Embedded System Hands-On 2

Topic: Report ESHO2
Student: Nguyen Tien Dat Tran 2871790
Date: Winter semester 29.12.2020

Fachgebiet Embedded Systems and Applications Group
Prof. Dr-Ing. Andreas Koch

Contents

1 Task 1

1.1 Build the Linux Kernel together with the appropriate drivers

Since the small Ultra96 board doesn't have the capacity to build the Xilinx Linux kernel directly on board, we have to build the kernel on the server first and copy it to the board later. In other word, we're using cross-compile to compile the Xilinx Linux kernel.

Before we build the drivers, we need the compiler provided by vivado, execute this command first:

```
export PATH=$PATH:/opt/cad/xilinx/vitis/Vitis/2020.1/gnu/aarch64/lin/aarch64-linux/
export XILINXD_LICENSE_FILE="/opt/cad/keys/xilinx"
source /opt/cad/xilinx/vitis/Vivado/2020.1/settings64.sh
```

Listing 1: Add path for compiler environment

Now we can now build the image with the following commands:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- xilinx_zynqmp_defconfig
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

Listing 2: Compile the kernel

After the menuconfig is executed, the user-friendly interface will show up, we have to look the driver of RTL8512(The Ethernet to USB device) and save this configuration. Now the kernel is already built, we will use these files for the next section where we put them into the SD card to boot the Ultra96-v2 board. In particular, we need the image file and the Avnet-ultra96-rev1.dtb.

1.2 Partition SD card

The SD card is partitioned to two parts. The first partition is FAT32 and the second partition is Ext4.

The first partition(name BOOT) must include 3 files:

```
image (located at /home/stud/username/echo2/linux-xlnx/arch/arm64/boot)
boot.bin (downloaded from moodle)
system.dtb (rename from avnet96.dtb to system.dtb)
```

Listing 3: First Partition

The system.dtb is actually the file avnet-ultra96-rev1.dtb. We just rename it to system.dtb. Avnet-ultra96-rev1.dtb is located at:

```
home/stud/username/echo2/linux-xlnx/arch/arm64/boot/dts/xilinx/avnet-ultra96-rev1.dtb
```

Listing 4: Location of dtb file

The second partition(name RootFS) must include the ARMv8 AArch64 Multi-platform, which can be cloned here:

```
https://archlinuxarm.org/about/downloads
```

Listing 5: Second Partition

1.3 First Boot and Package Install and Upgrade

Now we can plug SD card into the board and boot it up. After booting up, we must upgrade the packages, install sudo package.

```
pacman-key --init
pacman-key --populate archlinuxarm
pacman -S archlinuxarm-keyring
pacman -S sudo
pacman -Syu
```

Listing 6: Second Partition

Now we have **sudo package** installed. So we can execute "visudo" to change the alarm user to root user.

1.4 Vivado with MysteryRegs IP Core

1.4.1 Create Vivado project with Zynq and MPSoc

First, Vivado doesn't have the Ultra96-v2 board by default, so we have to import the board. First load the board from Github via :

```
https://github.com/Avnet/bdf
```

Listing 7: Clone avnet boatd repository

And execute this command to import the board to vivado:

```
set_param board.repoPaths PATH_TO_bdfRepository
```

Listing 8: Set path of Ultra96 board repository to vivado

Second, import our IP core (the MysteryRegs) to vivado by going to: And execute this command to import the board to vivado:

```
tools -> setting -> add path to our MysteryRegs repository
```

Listing 9: Add path of Mysteryreg IP core to vivado

Now we are ready to create vivado project and combine the necessary IP core blocks. The complete vivado block design is shown below:

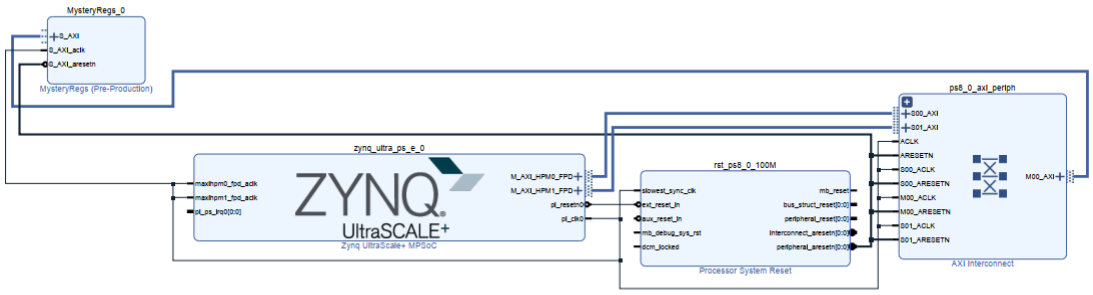

Figure 1: Task 1 vivado block design overview

1.4.2 Generate bit stream for Ultrascale 96 board

At this step, we create HDL Wrapper and simply generate the bitstream for our vivado design.

1.5 Use mmap for MysteryRegs

1.5.1 Load bitstream to board

The bitstream is generated and located on the server (ESA Lab). We now have to copy it to the board by using scp command. So, executing this command on the board's termial:

```
scp nt92homu@130.83.161.131:/path/to/bitstream /home/alarm
```

Listing 10: Copy bitstream from server to board

Now, we can load the bitstream onto board by executing this command:

```
echo 0 > /sys/class/fpga_manager/fpga0/flags
mkdir /lib/firmware
copy path/to/bitstream /lib/firmware/
echo name_of_bitstream >/sys/class/fpga_manager/fpga0/firmware
```

Listing 11: Load Bitstream

We can check it if the bitstream was successfully loaded by executing:

```
dmseg
```

Listing 12: Load Bitstream

And the result is listed below:

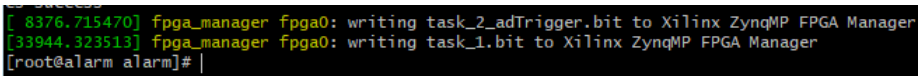

Figure 2: FPGA manager message

1.5.2 Write and Read Registers with mmap

At this step, we can use mmap to write to and read from 4 registers in the MysteryRegs IP core. The code written in C (task1.c) can be found in the Gitlab repository.

1.5.3 Guessing patterns of mystery registers

Compile the task1.c with gcc, we get the object file **task1**:

```
gcc -o task1 task1.c
```

Listing 13: Compile task1.c

Executing the compiled file **task1**, we got the pattern result:

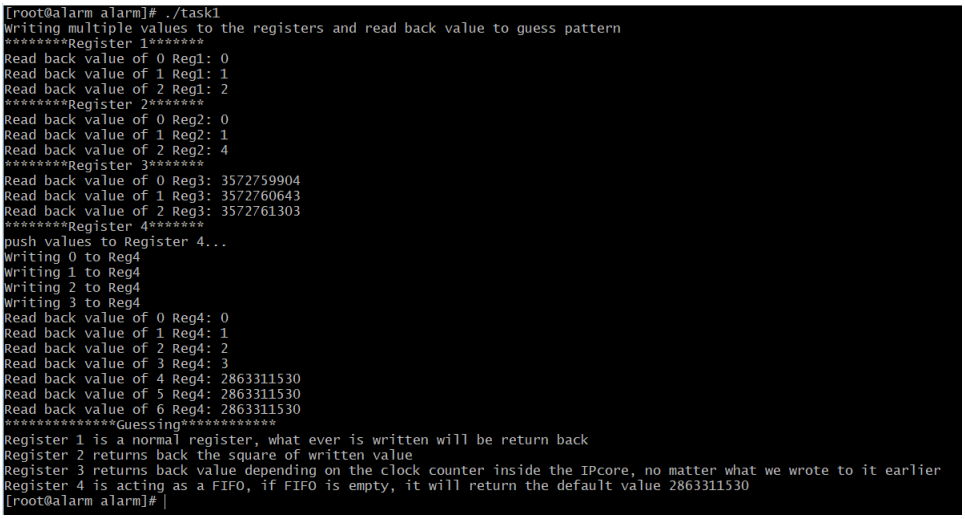

Figure 3: Guessing patterns of 4 registers

1.6 Summary

In summary:
Register 1 is a normal register, what ever is written will be return back.
Register 2 returns back the square of written value.
Register 3 returns back value depending on the clock counter inside the IPcore, no matter what we wrote to it earlier.
Register 4 is acting as a FIFO, if FIFO is empty, it will return the default value 2863311530.

2 Task 2

2.1 Setup Bluespec Path and Library Path

To write the AXI Multiplier Module, we need another 2 libraries, namely BlueAXI and BlueLib

```
git clone https://github.com/esa-tu-darmstadt/BlueAXI.git
git clone https://github.com/esa-tu-darmstadt/BlueLib.git
```

Listing 14: Clone BlueAXI and Bluelib liraries

To be able to run Bluespec and Vivado, we have to add its path directory to the environment path. Moreover, the BlueAXI and BlueLib libraries must also be included onto the bluespec path. A bash script would be suitable for this purpose:

```
source /opt/cad/xilinx/vivado/SDK/2019.1/settings64.sh
export PATH=/opt/cad/bluespec/latest/bin:$PATH
export BLUESPECDIR=/opt/cad/bluespec/latest/lib
export BLUESPEC_LICENSE_FILE=/opt/cad/keys/bluespec
export BSC_OPTIONS="-p :/home/stud/nt92homu/echo2/BlueAXI/src:/home/stud/nt92homu/echo2/BlueLib/src:+"
export XILINXD_LICENSE_FILE="/opt/cad/keys/xilinx"
```

Listing 15: bash script for adding BlueSpec and Vivado Path

To execute bash script, we can simply run:

```
source /path/to/bashscript
```

Listing 16: execute bash script

2.2 Write the AXI Slave Multiplier in Bluespec

The code Mul_AXI.bsv is available on gitlab. It basically create mkAXI4_Lite_Slave_Wr() and mkAXI4_Lite_Slave_Rd() object for AXI communication. The functions object.request.get() and object.response.put() are the main function we have to call to handle the communication. These 2 functions will return the AXI4_Lite_Read_Rq_Pkg, AXI4_Lite_Read_Rq_Pkg, AXI4_Lite_write_Rq_Pkg, AXI4_Lite_Write_Rs_Pkg,through which we can extract **addr** field and **data** field inside it.

2.3 Write testbench for AXI Slave Multiplier in Bluespec

There are two files for testing which are TestsMainTest.bsv and Testbench.bsv. They are also availible on gitlab repository.

TestsMainTest.bsv will receive the first operand, second operand and the expected result. Then it computes the result and compare it with the expected result. All these process is realized by a Finite State Machine(FSM):

```
module [Module] mkTestsMainTest#(Bit#(64) opA, Bit#(64) opB, Bit#(64) expectedResult)(TestHandler);
.
.
action
.
.
endaction
action
let r2 <- axi4_lite_read_response(rd);
    if (r2 == expectedResult) begin
      $display("TEST PASS %d mul %d : %d",opA, opB, r2);
    end else begin
      $display("TEST FAIL: %d != %d", r2, expectedResult);
    en
endaction

action
.
.
endaction
.
.
```

Listing 17: Implement test as FSM

Testbench.bsv is very simple module . It creates multiple data for testing and call the TestsMainTest.bsv.

```
module [Module] mkTestbench();
        Vector#(TestAmount, TestHandler) testVec;
 testVec[0] <- mkTestsMainTest(11,11,121);
 Stmt s = {
 .call sequential tests
 .
 .
 }
 mkAutoFSM(s);
```

Listing 18: Generate data for testing and call tests

2.4 Use BSVtools to package IP core AXI Slave Multiplier

Usually, we can compile and test the module by simply executing:

```
bsc -u -sim -g mkMul\_AXI.bsv Mul\_AXI.bsv.bsv
bsc -sim -o out -e testbench.bsv
```

Listing 19: Generate data for testing and call tests

But the BSVtools is already developed. It has a built in Makefile and it allows us to quickly build the project and package the IP core. We simply clone the BSVtool repository first:

```
git clone https://github.com/esa-tu-darmstadt/BSVTools.git
```

Listing 20: Clone BSVtools repository

And then execute:

```
path/to/BSVTools/bsvNew.py PROJECT_NAME
make SIM_TYPE=VERILOG ip
```

Listing 21: Build the project and package the IP core with BSVtools

2.5 Generate Bitstream

After building the IP core with BSVtools . We now can import the IP core to vivado as same as in task1.

```
tools -> setting -> add path to our AXI Multiplier repository
```

Listing 22: Add path of Mysteryreg IP core to vivado

Now we can create HDL Wrapper and generate bitstream.

2.6 Load bitstream to board and test with mmap

Copy the bitstream from server to the board and load it to the board as same as in task 1.

2.7 Write and Read Registers with mmap

The task2.c uses mmap to write two values to two registers and read back the value of the third register:

```
[root@alarm alarm]# gcc -o task2 task2.c
[root@alarm alarm]# echo task_2_adTrigger.bit > /sys/class/fpga_manager/fpga0/firmware
[root@alarm alarm]# ./task2
Read back value Reg3 ( 13 product 11): 143
[root@alarm alarm]#
```

Figure 4: Use mmap to write 13 and 11 to first and second register and read back the value 143 at the third register

2.8 Debug with ILA debug IP core

We can now get the ILA system IP core to debug our AXI.

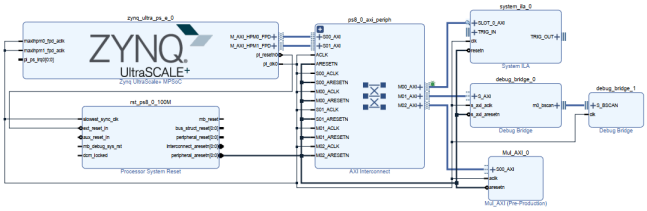

Figure 5: Debug AXI with ILA

Build the bitstream again to the board and run the xvcserver on the board to transmit the data over TCP/IP to the lab PC of ESA. On the lab PC of ESA, config the trigger and capture mode as follow:

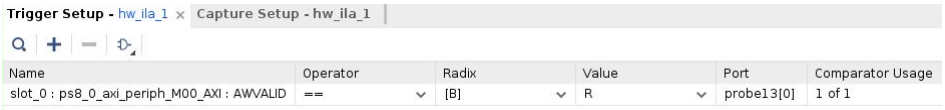

Figure 6: Trigger config

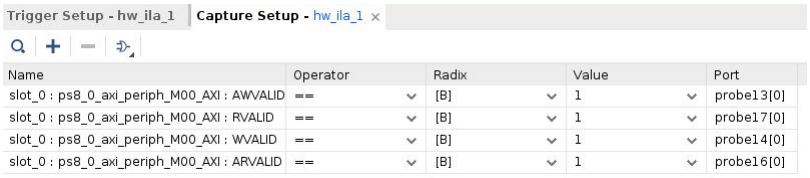

Figure 7: Capture config

Finally, we have to specify the .ltx file, which is the probe file.

```
Go to "open synthesized design" tab in vivado and execute the tcl command:
write_debug_probes filename.ltx
```

Listing 23: Specify the .ltx probe file

Now run the trigger and it will just wait for our AXI communication signal. Now at the same time we run the mmap code in the last section (task2.c) to generate AXI communication signal. Vivado can capture these signal from the board sending to ESA lab:

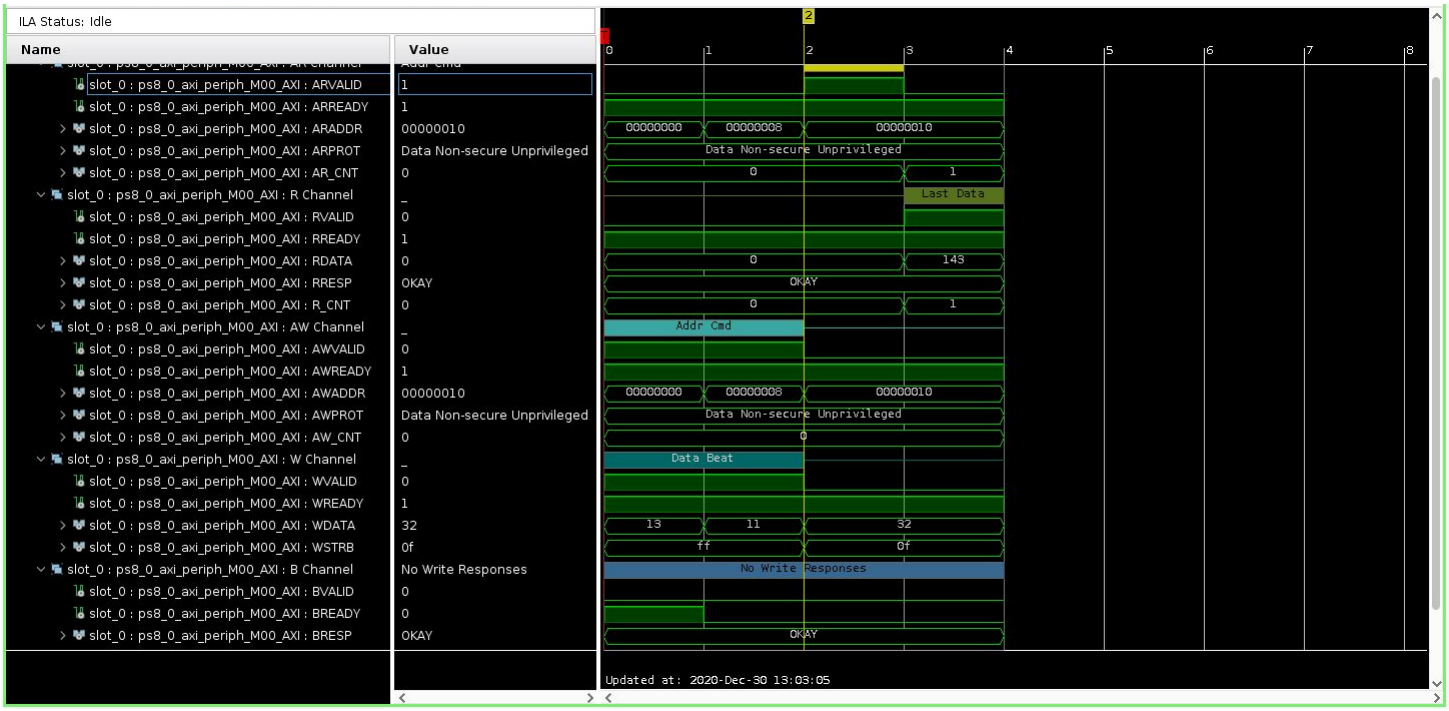

Figure 8: Multiply 13 and 11 and read back the value 143 at the third register. The first 2 channels for reading, the later 3 channels for writing

3 Task 3

4 Task 4

4.1 Cimg for User-Space

4.2 Driver for Kernel-Space

4.3 Bluespec for RGB to Grayscale in Ultra96

4.4 Put together

4.5 Debug

5 Task 5

5.1 Pipeline coding rules and hardware implementation

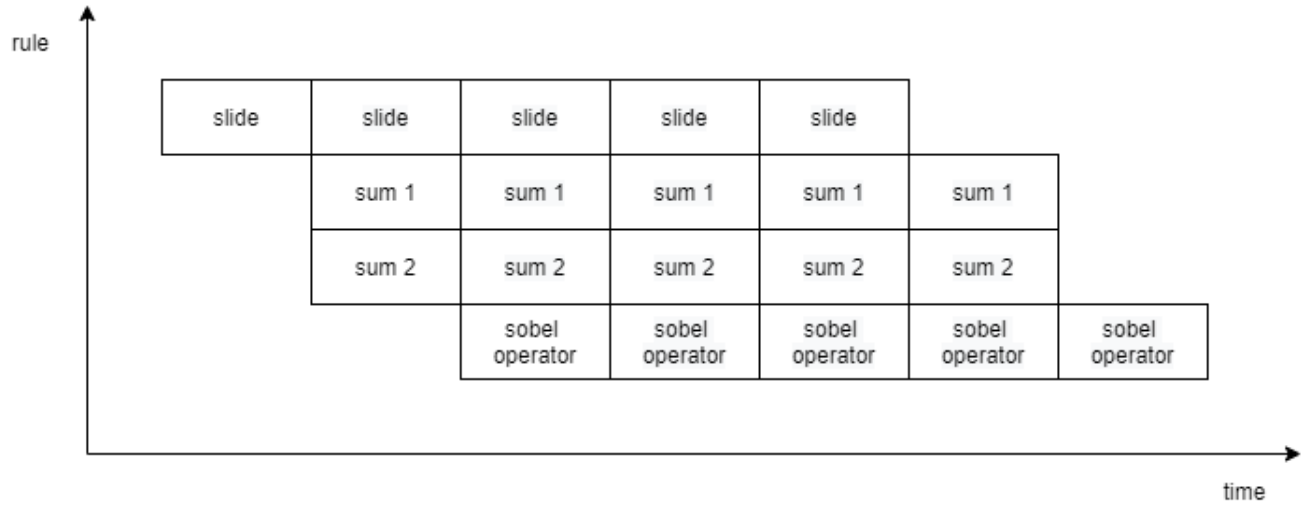

Figure 9: Coding Sobel Filter as pipeline structure

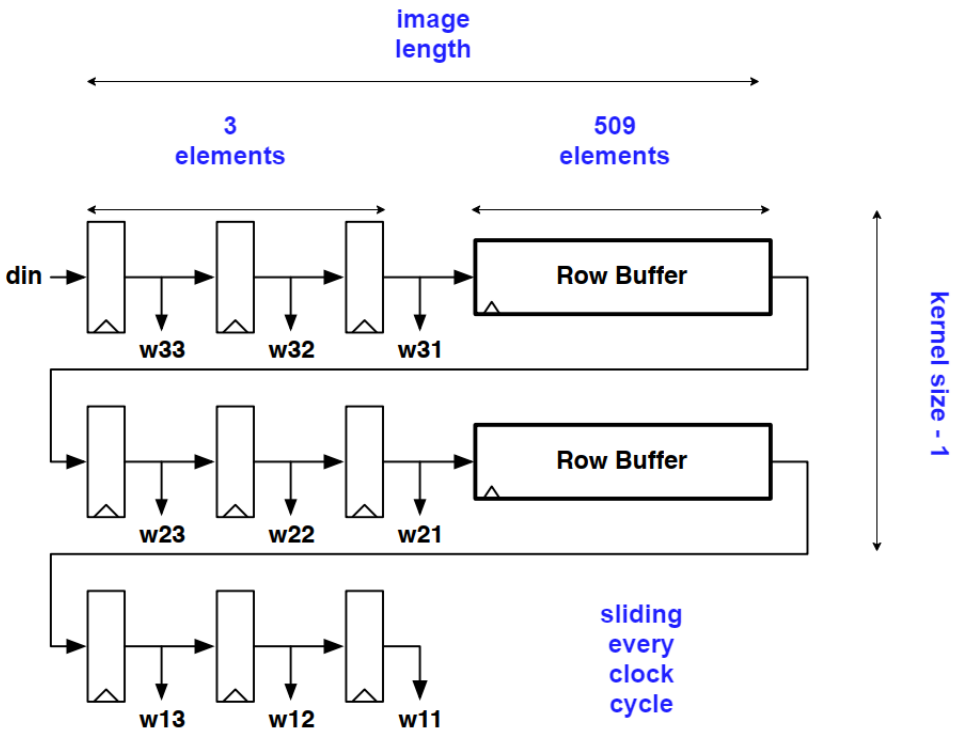

Figure 10: Hardware implementation as a sliding window

The hardware implentation is organized as in the figure 10 with 9 registers and 2 row buffers. In general the total number of registers and row buffer used depends on kernel size. If we have a kernel size ***n***, then we will have n^2 registers and ***n-1*** row buffers. I also used kernel 7x7, therefore I have 49 registered and 6 buffer rows. These 49 pixel values will be multiplied with 49 constant coefficients

[1] and sum up together to calculate sobel operator [2]. In fact, these 49 values will be changed by sliding the window 1 unit for every cycle.

The code to implement this hardware, which is a sliding window, is structured as in the figure 9. The most important rule is the sobel operator. It outputs 1 pixel every 1 cycle. It basically consists of 3 steps:

Step 1: Sliding window in 10 1 unit to the right.
Step 2: Two sums

$$sum_1 = \sum_{n=1}^{49} coefficients_n \cdot pixel_n$$

$$sum_2 = \sum_{n=1}^{49} coefficients_n \cdot pixel_n$$

Step 3: a norm of these two sums:

$$sum = abs(sum_1) + abs(sum_2)$$

Step 4: Normalization the ouput range to (0,255) and go to step 1 again.

5.2 Negative slack, split rule

In last section, we know that there are 4 basic steps in sobel operator. The naive ideas is to squeeze and **serialize** all of 4 steps above into 1 single rule in bluespec. Unfortunately, this can not work and will result **negative slack** after running implementation in vivado. The reason is because of the long critical path when we **serialize** them in 1 rule. If we want to remove the negative slack without having to lower the frequency below 100Mhz, we need to split the whole long rule intos multiple rules. In my case, I split to 4 rules:

- Rule "slide" will handle step 1.
- Rule "sum 1" and "sum 2" will handle step 2.
- Rule "sobel operator" will handle step 3 and step 4.

5.3 Serial and parallel rules

Rule of thumb: Independent rules are in parallel and dependent rules are in serial, since they wait result of each other before proceeding further. And finally, serial rules can also be **parallelize** under **phase difference**. This is a pipeline architecture as I implemented in the figure 9

Rule "sum 1" and "sum2" are independent, so they can run in parallel. Rule "sobel operator" can also be parallelized with rule "sum 1" and "sum 2" under the **phase difference** condition, because sobel operator needs to wait for the result of the "sum 1" and "sum 2" before calculating norm of these. Therefore, it delays 1 cycle compared to "sum 1" and "sum 2". But parallelism here means, in the mean time, it can calculate the norm of the previous "sum 1" and "sum 2". In summary, this actually is a pipeline architecture.

5.4 Character driver and its configuration registers

insert one configuration register There are 3 important functions while writing driver: No need to use **mmap** to transfer data to IP core. Instead, I wrote a character driver. All data needs to be communicated between user-space and IP core are implemented as configuration registers:

- gray image address.
- sobel image address.
- start signal.
- finish signal.
- image width.
- image length.
- kernel size.

- kmalloc().
- dma_map_single().
- ioremap().

5.5 Summary and Result

With this architecture, we can outpt 1 sobel pixel for every 1 cycle.

image size	kernel size	computation time(s)	fps	bandwidth(MB/s)
2048 x 1152	7	0.0683934s	14.62	32.895
	5	0.0687305s	14.55	32.7375
	3	0.0684598s	14.61	32.8725
512 x 512	7	0.0417292s	23.96	5.99
	5	0.0418141s	23.92	5.98
	3	0.0418034s	23.92	5.98

Figure 11: computational time measuring

References

[1] Sobel coefficients, 2020. https://stackoverflow.com/questions/31728948/what-are-the-kernel-coefficients-for-opencvs-sobel-filter-for-sizes-larger-than.

[2] Sobel coefficients, 2020. https://en.wikipedia.org/wiki/Sobel$_operator$.