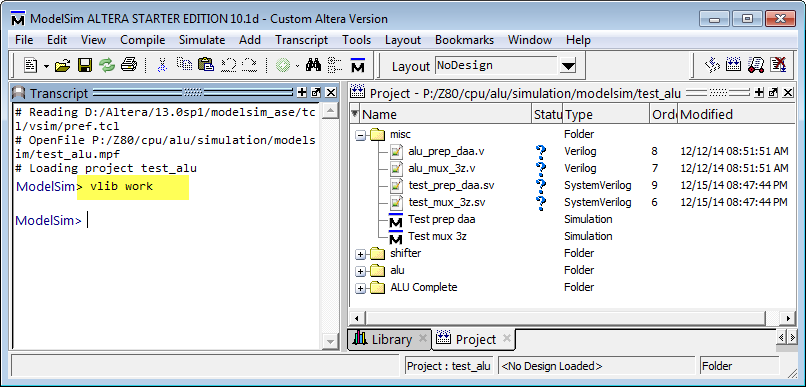
Each module in the “cpu” directory contains a ModelSim simulation project that verifies the functionality of each more important block. Before opening any project in ModelSim, you need to run “**modelsim\_setup.py**” script located in the project root directory. That script will set up relative file mappings to enable project to reside anywhere on your drive.

You may not need to run simulations since the project has already been tested. Should you want to do it, this chapter will show you how.

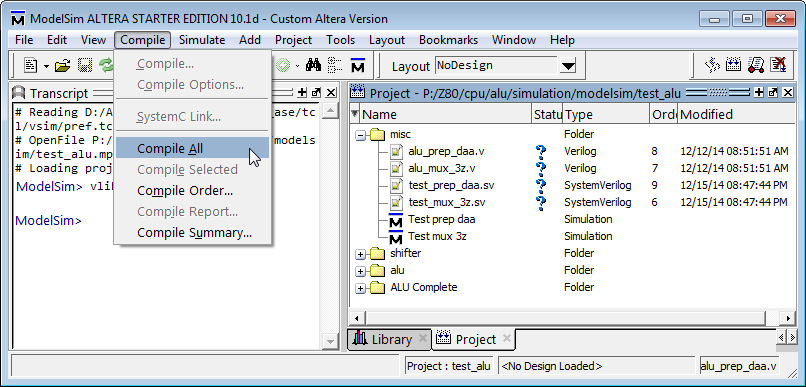
If you have installed and configured ModelSim properly, double-clicking on any **\*.mpf** file will open a project.

This particular example will illustrate setting up and starting a simulation in the **alu** module. Each module has its own ModelSim project.

Before you can compile simulation files, 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.



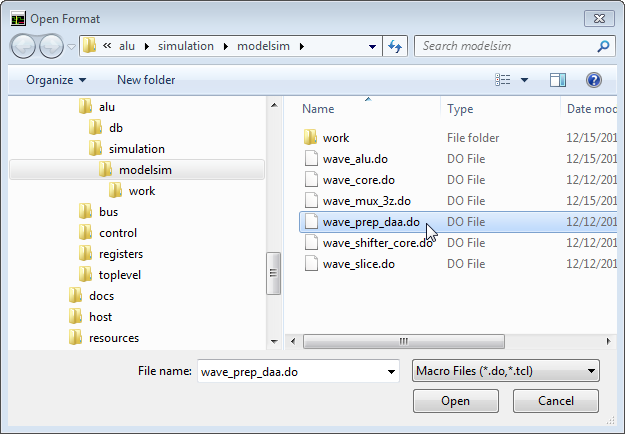
Each project has one or more Simulation Configurations, each configuration testing a specific block of logic. In addition, each configuration has its own wave file which you can load before you run a simulation. Wave files are customized for specific test.

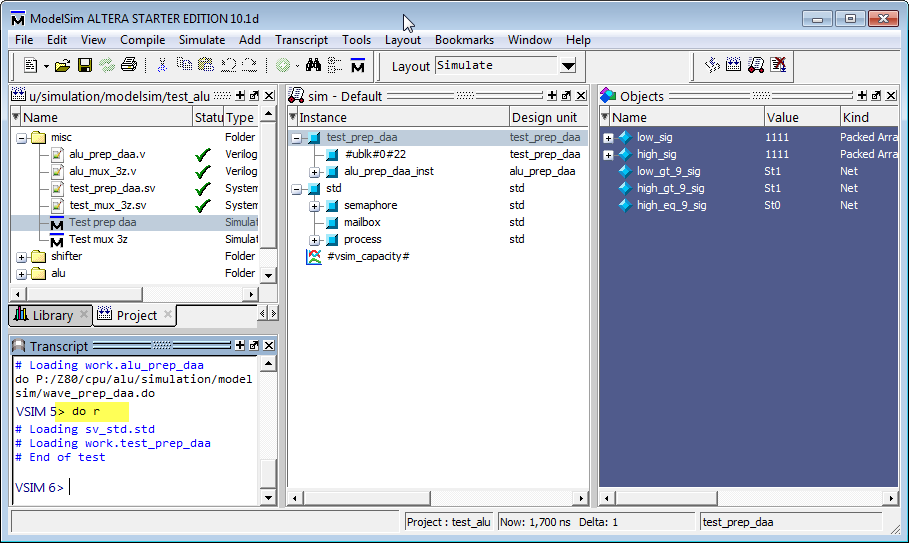
In this example, we will run “Test prep daa” configuration. DAA is a Z80 instruction that adjusts accumulator for a decimal operation. It requires calculating the adjustment addend based on the result of a previous operation. Hence, this test is written to verify the correctness of that calculation.

Each test configuration has a main test file – always written in *System Verilog*, a file with the extension **\*.sv,** to run the test. The file to run “Test prep daa” configuration is “**test\_prep\_daa.sv**”.

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

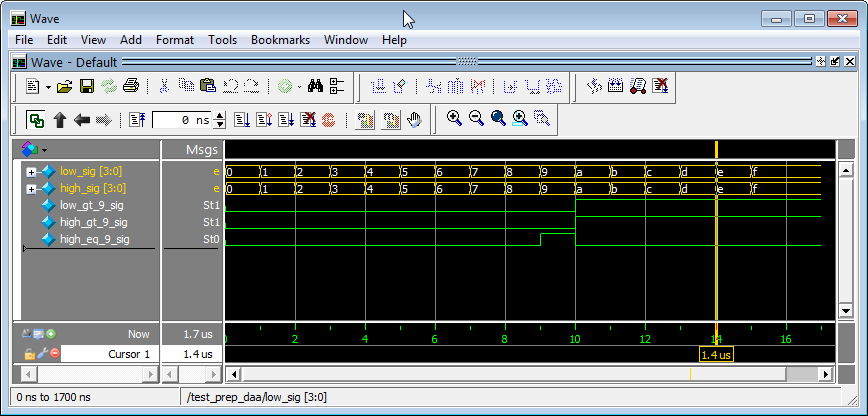
Open the wave window if it is not already visible and select File->Open to load a wave file as shown:





There is another shortcut to run a simulation: each ModelSim directory contains a small 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 the waveform of the DAA preparation block:



This is a very simple example, but the method of running simulations on other configurations and modules is the same.

Each main test file (for example “**test\_prep\_daa.sv**” contains a set of **assert()** statements to verify signal correctness.

Most simulations run for the predetermined number of clocks. The exceptions are top-level simulations (in the directory “**cpu\toplevel\simulation\modelsim**”) and 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

Two top-level simulations are designed to load an arbitrary Z80 assembly level code and execute it. A simple unidirectional UART model is provided to dump the output to the ModelSim console as well as to simulate the behavior of a synthesized design when run on the actual DE1 hardware.

|  |  |
| --- | --- |
| Module | Simulation project |
| Toplevel | cpu\toplevel\simulation\modelsim\test\_top.mpf |
| Basic host | host\basic\simulation\modelsim\test\_host.mpf |

Those simulation configurations can run any Z80 code, and several sample tests can be found in the directory “**tools\zmac**” along with the ZMAC assembler and few batch scripts that simplify the compilation and test setup.

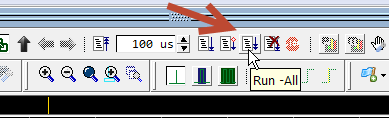
For this example, we will compile and run “Hello, world” test. Its source file is “**tools\zmac\hello\_world.asm**”. (Those Z80 test files are roughly based on the CP/M and BDOS interface for text printing)

Two MS DOS batch files are used to compile and run any of the tests in that folder (you can also create and run your own tests as well):

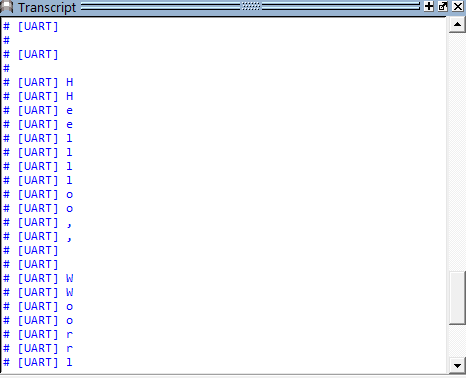
|  |  |
| --- | --- |
| Batch file | Description |
| tools\zmac\make\_modelsim.bat | Compiles and generates executable code for a ModelSim test at “cpu\toplevel\simulation\modelsim\test\_top.mpf”, for “test\_top” configuration. |
| tools\zmac\make\_fpga.bat | 1. Compiles and generates executable code 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 file (\*.asm) into one of the batch files and it will do compile them and copy them to proper directory after which you only need to recompile a project.

Drag and drop “**hello\_world.asm**” onto the “**make\_modelsim.bat**” and start the simulation. Shortly you should see the output.



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



# Verification

## Fuse tests

Fuse is a set of tests to verify Z80 instruction behavior.

## Selected functional tests

There are 3 tests that help verify some 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 instruction test Python scripts which implement corresponding algorithms. Lastly, the same text files are compared with ModelSim simulation of those instructions and also by running the same \*.asm executable on the Simple Host FPGA implementation.

Those “golden” files include flags and accumulator going into the instruction and the result:

|  |
| --- |
| 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 contain several Z80 assembly level tests.

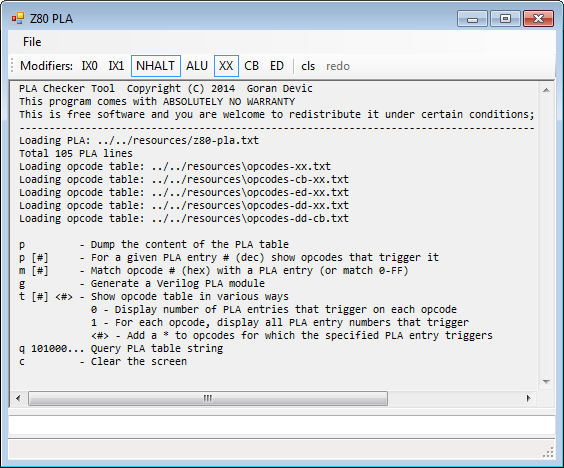
# Tools

## 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 including the raw PLA table definition reverse-engineered from the image of a Z80 die as well as opcode mnemonics.

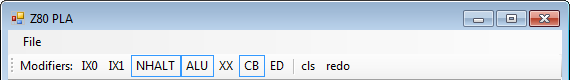


The tool was invaluable in the development phase of A-Z80 cpu and maintain its value as a cross-checker for the PLA code. Available commands are:

|  |  |
| --- | --- |
| 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.  C:\Users\gdevic\AppData\Local\Temp\SNAGHTML294146f7.PNG |
| 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.  C:\Users\gdevic\AppData\Local\Temp\SNAGHTML29487d90.PNG |
| 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.  C:\Users\gdevic\AppData\Local\Temp\SNAGHTML295b3710.PNGThe image shows the start of the PLA decode module implemented in Verilog. |

Z80 has several opcode tables and addressing modes selected either by the 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 when generating opcode dumps.



The modifier buttons directly correspond to modifiers in the PLA table and let you simulate the exact PLA logic behavior.

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 test instructions and monitor bus activity. You can read more about that dongle at [www.baltazarstudios.com](http://www.baltazarstudios.com).

It was heavily used to generate tables for the correct bus behavior. These tables and Python scripts that create them are checked with the project 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 other vendor such is Xilinx.

The integration involves adding all relevant source files. This table lists them:

|  |
| --- |
| 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, any one of the two sample implementations (*basic host* and *zxspectrum*) provide a good starting point and a working example.

# Pinout

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  ); |

The pinout is 100% identical to the Zilog Z80 package; bus timings also implement tri-state signaling. (While this is admittedly not optimal for an FPGA implementation, the goal of the project is to mimic the actual Z80 silicon).

Your system design should simply include all necessary files and instantiate a “**z80\_top\_direct\_n**” module.

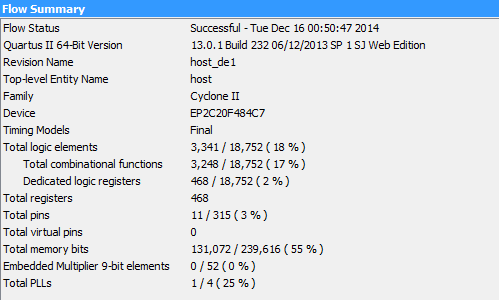
# Sample implementations

Two working sample implementations are included.

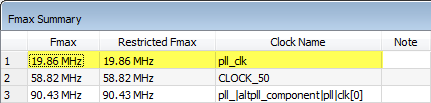
Warning: The synthesis and fMax results as shown might vary depending on the tool versions, applied timing constraints and the exact configuration.

## Simple host

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

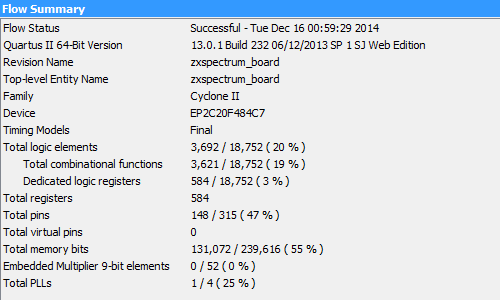


Since the CPU CLK is derived from the *pll\_clk*, the effective A-Z80 fmax for this compilation is 19.86 MHz.

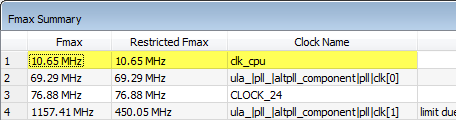


## Sinclair ZX Spectrum

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



The effective A-Z80 *clk\_cpu* fmax for this compilation is 10.65 MHz.



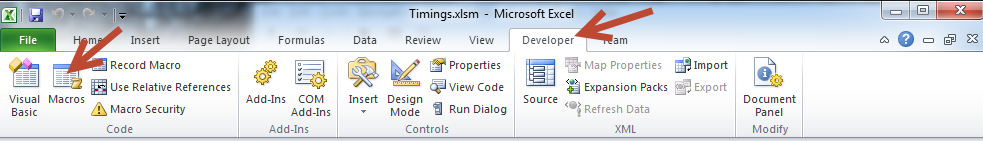
# Advanced Topics

## Modifying the A-Z80 CPU

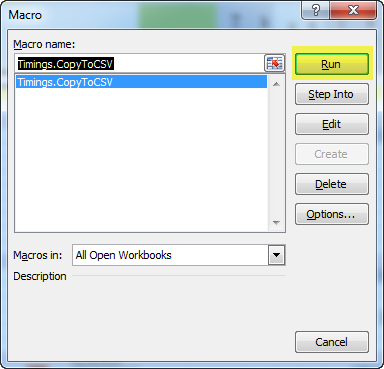
If you want to make a change to any instruction’s timing or a sequence of operations, do it in the file “**cpu\control\Timings.xlsm**”. This is a Microsoft Excel spreadsheet file that contains timing tables for each instruction group. Vertical columns are operations on specific blocks. Instruction groups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

Operations itself are represented by short tokens (for example, “PC” or “mr”, etc.) which are defined in the file “**cpu\control\timing\_macros.i**”. Every token is defined and translated into one or more concrete control signals.

After you change the timing spreadsheet export it into a TAB-separated file. The spreadsheet contains a macro to do that for you. Click on the “Developer” menu and run the Macros:



This will replace the existing CSV file which is ok. Both are checked in although one is generated from the other one.



After that is done, create a Verilog file from those timings by running a python script “**cpu\control\genmatrix.py**”. That script reads the CSV text file timing table and generates file “**exec\_matrix.i**” file which implements actual Verilog code that controls the timings.

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

If you change any *schematic file* and your change adds or removes that module’s 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 export 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.

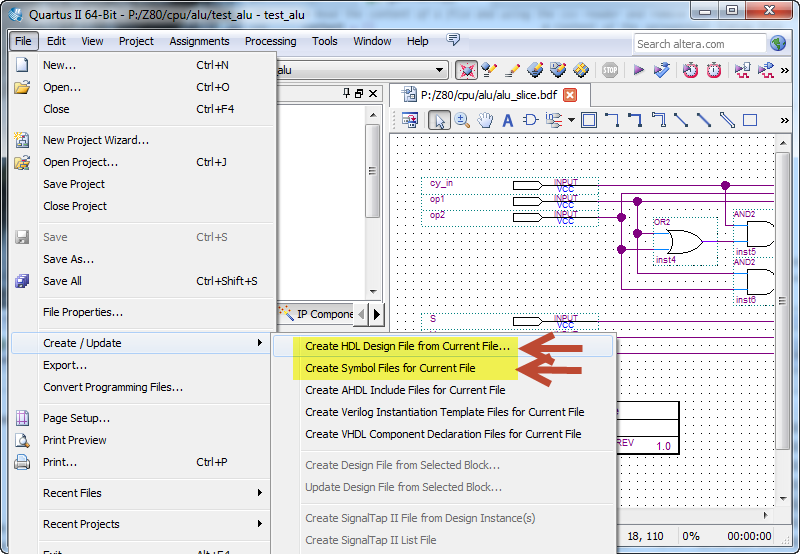
“**cpu\toplevel\genglobals.py**” – generates a list of global wire defines:

* “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 contains a top-level schematic diagram only 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 you are modifying a schematic (which are most of the A-Z80 blocks), open a Quartus container project for that module (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:



Verilog code is used to compile with the rest of the A-Z80 files in your master top-level design project while a symbol file is optional for now but could be used in the future to create a schematic top-level.