In [1]:

```
import onnx
import torch
```

In [2]:

```
build_dir = "/workspace/finn/notebooks/mnist_ex/verification"
```

In [3]:

```
import brevitas.onnx as bo
from finn.util.visualization import showInNetron
from finn.util.test import get_test_model_trained
from finn.transformation.streamline import Streamline
from finn.transformation.lower_convs_to_matmul import LowerConvsToMatMul
from finn.transformation.bipolar_to_xnor import ConvertBipolarMatMulToXnorPopcount
import finn.transformation.streamline.absorb as absorb
from finn.transformation.streamline.reorder import MakeMaxPoolNHWC, MoveScalarLinearPastInvariants, MoveFlattenPastAff
from finn.transformation.infer_data_layouts import InferDataLayouts
from finn.transformation.general import RemoveUnusedTensors

import finn.transformation.fpgadataflow.convert_to_hls_layers as to_hls
from finn.transformation.fpgadataflow.create_dataflow_partition import (
    CreateDataflowPartition,
)
from finn.transformation.move_reshape import RemoveCNVtoFCFlatten
from finn.custom_op.registry import getCustomOp
from finn.transformation.infer_data_layouts import InferDataLayouts
```

In [4]:

```
#import model .onnx into FINN
from finn.core.modelwrapper import ModelWrapper
ready_model_filename = "model_fmnist_notebook.onnx"
model_for_sim = ModelWrapper(ready_model_filename)
```

In [5]:

```
#member fuctions used to extract information about the structure and properties of the ONNX model
from finn.core.datatype import DataType

finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: inp.1
Output tensor name: 45
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 10]
Input tensor datatype: FLOAT32
Output tensor datatype: FLOAT32
List of node operator types in the graph:
['Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool',
'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul', 'Div', 'Add', 'Mul']
```

Note that the output tensor is (as of yet) marked as a float32 value, even though we know the output is binary. This will be automatically inferred by the compiler in the next step when we run the `InferDataTypes` transformation.

2. Network preparation: Tidy-up transformations

prepare our FINN-ONNX model. In particular, all the intermediate tensors need to have statically defined shapes.

Graph transformations in FINN. transform the model into a synthesizable hardware description.

In [6]:

```
from finn.transformation.general import GiveReadableTensorNames, GiveUniqueNodeNames, RemoveStaticGraphInputs
from finn.transformation.infer_shapes import InferShapes
from finn.transformation.infer_datatypes import InferDataTypes
from finn.transformation.fold_constants import FoldConstants

verif_model_filename = build_dir + "/model_fmnist_notebook_verified.onnx"

model_for_sim.set_tensor_datatype(model_for_sim.graph.input[0].name, DataType["UINT8"])
model_for_sim.save(verif_model_filename)

model_for_sim = model_for_sim.transform(InferShapes())
model_for_sim = model_for_sim.transform(FoldConstants())
model_for_sim = model_for_sim.transform(GiveUniqueNodeNames())
model_for_sim = model_for_sim.transform(GiveReadableTensorNames())
model_for_sim = model_for_sim.transform(RemoveStaticGraphInputs())
model_for_sim.save(build_dir + "/model_fmnist_notebook_tidy.onnx")
```

Let's view our ready-to-go model after the transformations. Note that all intermediate tensors now have their shapes specified (indicated by numbers next to the arrows going between layers). Additionally, the datatype inference step has propagated quantization annotations to the outputs of `MultiThreshold` layers (expand by clicking the + next to the name of the tensor to see the quantization annotation) and the final output tensor.

In [ ]:

```
showInNetron(build_dir + "/model_fmnist_notebook_tidy.onnx")
```

adding pre and post processing directly in the ONNX graph. In this case, the preprocessing step divides the input uint8 data by 255 so the inputs are bounded between [0, 1]. The postprocessing step takes the output of the network and returns the index (0-9) of the image category with the highest probability (top-1).

In [7]:

```
#PREPROCESSING
from finn.util.pytorch import ToTensor
from finn.transformation.merge_onnx_models import MergeONNXModels

model_for_sim = ModelWrapper(build_dir+"/model_fmnist_notebook_tidy.onnx")
global_inp_name = model_for_sim.graph.input[0].name
ishape = model_for_sim.get_tensor_shape(global_inp_name)
# preprocessing: torchvision's ToTensor divides uint8 inputs by 255
totensor_pyt = ToTensor()
chkpt_preproc_name = build_dir + "/model_fmnist_pre.onnx"
bo.export_finn_onnx(totensor_pyt, ishape, chkpt_preproc_name)
# join preprocessing and core model
pre_model = ModelWrapper(chkpt_preproc_name)
model_for_sim = model_for_sim.transform(MergeONNXModels(pre_model))
# add input quantization annotation: UINT8
global_inp_name = model_for_sim.graph.input[0].name
model_for_sim.set_tensor_datatype(global_inp_name, DataType["UINT8"])
```

```
/workspace/finn-base/src/finn/transformation/infer_data_layouts.py:119: UserWarning: Assuming 4D input i
s NCHW
  warnings.warn("Assuming 4D input is NCHW")
```

In [8]:

```
finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: global_in
Output tensor name: global_out
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 10]
Input tensor datatype: UINT8
Output tensor datatype: FLOAT32
List of node operator types in the graph:
['Div', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxP
ool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul', 'Div', 'Add', 'Mul']
```

In [9]:

```
#POSTPROCESSING
from finn.transformation.insert_topk import InsertTopK
from finn.transformation.infer_shapes import InferShapes

# postprocessing: insert Top-1 node at the end
model_for_sim = model_for_sim.transform(InsertTopK(k=1))
chkpt_name = build_dir+"/model_fmnist_pre_post.onnx"
# tidy-up again
model_for_sim = model_for_sim.transform(InferShapes())
model_for_sim = model_for_sim.transform(FoldConstants())
model_for_sim = model_for_sim.transform(GiveUniqueNodeNames())
model_for_sim = model_for_sim.transform(GiveReadableTensorNames())
model_for_sim = model_for_sim.transform(InferDataTypes())
model_for_sim = model_for_sim.transform(RemoveStaticGraphInputs())
model_for_sim.save(chkpt_name)
```

In [10]:

```
finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: global_in
Output tensor name: global_out
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 1]
Input tensor datatype: UINT8
Output tensor datatype: UINT32
List of node operator types in the graph:
['Div', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxP
ool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul', 'Div', 'Add', 'Mul', 'TopK']
```

In [15]:

```
showInNetron(build_dir+"/model_fmnist_pre_post.onnx")
```

```
Serving '/workspace/finn/notebooks/mnist_ex/verification/model_fmnist_pre_post.onnx' at http://0.0.0.0:8
081 (http://0.0.0.0:8081)
```

Out[15]:

In [11]:

```
model_for_sim = ModelWrapper(build_dir + "/model_fmnist_pre_post.onnx")
model_for_sim = model_for_sim.transform(MoveScalarLinearPastInvariants())
model_for_sim = model_for_sim.transform(Streamline())
model_for_sim = model_for_sim.transform(LowerConvsToMatMul())
model_for_sim = model_for_sim.transform(MakeMaxPoolNHWC())

model_for_sim = model_for_sim.transform(absorb.AbsorbTransposeIntoMultiThreshold())

model_for_sim = model_for_sim.transform(MakeMaxPoolNHWC())
model_for_sim = model_for_sim.transform(absorb.AbsorbTransposeIntoMultiThreshold())

model_for_sim = model_for_sim.transform(Streamline())
# absorb final add-mul nodes into TopK
model_for_sim = model_for_sim.transform(absorb.AbsorbScalarMulAddIntoTopK())
model_for_sim = model_for_sim.transform(InferDataLayouts())
model_for_sim = model_for_sim.transform(RemoveUnusedTensors())

model_for_sim.save(build_dir + "/model_fmnist_stream.onnx")
```

In [20]:

```
showInNetron(build_dir+"/model_fmnist_stream.onnx")
```

```
Stopping http://0.0.0.0:8081 (http://0.0.0.0:8081)
Serving '/workspace/finn/notebooks/mnist_ex/verification/model_fmnist_stream.onnx' at http://0.0.0.0:808
1 (http://0.0.0.0:8081)
```

Out[20]:

In [12]:

```
# HLS
model_for_sim = ModelWrapper(build_dir + "/model_fmnist_stream.onnx")

# choose the memory mode for the MVTU units, decoupled or const
mem_mode = "decoupled"


model_for_sim = model_for_sim.transform(to_hls.InferQuantizedStreamingFCLayer(mem_mode))
# TopK to LabelSelect
model_for_sim = model_for_sim.transform(to_hls.InferLabelSelectLayer())
# input quantization (if any) to standalone thresholding


model_for_sim = model_for_sim.transform(to_hls.InferStreamingMaxPool())
model_for_sim = model_for_sim.transform(to_hls.InferPool_Batch())

model_for_sim = model_for_sim.transform(to_hls.InferThresholdingLayer())
model_for_sim = model_for_sim.transform(to_hls.InferConvInpGen())
model_for_sim = model_for_sim.transform(to_hls.InferStreamingMaxPool())
# get rid of Reshape(-1, 1) operation between hlslib nodes
model_for_sim = model_for_sim.transform(RemoveCNVtoFCFlatten())
# get rid of Tranpose -> Tranpose identity seq
model_for_sim = model_for_sim.transform(absorb.AbsorbConsecutiveTransposes())
# infer tensor data layouts
model_for_sim = model_for_sim.transform(InferDataLayouts())


model_for_sim = model_for_sim.transform(InferDataTypes())
model_for_sim = model_for_sim.transform(RemoveStaticGraphInputs())
model_for_sim = model_for_sim.transform(RemoveUnusedTensors())
model_for_sim.save(build_dir + "/model_fmnist_final.onnx")
parent_model = model_for_sim.transform(CreateDataflowPartition())
parent_model.save(build_dir + "/dataflow_parent.onnx")
sdp_node = parent_model.get_nodes_by_op_type("StreamingDataflowPartition")[0]
sdp_node = getCustomOp(sdp_node)
dataflow_model_filename = sdp_node.get_nodeattr("model")
# save the dataflow partition with a different name for easier access
dataflow_model = ModelWrapper(dataflow_model_filename)
dataflow_model.save(build_dir + "/dataflow_model.onnx")
```

In [23]:

```
from finn.util.visualization import showInNetron

showInNetron(build_dir + "/model_fmnist_final.onnx")
```

```
Stopping http://0.0.0.0:8081 (http://0.0.0.0:8081)
Serving '/workspace/finn/notebooks/mnist_ex/verification/model_fmnist_final.onnx' at http://0.0.0.0:8081
(http://0.0.0.0:8081)
```

Out[23]:

3. Load the Dataset and the Brevitas Model

We'll use some example data from the quantized UNSW-NB15 dataset (from the previous notebook) to use as inputs for the verification.

In [3]:

```
#import model .onnx into FINN
from finn.core.modelwrapper import ModelWrapper
build_dir="./verification"
model_for_sim = ModelWrapper(build_dir + "/model_fmnist_notebook_tidy.onnx")
from dataset_loading import mnist
```

In [4]:

```
bsize=32
dataset_root="/workspace/finn/notebooks/mnist_ex/data2/FashionMNIST/raw"

trainx, trainy, testx, testy, valx, valy = mnist.load_mnist_data(dataset_root, download=False, one_hot=False)

test_imgs = testx
test_labels = testy
total = test_imgs.shape[0]
n_batches = int(total / bsize)
limit = n_batches*bsize
test_imgs = test_imgs[:limit,:,:].reshape(n_batches, bsize, -1)
test_labels = test_labels[:limit,].reshape(n_batches, bsize)
print(test_imgs.shape)
```

```
(312, 32, 784)
```

In [5]:

```
finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: global_in
Output tensor name: global_out
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 10]
Input tensor datatype: UINT8
Output tensor datatype: FLOAT32
List of node operator types in the graph:
['Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool',
'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul', 'Div', 'Add', 'Mul']
```

Let's also bring up the MLP we trained in Brevitas from the previous notebook. We'll compare its outputs to what is generated by FINN.

In [6]:

```
#MODEL

from torch import nn
from torch.nn import Module
import brevitas.nn as qnn
from brevitas.quant import Int8Bias as BiasQuant
from brevitas.quant import Uint8ActPerTensorFloat as ActQuant
from brevitas.quant import Int8WeightPerTensorFloat as WeighQuant
import torch.nn.functional as F

# Setting seeds for reproducibility
torch.manual_seed(0)

weight_bitx = 8
act_bitx = 8
bias_bitx = 8

model = nn.Sequential(
    #qnn.QuantIdentity(bit_width=act_bitx, return_quant_tensor=True),
    qnn.QuantConv2d(in_channels=1, out_channels=3, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, weight_quant=WeighQuant,
                                    bias_quant=BiasQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.MaxPool2d(kernel_size=2, stride=2),
    qnn.QuantConv2d(in_channels=3, out_channels=8, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, bias_quant=BiasQuant,
                                    weight_quant=WeighQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.MaxPool2d(kernel_size=2, stride=2),
    qnn.QuantConv2d(in_channels=8, out_channels=16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, bias_quant=BiasQuant,
                                    weight_quant=WeighQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.Flatten(),
    qnn.QuantLinear(in_features=16*7*7, out_features=10, bias=True, weight_bit_width=weight_bitx,
                                   weight_quant=WeighQuant, bias_quant=BiasQuant, return_quant_tensor=False)
)
trained_state_dict = torch.load("state_dict_self-trained.pth")
#trained_state_dict = torch.load("state_dict.pth")["models_state_dict"][0]
model.load_state_dict(trained_state_dict, strict=False)
```

Out[6]:

```
<All keys matched successfully>
```

In [7]:

```
def inference_with_brevitas(current_inp):
    input_tensors = current_inp.astype(np.float32)
    input_tensors = torch.from_numpy(input_tensors)
    brevitas_output = model.forward(input_tensors)
    y_pred=brevitas_output.argmax(1)
    return y_pred
```

4. Compare FINN & Brevitas execution

Let's make helper functions to execute the same input with Brevitas and FINN. For FINN, we'll use the `finn.core.onnx_exec` (https://finn.readthedocs.io/en/latest/source_code/finn.core.html#finn.core.onnx_exec.execute_onnx) function to execute the exported FINN-ONNX on the inputs. Note that this ONNX execution is for verification only; not for accelerated execution.

Recall that the quantized values from the dataset are 593-bit binary {0, 1} vectors whereas our exported model takes 600-bit bipolar {-1, +1} vectors, so we'll have to preprocess it a bit before we can use it for verifying the ONNX model.

In [8]:

```
import finn.core.onnx_exec as oxe

def inference_with_finn_onnx(current_inp):
    finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
    finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
    finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
    #print(finnonnx_model_in_shape)
    current_inp=current_inp.astype(np.float32)
    # reshape to expected input (add 1 for batch dimension)
    current_inp = current_inp.reshape(finnonnx_model_in_shape)
    # create the input dictionary
    input_dict = {finnonnx_in_tensor_name : current_inp}
    # run with FINN's execute_onnx
    output_dict = oxe.execute_onnx(model_for_sim, input_dict)
    #get the output tensor
    finn_output = output_dict[finnonnx_out_tensor_name]
    finn_output= torch.from_numpy(finn_output.argmax(1))
    return finn_output
```

Now we can call our inference helper functions for each input and compare the outputs.

In [9]:

```
import numpy as np
from tqdm import trange

n_verification_inputs=32
verify_range = trange(n_verification_inputs, desc="FINN execution", position=0, leave=True)
model.eval()

ok = 0
nok = 0

for i in verify_range:
    current_inp = test_imgs[0][i].reshape(1, 1, 28, 28)
    brevitas_output = inference_with_brevitas(current_inp)
    finn_output = inference_with_finn_onnx(current_inp)
    #print(brevitas_output, finn_output)
    # compare the outputs
    ok += 1 if finn_output == brevitas_output else 0
    nok += 1 if finn_output != brevitas_output else 0
    verify_range.set_description("ok %d nok %d" % (ok, nok))
    verify_range.refresh()
    #print(finn_output)
```

```
ok 32 nok 0: 100%|██████████| 32/32 [00:03<00:00, 10.50it/s]
```

In [10]:

```
if ok == n_verification_inputs:
    print("Verification succeeded. Brevitas and FINN-ONNX execution outputs are identical")
else:
    print("Verification failed. Brevitas and FINN-ONNX execution outputs are NOT identical")
```

```
Verification succeeded. Brevitas and FINN-ONNX execution outputs are identical
```

This concludes our second notebook. In the next one, we'll take the ONNX model we just verified all the way down to FPGA hardware with the FINN compiler.