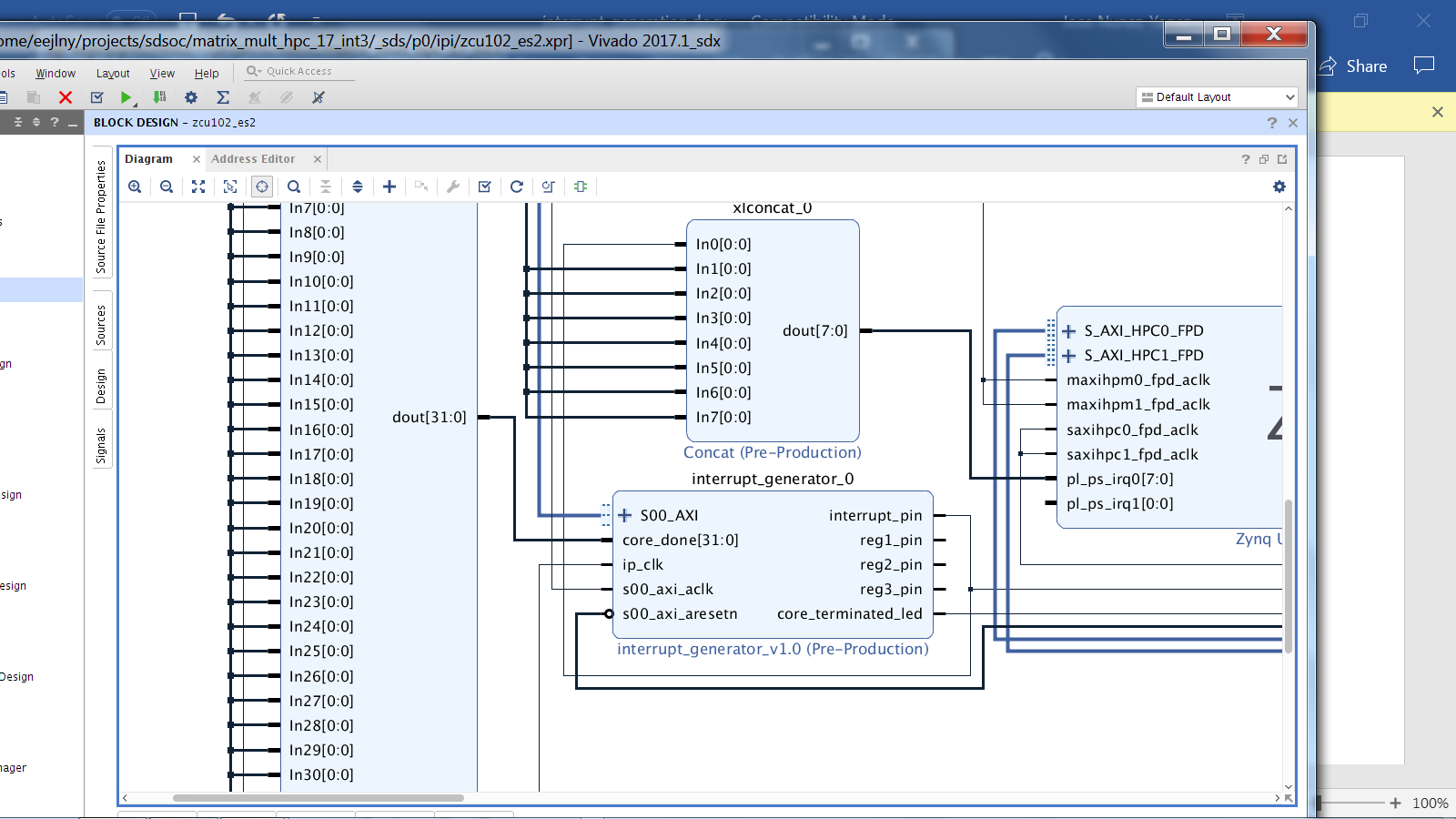
We have a new approach that creates a hardware implementation that directly uses the ap\_done signal originating from the hardware accelerator to signal termination to a interrupt generation controller. The advantage is that there is no need to modify the source code of the accelerator application to generate the interrupt so there is no need to add more ports to the accelerator application as done before. The available scripts described below take care of the implementation automatically.

**The following needed files are located in github https://github.com/eejlny/**gphcu/zcu102:

1.mydriver.ko (driver for the interrupt generation controller)

2.example project matrix\_mult\_hpc\_17\_int3. Check the paths in the Makefile,boot\_bin.bif and vivado\_int.tcl scripts as indicated below so they point to your own paths and do a make. If everything goes well you should get a project with interrupt generation IP included as shown in this figure.



**In the previous version published in SC journal the hardware accelerator code was modified so the last operation before finishing was to write a memory mapped register that corresponds to the interrupt register. That was the mechanism used to generate interrupts but the diasvantage is that the accelerator code had to be modified and that additional function parameters had to be added to pass information regarding the location of the interrupt register to the accelerator and the interrupt variable itself. In this new version the implementation scripts modifies the block diagram so the ap\_done control line is connected to the interrupt controller directly that monitors when this signal pulses and uses this to generate the interrupt. The advange is that the source code of the accelerator remains unmodified and the hardware connections can be done automatically in scripts that modified the sdsoc hardware.**

**Interrupt platform project generation:**

Makefile : the makefile will generate the sdsoc project using the interrupt platform. The makefile should point to use the interrupt platform with:

PLATFORM = /home/programs/Xilinx2/SDx/2017.1/platforms/zcu102\_es2\_ig/zcu102\_es2

The makefile also includes the following lines:

vivado -mode batch -source vivado\_int.tcl

bootgen -arch zynqmp -image boot\_bin.bif -w -process\_bitstream bin

The first line launches a script that modifies the project to correctly connect the interrupt generator IP and the accelerator. It also reimplements to take into account the modifications. The second line creates a bit.bin ready to moved to the FPGA device.

It is important to edit the vivado\_int.tcl to make sure that the paths match the project paths for a new user. Similarly it is important to edit boot\_bin.bif and make sure the paths match the user paths.

Once all these have been verified a simple make in the project directory will execute the Makefile and generate the sdsoc outputs zcu102\_es2\_wrapper.bit.bin (hardware description file in implementation directory impl1) and the library (for example libkernelmatrixmult.a in project directory) that needs to be copied to the host directory in the board. For example,

root@zcu102:~/parallel\_for\_xilinx/MM/LibMatrixmult# ls

libkernelMatrixmult.a

Then you can recompile the user application in the board with a make. You need to insert the driver with insmod mydriver.ko and then you are ready to run the user applicatoin with for example

./MM\_DSW 0 1 1024

After execution the user app reads a interrupt register to know which core terminated. if you type dmesg you can also see the interrupt generated and which core generated the interrupt.

This information can be used by the scheduler as follows:

**Interrupt scheduler integration:**

The new interrupt generation IP part of the platform has 2 registers reg0 at 0xa0000000 that generates interrupts and is reseted by the interrupt service routine and 0xa0000010 that contains information who generated the interrupt (at the moment just core 1)

The idea is that the cores will trigger interrupts as they terminate the work assigned to them. The scheduler will wake up when an interrupt is received and read interrupt register 1 that indicates wich core has terminated so the scheduler can assign more work to it. The draft steps are as follows:

1) the scheduler assigns work to all available cores and sets reg1 to 0 (so all cores busy) then goes to sleep by calling mydriver\_ioctl(as done in Zynq) so the thread goes to sleep waiting for interrupt

2) As core finish they generate interrupts and wake up scheduler thread. The corresponding bits in reg1 are set to 1 by the cores that have finished.

3) The scheduler checks if more work pending and if all work done then exit.

4) if more work available the scheduler reads reg1 to know which cores have finished assigns more work to them, sets the corresponding bits in reg1 to 0 and goes back to sleep by calling mydriver\_ioctl

5) scheduler goes back to 2)

7) all done