Yanchen Liu (yl4189)
Minghui Zhao (mz2866)

Columbia University
Spring 2022

CSEE 4840 Embedded System Design Final Project Report

Convolutional Neural Network

1. Introduction

Convolutional Neural Network (CNN) is widely used in the machine learning task in the computer vision and neural language processing area. In this project, we implement the convolutional neural network algorithm on the DE-1 SOC FPGA + HPS to run a pre-trained CNN-based network: VGG-11.

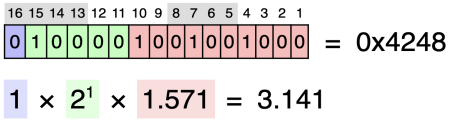
2. Data Flow

The figure below shows the structure of the VGG11, which contains these types of operator: conv2d, Relu, max pooling 2d, adaptive average pooling, linear(fully connection).

To limit the volume of data transmitted between the HPS and the FPGA, we did the following modification compared with the original VGG

1. Instead of the famous VGG16, we apply VGG11, which is lighter with less parameters compared with VGG16.
2. The input image shape is set to 32 by 32 by 3 rather than 224 by 224 by 3, and the output class number is changed from 1000 to 10. We use an open source 32 by 32 RGB dataset called CIFAR-10 to train the VGG11 network.
3. Float point 16 is used instead of float point 32. We use fp16 to make the data width shorter. The fp16 follows the IEEE 754 with one bit sign, five bit exponent and ten bit mantissa.

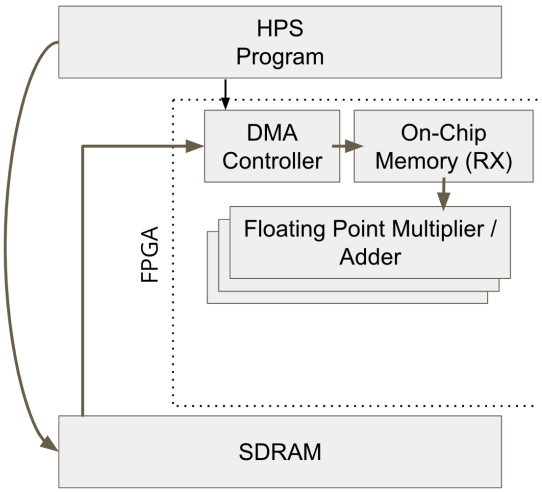
16-bit (half)



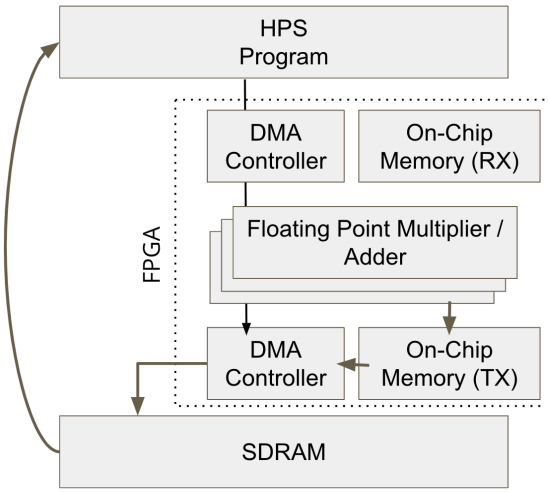
$1 \times 2^{1} \times 1.571 = 3.141$

```
VGG11_NET                                --                        --
├─VGG: 1-1                               [1, 25088]                --
│    └─Sequential: 2-1                   [1, 512, 1, 1]            --
│    │    └─Conv2d: 3-1                  [1, 64, 32, 32]           1,792
│    │    └─ReLU: 3-2                    [1, 64, 32, 32]           --
│    │    └─MaxPool2d: 3-3               [1, 64, 16, 16]           --
│    │    └─Conv2d: 3-4                  [1, 128, 16, 16]          73,856
│    │    └─ReLU: 3-5                    [1, 128, 16, 16]          --
│    │    └─MaxPool2d: 3-6               [1, 128, 8, 8]            --
│    │    └─Conv2d: 3-7                  [1, 256, 8, 8]            295,168
│    │    └─ReLU: 3-8                    [1, 256, 8, 8]            --
│    │    └─Conv2d: 3-9                  [1, 256, 8, 8]            590,080
│    │    └─ReLU: 3-10                   [1, 256, 8, 8]            --
│    │    └─MaxPool2d: 3-11              [1, 256, 4, 4]            --
│    │    └─Conv2d: 3-12                 [1, 512, 4, 4]            1,180,160
│    │    └─ReLU: 3-13                   [1, 512, 4, 4]            --
│    │    └─Conv2d: 3-14                 [1, 512, 4, 4]            2,359,808
│    │    └─ReLU: 3-15                   [1, 512, 4, 4]            --
│    │    └─MaxPool2d: 3-16              [1, 512, 2, 2]            --
│    │    └─Conv2d: 3-17                 [1, 512, 2, 2]            2,359,808
│    │    └─ReLU: 3-18                   [1, 512, 2, 2]            --
│    │    └─Conv2d: 3-19                 [1, 512, 2, 2]            2,359,808
│    │    └─ReLU: 3-20                   [1, 512, 2, 2]            --
│    │    └─MaxPool2d: 3-21              [1, 512, 1, 1]            --
│    └─AdaptiveAvgPool2d: 2-2            [1, 512, 7, 7]            --
│    └─Sequential: 2-3                   [1, 25088]                --
├─Sequential: 1-2                        [1, 10]                   --
│    └─Flatten: 2-4                      [1, 25088]                --
│    └─Linear: 2-5                       [1, 512]                  12,845,568
│    └─ReLU: 2-6                         [1, 512]                  --
│    └─Dropout: 2-7                      [1, 512]                  --
│    └─Linear: 2-8                       [1, 128]                  65,664
│    └─ReLU: 2-9                         [1, 128]                  --
│    └─Dropout: 2-10                     [1, 128]                  --
│    └─Linear: 2-11                      [1, 10]                   1,290
==========================================================================================
Total params: 22,133,002
```

3. Hardware System Architecture



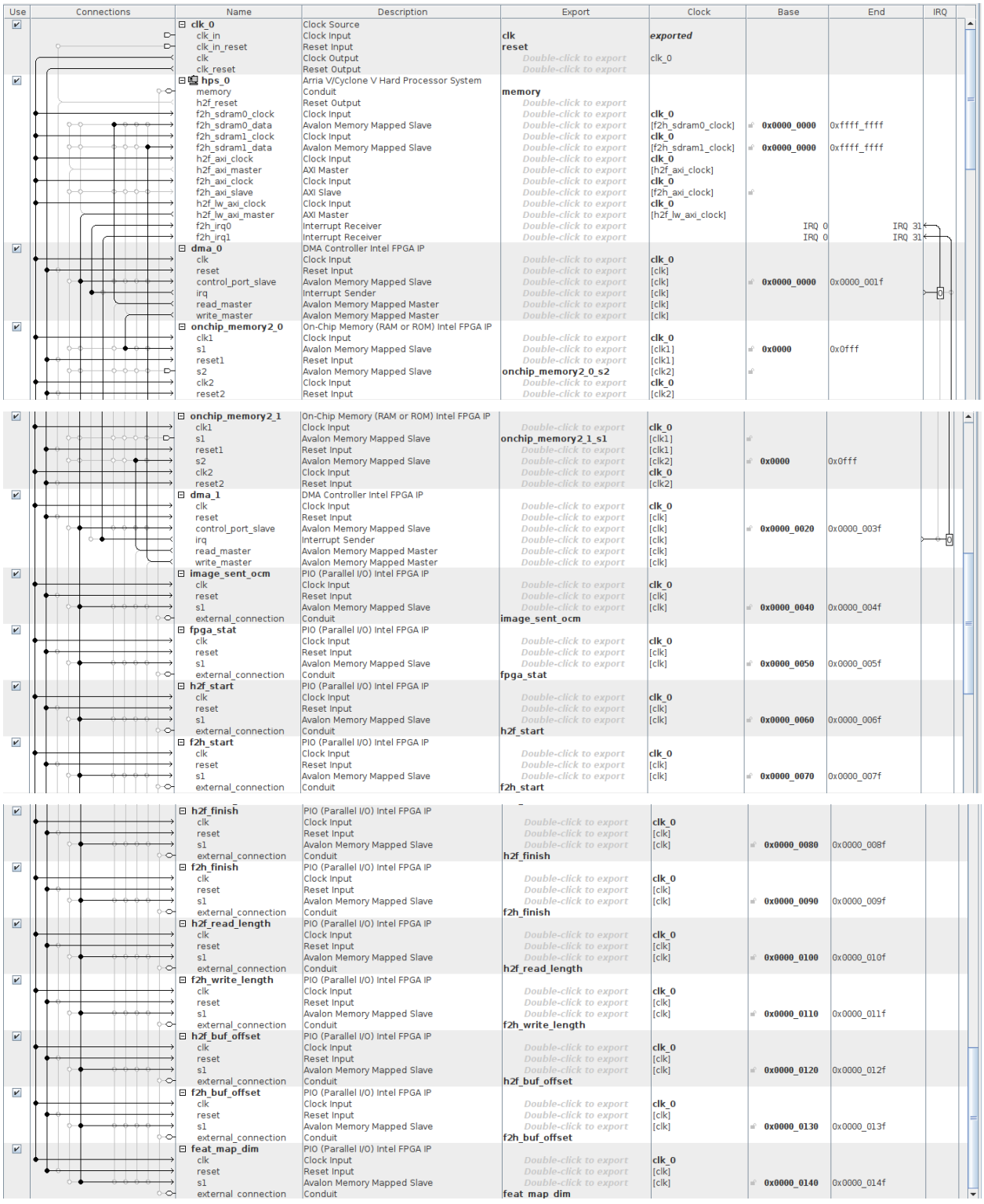
HPS Data to FPGA



FPGA Data to HPS

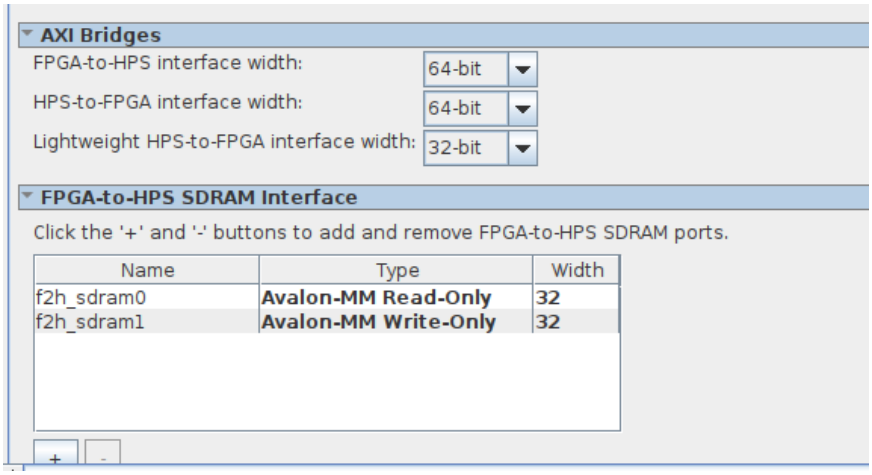
The system is built with two main components, HPS and FPGA that work with each other. From high level, the HPS is in charge of interfacing with the user and obtains the initial data, whereas the FPGA is in charge of low level computations.

The system is first configured and generated using qsys (platform designer). The qsys configuration is as follows:

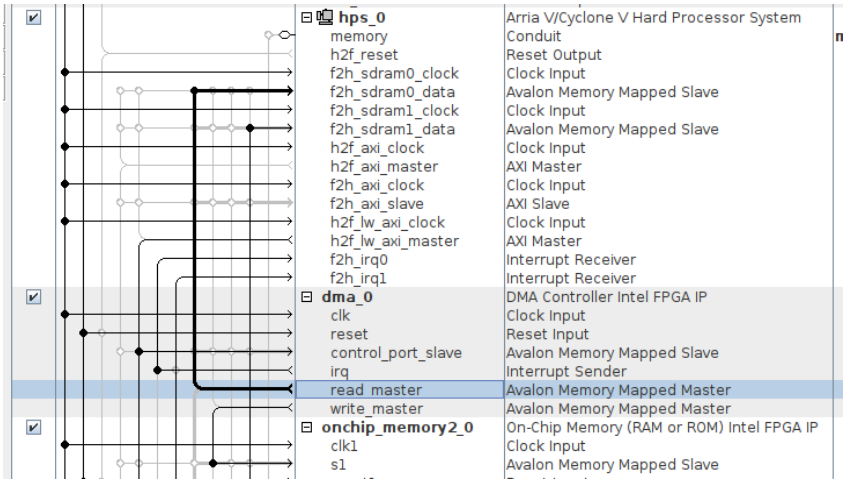


The detailed configuration of each block is as follows:

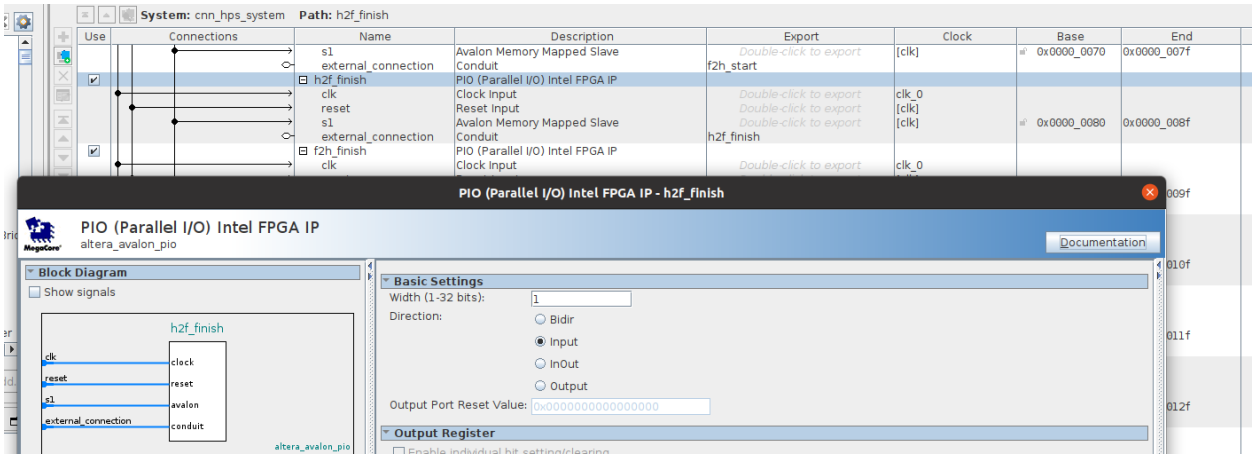
- The HPS system is configured with two DDR3 SDRAM interfaces, connected through the Avalon Memory Mapped interface as shown below. One is configured to read-only and the other as write-only.



- Two DMA-onchip_memory pairs are created for exchanging data between HPS and FPGA. As seen below, the first dma0 has its read_master connected to HPS's sdram0 (read-only), and write_master to onchip_memory0. This dma copies data from HPS to the onchip_memory. Vise versa, dma1 has its read_master connected to on_chip_memory1 and write_master connected to HPS's sdram1 (write-only).



- A list of PIO IP cores are added for HPS to send or receive control signals from / to FPGA. As seen below, the h2f_finish signal has one bit and is configured as an input, with conduit "h2f_finish" and base address 0x00000080.



This signal can be used in the top level module as follows, as part of the generated qsys module.

```
502     // PIO to start the image reading from OCM
503     .image_sent_ocm_export                  (start_ocm_wire),
504
505     .fpga_stat_export                       (fpga_stat_wire),
506
507     .h2f_start_export                       (h2f_start),
508     .h2f_read_length_export                 (h2f_read_length),
509     .feat_map_dim_export                    (feat_map_dim),
510     .h2f_buf_offset_export                  (h2f_buf_offset),
511     .f2h_start_export                       (f2h_start),
512     .f2h_write_length_export                (f2h_write_length),
513     .f2h_buf_offset_export                  (f2h_buf_offset),
514
515     .h2f_finish_export                      (h2f_finish),
516     .f2h_finish_export                      (f2h_finish),
517
```

In the HPS code, we first take advantage of the “sopc-create-header-files” tool from quartus to generate a header file that has the configured base addresses defined using the following command:

```
sopc-create-header-files "/path/to/qsys.sopcinfo" --single hps_0.h
--module hps_0
```

```
249  /*
250   * Macros for device 'h2f_finish', class 'altera_avalon_pio'
251   * The macros are prefixed with 'H2F_FINISH_'.
252   * The prefix is the slave descriptor.
253   */
254  #define H2F_FINISH_COMPONENT_TYPE altera_avalon_pio
255  #define H2F_FINISH_COMPONENT_NAME h2f_finish
256  #define H2F_FINISH_BASE 0x80      You, seconds ago • Uncommitted changes
257  #define H2F_FINISH_SPAN 16
258  #define H2F_FINISH_END 0x8f
259  #define H2F_FINISH_BIT_CLEARING_EDGE_REGISTER 0
260  #define H2F_FINISH_BIT_MODIFYING_OUTPUT_REGISTER 0
```

The generated header file includes the defines above, and we can use the PIO in the HPS C code as follows:

```
//lightweight HPS-to-FPGA bridge
void *virtual_base;
virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ),
MAP_SHARED, fd, HW_REGS_BASE );

uint32_t * h2f_finish = virtual_base + ( ( unsigned long  )(
ALT_LWFPGASLVS_OFST + H2F_FINISH_BASE) & ( unsigned long)(
HW_REGS_MASK ) );

*h2f_finish = 0;
```

Similarly, as the DMA controller is also on the lightweight HPS-to-FPGA bridge, we are able to command the DMA controller similarly:

```
// ========== HPS ---DMA----> FPGA ==========
void *h2p_lw_dma_addr0 = NULL;
h2p_lw_dma_addr0 = virtual_base + ( ( unsigned long  )(
ALT_LWFPGASLVS_OFST + DMA_0_BASE ) & ( unsigned long)( HW_REGS_MASK )
);

#define _DMA_REG_STATUS(BASE_ADDR) *((uint32_t *)BASE_ADDR+0)
#define _DMA_REG_READ_ADDR(BASE_ADDR) *((uint32_t *)BASE_ADDR+1)
#define _DMA_REG_WRITE_ADDR(BASE_ADDR) *((uint32_t *)BASE_ADDR+2)
#define _DMA_REG_LENGTH(BASE_ADDR) *((uint32_t *)BASE_ADDR+3)
#define _DMA_REG_CONTROL(BASE_ADDR) *((uint32_t *)BASE_ADDR+6)

_DMA_REG_STATUS(h2p_lw_dma_addr0) = 0;
```

```
_DMA_REG_READ_ADDR(h2p_lw_dma_addr0) = physical_addr1;  // read from
F2SDRAM_0
_DMA_REG_WRITE_ADDR(h2p_lw_dma_addr0) = 0;  // write to F2SDRAM_1
_DMA_REG_LENGTH(h2p_lw_dma_addr0) = <size>; //write <size>

    //start the transfer
_DMA_REG_CONTROL(h2p_lw_dma_addr0) = _DMA_CTR_BYTE | _DMA_CTR_GO |
_DMA_CTR_LEEN;

// ========== FPGA ----DMA----> HPS ==========
void *h2p_lw_dma_addr1 = NULL;
h2p_lw_dma_addr1 = virtual_base + ( ( unsigned long  )(
ALT_LWFPGASLVS_OFST + DMA_1_BASE ) & ( unsigned long)( HW_REGS_MASK )
);

_DMA_REG_STATUS(h2p_lw_dma_addr1) = 0;
_DMA_REG_READ_ADDR(h2p_lw_dma_addr1)  = 0; // read from OCM
_DMA_REG_WRITE_ADDR(h2p_lw_dma_addr1) = physical_addr2;  // write to
SDRAM (DDR3)
_DMA_REG_LENGTH(h2p_lw_dma_addr1) = <size>; // read <size> bytes

//start the transfer
 _DMA_REG_CONTROL(h2p_lw_dma_addr1) = _DMA_CTR_BYTE | _DMA_CTR_GO |
_DMA_CTR_LEEN;

// wait for DMA to be finished
waitDMAFinish(h2p_lw_dma_addr1);

void waitDMAFinish(void *BASE_ADDR) {
     while(!(_DMA_REG_STATUS(BASE_ADDR) & _DMA_STAT_DONE) &&
           (_DMA_REG_STATUS(BASE_ADDR) & _DMA_STAT_BUSY));
}
```

4. Main Modules

4.1 readOCM.sv

This module handles reading from the OCM, including both reading weight and bias, and loading the feature map.

After HPS finishes asking the DMA controller to copy data into OCM, it sends a signal to the “start” wire below, after which the readOCM module will load the weight and bias from OCM into “weight_bias” and raise “in_data_ready” when finished.

The module also reads feature maps from OCM at index “conv_idx”, after receiving a rising edge on “start_fm”. It will read the feature map into “feat_map_in” and raise “finish_fm” when done.

The detailed interface is as follows.

```
module readOCM(
        input    logic                                  clk, reset,

        // ------------ Read weight and bias ------------
        // Input from HPS
        input    logic                                  start,
        input    logic    [15: 0]                       read_length,
        input    logic    [5: 0]                        read_data_dim,

        // Output to pipeline
        output   logic                                  in_data_ready,
        output   logic    [(3*3+1)*16-1: 0]             weight_bias,
        // ---------------------------------------------

        // ------------ Read feature map 3x3 ------------
        // Input from pipeline
        input    logic                                  start_fm,
        input    logic    [15: 0]                       conv_idx,

        // Output to pipeline
        output   logic                                  finish_fm,
        output   logic    [3*3*16-1: 0]                 feat_map_in,
        // ---------------------------------------------


        // On-Chip RAM 0 s2 (read)
        input    logic    [ 7: 0]                       ocm0_readdata,
        output   logic    [16: 0]                       ocm0_addr,
        output   logic                                  ocm0_chip,
        output   logic                                  ocm0_clk_enab,

        // Debug
        output   logic    [ 2: 0]                       debug_state
);
```

The module is implemented using a finite state machine structure with the following states. Please refer to the code for detailed implementation.
S0: Reset

S1: Prepare to read weight bias
S2: Read weight bias
S3: Finished reading weight bias. Wait for feature map request
S4: Received feature map request, prepare to read
S5: Read feature map 8 LSB
S6: Read feature map 8 MSB
S7: Finished feature map reading

4.2 convOpt.sv

This module implements convolution. Input “in_data_ready” signals that weight and bias is ready, and “in_data_dim” is the dimension of the data. “Weight_bias” is the weight and bias read by readOCM module. The module uses a finite state machine to manage states. Please see the source code for detailed implementation.

```
module convOpt(
        input   logic                                   clk, reset,

        // Input from pipeline
        input   logic                                   in_data_ready,
        input   logic   [ 5: 0]                         in_data_dim,
        input   logic   [(3*3+1)*16-1: 0]               weight_bias,

        // Request FM from pipeline
        output  logic                                   start_fm,
        output  logic   [15: 0]                         conv_idx,

        input   logic                                   finish_fm,
        input   logic   [3*3*16-1: 0]                   feat_map_in,

        input   logic                                   finish_out,

        // Output to pipeline
        output  logic                                   start_out,
        output  logic   [16: 0]                         out_idx,
        output  logic   [3*3*16-1: 0]                   feat_map_out,

        // Output to HPS
        output  logic                                   finish,

        // Debug
        output  logic   [ 2: 0]                         debug_state
);
```

4.3 writeOCM.sv

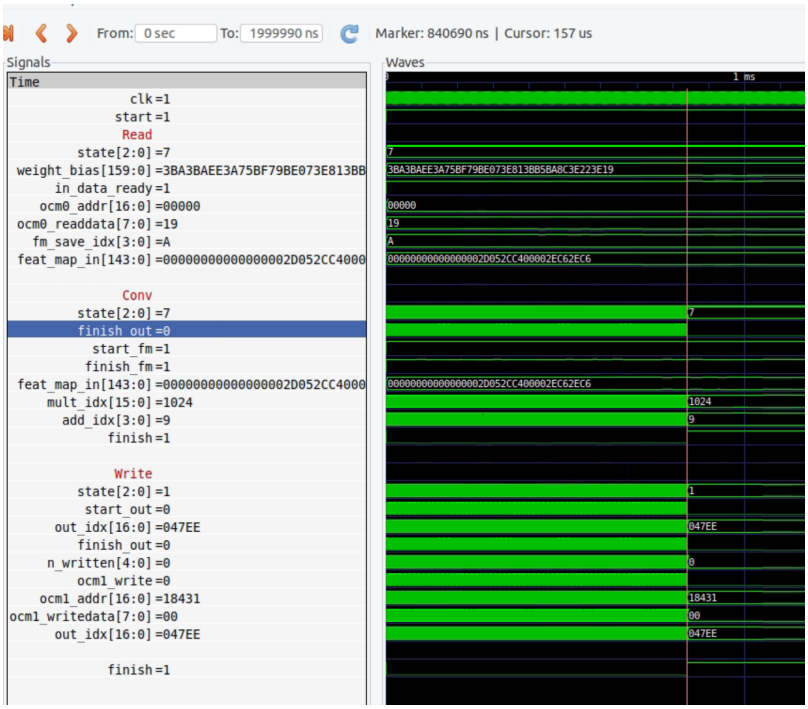
This module handles writing data to OCM that will be DMA'd back to HPS. The module uses a finite state machine to manage states. Please see the source code for detailed implementation.

```
module writeOCM(
        input   logic                                   clk, reset,
        input   logic                                   start_out,
        input   logic   [16: 0]                         out_idx,
        input   logic   [3*3*16-1: 0]                   feat_map_out,

        output  logic                                   finish_out,

        // On-Chip RAM 1 s1 (write)
        output logic    [ 7: 0]                         ocm1_writedata,
        output logic    [16: 0]                         ocm1_addr,
        output logic                                    ocm1_chip,
        output logic                                    ocm1_clk_enab,
        output logic                                    ocm1_write,

        output logic    [15: 0]                         count,

        // Debug
        output  logic   [ 2: 0]                         debug_state
);
```

5. Testbench

Version 1:
For one 32x32 feature map, total time is 472us
Loading 32*32: 40us
Computation: 390us
Sending 32*32: 40us

Version 2:
For one 32x32 feature map, total time is 840us



6. Synthesis result

Flow Status	Successful - Thu May 12 16:22:09 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	cnn_process
Top-level Entity Name	main
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	5,361 / 32,070 (17 %)
Total registers	3925
Total pins	97 / 457 (21 %)
Total virtual pins	0
Total block memory bits	73,728 / 4,065,280 (2 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	1 / 4 (25 %)

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	5455
2		
3	▾ Combinational ALUT usage for logic	8938
1	-- 7 input functions	156
2	-- 6 input functions	1540
3	-- 5 input functions	1262
4	-- 4 input functions	2119
5	-- <=3 input functions	3861
4		
5	Dedicated logic registers	3688
6		
7	I/O pins	97
8	I/O registers	80
9	Total MLAB memory bits	0
10	Total block memory bits	73728
11		
12	Total DSP Blocks	0
13		
14	Total DLLs	1
15	Maximum fan-out node	CLOCK_50~input
16	Maximum fan-out	3763
17	Total fan-out	52077
18	Average fan-out	3.95

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins
1	|main	8938 (1)	3688 (0)	73728	0	97	0
1	|cnn_hps_system:The_System|	3852 (0)	2896 (0)	73728	0	0	0
2	|tbOpt:opt|	5085 (0)	792 (0)	0	0	0	0
1	|convOpt:cv|	4385 (162)	250 (250)	0	0	0	0
1	|float_adder:fp_add|	265 (265)	0 (0)	0	0	0	0
2	|float_multi:gen_f...kernel[0].fp_mult|	437 (437)	0 (0)	0	0	0	0
3	|float_multi:gen_f...kernel[1].fp_mult|	438 (438)	0 (0)	0	0	0	0
4	|float_multi:gen_f...kernel[2].fp_mult|	438 (438)	0 (0)	0	0	0	0
5	|float_multi:gen_f...kernel[3].fp_mult|	438 (438)	0 (0)	0	0	0	0
6	|float_multi:gen_f...kernel[4].fp_mult|	437 (437)	0 (0)	0	0	0	0
7	|float_multi:gen_f...kernel[5].fp_mult|	438 (438)	0 (0)	0	0	0	0
8	|float_multi:gen_f...kernel[6].fp_mult|	438 (438)	0 (0)	0	0	0	0
9	|float_multi:gen_f...kernel[7].fp_mult|	442 (442)	0 (0)	0	0	0	0
10	|float_multi:gen_f...kernel[8].fp_mult|	419 (419)	0 (0)	0	0	0	0
11	|lpm_mult:Mult0|	33 (0)	0 (0)	0	0	0	0
1	|mult_ug11:auto_generated|	33 (33)	0 (0)	0	0	0	0
2	|readOCM:rd|	653 (503)	499 (499)	0	0	0	0
1	|lpm_mult:Mult1|	31 (0)	0 (0)	0	0	0	0
1	|mult_ug11:auto_generated|	31 (31)	0 (0)	0	0	0	0
2	|lpm_mult:Mult2|	119 (0)	0 (0)	0	0	0	0
1	|mult_li11:auto_generated|	119 (119)	0 (0)	0	0	0	0
3	|writeOCM:wr|	47 (47)	43 (43)	0	0	0	0

6. Other Modules & Test Files Developed

Main.sv: Top module for synthesis

readFMPipeline.sv: Version 1's read from OCM

readImg.sv: Test module

tbEchoWrite.sv: Test bench top module for writing to OCM and echo back

tbFPU.sv: Test bench top module for the float16 module used

tbOpt.sv: Test bench top module for version 2

tbPipeline.sv: Test bench top module for version 1

tbRWFeatureMap.sv: Test bench top module for reading and writing feature map

testEcho.sv: Test bench top module for echoing data from input OCM to output OCM

writeFeatMap.sv: Write feature map to output OCM initial test

writeFMPipeline.sv: Write feature map to output OCM version 1

writeOCM.sv: Write feature map to output OCM version 2

writeOCM8.sv: Initial test to write to OCM 8-bit data