EAT Assignment 3: Clash convolution

Revision	Date	Description
1.0	2025-04-22	Initial version this year

Table 1: Changelog

Contents

1 Introduction

The goal of this assignment is to design a convolution accelerator based on AXI4-Streaming for an FPGA using Clash and verify it in simulation, using already provided testbench/test cases.

Next to the design work a report must be delivered covering explanations of your simulations and containing motivated answers to the questions. Try to keep your answers as short as possible. Deliver as **a group on canvas**:

- Report (pdf) that contains the images, and code snippets of the various implemented functions, make sure you put your group number and student numbers and names on the frontpage.
- `ConvStudent.hs` (the only Clash file you have edited, if you have edited more files, deliver them as well).

2 Accelerating Sobel edge detection

This chapter goes over the main ideas behind the assignment and the design of the accelerator.

2.1 Sobel edge detection

Sobel kernels can be used in image processing to detect edges [1]. A grayscaled input image is convolved twice, first with the Sobel-X kernel K_x and after with the Sobel-Y kernel K_y. These kernels are designed to respond maximally to the edges running vertically and horizontally relative to the pixel grid.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Figure 1: Sobel X Kernel

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 2: Sobel Y Kernel

After the gradient magnitude is given by:

$$|K| = \sqrt{K_x^2 + K_y^2}$$

The combined result on a test image is visualized in Figure 3.

Figure 3: Edge detection on Rocky.

Computing the convolved feature matrix C involves many multiplications, linearly dependent on the number of horizontal and vertical pixels, and quadratically on the kernel size. The test image I is of size 762×1354 and the Sobel kernel K of 3×3. A stride S of 1 is used. This gives a convolved feature matrix with dimensions: $C_r = \frac{I_r - K_r}{S} + 1 \times C_c = \frac{I_c - K_c}{S} + 1 \rightarrow 760 \times 1352$. Each convolved feature requires $K_r \times K_c$ multiplications, thus $C_r \times C_c \times K_r \times K_c \rightarrow 9247680$ multiplications to apply a single Sobel kernel (additions are ignored due to their small impact compared to more expensive operations such as multiplication). Multiplications often require multiple instructions on a processor. An accelerator is introduced to speed up the calculations

The goal of this assignment is to design such an accelerator for an FPGA and verify it in simulation, using already provided testbench/test cases.

2.2 System design

The accelerator can be part of a much larger system of many blocks. In order to design a component in a large system you need to understand what goes in/out of your block using what protocol. To give you some idea behind the other components involved in such a setup, the basics are described for the popular Xilinx/AMD Zynq chip. Zynq devices containing both a hard core ARM processor and FPGA are showcased in Figure 4. The Zynq Processing System (PS) represents the ARM core, while the accelerator is implemented in the Programmable Logic (PL) FPGA. The DDR4 main memory contains the large image, the image should not be stored on the FPGA as it is a waste of expensive reprogrammable hardware. High-performance busses are available to exchange data between PL and PS, using *Direct Memory Access* (DMA) controllers. The ARM processor is used to control the DMA. The DMA controller is using the AXI4-Streaming protocol to offload data to and from the accelerator, as it also needs to write back the convolved features to main memory. This means that you need to design the accelerator in such a way that it supports AXI4-Streaming, so that the rest of the system can be taken for granted.

2.3 AXI4-Streaming

Note that there are multiple protocols AXI4-Streaming, AXI4-Full and AXI4-Lite. The latter two are control/status protocols, while AXI4-Streaming is for data streaming. AXI4-Streaming contains a master (transmitter) and slave (receiver). Only a few signals in AXI4-Streaming are mandatory to implement e.g. `tdata`, `tvalid`, `tready`, most others are design-specific. For the accelerator, the required signals are specified in Table 2 and Figure 5 is for visual understanding. The widths are

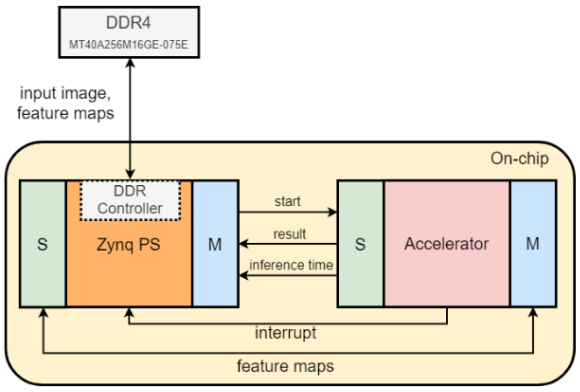

Figure 7. CloudScout characterization system.

Figure 4: Common accelerator setup [2].

as follows: `tdata` is of Signed 32 and tkeep of Unsigned 4. Although the data is a single word and does not need a keep, the DMA sees the data as 4 bytes, thus requires 4 keep bits.

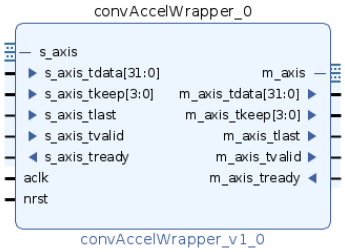

Figure 5: Accelerator IP.

Signal	Source	Description
ACLK	Clock	Global AXI4-Stream Clock
ARESETn	Reset	Global reset is active-LOW
TDATA[(8n-1):0]	Master	Data of the packet
TLAST	Master	Last packet of burst flag
TKEEP[(n-1):0]	Master	Which bytes are part of packet
TVALID	Master	Master is driven a valid transfer
TREADY	Slave	Slave can accept transfer in current cycle

Table 2: Accelerator AXI4-Stream signals.

2.4 Loading techniques

As this is your first time using Clash we will only implement the computation part of the accelerator on the FPGA. The sorting is done beforehand in the provided test environment.

The dimensions of the provided image are 290x364, the kernel is 3x3.

2.4.1 No reuse

Lets assume a use case where no reuse is applied. The kernel and image are loaded using a column-wise traversal technique showcased in Figure 6. First, the kernel is loaded on the AXIS slave and stored in the local registers of the accelerator. After the first sub-image is loaded and also stored in local registers. Convolution is applied and the convolved feature is written to the AXIS master. The process continues for the other sub-images till all convolved features are written on the AXIS master. In case of Sobel another kernel needs to be inserted, thus the reset is triggered, than the process repeats, but for the Sobel-Y kernel instead of Sobel-X. The accelerator does not take care of combining the Sobel subresults, lets assume this is done by the ARM processor.

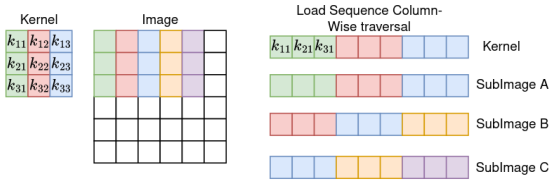

Figure 6: No reuse column-wise traversal load sequence.

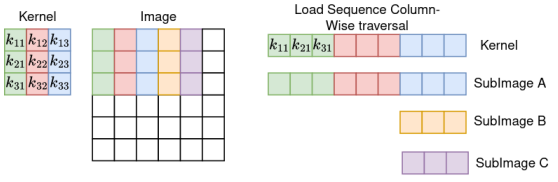

Figure 7: Reuse column-wise traversal load sequence.

Simulation waveforms for Sobel-Y are showcased in Figure 9 and Figure 8.

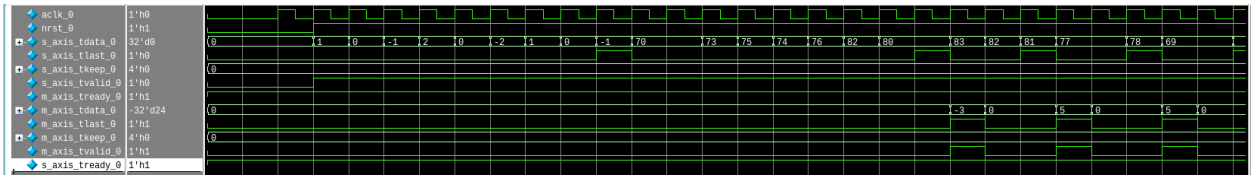

Figure 8: Simulation convolved unit Sobel-Y with reuse.

2.4.2 Reuse

In order to shorten the computation time of the image detection, the reuse optimization can be applied. Figure 7 showcases how six out of nine items can be reused in case the sub-image is not the first of the row.

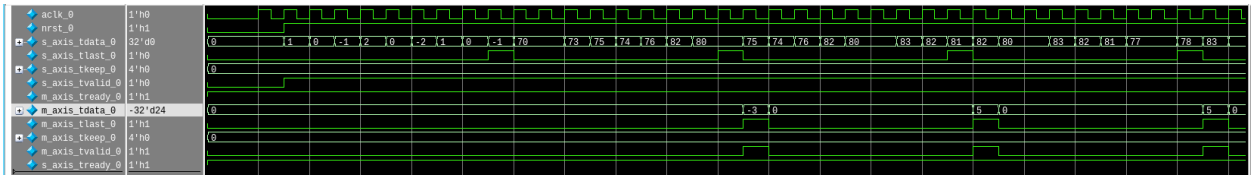

Figure 9: Simulation convolved unit Sobel-Y no reuse.

3 Clash questions [60 pts]

3.1 Provided files

We have provided the following files, make sure to place them into a directory called clash (already in zip file):

- `Axi.hs` contains the `Axi4Stream` data type and all `instances`.
- `Common.hs` includes global parameters such as the dimensions of the image, kernels and convolved features.
- `ConvTbData.hs` holds no data for the assignment, however it can be useful to play around with, before trying the large tests.
- `Rocky.hs` contains a test setup for simulating the entire system.
- `RockyTest*.hs` contains test vectors used by `Rocky.hs`
- `SobelStream.hs` contains the stream generator used in the testbench. Change the number of samples that are streamed to the accelerator in Common.hs.
- `SobelVerifier.hs` contains the stream verifier used in the testbench. Change the number of samples that are verified in Common.hs.
- `SobelTbData.hs` includes functions to access the image, kernels and expected convolved features as lists.
- `ConvStudent.hs` is the file used for your work.

3.2 Setup your environment

Before doing anything else make sure to change the path below to your workspace see `Common.hs`. Do not forget to add the "/" at the end of the folder path.

```
wsPath :: String
wsPath = "/some/path/to/your/project/directory/"
```

An approach to implement the accelerator is by doing only the computations on the FPGA. All sorting is done before streaming it to the accelerator. Figure 10 showcases a simplified dataflow graph for the logic within the accelerator. In this assignment we will create the unit for a 3x3 kernel and subimage using Signed 32 as data type.

A convenient way to design in clash is by developing the combinational logic first and then using a mealy machine to introduce registers. This step-wise approach allows to test each step separately.

The steps below can be used to get familiar with the design flow in clash and to solve the first exercise. Make sure that you can run your code in the interactive clashi environment. Go to the clash directory in your terminal, then start clashi by: `clashi ConvStudent.hs` Make sure that your interactive environment does not give you any errors, warnings are OK.

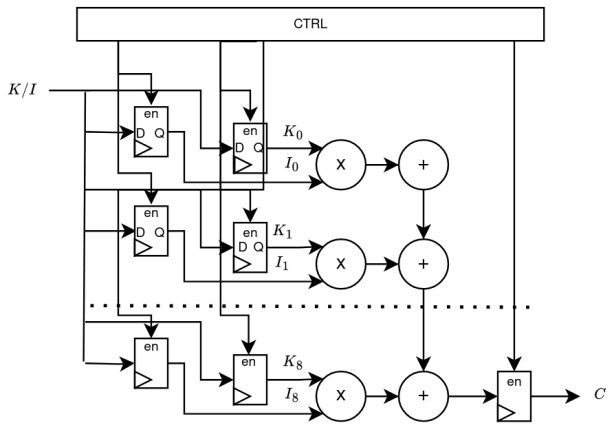

Figure 10: Data Flow Graph convolution unit.

3.3 Clash Q1: The convolution to AXIS [20 pts]

3.3.1 The *conv* function

Implement convolution function called `conv` which handles the combinational logic without any state or registers, thus basically a dot product operation. It has 2 `Vec a n` inputs (kernel and subimage) and outputs a `n` (convolved feature). Show your code of the convolution function in your report.

3.3.2 Clocked *conv* called *rConv*

Implement the `rConv` function that saves the output of `conv` function in a register. Note that it is necessary to lift the `conv` into the Signal domain. Show the implementation of the `rConv` function in your report.

3.3.3 Simulation of *rConv*

`rConv` currently has 2 inputs, while the internal clash simulation function `simulate` expects a single input. Implement the `rConvTb` which makes `rConv` compatible with `simulate`, and hence changes the 2 argument function into a function that takes 1 argument. Implement the `simRConvTb` function that simulates the `rConvTb` function with some inputs. Show the implementation of the `rConvTb` and `simRConvTb` function with the usage of the simulation function and its output in your report.

Tip: check the expected inputs for the simulate function with:

```
clashi> :t simulate
simulate :: (KnownDomain dom, NFDataX a, NFDataX b) =>
  (HiddenClockResetEnable dom => Signal dom a -> Signal dom b)
  -> [a] -> [b]
```

3.3.4 State machine to handle input streams

A state machine is required to load the kernel and image vector as the AXIS can only deliver a single item per clock cycle. The `State` data is already created for you see the snippet below. Later

on some extra states might need to be added, thus make sure to always forward "other" states to the default state. Implement `conv1D` which performs the correct step for a given state. The following states should be covered:

- `LOAD_KERNEL`, shift the incoming value in the kernel.
- `LOAD_SUBIMG`, shift the incoming value in the image.
- `CONV`, shift the incoming value in the image, perform the convolution.

The function `conv1D` should not change the state in this function, hence the state is given as in input. Changing the state is done at a higher level. The inputs are `State` containing the current state, `n` holding the new item from AXIS, `Vec a n` the current kernel and `Vec a n` the current sub-image. The outputs are `Vec a n` the new kernel and `Vec a n` the new sub-image. The given type signature already keeps in mind that this function might end up in a `mealy` machine, hence the type format: `s -> i -> (s, o)`. Show the implementation of the `conv1D` in your report.

```
data State = LOAD_KERNEL | LOAD_SUBIMG | CONV | CONV_NO_PIPE
deriving (Show, Generic, NFDataX, Eq)
```

3.3.5 Testing the *conv1D* function

Implement the `conv1D'` function to verify the functionality of the `conv1D` function. Create a state machine that starts with the state `LOAD_KERNEL`, wait till 8 clock cycles pass and than go to `LOAD_SUBIMG` state, wait again 8 cycles and move to `CONV` state. After the `CONV` state, go to the `LOAD_SUBIMG` again. It is important to note that during the `CONV` state, the system is still receiving pixels, so the `conv1D` function must be able to load a pixel during that state. The `conv1D'` function allows us to check whether `conv1D` works before implementation the AXIS state machine. If you make the `conv1D'` function adhere to the format `s -> i -> (s,o)`, then you can use the mealy machine to add registers for state in the next question. You can use the `simConv1DTbPrint'` function to test and visualize the input, state, next state, and output. Show the implementation of the `conv1D'` in your report.

3.3.6 Mealy machine and simulation

Implement the `mConv1D'` function that puts the `conv1D'` into a mealy machine (or registers). The `SNat a` determines the length of the kernel, in this case we can use `d9` as input, because our kernel is 3×3. Implement the `simMConv1DTb` function that test the `mConv1D'` function. So stream in the `rockyFirstNoReuseInps`, the first 9 values should end up in the kernel, from 9-18 should be the image, after which the output should be the convolved feature. Show the implementation of the `simMConv1DTb` function and its output of roughly 100 input samples in your report.

3.3.7 Making an AXIS version of the *conv1D* function

Implement the `axisConv1D` function that will use the `conv1D` function but it operates on the AXIS protocol. If you do not like to use the `conv1D` function, you may also specify the convolution behavior in this (`axisConv1D` function. Figure out how AXIS works and try to handle all edge cases well in the implementation. The appendix A gives a description on how to implement the AXIS protocol. Note that the `axisConv1D` function is both a slave and a master. It receives data as a slave over

an AXIS stream, but it also sends the convolved data over AXIS, so it behaves as a master in that case. The AXIS input of `tdata, tlast, tkeep` uses the `Axi4Stream a n` data type, notice that a `Maybe` is wrapped around it for the `tvalid` flag. You can use the `simAxisConv1DTbPrint` function to simulate your `axisConv1D` function. As simulation input the `mAxisConv1DTbNoReuseInp` is used. Inspect this simulation input in the `ConvTbData.hs` file. It contains a list of tuples that consist of the following:

- Maybe Axi4Stream from the Sobel generator (Master) that holds the data (kernel or image)
- A Bool that indicates whether the Sobel verifier (Slave) is ready to receive the convolved feature.

This `simAxisConv1DTbPrint` simulates the Sobel generator (Master) that provides a kernel and 3 images to the `axisConv1D` function. Most of the times the Sobel verifier (Slave) is ready to receive, as the Bool is most of the time True, but right after the master sends image1, when it is starting to send image2, the slave indicates that it is not ready to receive data. If this is the case, you can simply tell the Sobel generator (Master) that you are also not ready to receive data. Show the implementation of the `axisConv1D` function in your report.

3.3.8 The AXIS version in a mealy machine

Implement the `mAxisConv1D` function that uses a mealy machine on the `axisConv1D` function. The `mAxisConv1DTb` and `simMAxisConv1DTb` are already given. Run the `simMAxisConv1DTb` function and put the result in your report, this simulation function uses also the `mAxisConv1DTbNoReuseInp`, as explained in section 3.3.7, but it simulates the mealy machine with this input.

3.3.9 Testing using Rocky

Load `Rocky.hs` in clashi. In Appendix B.3 you can find more information on how to test using the Rocky image. Verify your design using the `tcRocky` function with `SIMPLE_NO_REUSE` sobel-combined on 30 convolved features in the report. Also run the `tcRockyFullImage` function, two test-cases will fail, but the first one will output a `rockyEdgesSimpleNoReuse.pgm` file in the img directory. Place this image in your report. You can use `convert.exe` (or on Linux `convert`) to convert this pgm into a png.

3.4 Clash Q2: Convolution over time [18 pts]

The accelerator can be optimized for area utilization. A way to do this is by reducing the number of multipliers and adders, folding it into a serial convolution unit as showcased in Figure 11.

This reduces the input registers, multipliers and adders.

3.4.1 Critical path

Highlight the critical path in the Figure 11 and compare it to the schematic from Figure 10. Discuss the changes in the report.

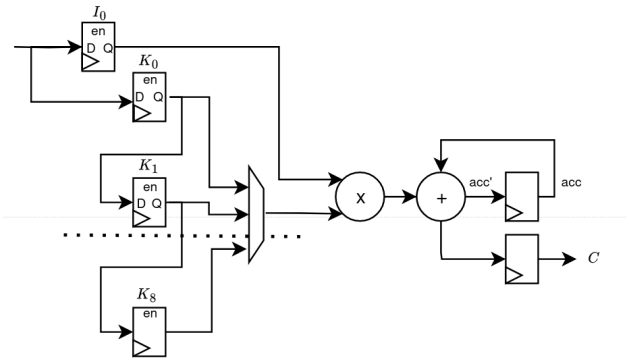

Figure 11: Data Flow Graph serial accelerator.

3.4.2 Implementation of *serConv1D*

Implement the `serConv1D` function that operates on a stream of input values based on the state. So if in the `LOAD_KERNEL` state, you update the kernel with the input. If in the `CONV` state, perform a partial convolution (mac) on the input and the corresponding value from the kernel. This means that there is no separate `LOAD_SUBIMG` state anymore, because the calculations for convolution are performed while the input values are streamed in. The accumulation must reset after computing every kernel. `serConv1D` does not modify the state, that happens at a higher level. Make sure that `serConv1D` outputs a `-1` when not outputing valid data. Show the `serConv1D` in your report.

3.4.3 Implementation of *serConv1D'*

Implement the `serConv1D'` function that operates the state variables for the `serConv1D` function. It must start in the `LOAD_KERNEL` state, and then after 8 clock cycles, go to the `CONV` state where the partial convolution is performed by the `serConv1D` function. You can use the `simSerConv1DTbPrint'` function to verify your behavior of the `serConv1D'` (you have to uncomment the helper functions in order to use `simSerConv1DTbPrint'`.) Show the `serConv1D'` in your report.

3.4.4 Testing the *serConv1D'* using *mSerConv1D'*

Implement the `mSerConv1D'` function and the `simMSerConv1DTb'` function. Show both functions in your report and show the output of the `simMSerConv1DTb'`

3.4.5 Implementation of *axisSerConv1D*

Implement the `axisSerConv1D` function that performs the convolution on an AXI-stream. You may use the `serConv1D` or perform the shifting and accumulation inside the `axisSerConv1D` function. Uncomment the `simSerAxisConv1DTbPrint` and its helper function if you want to test your `axisSerConv1D` and visualize the input, state, next state, and output. Show your implementation of `axisSerConv1D` in your report.

3.4.6 The AXIS version of *axisSerConv1D* in a mealy machine and simulation

Implement the `mAxisSerConv1D` function that uses a mealy machine on the `axisConv1D` function. Implement the `mAxisSerConv1DTb`, run the `simMAxisSerConv1DTb` and show its output in your report.

3.5 Clash Q3: RTL schematics [4 pts]

3.5.1 *convAccel* RTL schematic

The `convAccel` is a synthesis entity of the `mAxisConv1D` function. Synthesize your design, show the RTL schematic in your report. Shortly comment on the number of multipliers, input pins, and output pins.

3.5.2 *serConvAccel* RTL schematic

The `serConvAccel` is the synthesis entity of the `mAxisSerConv1D` function. Generate Verilog or VHDL and generate an RTL schematic. Show this schematic in your report. Shortly comment on the number of multipliers, input pins, and output pins. What are the differences between this `convAccel` design and `serConvAccel` design in terms of resources?

3.6 Clash Q4: Convolution with reusing sub images [18 pts]

Instead of optimizing for area we can optimize for time, by reusing part of the sub-image as showcased in Figure 7.

3.6.1 Reusing subimage

Figure out a way to optimize for time by reusing 6 out of 9 sub-image items in case it is not the first sub-image of a row. It is up to you how many multipliers and adders you are going to use. Implement the following functions:

- `conv1DReuse` that manages and performs a convolution on a stream of numbers, but with data reuse.
- `mConv1DReuse` that puts the `conv1DReuse` in a mealy machine.
- `axisConv1DReuse` that performs a convolution on a AXIS.
- `mAxisConv1DReuse` that puts the `axisConv1DReuse` in a mealy machine.
- `mAxisConv1DReuseTb` that makes the `mAxisConv1DReuse` compatible with the simulate function

You can use the simulation functions provided in the comments so simulate and debug your code. Deliver the following in your report:

- A description of your idea on how to optimize your architecture such that it reuses the sub-images
- The code of `conv1DReuse`

- The output of the `simMConv1DReuse` function
- The code of `axisConv1DReuse`
- The output of the `simMAxisConv1DReuseTb` function

3.6.2 RTL schematics

Generate an RTL schematic, comment on the number of multipliers, registers, input and output pins.

3.6.3 Testbench Rocky

Verify your work using the `rockyTc` apply the full image with `tcRockyFullImage`. Include the picture produced using `tcRockyFullImage` in your report.

References

[1] R. Fisher, "Feature Detectors - Sobel Edge Detector," 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

[2] E. Rapuano, G. Meoni, T. Pacini, G. Dinelli, G. Furano, G. Giuffrida, and L. Fanucci, "An FPGA-Based Hardware Accelerator for CNNs Inference on Board Satellites: Benchmarking with Myriad 2-Based Solution for the CloudScout Case Study," *Remote Sensing*, vol. 13, no. 8, p. 1518, Jan. 2021, number: 8 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2072-4292/13/8/1518

[3] Xilinx/AMD, "AXI4-Stream Interface • Soft-Decision FEC Integrated Block LogiCORE IP Product Guide (PG256) • Reader • AMD Adaptive Computing Documentation Portal," Nov. 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/pg256-sdfec-integrated-block/AXI4-Stream-Interface

A AXI4-Streaming Protocol Description

AXI4-Streaming protocol makes use of a slave and master channel. The local slave channel (convolution accelerator) receives data from an external master (stream generator) while the local master (convolution accelerator) channel transmits data to an external slave (stream verifier). See Figure 12 for an overview of the relevant signals for the convolution assignment, the clock and reset are left out for simplicity.

The `clock`, `reset`, `tdata`, `tvalid` and `tready` signals are always mandatory, while `tkeep` and `tlast` are added to be compatible with the AXI-DMA IP, which can be used in combination with our accelerator. The `tvalid` and `tready` are used for the handshake. The `tkeep` and `tlast` are control signals, `tdata` contains the actual content in the stream. It contains the kernel or subimage item on the slave and the convolved feature on the master.

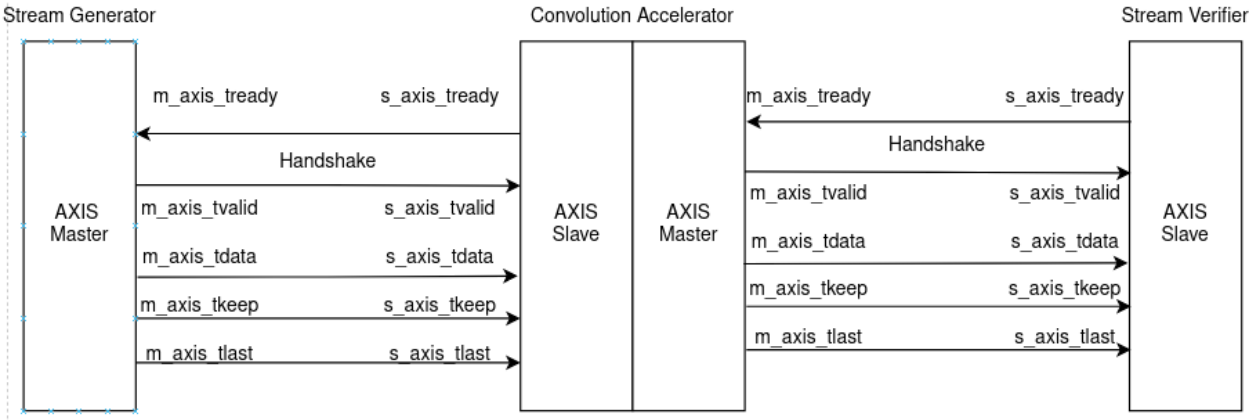

Figure 12: Test setup convolution accelerator.

A.1 Slave receive

The handshake determines when information is passed across the interface. In order for a transfer to occur, both `tready` and `tvalid` must be asserted. Either `tvalid` or `tready` can be asserted first or both can be asserted in the same clock cycle. The local slave can request data by putting its `tready` high. The external master will notice this and put the `tvalid` high, this is a *tready before tvalid handshake*. It is also possible that the master already has the `tvalid` asserted, this is called a *tvalid before tready handshake*. If this is the case it cannot deassert it.

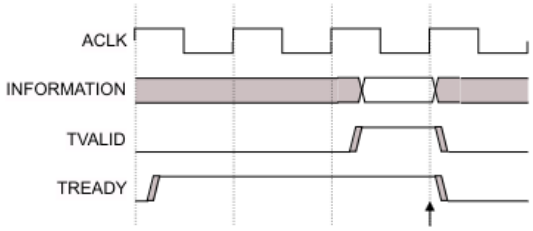

Figure 2-2 TREADY before TVALID handshake

Figure 13: *tready before tvalid hanshake* [3]

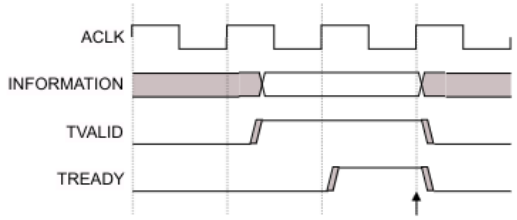

Figure 2-1 TVALID before TREADY handshake

Figure 14: *tvalid before tready handshake* [3]

To simplify the code we can define a local signal which is high in case the `tvalid` and `tready` are high.

```
vld =
  case (s_axis, s_axis_tready) of
    (Just _, True) -> True -- Incoming packet is valid & local slave is ready
    _              -> False
```

The local slave can now read out `tdata`, `tkeep` and `tlast`. In case the data is valid (`Maybe` is used as valid flag), each field can be accessed as shown below, notice that in case it is not valid default values must be inserted.

```
(s_axis_tdata, s_axis_tlast, s_axis_tkeep) =
      case s_axis of
        (Just x) -> (tData x, tLast x, tKeep x) -- extract data from axi record
        _        -> (0, False, 0) -- default data
```

The `tready` should be low in case the external slave cannot receive any data or if the local buffer is full. This only happens if we are in the `CONV` state and the kernel and image registers are full. Otherwise, the incoming data from the stream generator, will not be retransmitted next clock cycle and it will be lost forever, as it is not stored in the local kernel or subimage buffer during the `CONV` state. As we need to keep the current buffer, because the state is still `CONV` next clock cycle.

```
s_axis_tready = not (state == CONV && not m_axis_tready)
```

A.2 Master transmit

The local master should assert the `tvalid` in the `CONV` state as the convolved feature will be available on tdata. It cannot leave the `CONV` state until the tready from the external slave is asserted, which means that it has completed the handshake and holds our transmitted convolved feature.

```
m_axis_tvalid
  | state == CONV = True -- Convolutional feature available
  | otherwise = False -- Convolutional feature not available
```

The `tkeep` and `tlast` signals can be forwarded from the local slave to the local master without any changes, as our block does not support any control logic for these signals. Do not hard code them to zero as another IP in the stream could be using them and would thus discard all the data or introduce a deadlock.

```
m_axis
  | m_axis_tvalid =
    Just
      Axi4Stream
        {tData = cf, tLast = s_axis_tlast, tKeep = s_axis_tkeep}
  | otherwise = Nothing
```

B Clash

This appendix covers important beginner tips on how to use clash. Notice that the directory where the command is executed is notated within the brackets on the left side, e.g. from the workspace directory in subdirectory clash is denoted as

```
[~]$ cd clash
[~/clash]$
```

B.1 Navigating through Haskell and Clash documentation

Clash can be seen as a subset of Haskell. In a program the design is done with Clash specific functions, while for testing it is partly Haskell partly Clash. The Clash documentation only covers Clash specific functions.

- Haskell docs: `https://hoogle.haskell.org/`.
- Clash docs: `https://hackage.haskell.org/package/clash-prelude-1.8.1/docs/Clash-Prelude.html`

B.2 Manage overlapping Clash and Haskell library functions

Haskell uses lists and bounded/unbounded vectors, while Clash has only bounded vectors. Various libraries contain functions to operate on them and they often have functions with similar names. Thus you might want to call the `map` function from the Haskell Data.List library, but you end up with `map` from the Clash.Prelude library, or with an error in case it is unclear which one is called. To deal with this the libraries can be imported with a prefix, this way we can either call `CP.map` or `L.map`.

```
import Data.List as L (map)
import Clash.Prelude as CP
```

Secondly, note that from Data.List only map is imported and not all other functions.

```
clashi> :t CP.map
CP.map :: (a -> b) -> Vec n a -> Vec n b
clashi> :t L.map
L.map :: (a -> b) -> [a] -> [b]
```

B.3 Test case Rocky

A test case function is already designed for you it is called *tcRocky()*. Notice that the test case runs within the interactive environment. It covers a test for both non-reuse and reuse. *tcRocky()* can be launched with various arguments:

1. *TestState*: different test data is generated in each state.

```
-- BRAM loads row-wise, others load col-wise
data TestState = SIMPLE_NO_REUSE | SIMPLE_REUSE | PARALLEL_NO_REUSE | PARALLEL_REUSE | BRA
```

2. *subset*: Number of convolved features
3. *saveImage*: if True, than a .pgm image of the convolved features is created, which should match Figure 3.

E.g. no reuse, a subset of 100 convolved features and no save of the image:

```
clashi> tcRocky SIMPLE_NO_REUSE 100 False "../img/"
```

The expected output:

```
clashi> tcRocky SIMPLE_NO_REUSE 100 False "../img/"
Convolved Features: 100
Sobel X Output:
[-39,-29,13,50,46,15,0,0,1,1,0,8,16,15,-5,-28,-20,-5,-7,-7,-24,-59,-64,-15,29,30,29,27,12,-8,
-10,-2,-10,-26,-24,-2,14,11,-12,-11,11,14,23,28,10,-12,-43,-61,-55,-29,11,28,27,19,-4,-9,-11,
5,-5,-13]

Sobel Y Output:
[-3,5,5,-2,-8,-9,-4,4,13,21,22,16,8,1,3,14,22,25,23,13,-2,-15,-24,-29,-29,-28,-27,-25,-24,-24,
-18,-2,16,28,30,24,18,19,22,19,15,12,9,6,0,-8,-13,-9,5,21,31,34,31,25,26,35,41,37,25,13]

Sobel X Match: (True,0,Nothing)
Sobel Y Match: (True,0,Nothing)
Normalized Sobel Results:
[17,12,6,22,20,7,1,1,5,9,9,7,7,6,2,13,13,11,10,6,10,26,30,14,18,18,17,16,11,11,9,1,8,16,16,10,
10,9,11,9,8,8,10,12,4,6,19,27,24,15,14,19,18,13,11,15,18,16,11,8]

Normalized Sobel Expected Results:
[17,12,6,22,20,7,1,1,5,9,9,7,7,6,2,13,13,11,10,6,10,26,30,14,18,18,17,16,11,11,9,1,8,16,16,10,
10,9,11,9,8,8,10,12,4,6,19,27,24,15,14,19,18,13,11,15,18,16,11,8]

Normalized Combined Sobel Match: (True,0,Nothing)
Done!
```

In case there was an error it returns the clock cycle and payload of the non-matching pair, e.g. below at clock cycle 19:

```
Sobel X Match: (False,19,Just ((Just (Axi4Stream {tData = -39, tLast = True, tKeep = 15}),True)
```

Your result is:

```
(Just (Axi4Stream {tData = -39, tLast = True, tKeep = 15}),True)
```

While the expected result is:

```
(Just (Axi4Stream {tData = -3, tLast = True, tKeep = 15}),True)
```

It is also possible to run it on the full image using the dimension variables from *ConvTbData.hs*:

```
clashi> tcRocky True (ConvTbData.convFeaturesRows * ConvTbData.convFeaturesCols) True
```

B.4 Testbench Rocky

Next to the test case we have a testbench which generates a VHDL testbench to simulate in Vivado Simulator. This allows for waveform simulation.

First test the testbench inside clashi. E.g. for reuse enabled and 30 convolved features we expect

Clock Cycles $= 1 + K_r K_c + K_r K_c + K_c(N-1) + 1 = 9 + 9 + 3 * 29 + 1 = 107$.

Left $+1$ is the reset cycle, K_r kernel rows, K_c kernel cols, N number of convolved features and the right $+1$ is the fill-up cycle.

Thus after 107 cycles the resulting output matches the expected output.

In order to get this subset edit *SobelStream.hs and sobelVerifier.hs*.

```
clashi> convAccelTbSample 107
```

Expected output

```
[False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,True]
```

B.5 AXI4-Stream data type

The Axi4Stream data type is already created for you in *Axi.hs*.

```
data Axi4Stream a n = Axi4Stream {
  tData  :: a,
  tLast  :: Bool,
  tKeep  :: n
} deriving (Generic, NFDataX, Read, Show, ShowX)
```

Many variants can be created e.g. a *Vec 4 (Signed 8)* which gives 4 separately addressable *Signed* bytes, or *Unsigned 32* for a single signed word, depending on the desired implementation:

```
axi4StreamTc0 :: Axi4Stream (Vec 4 (Signed 8)) (Vec 4 Bool)
axi4StreamTc0 = Axi4Stream {
  tData  = 1 :> -2 :> 3 :> 14 :> Nil,
  tLast  = False,
  tKeep  = True :> True :> True :> False :> Nil
}

axi4StreamTc1 :: Axi4Stream (Unsigned 32) Bool
axi4StreamTc1 = Axi4Stream {
  tData  = 31,
  tLast  = False,
  tKeep  = True
}
```

The data type is not written as a VHDL user would expect it to be, e.g. the blocks below showcase the clash we put in and the VHDL generated when this data type is used in the accelerator.

```
convAccel
  :: Clock ConvAccelSystem -- aclk
  -> Reset ConvAccelSystem -- nrst
  -> Signal ConvAccelSystem (Maybe (Axi4Stream (Signed 32) (Unsigned 4)), Bool)
  -> Signal ConvAccelSystem (Maybe (Axi4Stream (Signed 32) (Unsigned 4)), Bool)
```

```
entity convAccel is
  port(-- clock
       aclk          : in convAccel_types.clk_ConvAccelSystem;
       -- reset
       nrst          : in convAccel_types.rst_ConvAccelSystem;
       s_axis_tvalid : in std_logic_vector(0 downto 0);
       s_axis_tdata  : in signed(31 downto 0);
       s_axis_tlast  : in boolean;
       s_axis_tkeep  : in unsigned(3 downto 0);
       m_axis_tready : in boolean;
```

```
        m_axis_tvalid : out std_logic_vector(0 downto 0);
        m_axis_tdata  : out signed(31 downto 0);
        m_axis_tlast  : out boolean;
        m_axis_tkeep  : out unsigned(3 downto 0);
        s_axis_tready : out boolean);
end;
```

This has to do with the standard functions available in clash, which all use a Maybe for the valid flag. The Maybe can either be a *Just a* (valid) or *Nothing* (not valid).

Many functions can be applied to check the valid or extract its data, some examples:

```
clashi> L.map isJust [Just (Axi4Stream (0 :: Unsigned 32) False (False :: Bool)), Nothing]
[True,False]
clashi> catMaybes [Just (Axi4Stream (0 :: Unsigned 32) False (False :: Bool)), Nothing]
[Axi4Stream {tData = 0, tLast = False, tKeep = False}]
clashi> fromJust $ Just (Axi4Stream (0 :: Unsigned 32) False (False :: Bool))
Axi4Stream {tData = 0, tLast = False, tKeep = False}
```

E.g. the blockRamBlob, which is used to generate the input stream and verify the output stream:

```
clashi> :t blockRamBlob
blockRamBlob
  :: (HiddenClock dom, HiddenEnable dom, Enum addr, NFDataX addr) =>
     MemBlob n m
     -> Signal dom addr
     -> Signal dom (Maybe (addr, BitVector m))
     -> Signal dom (BitVector m)
```

In order to comply with this standard (which is needed to be able to use the functions) the valid is included via:

```
Maybe (Axi4Stream (Unsigned 32) (Unsigned 4))
```

The ready signal should also be declared externally, as the valid of the Axi4Stream 'packet' has nothing to do with the validness of the ready flag, as it comes from another source.

B.6 Annotations

The generated ports have by defaults long and very unreadable names, annotations can be added to top modules to get useful names e.g. the names from the example showcased before are obtained with:

```
{-# NOINLINE convAccel #-}
{-# ANN convAccel
  (Synthesize
    { t_name   = "convAccel"
```

```
    , t_inputs = [
      PortName "aclk",
      PortName "nrst",
      PortProduct ""
        [ PortProduct "s_axis"
              [ PortName "tvalid"
              , PortProduct ""
                   [ PortName "tdata"
                   , PortName "tlast"
                   , PortName "tkeep" ] ]
        , PortName "m_axis_tready"
        ]
    ]
    , t_output = PortProduct ""
        [ PortProduct "m_axis"
              [ PortName "tvalid"
              , PortProduct ""
                   [ PortName "tdata"
                   , PortName "tlast"
                   , PortName "tkeep" ] ]
        , PortName "s_axis_tready"
        ]
    }) #-}
convAccel
  :: Clock ConvAccelSystem -- aclk
  -> Reset ConvAccelSystem -- nrst
  -> Signal ConvAccelSystem (Maybe (Axi4Stream (Signed 32) (Unsigned 4)), Bool)
  -> Signal ConvAccelSystem (Maybe (Axi4Stream (Signed 32) (Unsigned 4)), Bool)
```

Notice that these annotations are not just to make it look nice, but also define if the function should be synthesized as a design or testbench.

B.7 Custom Clock Domain

A custom clock domain called *convAccelSystem* is created to allow for a negative edge reset, which is often used in AXIS. In clash examples the *@System* is often used to refer to the default clock domain. However now you must refer to the custom clock domain e.g. by calling *@ConvAccelSystem*:

```
convAccel
  :: Clock ConvAccelSystem -- aclk
  -> Reset ConvAccelSystem -- nrst
  -> Signal ConvAccelSystem (Maybe (Axi4Stream (Signed 32) (Unsigned 4)), Bool)
  -> Signal ConvAccelSystem (Maybe (Axi4Stream (Signed 32) (Unsigned 4)), Bool)
```