Create and integrate IP for Cyclone V using the intel HLS compiler

Table of Contents

Requirements

The following steps assume a valid installation of Quartus 18.1 as well as the HLS compiler. This compilation process is done on a virtual machine using ubuntu 16.04 LTS.

Makefile

The Makefile sets some important flags and compilations settings and generally simplifies the compilation.

The following code is used:

```
SOURCE_FILES  := add_one.cpp
HLS_CXX_FLAGS :=
CXX := i++
override CXXFLAGS := $(CXXFLAGS)
VERBOSE := 1

# OS-dependant tools
ifeq ($(OS),Windows_NT)
  RM  := rd /S /Q
else
  RM  := rm -rfv
endif

ifeq ($(MAKECMDGOALS),)
  $(info No target specified, defaulting to test-x86-64)
  $(info Available targets: test-x86-64, test-fpga, test-gpp, clean)
endif

# Any tools installed with HLS can be found relative to the location of
i++
HLS_INSTALL_DIR := $(shell which i++ | sed 's|/bin/i++||g')

# Run the i++ x86 test by default
.PHONY: default
default: test-x86-64

# Compile the component and testbench using g++ and run them as a regular
program
.PHONY: test-gpp
test-gpp: CXX := g++
test-gpp: CXXFLAGS := $(CXXFLAGS) -I"$(HLS_INSTALL_DIR)/include" -
L"$(HLS_INSTALL_DIR)/host/linux64/lib" -lhls_emul -o test-gpp
test-gpp: $(SOURCE_FILES)
      $(CXX) $(SOURCE_FILES) $(CXXFLAGS)
      @echo "+-----------------------------------+"
      @echo "| Run ./test-gpp to execute the test. |"
      @echo "+-----------------------------------+"
```

```
# Run the testbench and the component as a regular program
.PHONY: test-x86-64
test-x86-64: CXXFLAGS := $(CXXFLAGS) $(HLS_CXX_FLAGS) -march=x86-64 -o
test-x86-64
test-x86-64: $(SOURCE_FILES)
	$(CXX) $(SOURCE_FILES) $(CXXFLAGS)
	@echo "+----------------------------------------+"
	@echo "| Run ./test-x86-64 to execute the test. |"
	@echo "+----------------------------------------+"

# Run a simulation with the C testbench and verilog component
.PHONY: test-fpga
ifeq ($(VERBOSE),1)
  test-fpga: CXXFLAGS := $(CXXFLAGS) -v
endif
test-fpga: CXXFLAGS := $(CXXFLAGS) $(HLS_CXX_FLAGS) -march=CycloneV -o
test-fpga
test-fpga: $(SOURCE_FILES)
	$(CXX) $(SOURCE_FILES) $(CXXFLAGS)
	@echo "+-------------------------------------+"
	@echo "| Run ./test-fpga to execute the test. |"
	@echo "+-------------------------------------+"

# Clean up temprary and delivered files
CLEAN_FILES := test-gpp \
               test-gpp.prj \
               test-fpga \
               test-fpga.prj \
               test-x86-64 \
               test-x86-64.prj
clean:
	-$(RM) $(CLEAN_FILES)
```

The target is selected with the -march flag. Here the CycloneV was chosen. This makefile can also be found in the hls/examples/counter folder. Make sure to change the name of the cpp file in the first line as well as the argument of the -march flag if you copy this file.

C/Cpp Code for component

For this example, a very basic component will be created, that simply adds 1 to a given value. The component keyword is used to declare the hardware that shall be implemented on the FGPA. The content of the main method is used to create a testbench, as we will later see.

The Makefile and the add_one.cpp need to be in the same directory.

add_one.cpp

```
#include "HLS/hls.h"
#include <stdio.h>

using namespace ihc;

component unsigned long add_one(unsigned long in_val)
{
    return in_val +1;
}

int main(void)
{
    // very basic testbench

    bool pass = true;
    long TEST_VAL = 1000;
    TEST_VAL = add_one(TEST_VAL);
    if(TEST_VAL != 1001)
    {
        printf("ERROR, expected %lu, found %lu \n",(unsigned long)
1001,TEST_VAL);
        pass = false;
    }

    if(pass)
    {
        printf("PASSED \n");
    }
    else
    {
        printf("FAILED \n");
    }

    return 0;


}
```

Compiling the component

Now we can compile a x86 binary as well as a HDL representation of the code using the following commands:

```
make test-x86-64 -o add_one
make test-fpga -o add_one
```

Testing the component

In both cases we can invoke the testbench with ./test-x86-64 or ./test-fpga
In the latter case the hardware is simulated to reflect the behavior of the fpga.

For more complex components it is often of interest to look at the signal flow. This can be done by using the -ghdl flag, where all signals are saved. Instead of editing the makefile, we can simply invoke the i++ command directly:

```
i++  -ghdl add_one.cpp -v  -march=CycloneV  -o add_one
```

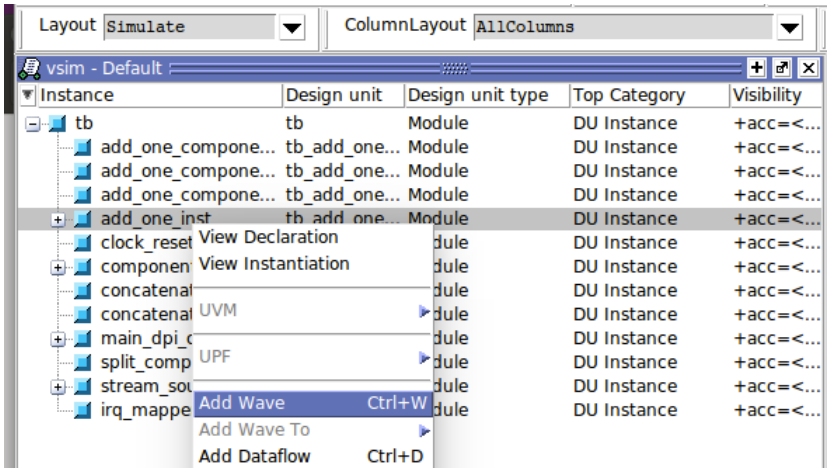
Next, we run the simulation once:

```
./add_one
```

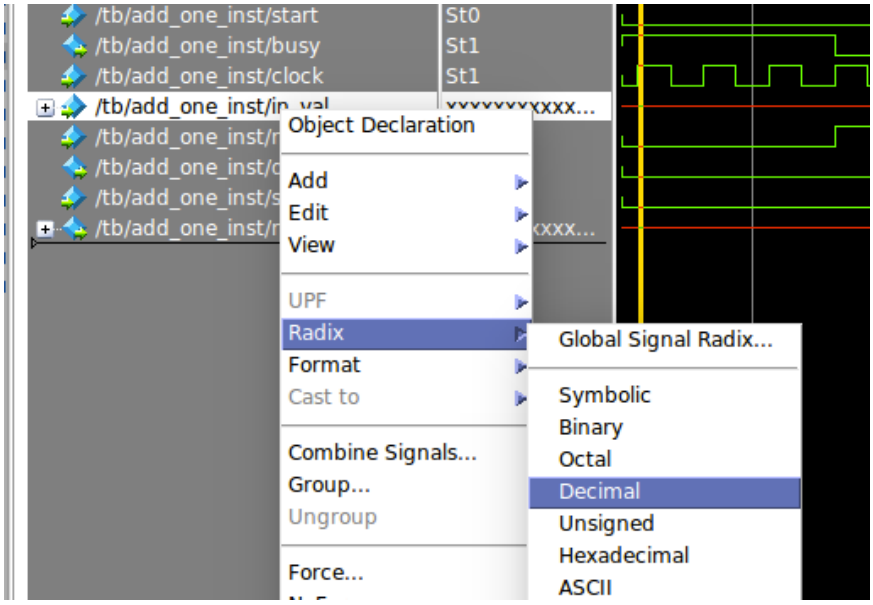
Now we can inspect the component in modelsim:

```
vsim add_one.prj/verification/vsim.wlf
```

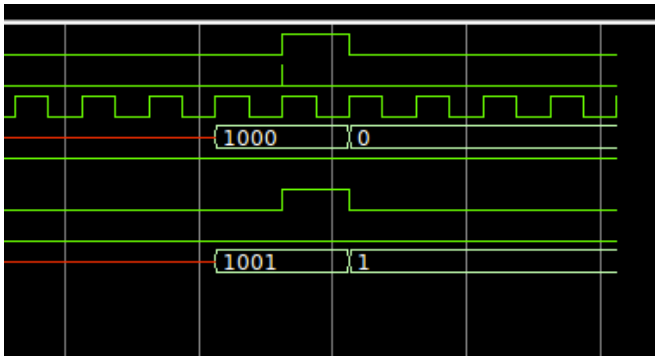
In Modelsim we find the instance of our component called add_one_inst and add the wave with a right click:



After, we press F to view the whole wave form.
Finally, we right click on in_val and return data and select Radix->Decimal



We can now see that the hardware acts as expected and simply adds 1 to the value it was given.

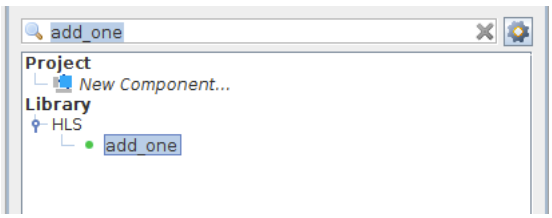


Use IP on the FPGA

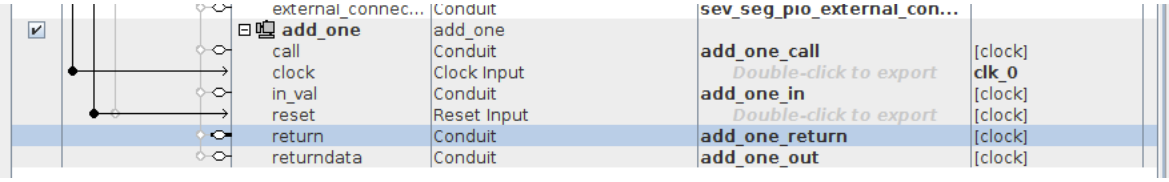
In this step we will integrate the previously generated IP into an existing quartus project.

Open Quartus and the Platform Designer. In the Platform Designer go to Tools->Options… Here we can add an additional search path for the IP Catalog. We will add the directory in which we have just created the IP. <your_path>/add_one.prj/components/add_one.

Confirm with finish. Now we can search for our custom IP:



We add it with a double click and connect the clock and reset signals. For this IP we want to export the call, in_val, return and returndata signals:



Now we need to recreate the system. Click Generate-> Generate HDL and click Generate at the bottom.

If you have not added the qip file that is generated by the platform designer before, you should now do this. Next, we can add the IP to an existing project. The relevant file is called <your_project_name>_add_one.v in this case. Insert the following code into the top level entity:

```
soc_system_sph_add_one add_one_inst(

            .clock      (CLOCK_50),      //      clock.clk

            .resetn     (1'b1),     //      reset.reset_n

            .start      (1'b1),      //       call.valid

            //.busy       (busy),       //           .stall

            //.done       (done),       //     return.valid

            .stall      (1'b0),      //           .stall

            .returndata (sev_seg_internal_processed), // returndata.data

            .in_val     (sev_seg_internal)      //     in_val.data

);
```

For now, we will not use any flow control and hence comment out the busy and done signals and set the stall signal to a fixed 0 (never stall). In this example the data is written to a PIO port address by a C program running on the HPS. The data is accessed through the sev_seg_internal wire. The IP then returns the value of sev_seg_internal +1 to the sev_seg_internal_processed wire, which then in turn can be returned to the user or shown on the seven-segment display. The value displayed is now always one more than the value of the data that was written to the PIO address, which shows that the IP integration was successful.

The full quartus project can be found here:
https://github.com/yanschneider/Cyclone_V_SoC