CNN - FPGA Acceleration

[report]

2012131020 Yang yee chang

[Contents]

▶ 1. Proposal

We implement two modules. Named moduleA and moduleB as we did in week9. moduleA reads the start signal and check if moduleB is done. In moduleB, we implement the CNN as instructed. we will do 3 layers, which are convolution, max-pooling, and fully-connected layer. We reduce the CNN total calculation time by adjusting the operations. we will explain it in detail of the code.

As a whole thing, we did 3 steps to calculate.

Step 1 : 2D convolution (inner product)

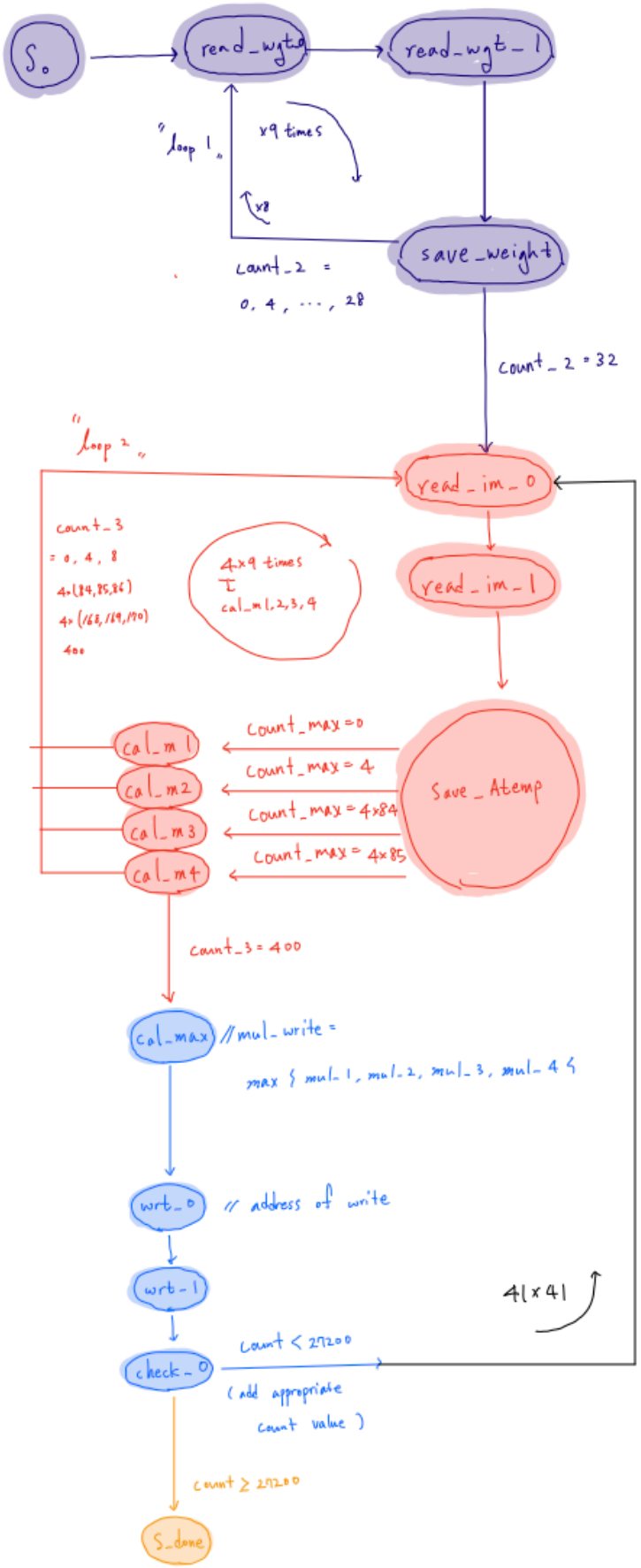
Step 2 : Max pooling

Step 3 : ReLu

The detail state diagram of moduleB is as follows.

```
********************************************************************************************** states

    parameter S0 = 4'd0; // check start signal

    ////////////////////////////read convolution weight/////////////////////////////////  loop1
    parameter read_wgt_0 = 4'd1; // assign address to read weight value ( 4001_0000[i] ) i < 9
    parameter read_wgt_1 = 4'd9; // change ready in and trans to 0 to use bus ( for convolution matrix )
    parameter save_weight = 4'd2; // save the weight to B11 ~ B33.
    // cheack count_2 if saving the weight is done
    ////////////////////////////read image and do CNN ( whole calculation )//////////////////  loop2
    parameter read_im_0 = 4'd3; // try to read image data ( 4000_0000[j] ) j < 82*82
    parameter read_im_1 = 4'd10; // change ready in and trans to 0 to use bus ( for image data )
    parameter save_Atemp = 4'd4; // save the image data to A_temp

    parameter cal_m1 = 4'd11; // calculate 2D convolution & ReLU for matrix 1 ( with moving to read_im 0 )
    parameter cal_m2 = 4'd12; // calculate 2D convolution & ReLU for matrix 2 ( with moving to read_im 0 )
    parameter cal_m3 = 4'd13; // calculate 2D convolution & ReLU for matrix 3 ( with moving to read_im 0 )
    parameter cal_m4 = 4'd14; // calculate 2D convolution & ReLU for matrix 4 ( with moving to read_im 0 )

    parameter cal_max = 4'd15; // find max value from cal_m1-cal_m4
    ////////////////////////////////write ////////////////////////////
    parameter wrt_0 = 4'd5; // write the data to activation_fpga ( 4002_0000[k] ), k < 1681 (41^2)
    parameter wrt_1 = 4'd6; // change ready in and trans to 0 to use bus ( do write )

    parameter check_0 = 4'd7; // check whether the count(img_matrix position) reaches 4*( 80*84 + 80 )
    // if not count doesn't end, go to read_im_0 ( loop 2 )
    ////////////////////// done
    parameter S_done = 4'd8; // end of the whole loop. get out finish = 1
```

[states]



< loop 1 >

: save 3×3 = 9 weight values to B_11, B_12, ..., B_33

< loop 2 >

: calculate cal_m1, 2, 3, 4

< loop 3 >

count_4 / count: 0, 1, ..., 40

0

$\frac{8\times40+16+4\times84}{A}$

A + 8×40 + 16 + 4×84

⋮

→ 41×41 rotation

Total state passed

$= 1 + (3\times9) + \left((4\times9\times4)+4\right)\times41\times41 + 1$

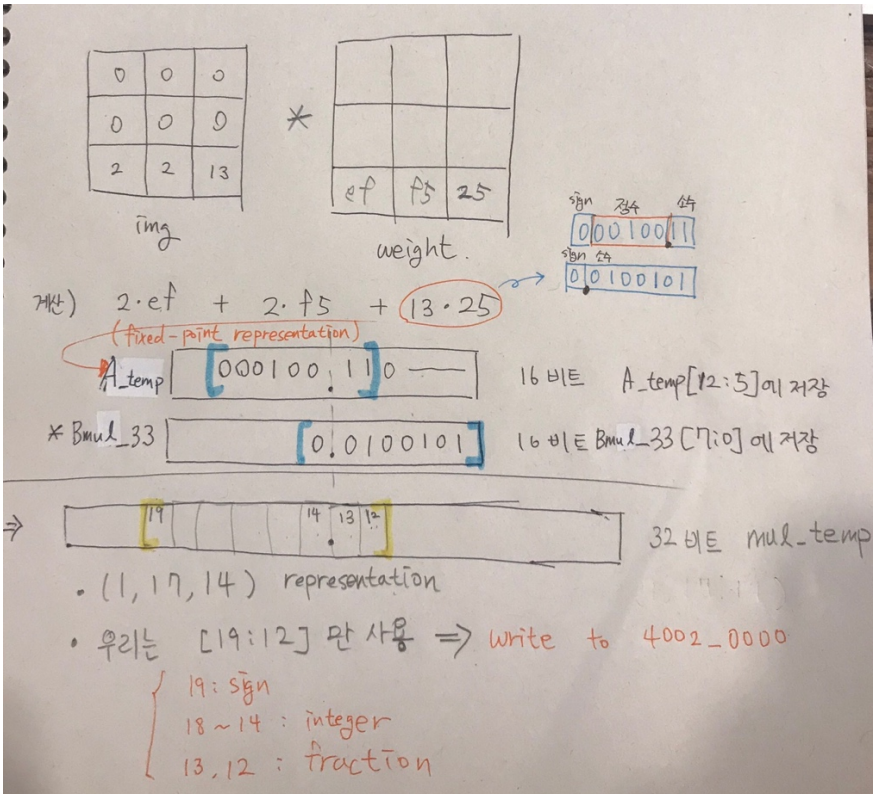
$= 248,826$

To speed up the time, we did following things.

First) save the convolution weight values in B11 to B33. It is done in states read_wgt_0 to save_weight. 9 values are saved in register as we announced. Since we don't get access to register 4001_0000 after these states, the time will be reduced.

Second) In case of step1,2 we did whole calculation in only one loop (read data from image file). we use count, count_3, count_4, count_max to calculate as a whole. states read_im0 ~ save_Atemp reads current data from img (4000_0000[j]) and cal_m1 do convolution (dot product, i.e multiply A_temp with Bmul_11) from inner_matrix 3x3. then move the bias point with count_max's change. State is then read_im_0 and read the data and then do state cal_m2. cal_m1,m2,m3,m4 saves the 2D convolution data to mul_1 to mul_4. If done, state changes to cal_max which implement max pooling.

Third) we use the fixed point representation and cutting as efficient as we can. which does the calcultion time lowerd. it is explained in fig1.



[fig 1 : fixed point representation]

In multiplication, it is important to adjust the dot (fraction start point). we use intermediate 16 bits A_temp and Bmul_ij (i,j = 1,2,3). to implement, RDATA from img is saved to A_temp[12:5] and weights are in B_11[7:0]. then the multiplied value can extended to 32 bit. Thus we use 32 bit (mul_temp) mul_temp do sumation of whole multiplied values. Then it will be checked in max-pooling. The point is that we only use [19:12] (slicing the rest). as write data as instruction. we do slice beforehand.

```
save_Atemp: begin
            if ( READY_out == 1 ) begin

            A_temp[12:5] <= RDATA[31:24]; // { D , D , D , D }
            //A_temp is [12:5] to match the fraction point with weight
```

```
mul_1[7:0] <= mul_temp[19:12]; // this is core statement of getting output.
            // since we multiplied two fixed point (1,8,7) intermediate values, the multiplied
number
            // is (1,17,14) representation
```

Therefore it will reduce the time. Also, to make multiplication of 2's complement, we do follow assign that make weight as absolute value.

```
assign Bmul_11[7:0] =(B_11[7]==0)? B_11:~B_11+1'b1;
```

In calculation, we checked the data file [convolution weight] to see when the number is minus. B(1,2) , B(2,1) B(2,3) B(3,1) B(3,2) are minus. thus we assign those value as follows.

```
mul_temp <= mul_temp - A_temp*Bmul_12;
```

if weight is plus, we did as follows. (B(1,1) B(1,3) B(2,2) B(3,3)

```
mul_temp <= mul_temp + A_temp*Bmul_11;
```

The detailed things will be explained in < 2. verilog code.

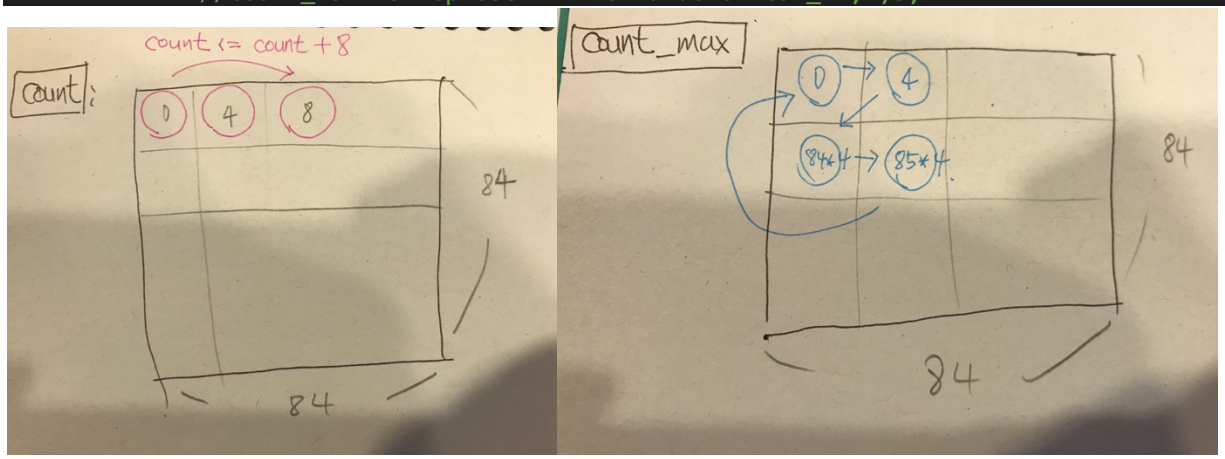
- ReLU

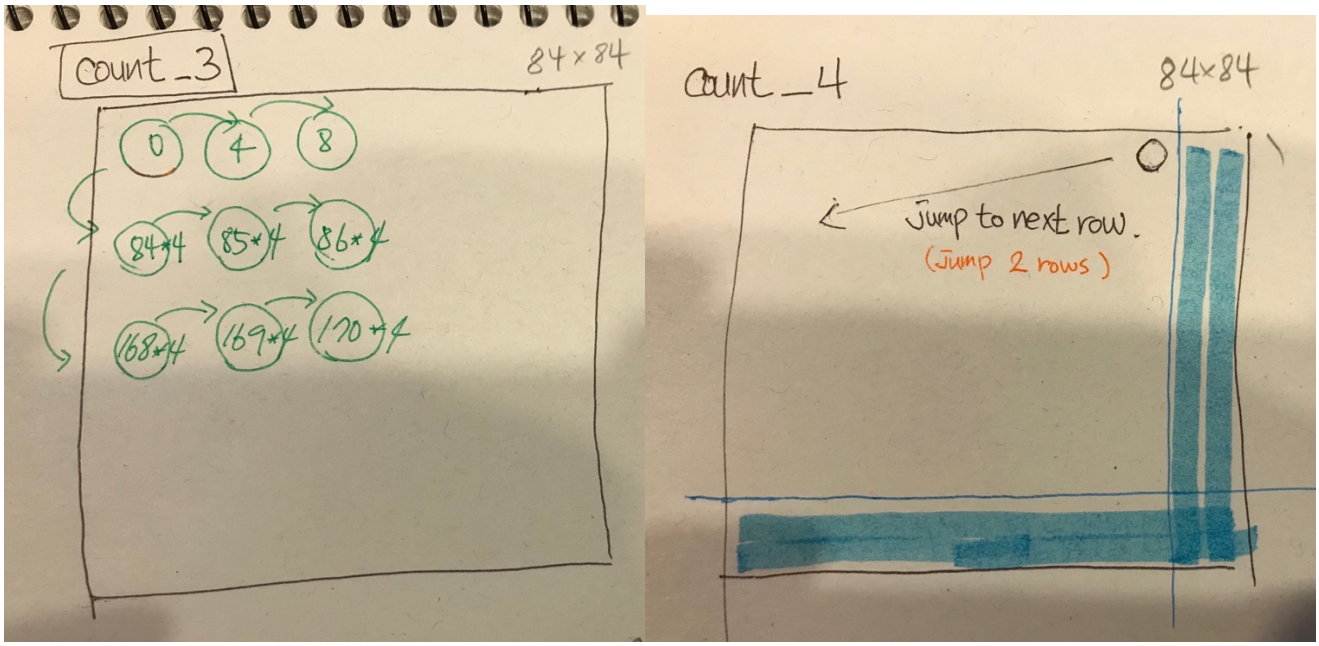
To implement ReLu, we do following code. it make mul (max of sum of inner product) when it is minus sign, make it 0, when it is plus sign, just left as it was.

```
assign mul = (mul_write[7]==0)? mul_write : 0 ;
```

-count

```
read_im_0: begin
          if ( READY_out == 1 ) begin

              ADDR <= 32'h4000_0000 + count + count_3 + count_max; // address of image data.
              // count is jump for whole 2D convolution position jump by two rows.
              // count_3 is jump for inner matrix of image_data ( 3x3 ) jump by one row
              //count_max is representitive value of cal_m1,2,3,4
```

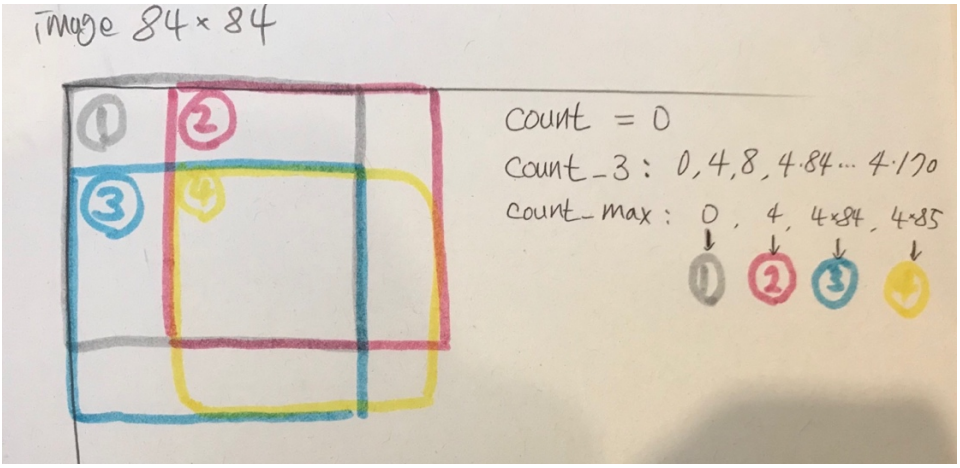




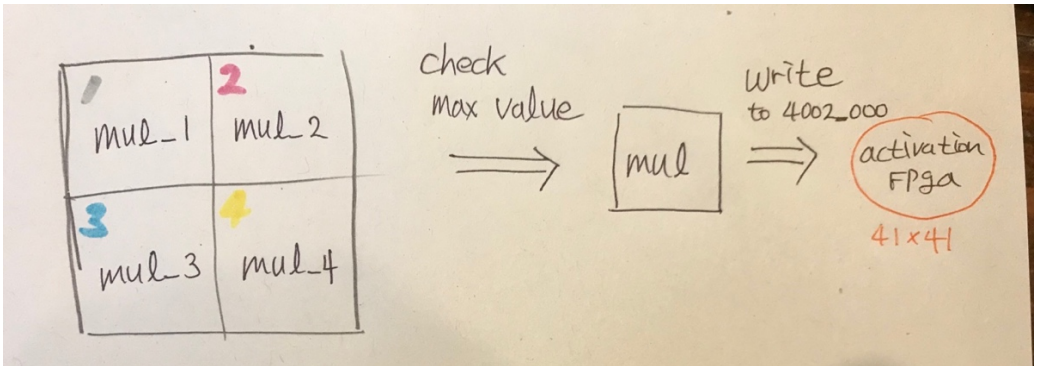

```
// count : for whole loop of calculation ends at 4*(80*84 + 80 ). it indicates convolution
position ( stride 1 )
// count_2 : for loop1 of reading weight
// count_3 : present matrix position
// count_4 : cut at 82th row position.
// count_max : checking max pooling done
```

Since count + 4 is one PC movement in data table, the base of count is multiplied by 4. For count we jump by 8 to meet the proper position of image matrix. image's size is 84X84. Thus if we have to jump rows, we have to plus 4*84. count_max is 0 ,4 , 4*84, 4*85, it is bias position of present matrix position. this is used for max-pooling. count_3 is for 2D convolution. it set the position of present position of image data. count_4 jumps when it meets the last position. it make count to jump 2 rows.

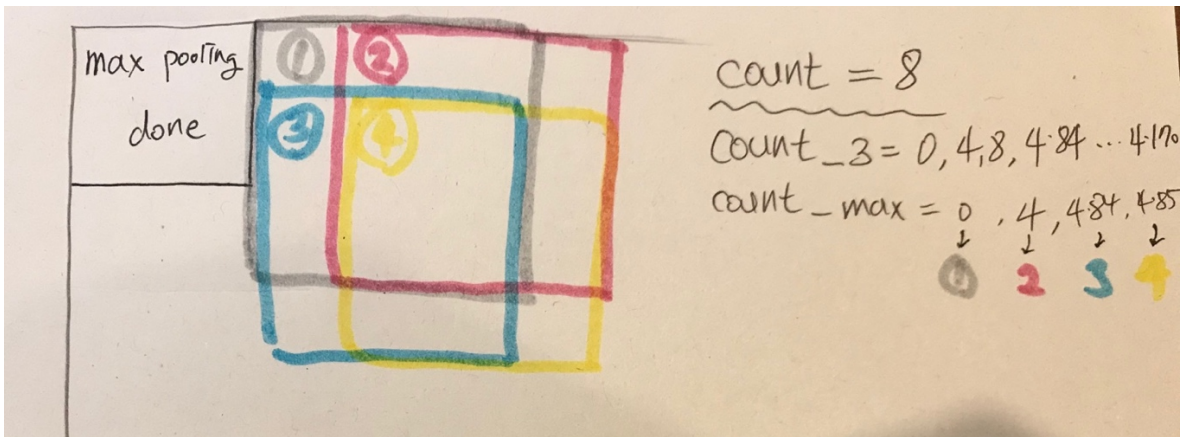
========Details of calculation==========



[fig 1-1 : when count is 0]



[fig 1-2 : saved in mul_1 ~ 4 then check max value, save to mul then write]



[fig 1-3 : when count 0 is done, move count : +8 since we have to jump 2 columns]

From read_im_0 state to check_0 state, we implement above calculations (repeatedly).

(fig 1-1) when count is 0, each value is chosen with count_3 (changes 0, 4, 8, 4*84, 4*85, 4*86, 4*168, 4*169, 4*170) then multiplied with corresponding weight (saved in Bmul_11 ~ Bmul_33).

(fig 1-2) Then sum all the values. The sum (mul_temp in verilog code) is then saved to mul_1. And we plus the count_max, Do things again then saved in mul_2. When saving in mul_1,2,3,4 is done, we go to cal_max state. In this state, check the max value of mul_1,2,3,4. then saved in mul_write. mul_write will be checked to implement ReLU. Then it is saved to mul. Then we change to wrt_0 state to write the data to activation_fpga (0x4002_0000).
(fig 1-3) When fig1-1,1-2 is done, we add 8 to count to jump two columns. The same thing is done. And so on.

▶ 2. Verilog code

==========moduleA==========

```
module moduleA(

  input [31:0]RDATA,
  input READY_out,
  input RESP,

  output reg [31:0]ADDR,
  output reg READY_in,

  output reg [1:0]TRANS,
  output reg [2:0]BURST,
  output reg [2:0]hsize,

  output reg SEL,
  output reg [3:0]PROT,

  output reg HWRITE,
  output reg [31:0]WDATA,

  output reg start,
  input finish,

  input clk,
  input reset
);

reg [3:0] state;


reg [31:0] temp;

   parameter S0 = 4'd0; // idle try to read ( address == 5000_0000 )
  parameter S1 = 4'd1; // read the data
  parameter S2 = 4'd2; // check if read data is same as 01020304
  parameter S3 = 4'd3; // waiting fin from module B.
  parameter S4 = 4'd4; // if fin == 1 ( add is end ) change the PC to 5000_0004
  parameter S5 = 4'd5; // trans = 0 and readyin = 0 ( ending the case )
  parameter S6 = 4'd6; // ready = 1 to implement the AHB bus end
  parameter S7 = 4'd7; // idle


    parameter start_signal_0 = 32'h01020304;
    parameter end_signal_0 = 32'h04030201;

always @(posedge clk) begin
   if ( !reset ) begin

      start <= 0;
```

```
        ADDR <= 0;
        READY_in <= 0;

        TRANS <= 0;
        BURST <= 0;
        hsize <= 0;

        SEL <= 0;
        PROT <= 0;

        HWRITE <= 0;
        WDATA <= 0;

        state <= S0;
    end
    else begin
    case(state)
        S0 : begin // assign address for start signal
            if ( READY_out == 1 ) begin

                ADDR <= 32'h5000_0000; // register of start signal
                READY_in <= 1;        // ready ( output )

                TRANS <= 2'b10;        // 00 idle, 10 nenseq
                BURST <= 0;           // burst gogo
                hsize <= 3'b010;       // we use only 010 ( word ), 32bit

                SEL <= 1;             // sel 0
                PROT <= 1;            // 0000 idle, 0001 read 1001 write

                state <= S1 ;
            end
        end
        S1: begin // read the data from 0x5000_0000[0] ( start_ signal )
            if ( READY_out == 1 ) begin
                READY_in <= 1;

                SEL <= 0;
                PROT <= 0;

                temp <= RDATA;

                state <= S2;
            end
            else begin
                READY_in <= 0;
            end
            TRANS <= 0;
        end
        S2: begin // compare the data to 0x01020304
            if ( temp == start_signal_0 ) begin
                state <= S3;

                start <= 1; // starting moduleB ( calculation )
```

```
        end
        else begin
            state <=  S0;
        end
    end
    S3 : begin // wait until calculation done
        if ( finish == 1 ) begin
            state <= S4;
        end
        ADDR <= 0;
        READY_in <= 0;

        TRANS <= 0;
        BURST <= 0;
        hsize <= 0;

        SEL <= 0;
        PROT <= 0;

        HWRITE <= 0;
        WDATA <= 0;

    end
    S4: begin // assign address for end signal and write ( resp = 0 )
        if ( READY_out == 1 && RESP == 0) begin
            ADDR <= 32'h5000_0004;  // address
            READY_in <= 1;         // ready (output)

            TRANS <= 2'b10;        // 10 nonseq
            BURST <= 0;           // burst
            hsize <= 3'b010;       // 32 bit

            SEL <= 1;
            PROT <= 9;            // prot 9 is write transfer

            HWRITE <= 1;          // do write
            WDATA <= end_signal_0; // write 0x04030201

            state <= S5;
        end
    end
    S5: begin // write the data

        READY_in <= 0;
        TRANS <= 0;

        state <= S6;
    end
    S6: begin // end of write, then go to default state.
        if ( READY_out == 0 && RESP == 0) begin

            READY_in <= 1;

            SEL <= 0;
            PROT <= 0;
```

```
            HWRITE <= 0;
            WDATA <= 0;

            state <= S7;
         end
      end
      default: begin

      ADDR <= 0;
      READY_in <= 0;

      TRANS <= 0;
      BURST <= 0;
      hsize <= 0;

      SEL <= 0;
      PROT <= 0;

      HWRITE <= 0;
      WDATA <= 0;

      end

   endcase
   end
end
endmodule
```

moduleA is same as we coded as week9. we assign the values referring to AHB spec (provided pdf in week9). The details are explained in week9.

		S0	S1	S2	S3	S4	S5	S6	S7
	if			temp==01020304	wait for finish vec_add				
input	HRDATA								
	HRESP					if==0		if==0	
	HREADYin	if ==1	if == 1			if==1		if==0	
output	HADDR	5000_0000	5000_0000	5000_0000	0	5000_0004	5000_0004	5000_0004	5000_0004
	HWDATA	0	0	0	0	4030201	0	0	0
	HBURST	0	0	0	0	0	0	0	0
	HREADY	1	1	1	0	1	0	1	0
	HTRANS	2'b10	2'b00	2'b00	2'b00	2'b10	2'b00	2'b00	2'b00
	HWRITE	0	0	0	0	1	1	0	0
	HSEL	1	0	0	0	1	1	0	0
	HPROT	4'b0001	4'b0000	4'b0000	0	4'b1001	4'b0000	4'b0000	0
	HSIZE	3'b010	3'b010	3'b010	0	3'b010	3'b010	3'b010	3'b010
output	start	0		1					
input	fin				checking				

[module A variables]

we use 8 states and each state is implemented as above. READYin and RESP is core input that checks if we do the read or write or wait. as we explained in week8. at first, we initial begin state to S0. S0 is the state that gets the address of the register PC (program counter) (checked as posedge of clk). but in case of S0, the followings are implemented when READYin (AHB_INTERFACE_hready_in) is 1. PC is 5000_0000 which is address of start_sig. then

we go to S1 which does the read the data. in case of S2, if read_data is 01020304 (which means C code of ARM vector add processing is ended). else, goes again to S0. when start output is 1, we go to the S3 which is waiting state to get the finish signal if fin ==1 we go to S4 which implements the end_sig_code. change the address to 5000_0004 (end_sig array) and write the data(end_sig_code = 04030201) to end_sig[0]. SDK ends, the rest states S5,S6.S7 is tedious states that will do the arrange the interfaces to 0.

==========moduleB==========

```
module moduleB(

  input [31:0]RDATA,
  input READY_out,
  input RESP,

  output reg [31:0]ADDR,
  output reg READY_in,

  output reg [1:0]TRANS,
  output reg [2:0]BURST,
  output reg [2:0]hsize,

  output reg SEL,
  output reg [3:0]PROT,

  output reg HWRITE,
  output reg [31:0]WDATA,

  input start,
  output reg finish,

  input clk,
  input reset

);

reg [3:0] state;

reg [31:0] count, count_2, count_3, count_4, count_max;
// count : for whole loop of calculation ends at 4*(80*84 + 80 ). it indicates convolution
position ( stride 1 )
// count_2 : for loop of read weight
// count_3 : present matrix position
// count_4 : cut at 82th row position.
// count_max : checking max pooling done

reg [7:0] B_11, B_12, B_13, B_21, B_22, B_23, B_31, B_32, B_33;
reg [15:0] A_temp;     // [12:5] is used. (1,5,2) sign-int-fraction. // 16 bit is used for
multiplication
                    // [6:5] is fractional number. setting this make multiplication possible.
( fixed position is implemented)
reg [31:0] mul_temp;    // since we multiply A_temp with Bmul_ij ( 16bit * 16bit ) => max
position can be 32bit.
```

```
reg [7:0] mul_1, mul_2, mul_3, mul_4, mul_write; // variables for max pooling.
// Bmul_11~33 is weight(abs). fixed position implemented ( [6:0] is fraction )
wire [15:0] Bmul_11, Bmul_12, Bmul_13, Bmul_21, Bmul_22, Bmul_23, Bmul_31, Bmul_32, Bmul_33;
wire [7:0] mul; // max value of 2x2 inner matrix

// these are variables for weight. get abs to calculate multiplication.
assign Bmul_11[7:0] =(B_11[7]==0)? B_11:~B_11+1'b1;
assign Bmul_12[7:0] =(B_12[7]==0)? B_12:~B_12+1'b1; // - sign
assign Bmul_13[7:0] =(B_13[7]==0)? B_13:~B_13+1'b1;
assign Bmul_21[7:0] =(B_21[7]==0)? B_21:~B_21+1'b1; // - sign
assign Bmul_22[7:0] =(B_22[7]==0)? B_22:~B_22+1'b1;
assign Bmul_23[7:0] =(B_23[7]==0)? B_23:~B_23+1'b1; // - sign
assign Bmul_31[7:0] =(B_31[7]==0)? B_31:~B_31+1'b1; // - sign
assign Bmul_32[7:0] =(B_32[7]==0)? B_32:~B_32+1'b1; // - sign
assign Bmul_33[7:0] =(B_33[7]==0)? B_33:~B_33+1'b1;
// Do ReLU
assign mul = (mul_write[7]==0)? mul_write : 0 ;
// ***************************************************************************************
state

    parameter S0 = 4'd0; // check start signal

    ////////////////////////////read convolution weight/////////////////////////////////
loop1
    parameter read_wgt_0 = 4'd1; // assign address to read weight value ( 4001_0000[i] ) i < 9
    parameter read_wgt_1 = 4'd9; // change ready in and trans to 0 to use bus ( for convolution
matrix )
    parameter save_weight = 4'd2; // save the weight to B11 ~ B33.
    // cheack count_2 if saving the weight is done
    ////////////////////////////read image and do CNN ( whole calculation )//////////////////
loop2
    parameter read_im_0 = 4'd3; // try to read image data ( 4000_0000[j] ) j < 82*82
    parameter read_im_1 = 4'd10; // change ready in and trans to 0 to use bus ( for image data )
    parameter save_Atemp = 4'd4; // save the image data to A_temp

    parameter cal_m1 = 4'd11; // calculate 2D convolution & ReLU for matrix 1 ( with moving to
read_im 0 )
    parameter cal_m2 = 4'd12; // calculate 2D convolution & ReLU for matrix 2 ( with moving to
read_im 0 )
    parameter cal_m3 = 4'd13; // calculate 2D convolution & ReLU for matrix 3 ( with moving to
read_im 0 )
    parameter cal_m4 = 4'd14; // calculate 2D convolution & ReLU for matrix 4 ( with moving to
read_im 0 )

    parameter cal_max = 4'd15; // find max value from cal_m1-cal_m4
    ////////////////////////////write ////////////////////////////
    parameter wrt_0 = 4'd5; // write the data to activation_fpga ( 4002_0000[k] ), k < 1681
(41^2)
    parameter wrt_1 = 4'd6; // change ready in and trans to 0 to use bus ( do write )

    parameter check_0 = 4'd7; // check whether the count(img_matrix position) reaches 4*( 80*84
+ 80 )
    // if not count doesn't end, go to read_im_0 ( loop 2 )
    ////////////////////// done
    parameter S_done = 4'd8; // end of the whole loop. get out finish = 1
```

```
always @(posedge clk) begin
    if ( !reset ) begin
        ADDR <= 0;
        READY_in <= 0;

        TRANS <= 0;
        BURST <= 0;
        hsize <= 0;

        SEL <= 0;
        PROT <= 0;

        HWRITE <= 0;
        WDATA <= 0;

        count <= 0;

        finish <= 0;
        state <= S0; //S0 initial state
    end
    else begin //not reset
        case(state) // case ///
        S0: begin
            if ( start ) // check start sig from moduleA //start_sig[0] is 0x01020304
                state <= read_wgt_0;  //go to state read_wgt_0
             end

////////////////////////////////////////////////////////////// loop 1
        read_wgt_0: begin
            if ( READY_out == 1 ) begin
                ADDR <= 32'h4001_0000 + count_2; // address of convolution weight.
                // until count_2 < 4*9 ( 3x3 matrix )
                READY_in <= 1;

                TRANS <= 2'b10;
                BURST <= 0;
                hsize <= 3'b010; //hsize is fixed to word(4byte) for bus

                SEL <= 1;
                PROT <= 1;

                state <= read_wgt_1; //go to state read_wgt_1
            end
        end

        read_wgt_1: begin
            READY_in <= 0;
            TRANS <= 0;
            state <= save_weight; //go to state save_weight
        end
```

```
      save_weight: begin
         if ( READY_out == 1 ) begin

             SEL <= 0;
             PROT <= 0;
             READY_in <= 1;
             end
             if ( count_2 == 0) begin // (1,1) of weight matrix
             B_11 <= RDATA[31:24];    // we use 8 bit-binary //RDATA[31:24] == RDATA[23:16] ==
RDATA[15:8] == RDATA[7:0]
             count_2 <= count_2 + 4;  // go to next matrix position
             state <= read_wgt_0;     // go to first read section and will read B_12 next
             end
             else if (count_2 == 4) begin // (1,2) of weight matrix.
             B_12 <=RDATA[31:24]; //RDATA is changed at read_wgt_1 state. Address <= Address + 4
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end
             else if (count_2 == 8) begin
             B_13 <=RDATA[31:24];
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end
             else if (count_2 == 12) begin
             B_21 <=RDATA[31:24];
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end
             else if (count_2 == 16) begin
             B_22 <=RDATA[31:24];
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end
             else if (count_2 == 20) begin
             B_23 <=RDATA[31:24];
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end
             else if (count_2 == 24) begin
             B_31 <=RDATA[31:24];
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end
             else if (count_2 == 28) begin
             B_32 <=RDATA[31:24];
             count_2 <= count_2 + 4;
             state <= read_wgt_0;
             end

             else if ( count_2 == 32 ) begin //last weigt matrix component (3,3)
                B_33 <=RDATA[31:24];

                state <= read_im_0; //all of the weight component is read, now we have to read
image
```

```
            end

        end



////////////////////////////////////////////////// loop 2
        read_im_0: begin
            if ( READY_out == 1 ) begin

                ADDR <= 32'h4000_0000 + count + count_3 + count_max; // address of image data.
                // count is jump for whole 2D convolution position jump by two rows.
                // count_3 is jump for inner matrix of image_data ( 3x3 ) jump by one row
                //count_max is representitive value of cal_m1,2,3,4
                READY_in <= 1; // do read or write bus

                TRANS <= 2'b10;
                BURST <= 0;
                hsize <= 3'b010;

                SEL <= 1;
                PROT <= 1;

                state <= read_im_1;
            end
        end

        read_im_1: begin
            READY_in <= 0;
            TRANS <= 0;
            state <= save_Atemp;
        end

        save_Atemp: begin
                if ( READY_out == 1 ) begin

                A_temp[12:5] <= RDATA[31:24]; // { D , D , D , D }
                //A_temp is [12:5] to match the fraction point with weight
                READY_in <= 1;

                SEL <= 0;
                PROT <= 0;

                end // count_max is jump for inner matrix of ( 2x2 ) it acts like biased point
                if ( count_max ==0 ) state <= cal_m1;
                else if (count_max ==4 ) state <= cal_m2;
                else if (count_max ==4*84 ) state <= cal_m3;
                else if (count_max ==4*85 ) state <= cal_m4;

            end
        cal_m1 : begin
            if ( READY_out == 1 ) begin
             if ( count_3 == 0) begin
                    count_3 <= count_3+ 4;
                    mul_temp <= mul_temp + A_temp*Bmul_11; //Bmul_11 = abs(B11). B11>0 so just add
```

```
                state <=read_im_0; //count_3 is increased -> A_temp address is increased to
next value and repeat the loop

            end
            else if ( count_3 == 4) begin
                count_3 <= count_3+ 4;
                 mul_temp <= mul_temp - A_temp*Bmul_12; //Bmul_12 = abs(B12), B12<0 so we have
to subtract

                state <=read_im_0;

            end
            else if ( count_3 == 8) begin
                count_3 <= 4*84; //indicate the next line of A_temp (84th index address)
                 mul_temp <= mul_temp + A_temp*Bmul_13;

                state <=read_im_0;

            end
            else if ( count_3 == 4*84) begin //line is changed so A_temp has to increase
                count_3 <= count_3+ 4; //change to next index address

                mul_temp <= mul_temp - A_temp*Bmul_21; //B21 is negative value

                state <=read_im_0;

            end
            else if ( count_3 == 4*85) begin // count_3 == 4*84 + 4 = 4*85
                count_3 <= count_3+ 4;

                 mul_temp <= mul_temp + A_temp*Bmul_22;

                state <=read_im_0;

            end
            else if ( count_3 == 4*86) begin
                count_3 <= 4*168; //change address to index 2*84 which is 3rd line
                mul_temp <= mul_temp - A_temp*Bmul_23;

                state <=read_im_0;

            end
            else if ( count_3 == 4*168) begin
                count_3 <= count_3 +4;
                 mul_temp <= mul_temp - A_temp*Bmul_31;

                state <=read_im_0;

            end
            else if ( count_3 == 4*169) begin
            count_3 <= count_3 +4;
                 mul_temp <= mul_temp - A_temp*Bmul_32;

                state <=read_im_0;
```

```
            end
            else if ( count_3 == 4*170) begin
            count_3 <= 400; // 400 go next (random number 400)
               mul_temp <= mul_temp + A_temp*Bmul_33;

               state <=cal_m1;
            end
            else if ( count_3 == 400) begin
            count_3 <=0; //count_3 is used for loop on cal_m1. We have to initialize to use in
cal_m2
            count_max <= 4; ///// core count
            mul_1[7:0] <= mul_temp[19:12]; // this is core statement of getting output.
            // since we multiplied two fixed point (1,8,7) intermediate values, the multiplied
number
            // is (1,17,14) representation
            mul_temp <=0; //initialize to use this on cal_m2
            state <= read_im_0;


            end
         end
        end
        // 9 iteration on cal_m1 for calculating inner product A_temp * weight
        // next cal_m2,3,4 is same with cal_m1 except that the address A_temp is increased by 4
(count_max)
      cal_m2 : begin
        if ( READY_out == 1 ) begin
         if ( count_3 == 0) begin
               count_3 <= count_3+ 4;
               mul_temp <= mul_temp + A_temp*Bmul_11;

               state <=read_im_0;

            end
            else if ( count_3 == 4) begin
               count_3 <= count_3+ 4;
                mul_temp <= mul_temp - A_temp*Bmul_12;

               state <=read_im_0;

            end
            else if ( count_3 == 8) begin
               count_3 <= 4*84;
                mul_temp <= mul_temp + A_temp*Bmul_13;

               state <=read_im_0;

            end
            else if ( count_3 == 4*84) begin
               count_3 <= count_3+ 4;

               mul_temp <= mul_temp - A_temp*Bmul_21;

               state <=read_im_0;
```

```
            end
            else if ( count_3 == 4*85) begin
               count_3 <= count_3+ 4;

                mul_temp <= mul_temp + A_temp*Bmul_22;

               state <=read_im_0;

            end
            else if ( count_3 == 4*86) begin
               count_3 <= 4*168;
               mul_temp <= mul_temp - A_temp*Bmul_23;

               state <=read_im_0;

            end
            else if ( count_3 == 4*168) begin
               count_3 <= count_3 +4;
                mul_temp <= mul_temp - A_temp*Bmul_31;

               state <=read_im_0;

            end
            else if ( count_3 == 4*169) begin
            count_3 <= count_3 +4;
                mul_temp <= mul_temp - A_temp*Bmul_32;

               state <=read_im_0;

            end
            else if ( count_3 == 4*170) begin
            count_3 <= 400;
               mul_temp <= mul_temp + A_temp*Bmul_33;
               state <=cal_m2;
             end
            else if ( count_3 == 400) begin
            count_3 <=0;
            count_max <= 4*84;
            mul_2[7:0] <= mul_temp[19:12];
            mul_temp <=0;
            state <= read_im_0;


            end
           end
          end
           // 9 iteration on cal_m2 for calculating inner product A_temp * weight
        // next cal_m3 is same with cal_m1,2 except that the count_max is 4*84 which indicates
next line, index of mul_3
        // mul_3 is third component of inner matrix

       cal_m3 : begin
         if ( READY_out == 1 ) begin
          if ( count_3 == 0) begin
```

```
            count_3 <= count_3+ 4;
            mul_temp <= mul_temp + A_temp*Bmul_11;

            state <=read_im_0;

        end
        else if ( count_3 == 4) begin
            count_3 <= count_3+ 4;
             mul_temp <= mul_temp - A_temp*Bmul_12;

            state <=read_im_0;

        end
        else if ( count_3 == 8) begin
            count_3 <= 4*84;
             mul_temp <= mul_temp + A_temp*Bmul_13;

            state <=read_im_0;

        end
        else if ( count_3 == 4*84) begin
            count_3 <= count_3+ 4;

            mul_temp <= mul_temp - A_temp*Bmul_21;

            state <=read_im_0;

        end
        else if ( count_3 == 4*85) begin
            count_3 <= count_3+ 4;

             mul_temp <= mul_temp + A_temp*Bmul_22;

            state <=read_im_0;

        end
        else if ( count_3 == 4*86) begin
            count_3 <= 4*168;
            mul_temp <= mul_temp - A_temp*Bmul_23;

            state <=read_im_0;

        end
        else if ( count_3 == 4*168) begin
            count_3 <= count_3 +4;
             mul_temp <= mul_temp - A_temp*Bmul_31;

            state <=read_im_0;

        end
        else if ( count_3 == 4*169) begin
        count_3 <= count_3 +4;
             mul_temp <= mul_temp - A_temp*Bmul_32;

            state <=read_im_0;
```

```
        end
        else if ( count_3 == 4*170) begin
        count_3 <= 400;
           mul_temp <= mul_temp + A_temp*Bmul_33;
           state <=cal_m3;
         end
        else if ( count_3 == 400) begin
        count_3 <=0;
        count_max <= 4*85;
        mul_3[7:0] <= mul_temp[19:12];
        mul_temp <=0;
        state <= read_im_0;


        end
    end
end
 cal_m4 : begin
   if ( READY_out == 1 ) begin
    if ( count_3 == 0) begin
           count_3 <= count_3+ 4;
           mul_temp <= mul_temp + A_temp*Bmul_11;

           state <=read_im_0;

        end
        else if ( count_3 == 4) begin
           count_3 <= count_3+ 4;
            mul_temp <= mul_temp - A_temp*Bmul_12;

           state <=read_im_0;

        end
        else if ( count_3 == 8) begin
           count_3 <= 4*84;
            mul_temp <= mul_temp + A_temp*Bmul_13;

           state <=read_im_0;

        end
        else if ( count_3 == 4*84) begin
           count_3 <= count_3+ 4;

           mul_temp <= mul_temp - A_temp*Bmul_21;

           state <=read_im_0;

        end
        else if ( count_3 == 4*85) begin
           count_3 <= count_3+ 4;

            mul_temp <= mul_temp + A_temp*Bmul_22;

           state <=read_im_0;
```

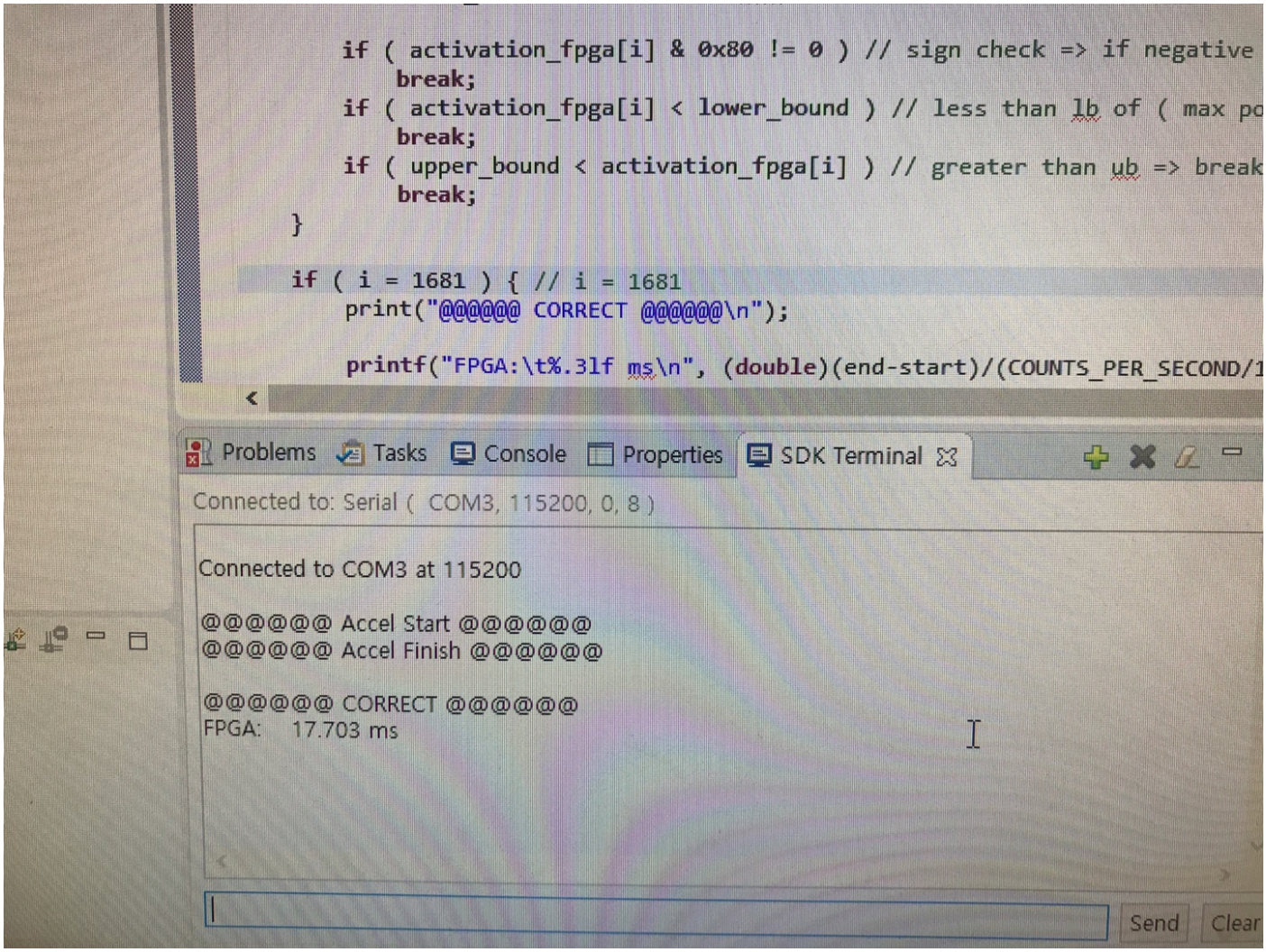
```
            end
            else if ( count_3 == 4*86) begin
               count_3 <= 4*168;
               mul_temp <= mul_temp - A_temp*Bmul_23;

               state <=read_im_0;

            end
            else if ( count_3 == 4*168) begin
               count_3 <= count_3 +4;
                mul_temp <= mul_temp - A_temp*Bmul_31;

               state <=read_im_0;

            end
            else if ( count_3 == 4*169) begin
            count_3 <= count_3 +4;
                mul_temp <= mul_temp - A_temp*Bmul_32;

               state <=read_im_0;

            end
            else if ( count_3 == 4*170) begin
            count_3 <= 400;
               mul_temp <= mul_temp + A_temp*Bmul_33;
               state <=cal_m4;
             end
            else if ( count_3 == 400) begin
            count_3 <=0;
            count_max <= 0;
            mul_4[7:0] <= mul_temp[19:12];
            mul_temp <=0;
            state <= cal_max; //we have saved mul_1,mul_2,mul_3,mul_4 value so now we need to
know which one is the largest


            end
      end
   end

      cal_max : begin
         if ( READY_out == 1 ) begin
            if ( mul_1 >= mul_2 ) begin
               if( mul_1>= mul_3) begin
                  if (mul_1 >= mul_4) begin //mul_1 will be the maximum of four values
                  mul_write <= mul_1; //pick and save mul_1 to mul_write
                  state <= wrt_0;
                end
                  else begin // 4 > 1,2,3
                  mul_write <= mul_4;
                  state <= wrt_0;
                end
                end
                else begin // mul_3 > 1,2
```

```
                    if (mul_3 >= mul_4) begin
                        mul_write <= mul_3;
                        state <= wrt_0;
                    end
                    else begin
                        mul_write <= mul_4;
                        state <= wrt_0;
                    end
                   end
                  end
            else begin // 2 >1
                if( mul_2>= mul_3) begin
                 if (mul_2 >= mul_4) begin
                     mul_write <= mul_2;
                     state <= wrt_0;
                   end
                     else begin // 4 > 1,2,3
                     mul_write <= mul_4;
                     state <= wrt_0;
                    end
                   end
                         else begin // mul_3 > 1,2
                            if (mul_3 >= mul_4) begin
                                mul_write <= mul_3;
                                 state <= wrt_0;
                                 end
                             else begin
                                mul_write <= mul_4;
                                  state <= wrt_0;
                              end
                              end
            end
           end

        end
        //untill now we have chosen the maximum value among mul_1,2,3,4 and continued to state
wrt_0

      wrt_0: begin // write
          if ( READY_out == 1 && RESP == 0 ) begin
             ADDR <= 32'h4002_0000 + count; //write mul_write at address starting from 4002_0000
with offset of 4
             READY_in <= 1;

             TRANS <= 2'b10;
             BURST <= 0;
             hsize <= 3'b010;

             SEL <= 1;
             PROT <= 9;

             HWRITE <= 1;
             WDATA <= {mul,mul,mul,mul}; // mul is 8 bit, and paste it as write data

             state <= wrt_1;
```

```
        end
    end
    wrt_1: begin

        READY_in <= 0;

        TRANS <= 0;

        state <= check_0;
    end
    check_0: begin
        if ( READY_out == 0 && RESP == 0) begin

            READY_in <= 1;

            HWRITE <= 0;
            WDATA <= 0;

            SEL <= 0;
            PROT <= 0;

            if ( count >= 27200 ) begin // = 32'hfffc // 56448, 28224 , 4*1681=6724 // 27536-8
                state <= S_done;
            end
            else begin
                if ( count_4 < 41) begin //count_4 is just an column index of activation FPGA
(0~40)
                    count <= count + 8;
                    //convolution activation matrix column index increases by 2 because
                    //we take one maximum value from 2*2 inner matrix
                    count_4 <= count_4 + 1'd1; //go to next column index
                end
                else if (count_4 == 41) begin //if one line ends (all 41 column calculated)
                    count_4<=0; //initialize column index
                    count <= count +16 + 4*84; //go to two lines ahead
                    //at first count = 8*40, +16 -> next line, +4*84 -> next line
                  end
                state <= read_im_0;
            end
        end
    end
    S_done: begin // out finish signal

        ADDR <= 0;
        READY_in <= 0;

        TRANS <= 0;
        BURST <= 0;
        hsize <= 0;

        SEL <= 0;
        PROT <= 0;

        HWRITE <= 0;
        WDATA <= 0;
```

```
            finish <= 1;
        end



        endcase
    end
end

endmodule
```

▶ 3. SDK result



We adjust the SIZE to 1681 as intended. The FPGA time calculated as 17.703ms. If we consider week9's ARM time (about 56.034ms), it would be far faster than ARM since week9's ARM only operates vector addition. if we implement CNN in c-code and check the ARM time, it will take much time than 17.703ms.

▶ 4. Discussion

There were few things to lower the calculation time, but since we didn't have much time to implement, we left it as discussion.

1) Make another module that only handles with calculation.

Let new module be module-C. on module-B, make read data as output and calculation with counting on module-C will reduce the time. In this case, module-B only do read data and write data. So the time will be reduced.

2) Do the calculation out of state.

If we do the calculation by assign statement outside of the case-state (as we did in week9) we calculate in parallel, which will make time lowered.

The number of state is fixed in each three cycles(loops). For example, for loop 1, we need 3 state of read_wgt_0, read_wgt_1 and save_weight states. For loop 2, there are read_im_0, read_im_1, save_Atemp and cal_m. For loop 3, there are cal_max, wrt_0, srt_1 and check_0. These are fixed essential states that we need at each loop. We thought that the most affective component for the execution time is number of states passed for total calculation. Since the number of states per loops is fixed, we have to reduce the number of cycles.

For example, one of the most stupid code will be taking loop1 inside loop2 which means that bringing the weight matrix(loop1) all the time when we calculate one component of convolution activation matrix. Total cycle can be calculated through

1. 9 cycles to bring weight matrix
2. 9 * 4 * 9 cycles to make one component of activation FPGA matrix
3. 9 * 4 * 9 * 41 * 41 cycles to make total activation FPGA matrix

   Total 9 * 4 * 9 * 41 * 41 = 544,644 cycles

   ⇨ Stupid code.

For another reasonable example, if we make a code that makes 82*82 convolution activation matrix and then calculate the maximum value of inner 2*2 matrix which we can make 41*41 activation FPGA matrix, we will need

1. 9 cycles to make one component of convolution activation matrix,
2. 9 * 82 * 82 cycles to make convolution activation matrix,
3. 41*41 cycles to make activation FPGA matrix

   Total = 9 * 82 * 82 + 41 * 41

   = 62,197 Cycles

In case of our code, we don't make convolution activation matrix because we will only use small portion (2*2 inner matrix) of that convolution activation matrix. We will directly calculate the component of activation FPGA matrix (mul_write) from the inner matrix. In that manner, we can calculate number of cycles by following equations.

1. 4 * 9 cycles to make four component (2*2 inner vector) of convolution activation matrix
2. 4 * 9 * 41 * 41 cycles to make activation FPGA matrix

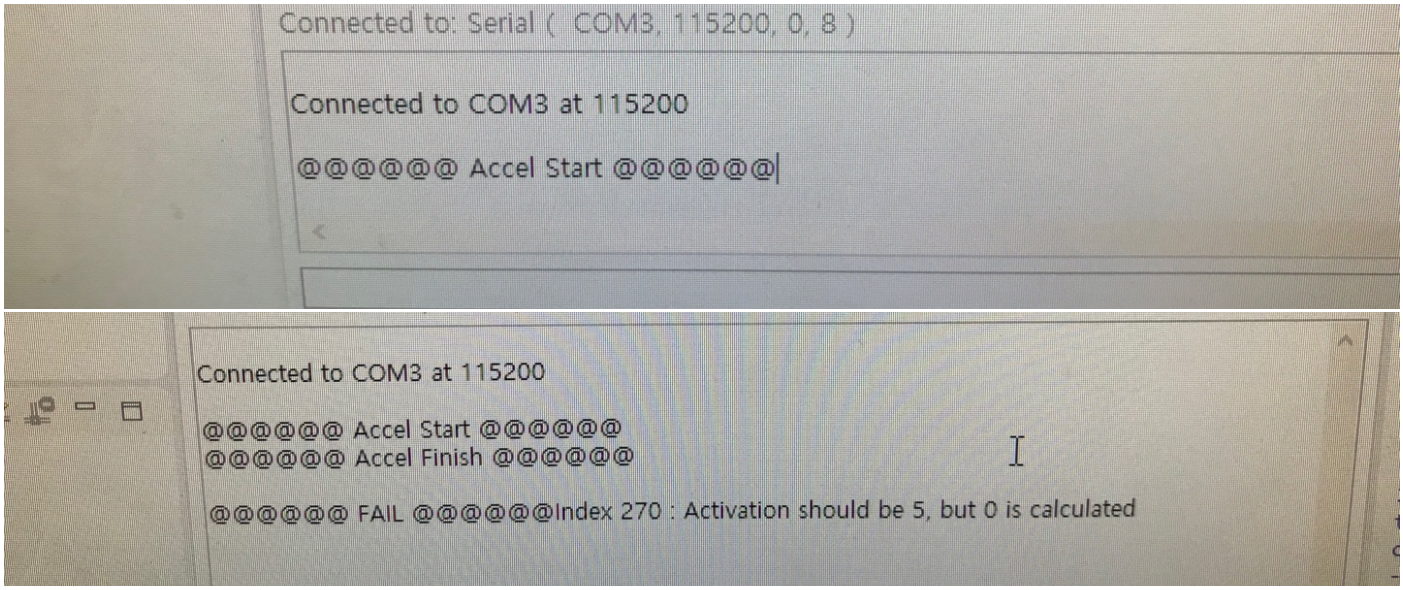
   Total = 60,516 Cycles

Of course we could make the code efficiently by changing the code of earning the maximum value of inner 2 * 2 matrix = mul_write(one component of activation FPGA matrix) or so. But we thought that our code is fine enough to make the execution time fast as we need.

And few things to be modified (not faster the time)

1) we read the data overlapped, since we do whole calculation in loop2. If we set another temporary value and save the read data, we could have reduce the get-access-and-read-data-overlapped and as a result we can reduce the time.
2) make convolution first, then save the data to another register, Then implement max pooling. It would also reduce the reading address but we need more WRITE transition which would make module slower.

-Appendix (Failed moments)



we failed several times. The first one is that we calculate the wrong indexes thus state couldn't go to write and finish section. The second one is we do wrong slicing (fixed point representation). Therefore the output is not matched with intended one.

▶ 5. References

1) AMBA® 3 AHB-Lite Protocol v1.0 Specification by www.arm.com
2) s-space.snu.ac.kr/bitstream/10371/122933/1/000000009022.pdf
3) http://recipes.egloos.com/4991780
4) https://m.blog.naver.com/PostView.nhn?blogId=esoclab&logNo=20174360379&proxyReferer=https%3A%2F%2Fwww.google.com%2F
5) http://egloos.zum.com/donghyun53/v/4087409
6) https://en.wikipedia.org/wiki/Wiki