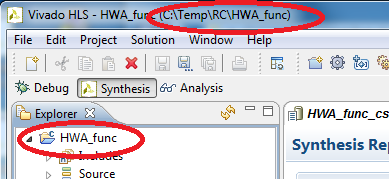
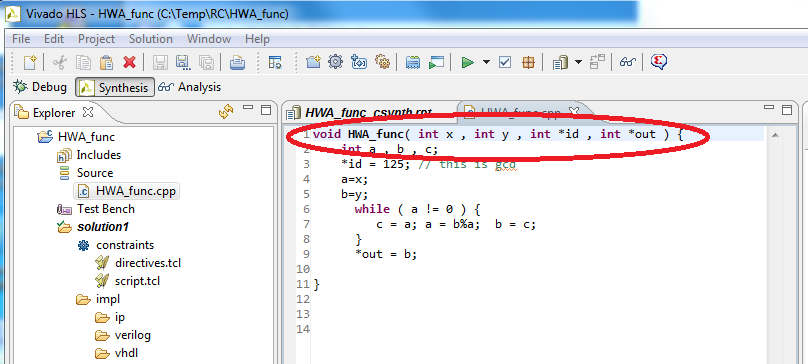
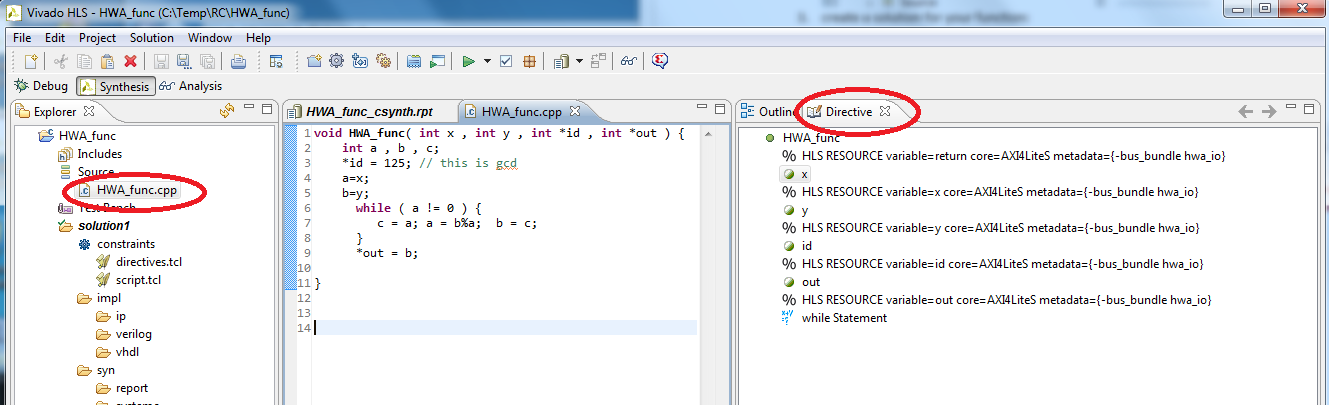
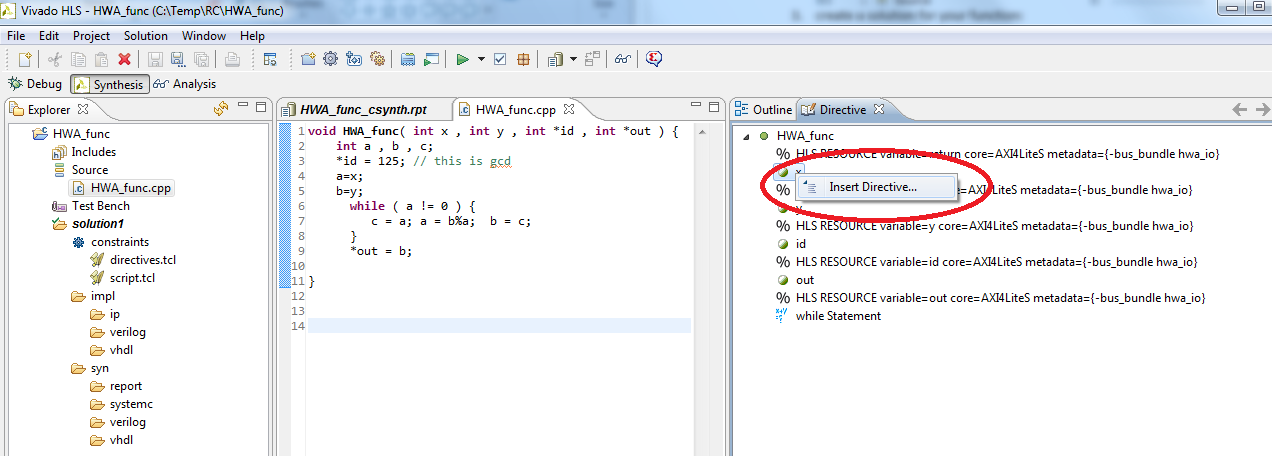
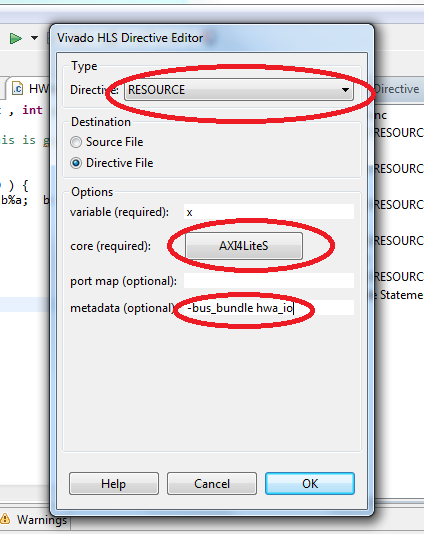
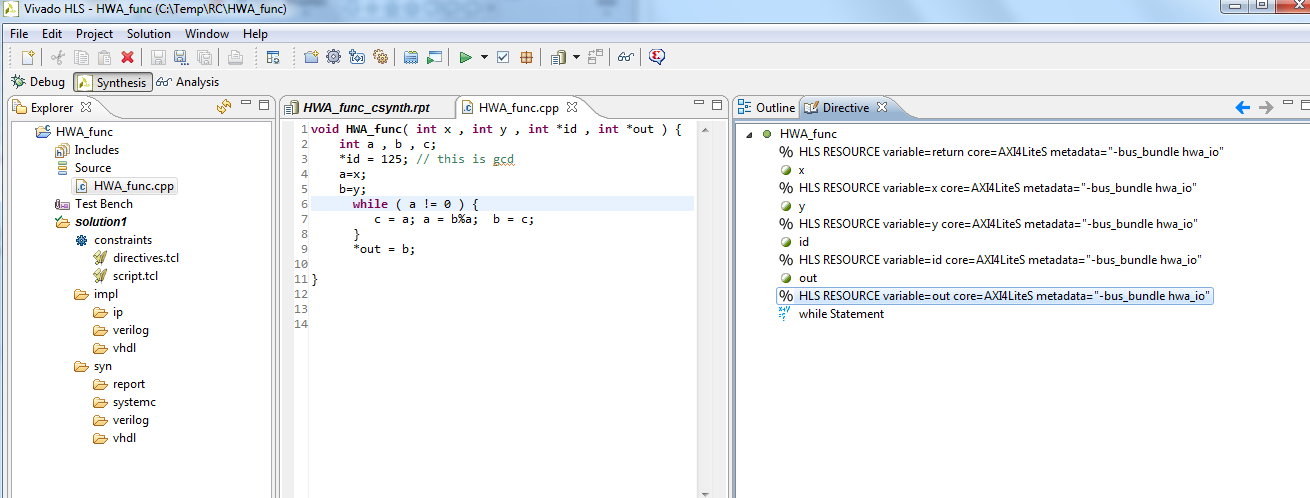
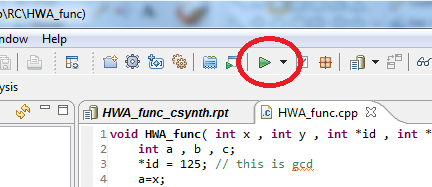
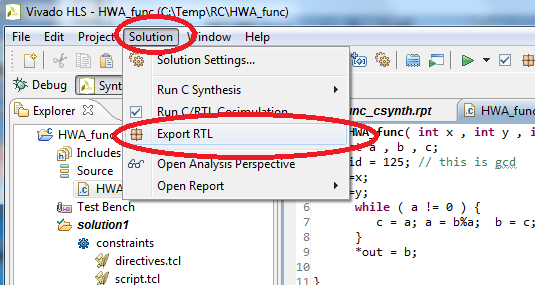
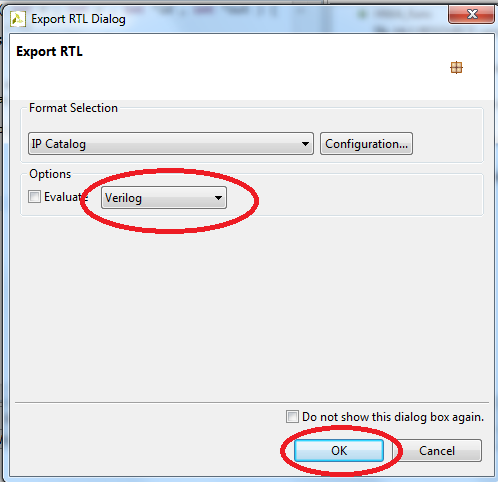
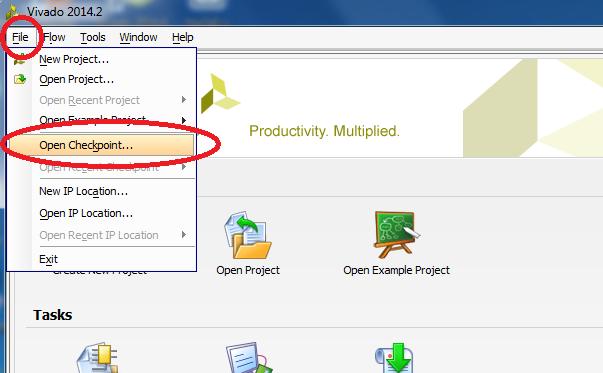
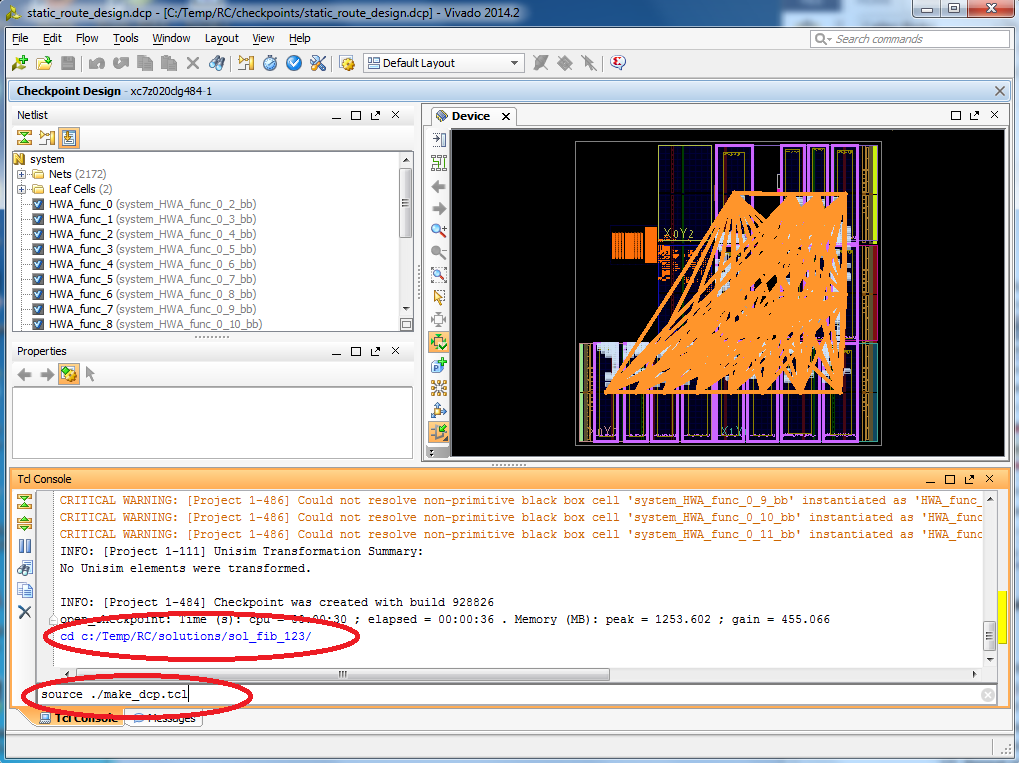
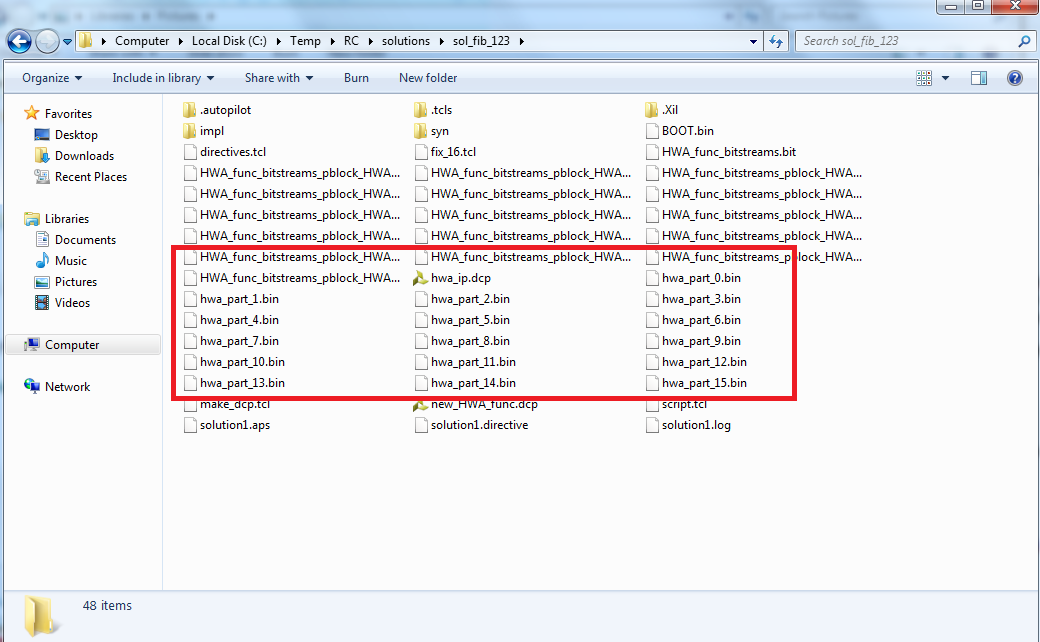
1. Open Vivado HLS (project built on version 2014.2).
2. Open/create project located in Vivado HLS named "HWA\_func"  
   reference project is in <project root>\HWA\_func.  
   
3. create a solution for your function:
   1. It should have 32 bit inputs and pointers to 32 bit outputs.  
      (unsigned int)
   2. In the function signature all inputs come first.

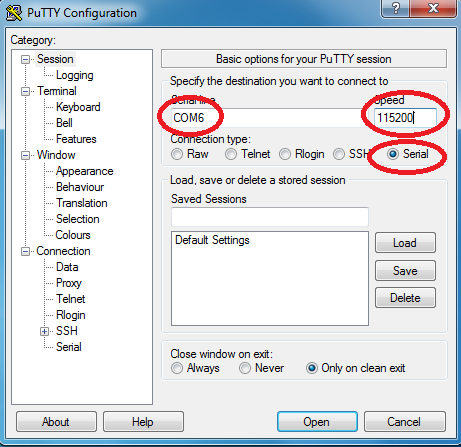
Note: you can edit the code in the reference.  


1. Add a directive to all inputs/outputs of the function and the function itself:
   1. Open the source file from the project explorer.
   2. Click the directive tab  
      
   3. Right click on a variable/main function and select insert directive.  
      
   4. Select RESOURCE in the directive dropdown menu.
   5. Select AXI4Lite in core by clicking the box next to core.
   6. Add "-bus\_bundle hwa\_io" in the metadata.  
      
   7. The end result should look like this:  
      
2. Run C synthesis by clicking the green arrow.  
   
3. Export RTL (use Verilog) by clicking solution and export RTL.  
   
4. Make shore it says varilog in the box and click OK.  
   
5. Wait for the proses to finish.
6. Copy the "solution" folder from HLS project directory to  
    <project root>\solutions\<your function>
7. Repeat steps 4-6 for every solution
8. Copy the "fix\_16.tcl", "make\_dcp.tcl" and "make\_bin.bat" files from:  
    <project root>\solutions   
   to:  
   <project root>\solutions\<your function>\
9. Run Vivado (tested on 2014.2).
10. In Vivado, go to: "file -> open checkpoint" and select the file:   
    <project root>\checkpoints\static\_route\_design.dcp and click "ok"  
    
11. Dismiss the warning by clicking OK.
12. In the tcl console below go to <project root>\solutions\<your function>\  
    using: cd <project root>/solutions/<your function>/
13. Run: "source ./make\_dcp.tcl"  
    note: this step takes some time.  
    
14. Repeat steps 10-12 for every solution.   
    WARNING: don’t save data if asked!
15. Brows to your solutions folder <project root>/solutions/<your function>/ and double click on the "make\_bin.bat" file, do this for every solution.

**Results:**

Now in every solution folder you will have the files: hwa\_part\_#.bin  
while # runs from 0-15. The file hwa\_part\_#.bin in folder <project root>\solutions\<your function>\ is the file you need to program the FPGA PR block # with <your function>.  


**Programing the FPGA using a partial reconfiguration bin file manually:**

1. Open a Linux shell on the system:
   1. Power on the board with the provided SD-Card.
   2. Connect the board UART port to the PC USB port.
   3. Run putty
   4. Select serial connection type.
   5. Select the com provided by windows.
   6. Set the speed to 115200  
      
   7. Click open.
2. In the provided Linux version on the SD-card the password for root is "root".  
   note: next steps depend on the Linux version and may vary.
3. Run:  
   echo 1 > /sys/devices/amba.0/f8007000.ps7-dev-cfg/is\_partial\_bitstream  
   to indicate to the Xilinx driver that the next bitstream is a partial bitstream.
4. Run:  
   cat hwa\_part\_#.bin > /dev/xdevcfg  
   to program the bin file.
5. Run:  
   cat /sys/devices/amba.0/f8007000.ps7-dev-cfg/prog\_done  
   to see if programming is done (1 means its done)

**Scripts:**

Scripts are used for automation and optimization.

Two issues need to be solved before we can insert the accelerator into the design:

1. The HDL files are built with the intent of being added to the design during its creation, this means the HDL builder assumes they will be part of the project file and there is no need for includes (since Vivado project does that automatically).  
   We need to add the includes manually (using a script).
2. The HDL is built with the minimum number of address pins needed, since our design is built with the maximum number on address pins we need to resize the address vector in the HDL file.

fix\_16.tcl: this script creates a temp.v file that has all the includes needed and the corrected address vector.

1. It scans the HDL folder and adds "`include file" for every file in this folder that is not the main HDL file (hwa\_func\_top.v ) to the temp.v file.
2. It copies all the data from hwa\_func\_top.v file to temp.v.
3. In file temp.v's main hierarchy the script changes the address vector size to 16 and reduces the number of bits for every recipient of this vector to the previous vector size.

Make\_dcp.tcl: this script synthesizes the design.

1. The script runs fix\_16.tcl script.
2. The script opens the temp.v file created, synthesizes it and creates a design checkpoint from it (DCP file).
3. The script loads the DCP too every PR block.
4. The script runs the necessary functions (opt\_design , place\_design and reute\_design).
5. The script builds a DCP file for the full design with the accelerator inserted and compares it the base design to know if it's PR compatible.
6. The script generates bitstreams of the design, both full and partial.

Make\_bin.bat: this is a shell script that converts bit files to bin files that are needed for programing the FPGA using Linux using promgen.

WARNING: using the exec function in TCL to run promgen to generate the bin files, corrupts them.

**Using the demo:**

There are 2 programs provided to demo the system:

hwa\_start\_stop:

This program builds all the necessary files for the demo to work.

Usage:   
hwa\_start\_stop.elf 1 ; starts the system  
hwa\_start\_stop.elf 0 ; stops the system

demo\_tester:  
This is the main demo program.

The demo\_tester has the following dependencies:

1. Hwa\_start\_stop ran in the same folder before running demo\_tester.
2. For every accelerator planned to be used there needs to be a folder with only the bin files of that accelerator.

Usage:   
Run demo tester with accelerator paths as arguments, example:  
./demo\_tester.elf 123\_fib/ 124\_prime/ 125\_gcd/

The program will ask you to provide the number of inputs and outputs to each accelerator and then give you a simple menu.

press 0 to exit  
press 1 to load a new HWA  
press 2 to unload an HWA  
press 3 to use an HWA  
press 4 to read accelerator registers for debug  
press 5 to test in SW with time  
press 6 to test in HW with time

The demo accelerators:

|  |  |  |  |
| --- | --- | --- | --- |
| accelerator folder | inputs | outputs | explanation |
| 123\_fib | 3 | 2 | Find the number in Fibonacci series that is represented by the sum of the 3 inputs. First output is always 123, the second is the result. |
| 124\_prime | 1 | 2 | Find the prime number indexed by the input. First output is always 124, the second is the result. |
| 125\_gcd | 2 | 2 | Find the greatest common denominator between 2 numbers that are represented by the 2 inputs. The first output is always 125, the second is the result. |

Demo accelerators timing analysis:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| accelerator | time SW | time HW | difference | % saved | inputs | notes |
| Fibonacci | 3.15004 | 1.234562 | 1.915478 | 60.81 | 123456789 |  |
| prime | 0.393612 | 2.03907 | -1.64546 | -418.04 | 1234 |  |
| GCD |  |  |  |  |  | algorithm to fact to compare |

Fibonacci algorithm was solved by HLS in a way that benefits HW, from the results we can see that the HW is 60% faster.

Prime algorithm is match faster on SW that on HW (HW is 418% slower). This might be because HLS default solution is not optimized well for HW or the algorithm itself is faster on SW.

GCD algorithm got results in single micro seconds on both SW and HW regardless of input size so it's not comparable.