Getting Started with Building a HARP System

Git Repo: *https://github.com/nnigania3/harp-system.git*

Contents:

1. Altera Licensing

Setting up license:

You need to run a license server on at least 2 out of the 3 machines which have been authorized:

Machines authorized to run the license file:

sunnylotus.cc.gt.atl.ga.us (main server)
damint.cc.gt.atl.ga.us
lamint.cc.gt.atl.ga.us

Log-in to these machines and run this command:

<quartus directory>/linux/lmgrd -c <license file directory>/1-BFS3BX_License.dat -l log

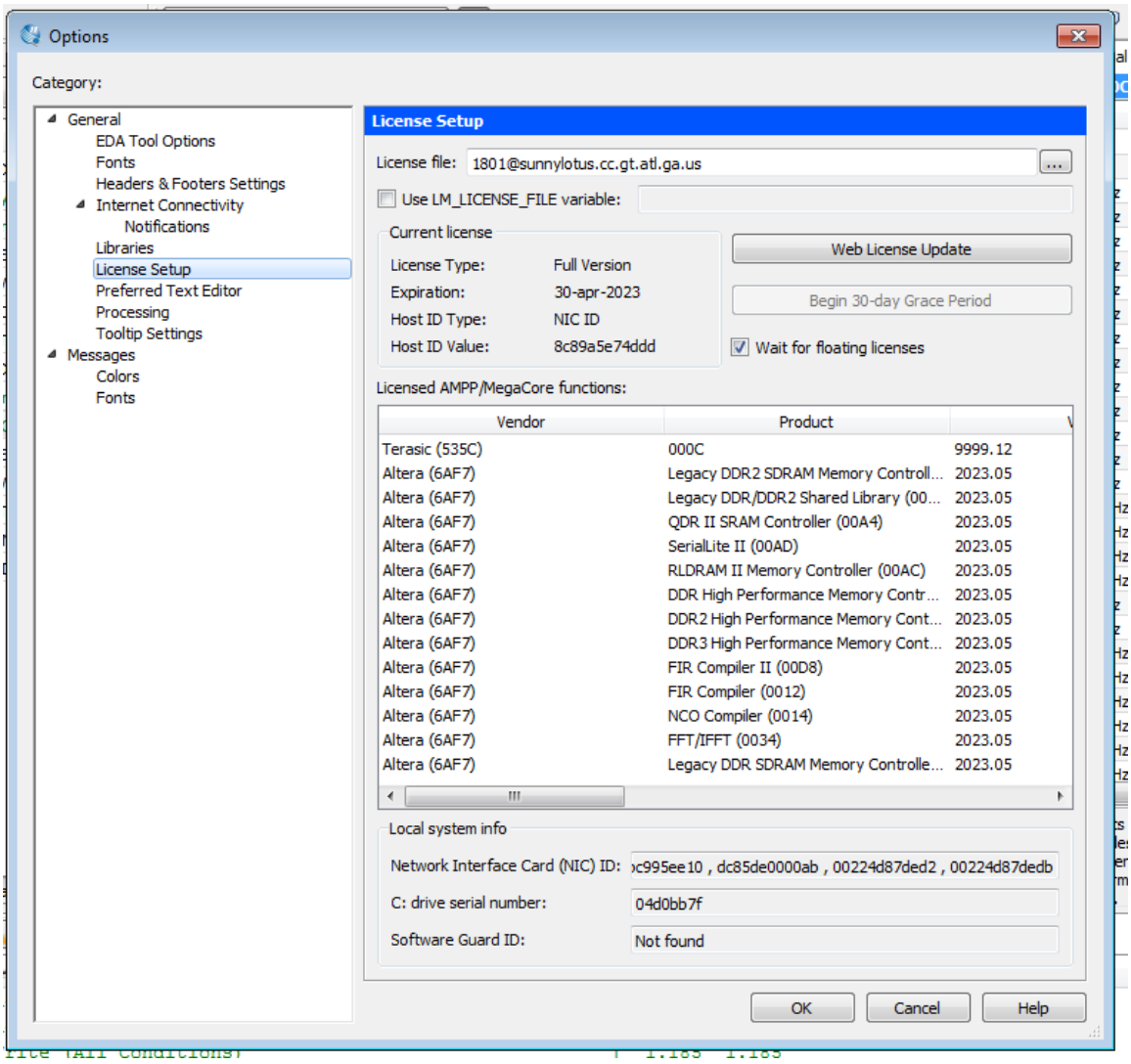
License file location here: "/harp-system/src/extras/1-BFS3BX_License.dat "

Just login to sunnylotus and any of damint/lamint machine and run the above commands and the license server should be ready for you.Need to run on atleast two machine including "sunnylotus".

If above dont work, look here for instructions to run license server:
http://www.altera.com/download/licensing/setup/lic-setup-float-unix.html

Specify License in quartus:

Once you have the license server running you need to tell Quartus its location. In Quartus go to "->tool->license setup" and you should see a window like below.

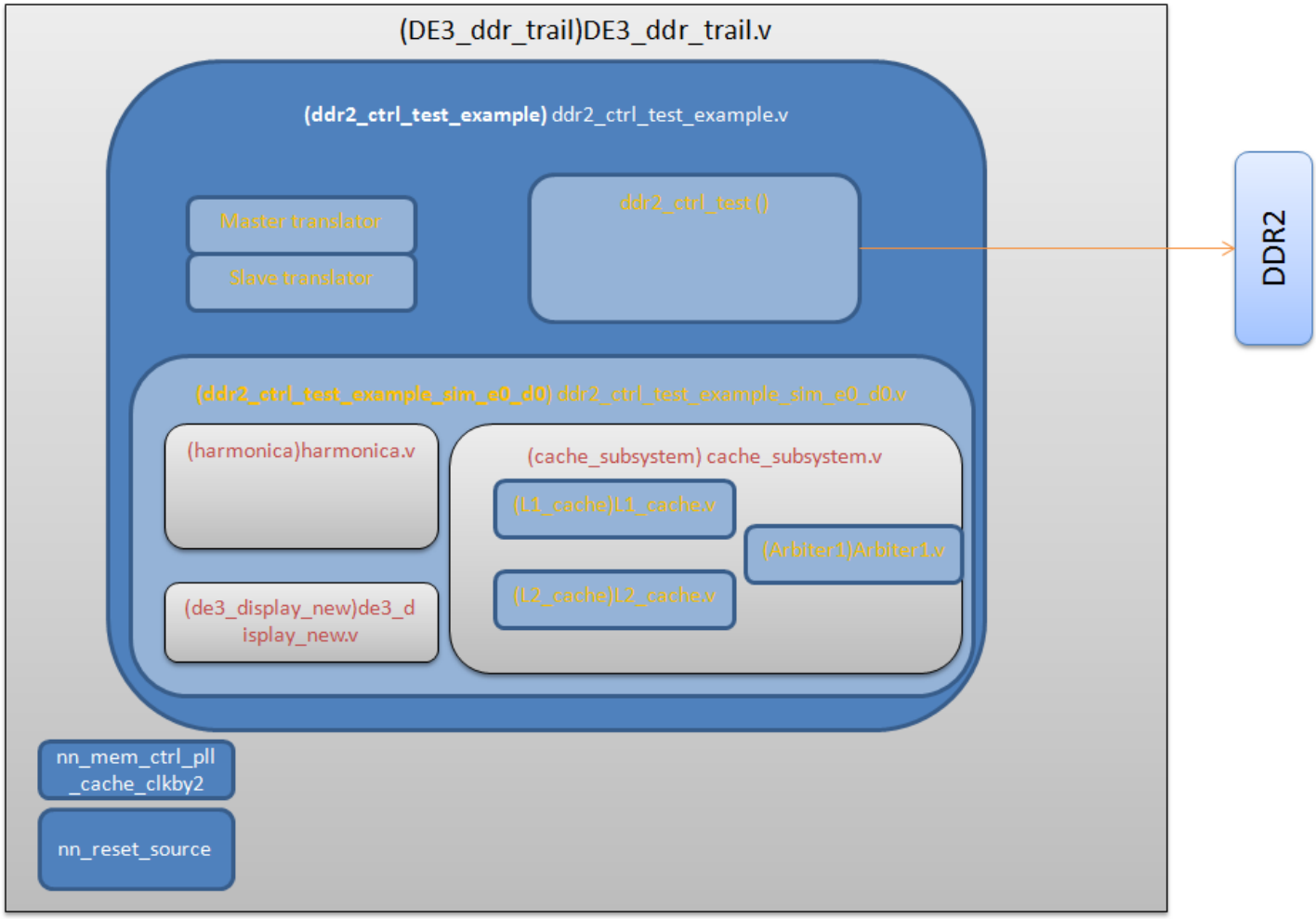


At the location of “license file:” just use any of (depending on which machine has server running):

1801@sunnylotus.cc.gt.atl.ga.us
1802@damint.cc.gt.atl.ga.us
1803@lamint.cc.gt.atl.ga.us

If servers are running then all supported things show in the same window as you can see in the above figure.

2. Overview Of Basic Blocks



These are the main files in the project. The top level file is **"DE3_ddr_trail.v"**. This block has the general 50MHz clock as the input and the signals to the DDR going as output ('avl_* ' signals). There is a pll for generating the 2x clock needed by us **(nn_mem_ctrl_pll_cache_clkby2)** and also a reset generator **(nn_reset_source)** which can generate reset for as many cycles as specified.

"ddr2_ctrl_test_example_sim_e0_d0.v" is the file which instantiates the DDR2 controller (**ddr2_ctrl_test**) and the ddr2 controller driver (**ddr2_ctrl_test_example_sim_e0_d0.v**). There are also some signal translator files which should be left as it is (master and slave translator).

ddr2_ctrl_test.v: is the name of the DDR2 Ip which was generated from Megacore. It is recommended to use the same file as preset in the project in the git repository.

All these files remain the same for all our systems. As we need a DDR2 controller and the top level file for PLL and reset logic. The only file which changes for different systems is the ddr2_ctrl_test_example_sim_e0_d0.v

ddr2_ctrl_test_example_sim_e0_d0.v: is the main driver of the DDR2 this is the file which generates signals from our system to the outside world (which is the DDR2 right now and also LED's and 7 segment displays) . This file instantiates the core and the cache. The number of cores and caches instantiated depends on the desired system. I have created a lot of example files for each kind of system, this is discussed more later.

Before that, we will discuss the files for our core and cache.

3. Memory / DDR2 Code

We have the option to choose either a dummy memory created using block RAM or we can use an actual DDR2. This is controlled by a flag called "DUMMY_MEM" which is used in the L2 cache code to either use a dummy memory or use an interface for the DDR2 and drive the DDR2 signals.

If creating a new project it is recommended to copy the example project and use your HARP system by creating a corresponding "**ddr2_ctrl_test_example_sim_e0_d0.v"** and keeping everything else unchanged. Some snapshots of the DDR2 settings in the megacore wizrd is here: "/harp-system/src/extras/ddr2_params_snaps/"

DDR2 controller is instantiated using the DDR2 Megacore wizard. Use the parameters and the pin configuration used in the example project. Be very careful in changing these as this can make the DDR to not function properly. You need to only do this once and no other changes will be needed. The DDR2 wizard also creates some example projects which you can use for simulation of just the DDR2 model.

DDR2 RTL Simulation of example project:

To simulate just the DDR2 model using model sim. Open ModelSim and run this command:
"do .../<ip_name>/<ip_name_example_project>/simulation/verilog/mentor/run.do "
[You might have to create the verilog simulation file by running "generate_sim_verilog_example_design.tcl" script in Quartus. This script is located at .../<ip_name>/<ip_name_example_project>/simulation/]
These simulations use a DDR2 model from Altera and can be slow. You can take these as an example to understand how the memory controller is organized but its not required.

4. HARP Core

Main files:

You can get all HARP tools from the harmonica and harptool git repository. I have made only minor changes to the files and I have only included those file in my git repo.

- harmonica.cpp: This is the main file which has all pipeline stages and instantiates all the blocks. I have made only 1-2changes (using ifdef statements)

- lsu.h: has the unit 'SimtLsu" which is the load store or coalescing unit. To enable this unit use the flag "USE_SIMT_LSU" when compiling harmonica. If you want the lanes to interface with the cache then also use the flag "USE_CACHE", else by default each lane will get its own private RAM.

- funcunit.h:has the units for the execute stage. It also has the 'SramLsu" which is the basic MMU for blocking requests. There are two variations of it. By default we will create a RAM for each lane for use and also a ROM without any interface with cache. But if you want to interface a cache with your harmonica core then use the flag "USE_CACHE" when compiling harmonica.

- config.h: This sets up the basic config. Set the number of registers and the number of lanes here. I have only made small changes to change the width of the address bits to 30 when using a cache (we use a DRAM with 1GB address space).

- Makefile: You need to define the flags "-DUSE_SIMT_LSU -DUSE_CACHE " to enable certain configurations as mentioned above and also set the ARCH variable when doing SIMD simulations to set up the number of lanes.

/testprog/*: This has all the applications used for the thesis work.

harmonica.v: This is the verilog file generated once you 'make' harmonica. This is the verilog file for the core used for our projects. It has signals for I/O (char_out*) and cache.

Building harmonica: Just do 'make' and set the appropriate flags in Makefile and change the config.h according to what type of core is need.

I/O read/write: The harmonica core has the signals "char_out" and "char_out_val". "char_out_val": says the data on char_out is valid. "char_out" is some data which the core wants to output to I/O or to external modules which they can use for whatever purpose. An read/writes done at address 0x80000000 are sent to "char_out" and asserts "char_out_valid".

Sample Harmonica cores:

Some sample harmonica cores for both single lane and multiple lane applications are placed here for use:
../harp-system/src/extras/sample_harp_cores/harmonica.v.*
You can use any of them as your core but be sure to set the flags when using SIMD cores for SIMD simulations (the cache code uses these flags as well). The data generated in the thesis is using all these harmonica cores.

Single Lane benchmarks:

-bubble.s: single lane bubble sort
- sieve.s: find prime numbers in 1-100
-matmul.s: matrix multiplication
-sum_mem.s: array sum

SIMD benchmarks:

(register %r0 is initialized with the Lane id for each lane and the same instruction is executed for all the lanes)!
matmul_mt.s: Matrix multiplication with SIMD instructions.
sum_simd8.s: Do a un-coalesced sum of a an array of numbers
sum_sim8coal.s: Do a coalesced sum of a an array of numbers
simd_randtest*.s: random tests to stress the cache and load/store unit.

5. Cache and System Driver Code

For overview of the cache read the thesis write up and also the "/doc/Final_Report_Format_Kani_Hassan.pdf"

Main Files:

cache_subsystem.v : This is the L1+L2 cache top level file. It instantiates 1 L1 and 1 L2
cache_shared.v: This is the 2x L1+L2 cache top level file. It instantiates 2 L1's and 1 L2, along with an arbiter.
Arbiter1.v: This is the arbiter used for shared cache. Right now this is for 2 L1 caches. To support more caches just changes the input signal list to add the extra signals and change the arbitration code a little (at line: 84)
L2_cache.v: L2 cache, this nstantiates lot of components. But mainly it instantiates either a "Mem_dummy" which is a dummy memory or it can instantiate a block to interface with the DDR2 ("mem_ctrl_wrapper"). By default it will instantiates a block to interface with the DDR but if you want to use the dummy memory then use the DUMMY_MEM flag while compiling your verilog code. (Example: iverilog -DDUMMY_MEM ….)
L1_cache.v: L1 cache. Main components are the "tag ram", "data ram", "mshr" and the "non blocking fsm".

nonblocking_fsm.v: cache controller used by both L1 and L2.

Other Files: FIFO.v, fill_list_shift_register.v, free_list_FIFO.v, MSHR_2.v, Mem_dummy.v, L1_tag.v, tag_array.v,L2_ram.v, GenericOneHotMux.v, array.v, data_ram.v, data_ram_simd.v, GenericMux.v.

System Driver File (ddr2_ctrl_test_example_sim_e0_d0.v)

This is one of the main files of the project to design different systems, hence we will discuss the structure of this file in some detail. This file mainly takes in the clock signals, signals(data) from DDR2 controller, signals(requests) going out to DDR2 controller, and also generates test signals which drive the LEDs.

```
module ddr2_ctrl_test_example_sim_e0_d0 #(
        parameter …. //you can ignore most parameters except address,data width
) (
        input/output avl_* //input output signals to DDR2 controller
        ….
        output pass/fail/test_complete //signals to drive LEDs
        output disp1/disp2 //signals to drive 7-segment displays
);

harmonica1 harp1(
….
harp_ready_out,       //core sending data valid signal for I/O display etc..
harp_data_out;        //core sending data for I/O display etc..
);

//depending on system more harp cores,  shared cache etc..

cache_subsystem  cache(
….
);

always@(posedge clk) begin
      if(harp_ready_out == 1'b1) begin
          data_sum <= data_sum + harp_data_out;          //Calculating checksum of data sent out by the core
          stop    <= 1'b1;  // we assert stop signal as soon as Harp core starts sending outputs to I/O displays
      end
end
….
assign test_complete = stop;
assign pass = (data_sum == 13)? 1'b1 : 1'b0 ;        //checksum logic depending on the application
….
perf_counter <= perf_counter + 1; //perf counter increment logic
….
de3_display_new  display(clk, perf_counter, pass, disp1, disp2, led);  //logic to display perf. on 7-seg displays!
```

6. Build and Simulation

Steps for RTL and Board Runs:

1. Get the latest harmonica core from https://github.com/cdkersey/harmonica
Then use the files in /harp-system/src/core/ instead of corresponding file in harmonica. You should merge the changes possibly, as orthogonal improvements keep happening in the Harmonica core which does not affect the MMU or the cache interface (These are the only extra features which we need to enable for our use, so merge should be conflict free!!)*
For SIMD core: use the "-DSIMD -DUSE_SIMT_LSU" flags in Makefile

*2. Select the **application** you want to run. Choose sample applications present in "/harp-system/src/core/testprogs/* ". Link the "rom.s" file in the harmonica directory to this file. (Simlink in Unix: ln -s <testprogs_dir/testfile.s> rom.s)*

*3. Compile Harmonica: do "**make**" in the harmonica directory. Take care of the flags to make sure they are set for corrects no. of lanes. Also set the **config.h** parameters properly. This will create a **"harmonica.v"** for the application you chose before. Such sample cores are also here "*/harp-system/src/extras/sample_harp_cores/harmonica.v.* "

Now we are done with the core part! Lets look at other things.

*4. Choose system driver file: This means what should you use as '**ddr2_ctrl_test_example_sim_e0_d0.v'.** This file defines the system you want to simulate or drive the DDR2 with.There are several options in **/harp-system/src/extras/system_drivers/***

Doing Single Core runs:	*ddr2_ctrl_test_example_sim_e0_d0_harp.v*
Doing dual core runs:	*ddr2_ctrl_test_example_sim_e0_d0_2core_harp.v*
Doing SIMD core runs:	*ddr2_ctrl_test_example_sim_e0_d0_harp_simd.v*
Doing Big.Little core runs:	*ddr2_ctrl_test_example_sim_e0_d0_harp_1Big2Little.v*

Just copy any of these to ddr2_ctrl_test_example_sim_e0_d0.v
Each of these driver files also have code to set test flag (LED) for each application as I mentioned before. So comment/uncomment the lines for testing the application that your HARP core is running.

*5. Now you have the harmonica.v code for the core and all the codes for the **cache** in "/harp-system/src/cache/ ". You also have the system driver file. Now we are all set for **RTL simulation.***

All RTL simulations were done using iverilog and all gate level simulations were done using Quartus. Dont do RTL sim in ModelSim/Quartus as they will complain for relaxed verilog syntax of the harmonica.v. To do RTL simulation using iverilog, specify all files(all verilog files and testbenches) to be simulated in a file (lets say file_list.txt) and then do:

iverilog -DSIMD -DDUMMY_MEM -c file_list.txt;

*An example "**file_list.txt**" can be found here: **/harp-system/src/cache/file_list.txt** (just change the location of the files if needed). This also includes a testbench for RTL simulations. The top level testbench used for RTL simulations can be found here **"harp-system/src/extras/testbench.v"***

We have used the two extra flags in above command, we always use a dummy memory when doing RTL simulations and also the SIMD flag is when you want to do SIMD simulations. Then to run just do **"./a.out"**

It will dump out a waveform (dump.vcd) which can then be viewed using **"gtkwave dump.vcd"**

*6. Verify if your application passes by looking at the **'pass', 'fail', 'test-complete'** signals in gtkwave. These are present in "ddr2_ctrl_test_example_sim_e0_d0.v" module.*

*7. Once your RTL simulation is done. We are now read to create the **FPGA binary**.*

8. Open the archived Project in Quartus. This should have all the top level files. Make sure the DDR2 controller is present and the pins are already configured. If verilog files are missing add all the relevant files to your project. Mainly all files are present in "harp-system/src/ " (remove the testbenches if you want)*

9. Once all files are included mainly for core, cache and the driver file is set properly. Then just make one last check to set the flags in Quartus. Remember for SIMD compilation we need to use the SIMD flag. For DDR2 we don't have to use the DUMMY_MEM!. Go to "assignments->settings-> Verilog HDL input'. Set the macro in the section called 'Verilog HDL Macro'.

Now all setting for quartus are done so ahead and run all the analysis, synthesis and generate programming file and burn it on the FPGA !! You will know if you are successful by looking at the LEDs described below. We also display the performance counters on 7-segment displays.

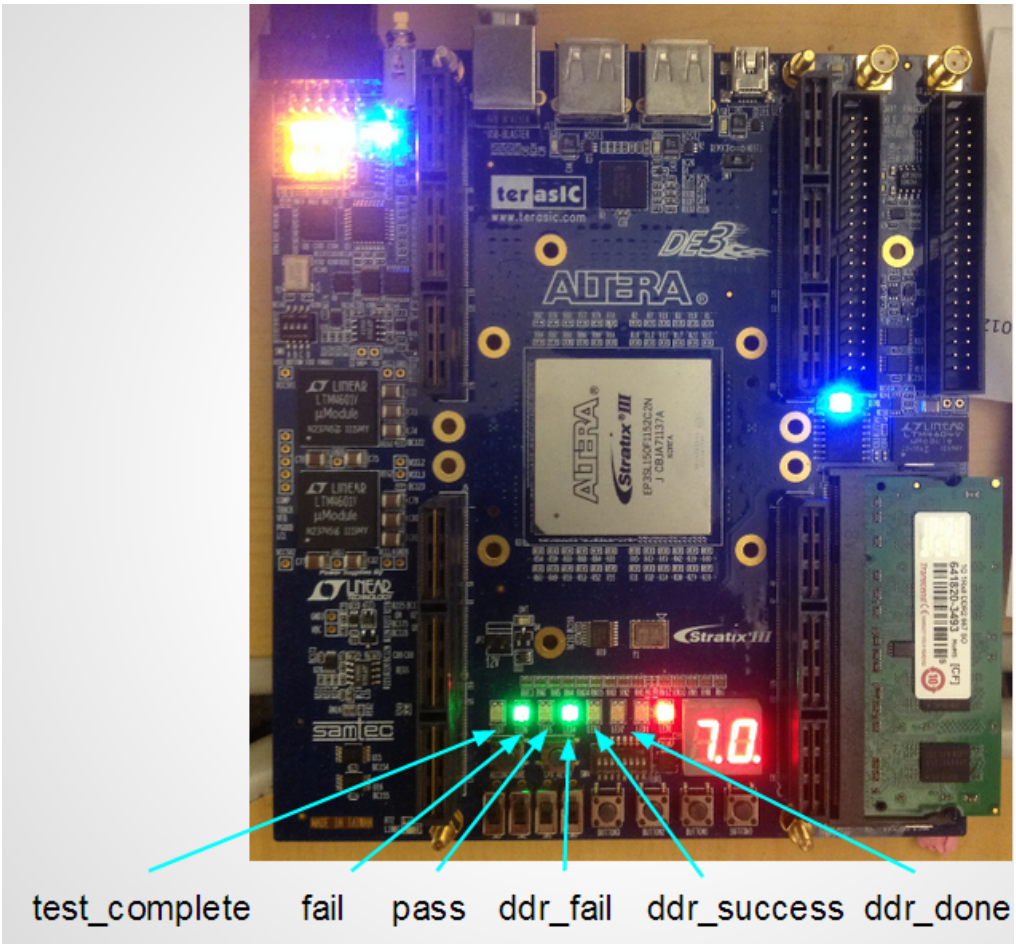
*P.S: If you want to do Gate Level Simulation:*These can be directly done from the quartus tool. It will be better to use the dummy memory for gate level simulation as well. As this can significantly reduce the simulation time (the DDR2 controller will take more than 10hrs to just initialize).

LEDs/Flags:

signal(label) : description

.local_init_done (**ddr_done**): if DDR2 finished initialization.
.local_cal_success (**ddr_success**): if DDR2 passed the initialization.
.local_cal_fail (**ddr_fail**): if DDR2 failed the initialization.
.drv_status_pass (**pass**): if application passed the test in the driver file.
drv_status_fai (**fail**): if application failed the test in the driver file.
drv_status_test_complete(**test complete**): application reached the end

The Figure below shows these LEDS for each signal on board. The LEDs are active low so when any signal is 'high' it is off and when a signal is 'low' it glows. So below we can see 'fail' and 'ddr_fail' singal are '0' or low and hence LEDs corresponding to them are glowing.



Performance Counter: The number of cycles taken is show on the 7-segment displays in 'hex' format. The performance counter is 32bits, and we display 4bits (one hex digit) at a time on the display one by one. The first number ('7') is the actual hex digit and the second number('0') is the bit no. which it represents. So to display 32 bits (8 hex bits from h7...h0) we display ('h7 7', 'h6 6',...'h0 0'). Fro the above figure h0 is '7'.

7. ISSUES

The main issue is the clock. Right now the operating frequency is 62.5 MHz. The DDR2 runs at 125 MHz. DDR2's minimum operating frequency is 125MHz and its functionality below that is not guaranteed.

The timing requirement of the design as reported by Quartus is about 62.5MHz for single lane and about 50 MHz for the SIMD lane. Which means running it will be slightly risky. Even when we run the design at 62.5MHz we are still able to run the single lane and SIMD core for all the applications in the 'testprogs' folder. I was also able to run the heterogeneous systems. But I did start running into problems when I started running 8Little-1Big Core design. The cache by itself can run at around 125MHz so the main bottleneck is the harmonica core.

Debugging:

Suppose we take a random application and it does not run on the board then there are few ways to debug it. These is a suggested way:

- Enable the "DUMMY_MEM" knob and make sure if RTL simulation passes.
- If RTL simulation passes then try to run it on the board using 'DUMMY_MEM' instead of DDR2.
-