Design of a parameterized CNN accelerator

Cheryl (Yingqiu) Cao
2022-08-06

Goal

To design and to implement a systolic array of MACs, double buffers for the ifmap, weights and ofmap, address generation logic for each buffer, and the top level state machine that orchestrates the flow of data to produce desired outputs.

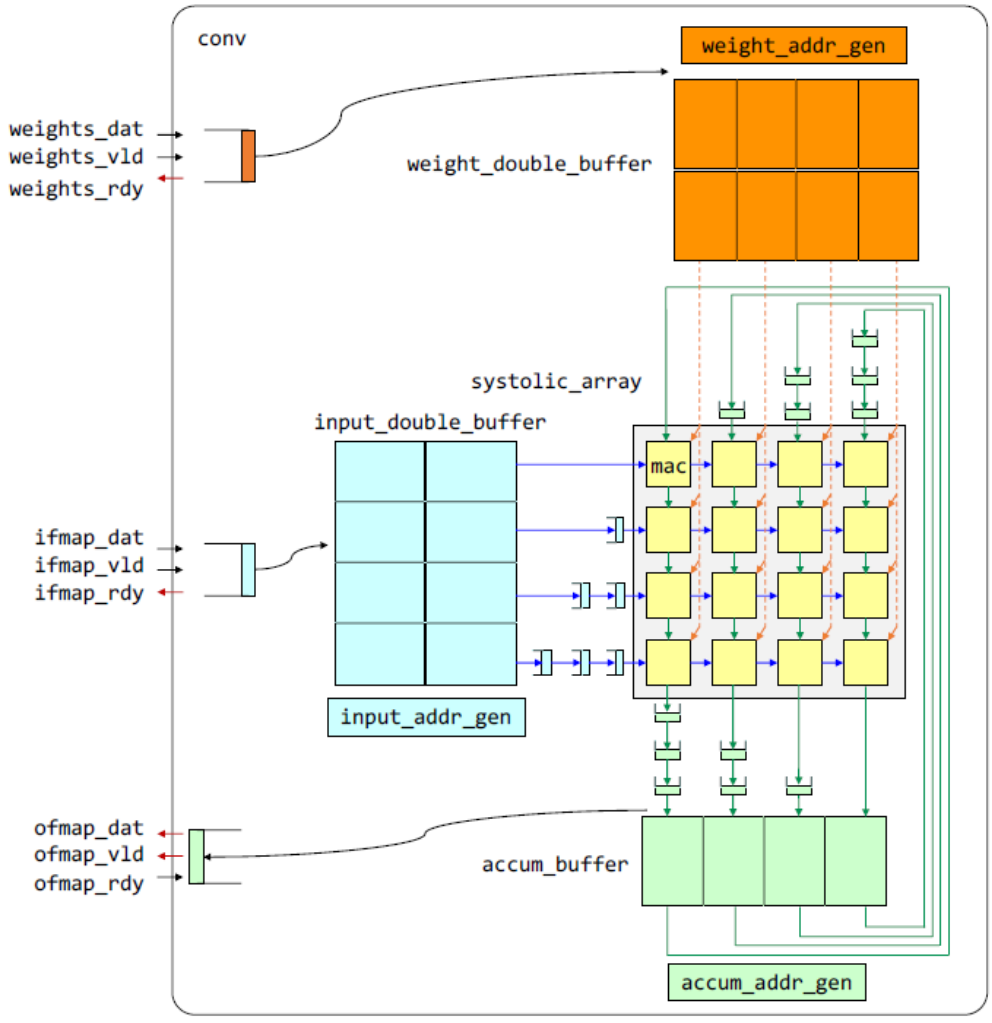
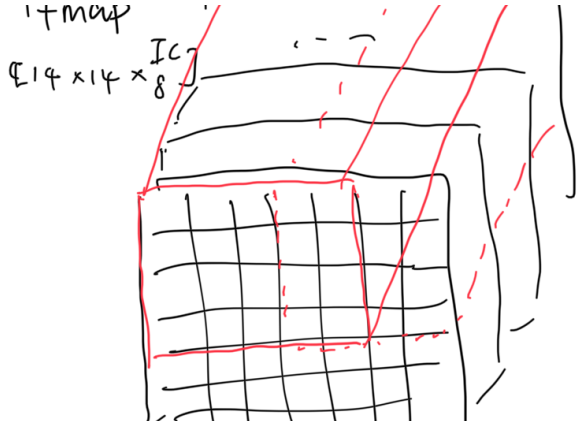


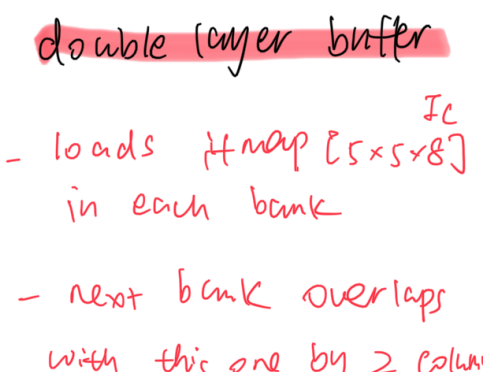
Figure 1: CNN accelerator block diagram.

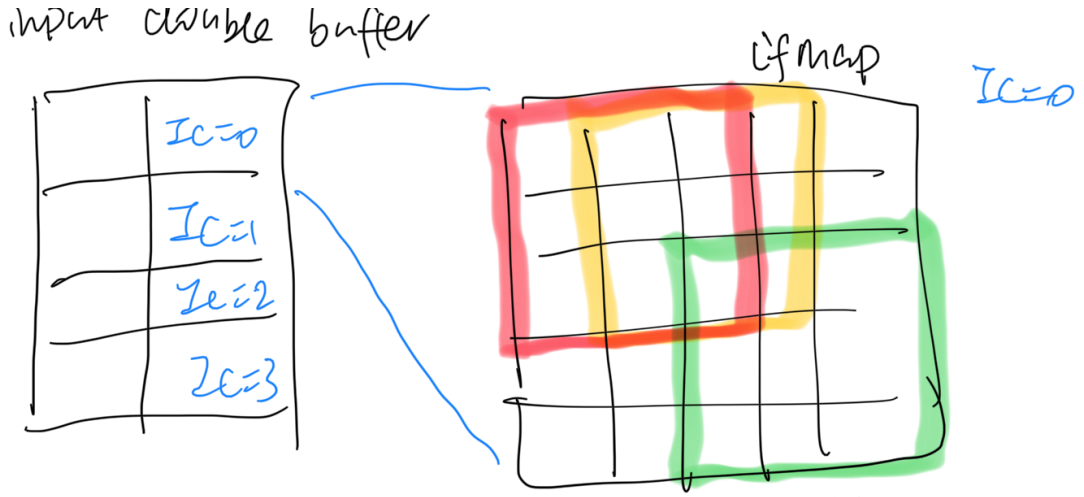
Analysis

input double buffer

ifmap buffer saves IC1 = 2, IY0 = 5, IX0 = 5, IC0 = 4 in each bank
There are a total of OY1 * OX1 * OC1 = 4 * 4* 4 = 64 banks

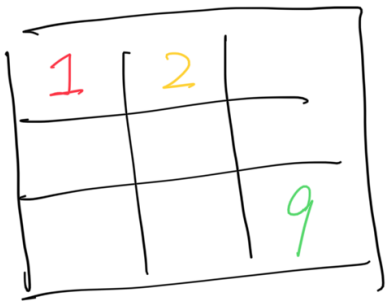






○ 框的大小和 ifmap 3×3一样

○ 框的个数和 feature map Size 一样



feature map

W(:,:, Ic=0, Oc=x)

为了得到 3×3 的 ofmap,

只 red 框里的 ifmap data

需要 convolve W(0,0,:,:)

input buffer 输出到 MAC array 的顺序是

红框 I(0,0,0) (1,0,0) (2,0,0)

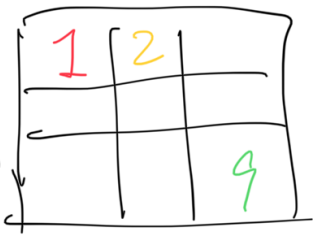
(0,1,0) (1,1,0) (2,1,0)

(2,2,0)

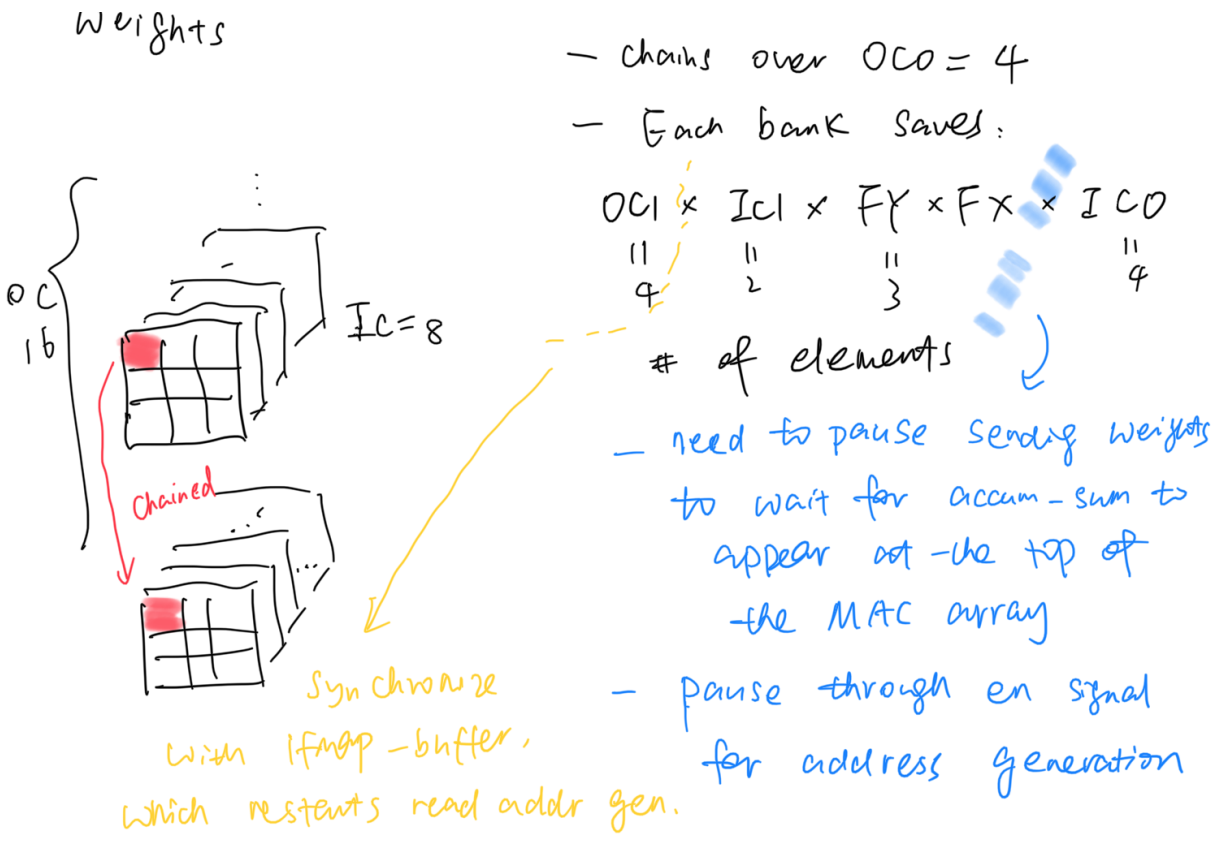
黄 → 绿 I(2,2,0) (3,2,0), (4,2,0)

(4,4,0)

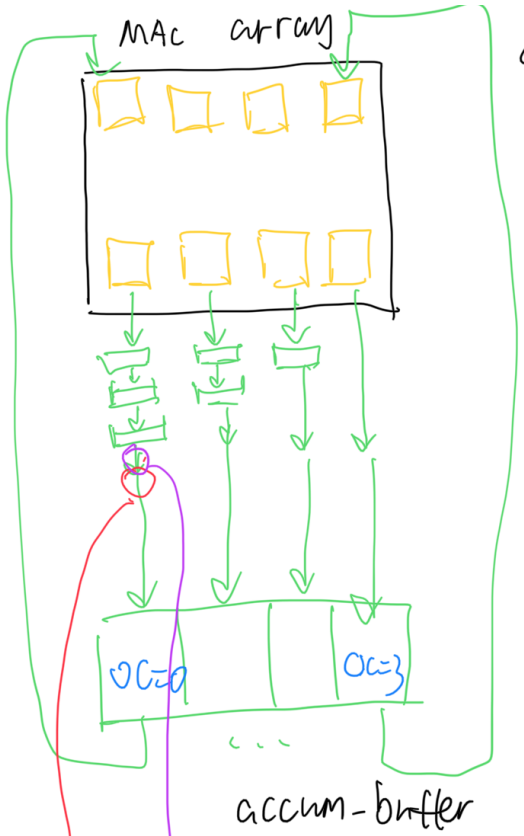
ofmap



Weight double buffer



Accum_buffer

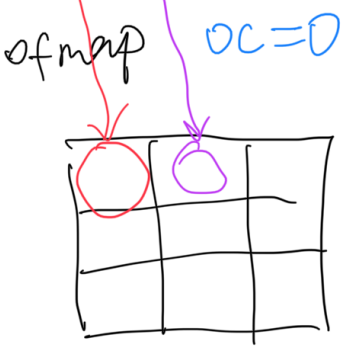


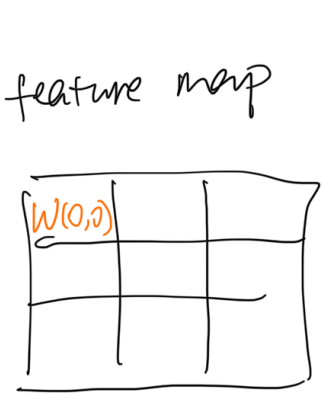
△ first output is a partial sum of O(0,0,OC=0) for IC=0:3

△ 2nd output is a partial sum of O(1,0,OC=0) for IC=0:3

△ After 9 (ofmap size) cycles, all elements in ofmap convolved with W(0,0,IC=0:3,OC=0)

△ "feed" accum_buffer results back to MAC array.





The 1st output is O(0,0,oc = 0) for w(0,0), IC = 0:3
The 2nd output is O(1,0, oc = 0) for W(0,0).
...
The 9th output (9 = ofmap size)
is O(2,2, oc = 0) for W(0,0)

The 10th output is O(0,0, oc = 0) for W(1,0) -- the 1st output needs to be added to this from the accum_buffer.

Once we iterate through one bank of weight buffer,we are done for O(0:2, 0:2, oc = 0:3) for IC = 0:3.
- next step is to feed Ofmap back through the accum_buffer.
- The weight bank will cover W(:,:, oc = 0:3) for IC = 4:7.

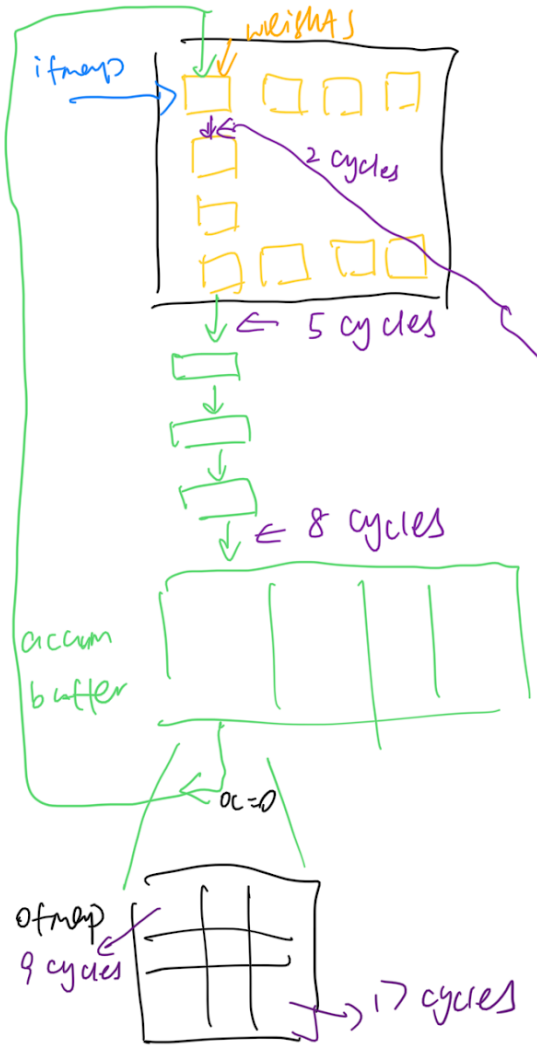
After that we need to reset accum_buffer because OC will move to 4:7.

After we iterate through all weight banks in the weight buffer, one blocking of ofmap is done -- O(0:3, 0:3, oc = all).

DRAM then reads the next block from the original ifmap.

Synchronization of the MAC array and the double buffers

- See the figure below for general analysis



t=-1 enable read weight_addr_gen

t=0 W(0,0,ic=0) weights appears at the top
→ enable ifmap_addr_gen

t=1 ifmap (0,0, oc=0, ic=0) appears
Fx=0, Fy=0
W(0,0,ic=2) appears

t=2 partial sum for O(0,0, oc=0, ic=0) appears here

t=8 - partial sum appears at the input of accum_buffer
// Ico+oco
- enable addr_gen for accum_buffer
- write / or enque
- W(1,0,ic=0) appears

t=9 - can dequeue - ifmap(2,2,oc=0) at input
- enable ifmap

t=10 - partial sum appears at the top of MAC array
- ifmap(0,0, oc=0, ic=0) appears
Fx=1, Fy=0

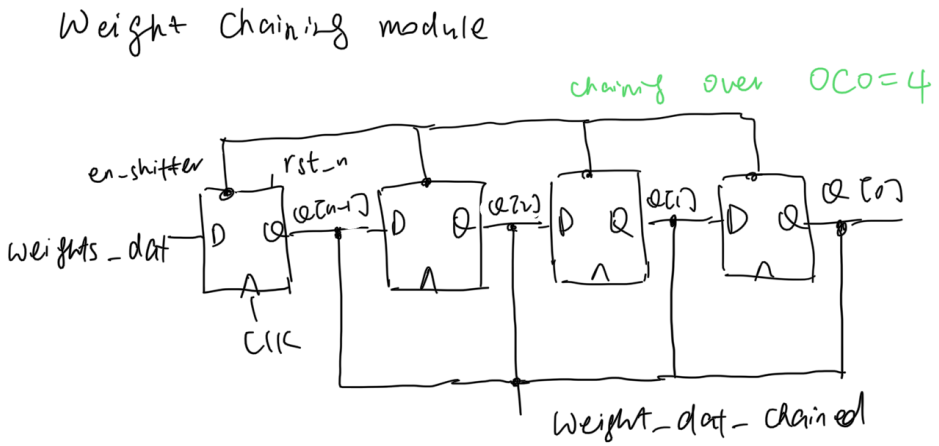
- accum-buf 1st element ready to deque w/ 9 cycles
  accum-cycle = ICO + OCO + 1 = 9
- ifmap iteration over 1 weight takes 9 cycles
  ifmap-cycle = OYo x OXo = 9
- if ifmap-cycle is larger(≥) T = max(accum-cycle, ifmap-cycle)
  - pauses weights longer for T - ICO = 9 - 4 = 5 cyles
  - if accum_buf_cycle is longer:
    - pauses ifmap_read_addr_gen for T - ifmap_cycle
    -

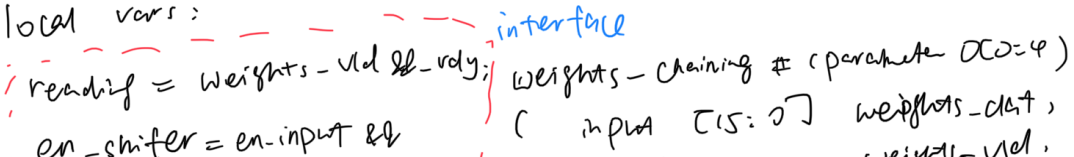
- Synchronization case 2: ifmap_cycle >= accum_cycle
    - It takes longer to iterate through the ifmap elements for weight (Fx = 0, Fy = 0), than for the 1st partial sum ofmap(0,0, IC = 0:3) to get out of the MAC array
    - need to wait to feed back the accum_buf to MAC array

Design

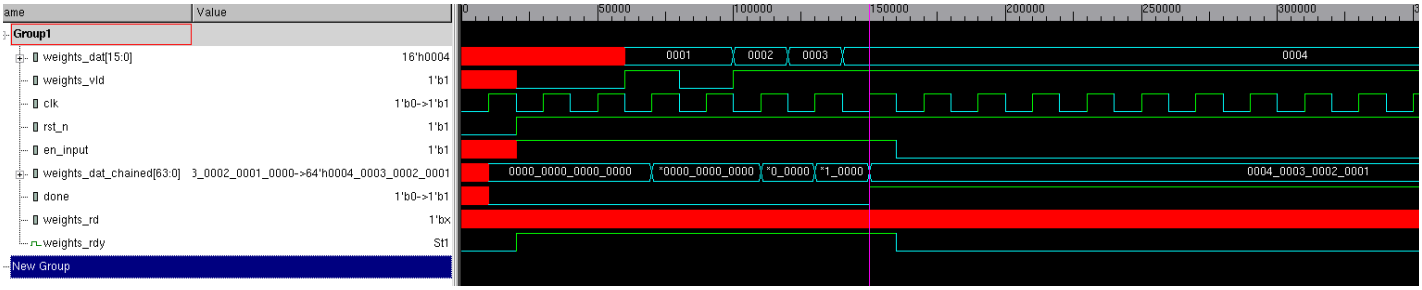
input/weight chaining module

- takes ifmap_data from the testbench, chains them along IC0 = 4, outputs them to the input_double_buffer
- takes weights_data from the testbench, chains them along OC0 = 4, outputs them to the weights _double buffer
- The following figure shows the design for the weight chaining module.
    - for the input changing, replace “weights” with “input”, replace “OC” with “IC”.



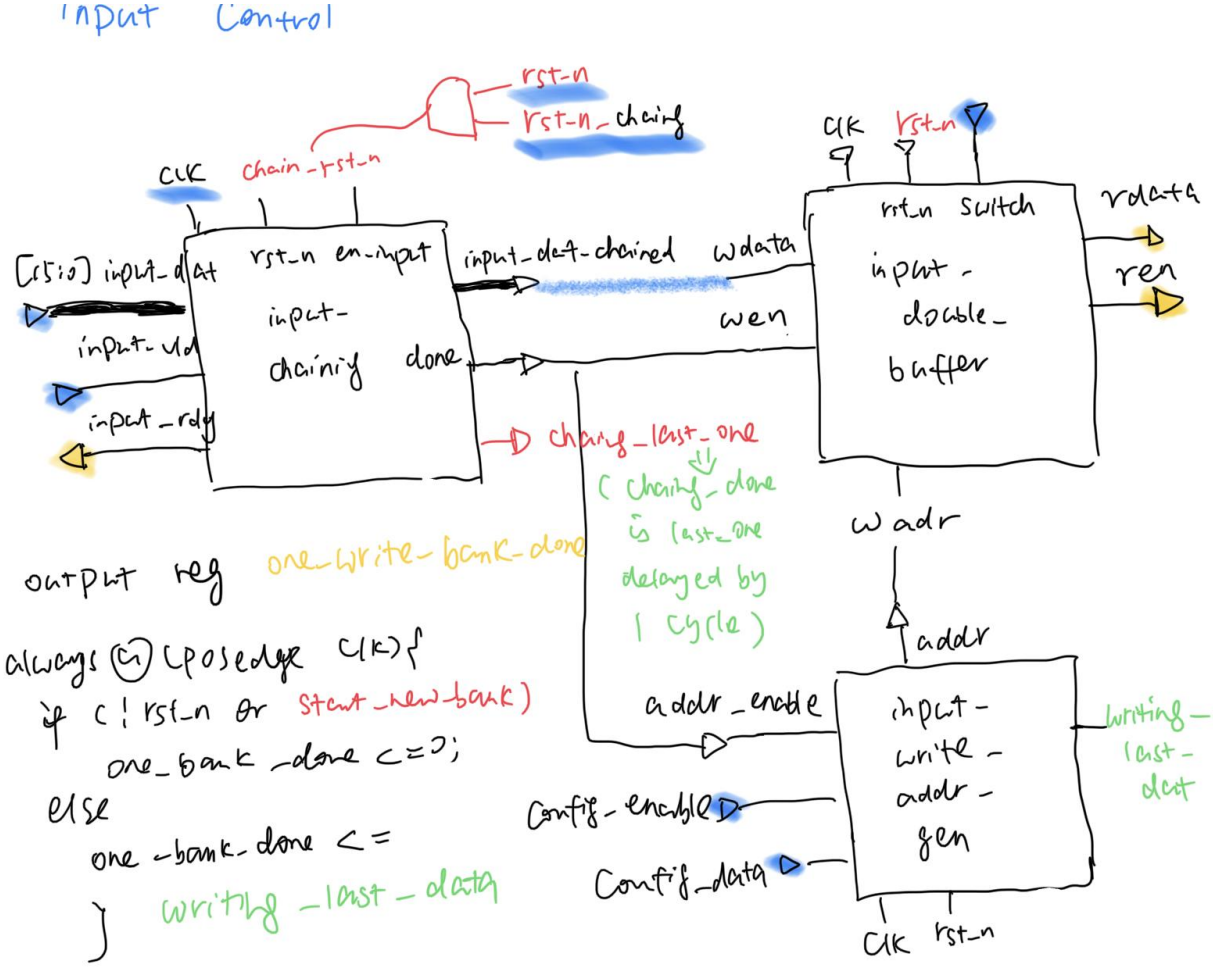


- The simulation waveform is as below
    - “done” goes high when chained_data appears at the output, and stays high until cleared by the state-machine/controller.

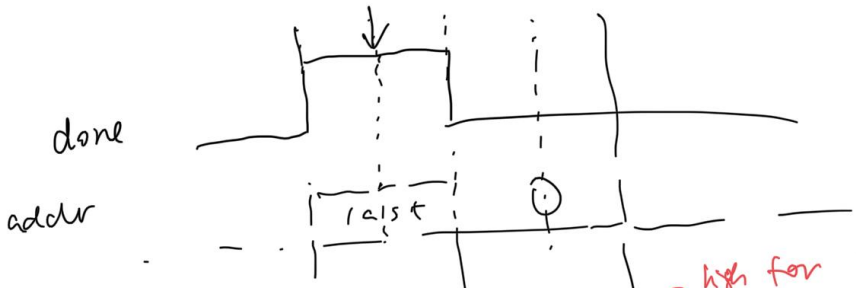


ifmap input control

- Takes ifmap data from the test bench
- sends ifmap data from the double buffer to the MAC array
- write_bank_count tracks the # of banks we have finished writing to.

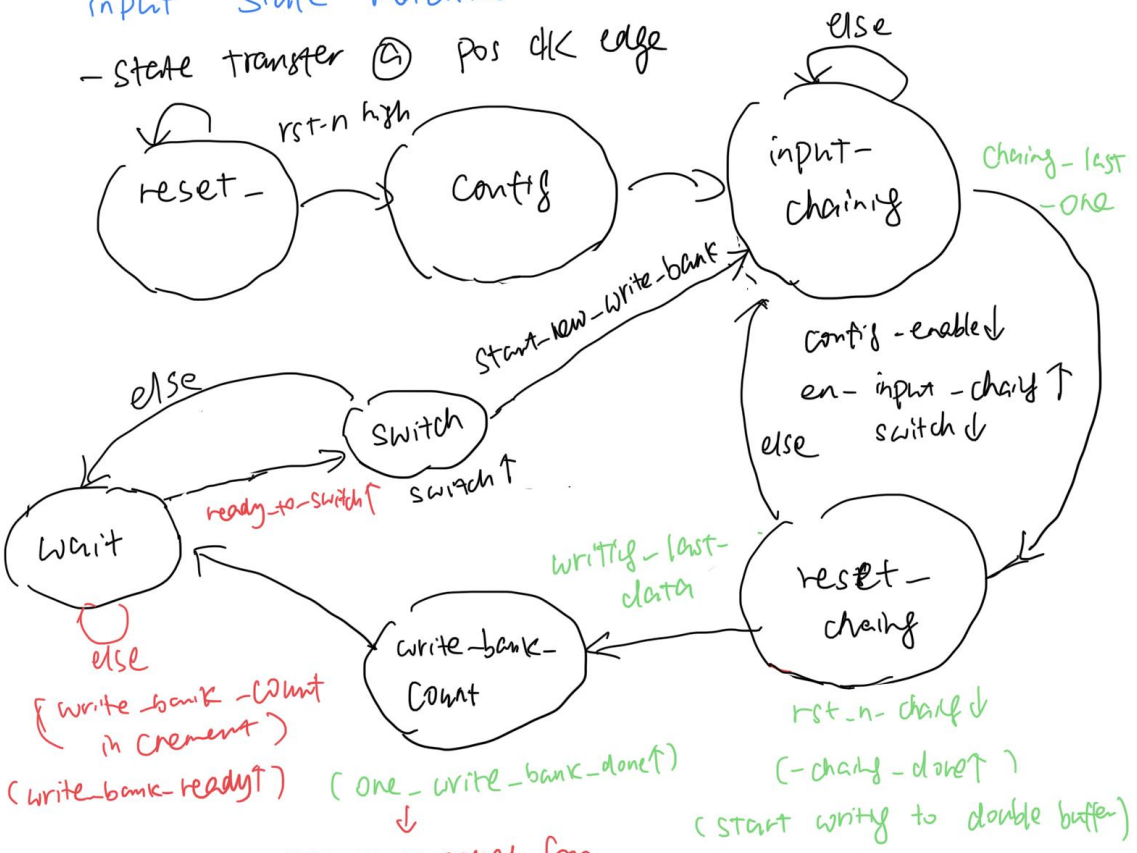


- we want "done" to stay high for only 1 cycle
- therefore, need to: reset "chaining" module immediately
  - to clear "done" flag
  - to clear SZPO
  - to block additional input



input state machine

- state transfer @ pos clk edge



use as en signal for

write_bank_counter: max = 0Y1 x 0X1

```
always @(pos clk){
    if (!rst_n or switch)
        write_bank_ready <= 0;
    else if (one_write_bank_done)
        write_bank_ready <= 1; }
```

stays 1 during wait/switch states

logic read/write_bank_ready_to_switch;

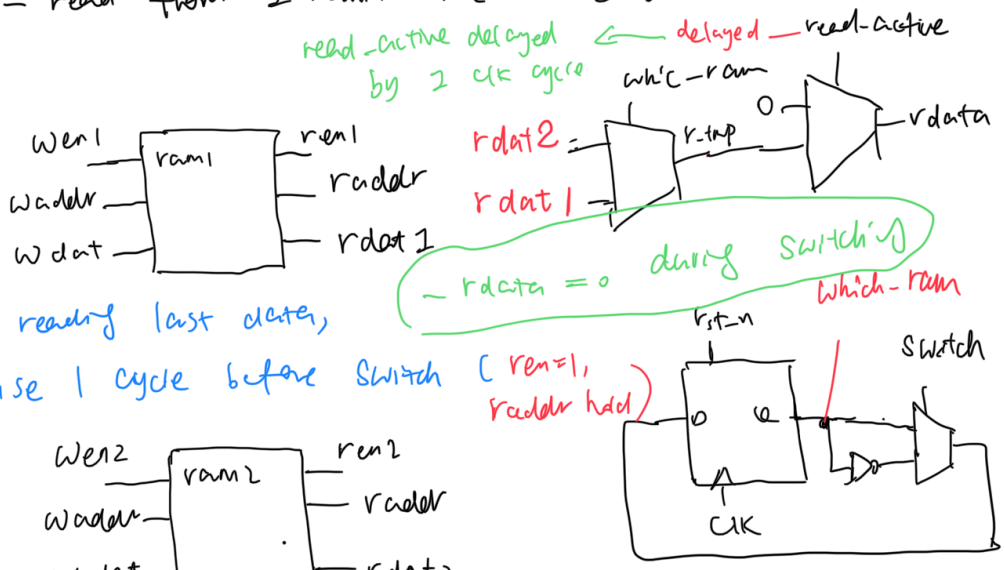
logic ready_to_switch;

double buffer module

double buffer

- read from 1 ram, write to the other

read_active delayed ← delayed — read_active by 1 clk cycle



- rdata = 0 during switching which_ram

- After reading last data, pause 1 cycle before switch (ren=1, raddr hold)

- switch bank goes 1 at neg clk cycle
- switch happens at the next clk edge

local signals:

```
logic which_ram;
logic active;
logic write_active;
logic read_active;
logic wen1, wen2, ren1, ren2
logic [Data_wid-1:0] rdat1,
```

active = rst_n && (!switch),

write_active = active && wen;

read_active = active && ren;

wen1 = write_active && (!which_ram)

wen2 = write_active && (which_ram)

ren1 = ..

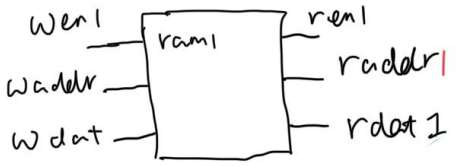
- must rst the double buffer before use
- initialize switch_back to 0 during reset

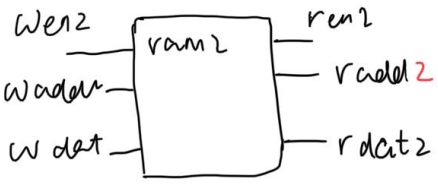
*Notes:

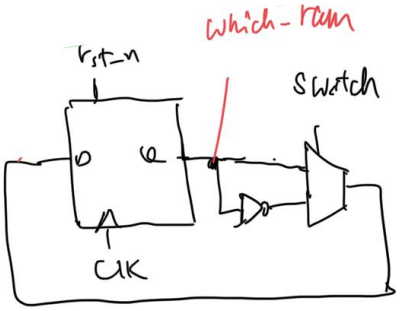
- must reset the double buffer before use
- during reset, initiate switch_banks to 0
- after reading the last data, keep ren =1 w/ the same raddr for another cycle before pulling switch high

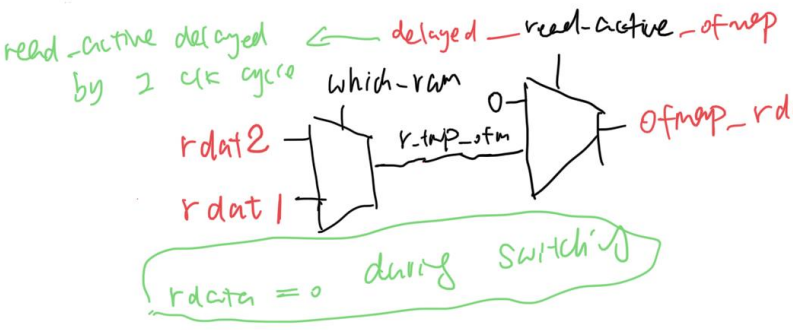
accum double buffer

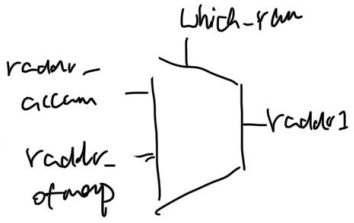
- different from the general double buffer used for ifmap and weight data, accum double buffer has 1 write port but **2 read ports**.
- one read port feeds ofmap partial sum back to the MAC array. The other one sends completed ofmap data out to the testbench.

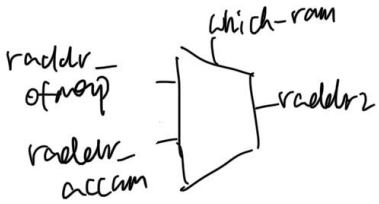


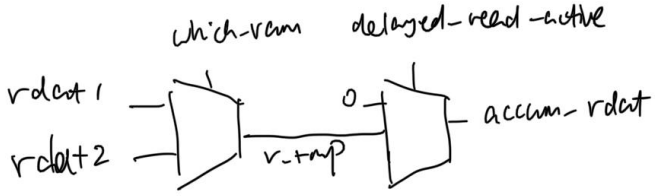


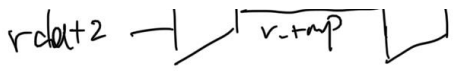


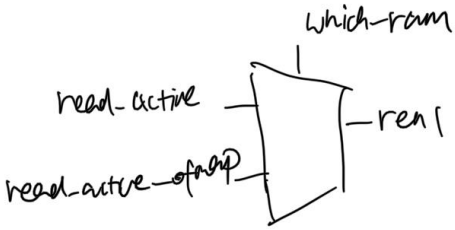


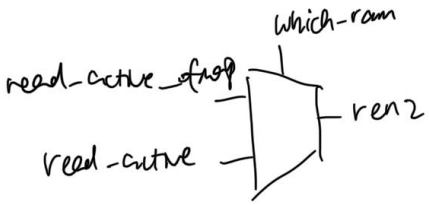












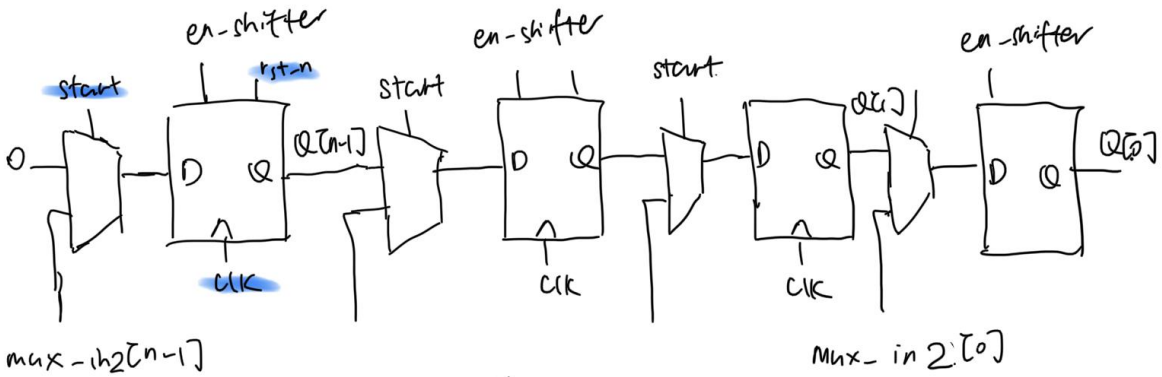
local signals:

```
logic  which_ram;
logic  active;
logic  write_active;
logic  read_active, read_active_ofmap
logic  wen1, wen2, ren1, ren2
logic  [Data_width-1:0] rdata2;

active = rst_n && (!switch);
write_active = active && wen;
read_active = active && ren;
read_active_ofmap = active && ren_ofmap;

wen1 = write_active && (!which_ram)
wen2 = write_active && (which_ram)
```

ofmap_PISO

ofmap_PISO

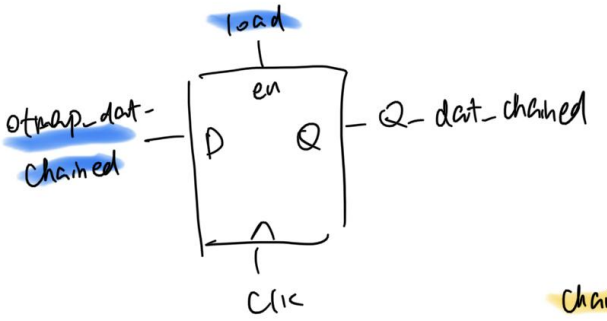
n = OCo



mux_in2[n] = Q_dat_chained [-1 : n×32] (n+1)×32

ready_to_unchain = ofmap_rdy;

en_shifter = en_PISO && ready_to_unchain;



- use en_shifter as en signal for unchain_counter;

chainy_last_one = en_shifter && (counter == (n-1));

local signals

interface

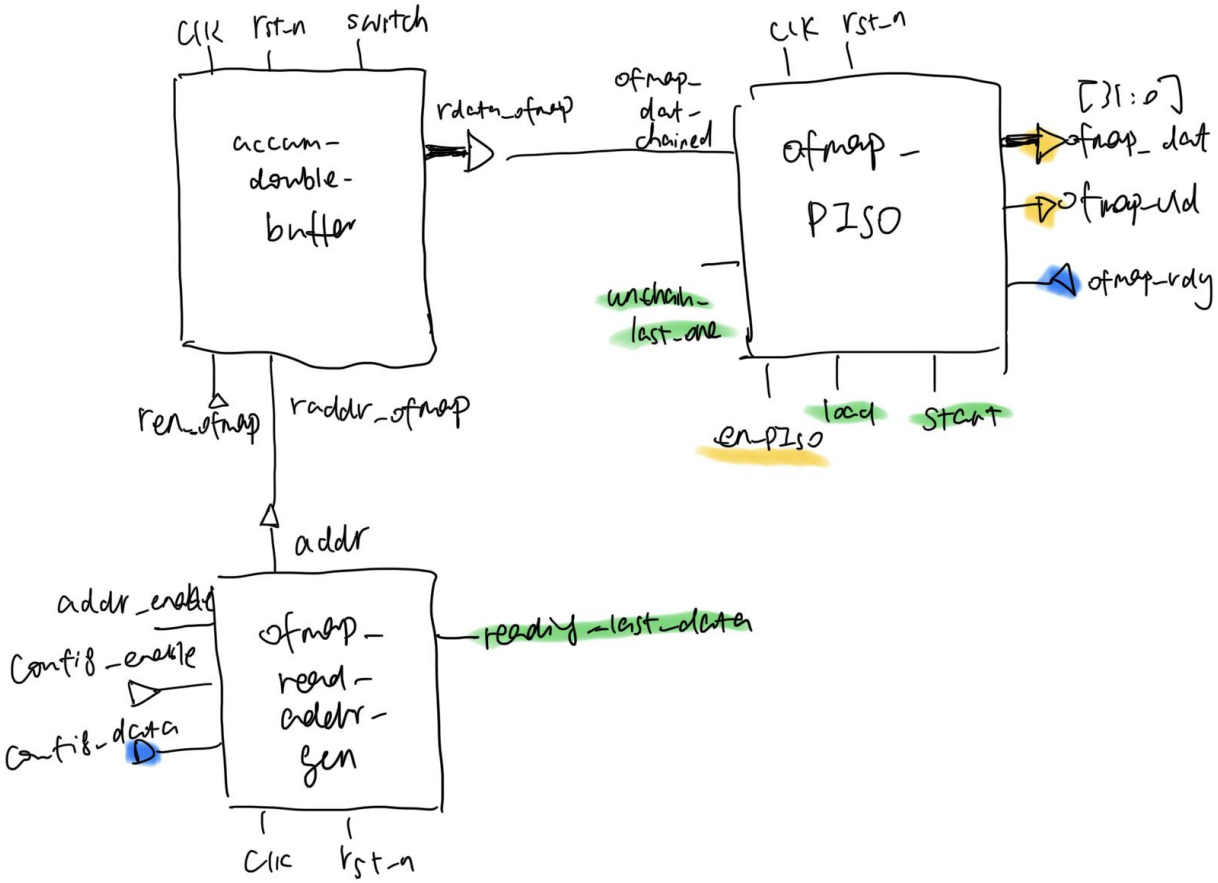
```
ofmap_PISO #( parameter OCO=4 )
(
    input clk,
    input rst_n,
    input en_PISO,
    input load,
    input start,
    output chainy_last_one,
    input [32*OCO-1:0] ofmap_dat_chained,
    input ofmap_rdy,
    output [32:0] ofmap_dat,
    output ofmap_vld
);
```

ofmap_vld → only 1 for one cycle per unchained data

ofmap output control

```
+1 ofmap_FSM.v | 3 ifmap_input_FSM.v
 4 // Author: Cheryl (Yingqiu) Cao
 5 // Date: 2022-03-12
 6
 7 module ofmap_FSM
 8 (
 9   input logic clk,
10   input logic rst_n,
11
12   // for the read_addr_gen
13   input logic last_ofmap_data,     // the unchained data in the last one in this ofmap bank
14   output logic config_enable,
15   output logic raddr_gen_en,
16
17   // for accum_double_buffer
18   output logic ren,
19   output logic siwtch,
20
21   // for ofmap unchaining PISO
22   input logic unchaining_last_one,
23   input logic ready_to_unchain,
24   output logic en_PISO,
25   output logic load,
26   output logic start,
27
28   // for the ofmap read bank counter
29   output logic one_read_bank_done,   // en signal for the counter
30
31   // for the main FSM
32   input logic ready_to_switch,
33   input logic start_new_read_bank,
34   output logic read_bank_ready_to_switch
35
36 );
37
```

ofmap_output_Controller



always @ (pos clk) {

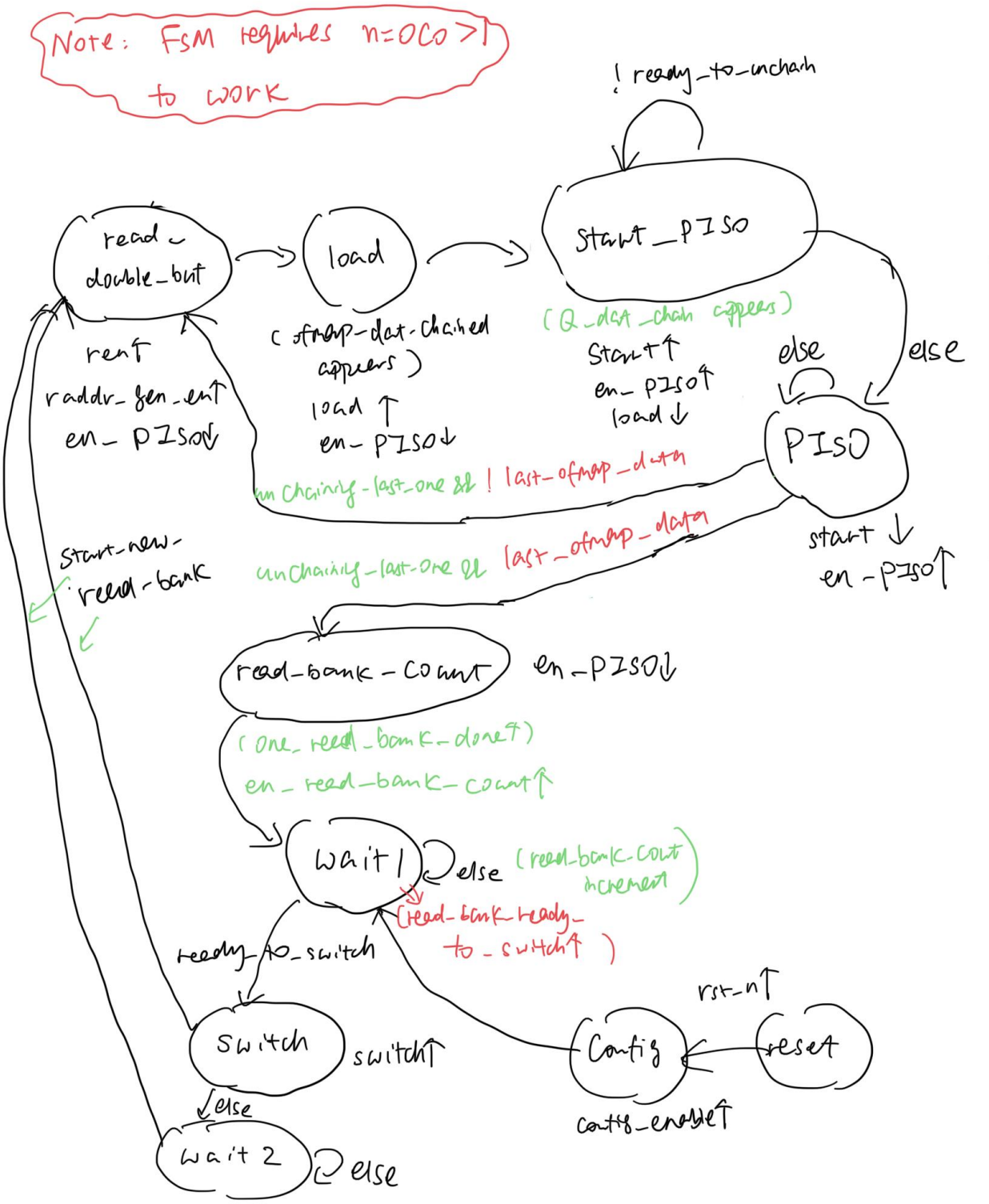
if (! rst or switch)

last_ofmap_data <= 0;

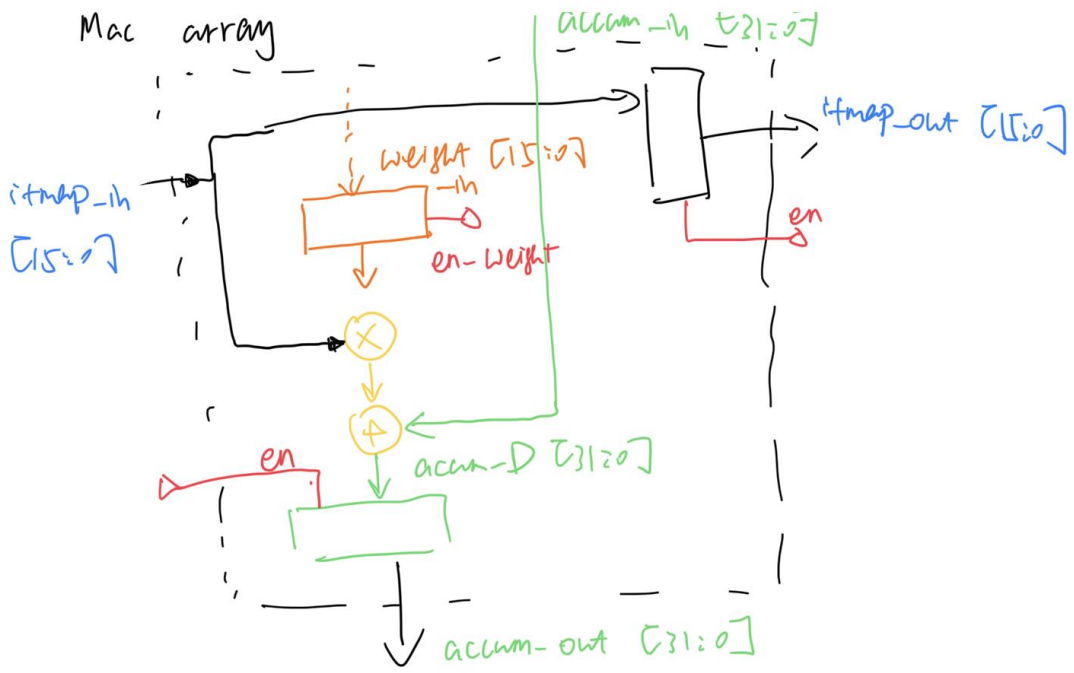
else if (ready_last_data)

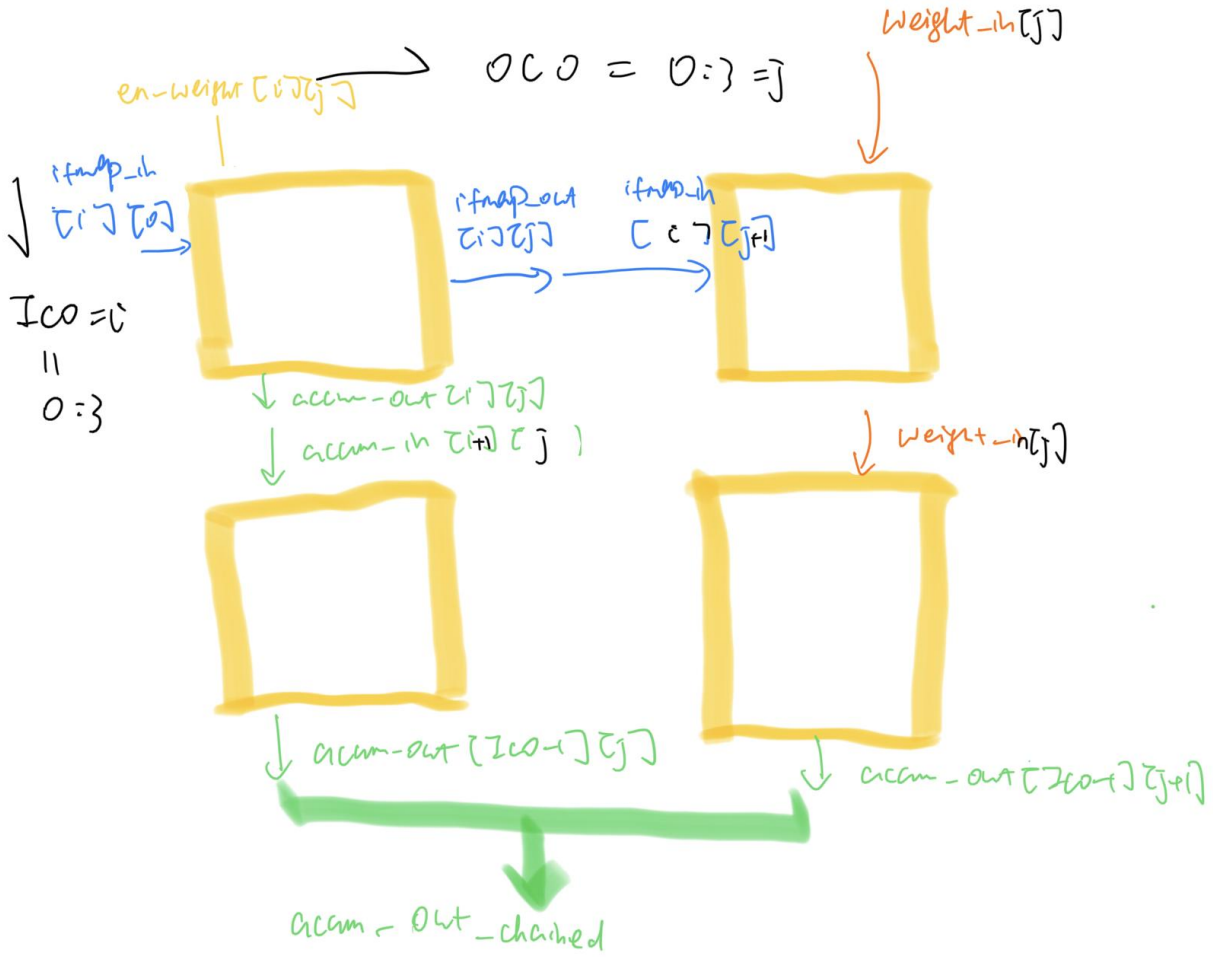
last_ofmap_data <= 1;

}


Note: FSM requires n=0Co >1 to work
! ready_to_unchain
read_double_buf
load
start_PISO
(ofmap_dat_chained appears)
load↑
en_PISO↓
(Q_dat_chain appears)
start↑
en_PISO↑
load↓
else
else
PISO
rent
raddr_gen_en↑
en_PISO↓
unchaining_last_one && ! last_ofmap_data
start ↓
en_PISO↑
start_new_read_bank
unchaining_last_one && last_ofmap_data
read_bank_count
en_PISO↓
(one_read_bank_done↑)
en_read_bank_count↑
wait1
else
(read_bank_count increment)
(read_bank_ready_to_switch↑)
ready_to_switch
rst_n↑
switch
switch↑
Config
reset
config_enable↑
else
wait 2
else

Mac array


Mac array
accum_in [31:0]
ifmap_out [15:0]
weight [15:0]
ifmap_in
[15:0]
en
en_weight
en
accum_D [31:0]
accum_out [31:0]



ifmap_in[i][0] = ifmap_dest_chained[16(i+1)-1 : 16i]

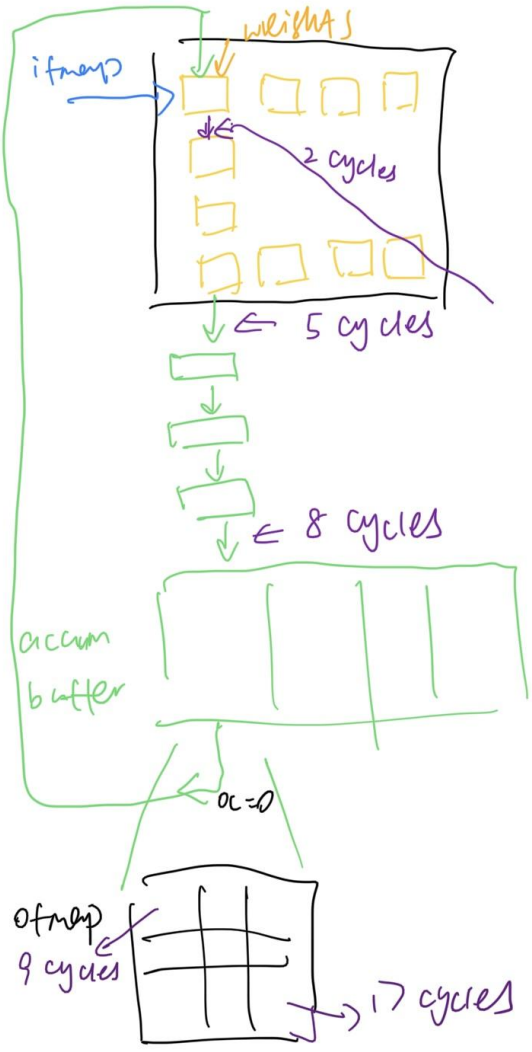
weight_in[j] = weight_dest_chained[16(j+1)-1 : 16j]

accum-in[0][j] = accum-in_chained[32(j+1)-1 : 32j]

accum-out[Ico-1][j] = accum-out_chained
[32(j+1)-1 : 32j]

Sync of MAC array

MAC array sync



t=0 weight_double_buffer: ren↑
ifmap_double_buffer: ren↓

t=1 (weight appears @ top of mac)
W(0,0,ico=0)

weight_fifo_enq, ↑
ifmap_fifo_enq ↓
mac_more: en_weight00 ↑
ifmap_double_buffer = ren↑
accum_double_buffer:
if (!1st_Fx_Fy_cyle) ren↑

t=2 (ifmap/accum_in data appears @ top/left of mac)
weight_fifo_enq ↑
mac_more: en_weight00↓
ifmap_fifo_enq ↑
accum_in_fifo_enq
<= (~1st_FxFy_cyle)
weight_double_buffer: ren↑
ifmap_double_buffer: ren↑
accum_double_buffer:
<= (~1st_FxFy_cyle)

t = Ico (pause weight ready)
weight_double_buffer: ren↓

$t = ICO + 1$

weight_fifo_enq ↓

mac_move: en_weight00 ↓

accum_out_fifo_enq ↓

$t = ICO + 2$

(accum_out appears @ top of fifo)

accum_out_fifo_enq ↑

$t = ICO + OCO + 1 = 9$

(accum_out appears @ top of double_buffer)

accum_double_buffer = wen ↑

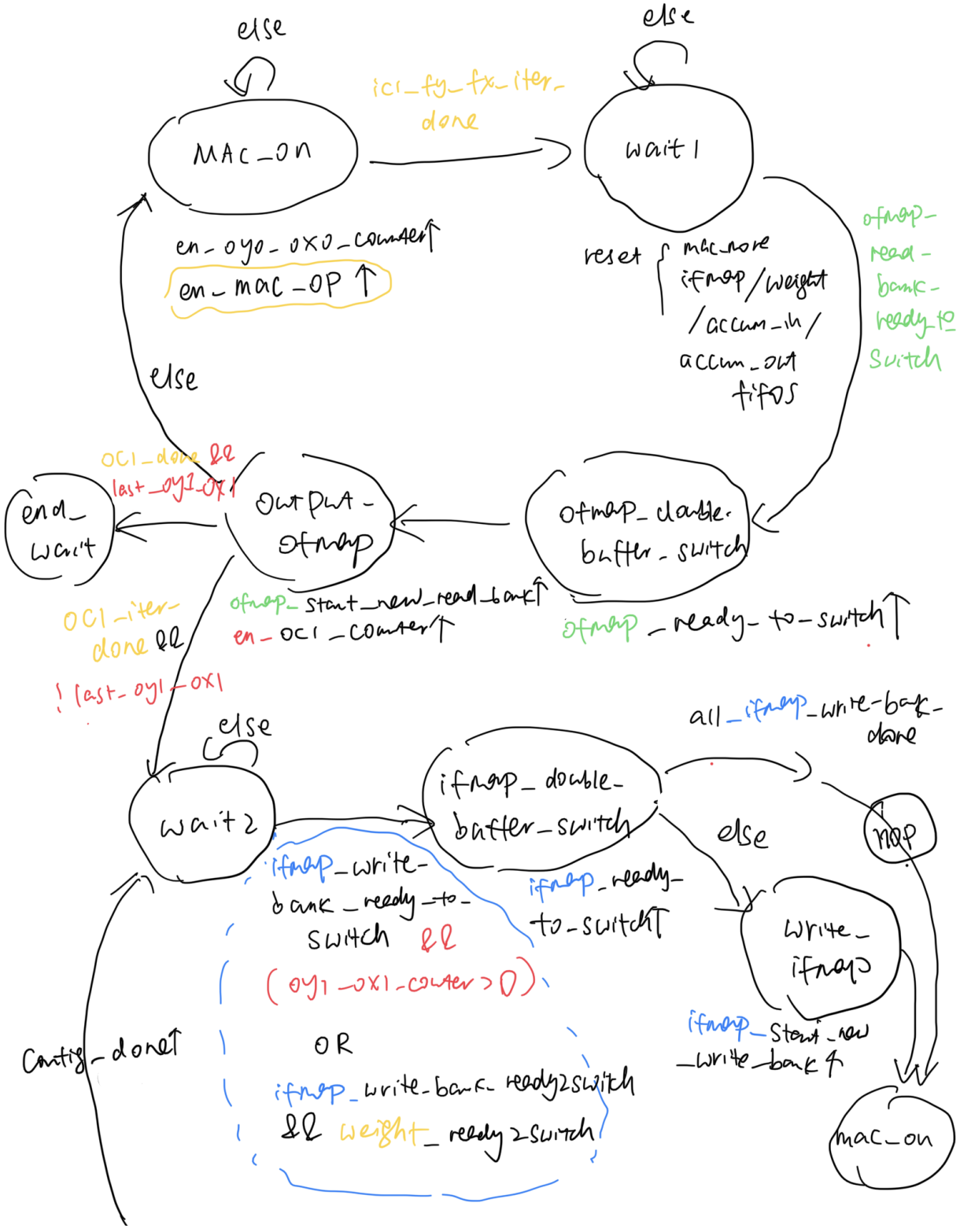
```
assign en_ic1_fy_fx_counter = (oy0_ox0 == max);
assign ic1_fy_fx_iter_done = (ic1_fy_fx == max) && en_ic1_fy_fx_counter;
assign oc1_iter_done = (oc1 == (OC1-1)) && en_oc1_counter;
```

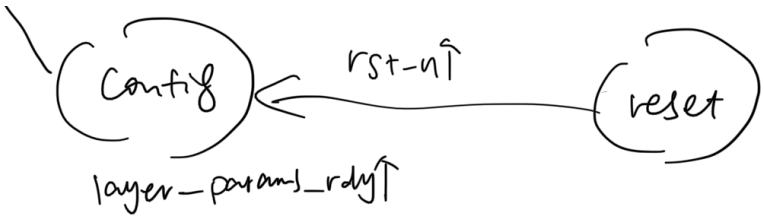
```
assign en_oyi_ox1_counter = OC1_iter_done
```

⇒ for loop

```
assign all_ifmap_write_bank_done = (write_bank_count == oy1_ox1);
assign weight_ready_to_switch = ifmap_ready_to_switch && (oy1_ox1_counter == 0);
```

weight ⇓ only 1 weight bank


else
MAC_on
ic_fy_fx_iter_done
else
wait 1
en_oyo_oxo_counter↑
en_mac_OP ↑
reset { mac_done ifmap/weight /accum_in/ accum_out fifos
ofmap_read_bank_ready_to_switch
else
oc1_done &&
last_oy1_ox1
end_wait
output_ofmap
ofmap_double_buffer_switch
ofmap_start_new_read_bank↑
en_oc1_counter↑
ofmap_ready_to_switch↑
oc1_iter_done &&
! last_oy1_ox1
else
wait 2
all_ifmap_write_back_done
ifmap_double_buffer_switch
else
nop
ifmap_write_bank_ready_to_switch &&
(oy1_ox1_counter > 0)
OR
ifmap_write_bank_ready2switch
&& weight_ready2switch
ifmap_ready_to_switch↑
write_ifmaps
ifmap_start_new_write_bank ↑
Config_done↑
mac_on



- Assuming OY0 * OX0 >= OC0 + IC0 + 1
- oy0_ox0, a counter that increments each clk cycle
  - range is [0, OY0_OX0]
  - Use **OY0_OX0 +1** = 3*3 +1 = 10 as **period**
- ic1_fy_fx, a counter that increments each OY0_OX0 iteration completes
  - range is [0, IC1_FY_FX], saves the # of OY0_OX0 cycle that was completed

For mac_more module:

Control signal	en_weight00	weight_fifo_enq	ifmap_fifo_enq		accum_in_fifo_enq		accum_out_fifo_enq	
On cycle : oy0_ox0	[1,1] oy0_ox0_first_cycle	[1,IC0]	[2, OY0_OX0]	[0,0]	[2, OY0_OX0]	[0,0]	[IC0 + 2, OY0_OX0]	[0, IC0]
Condition: ic1_fy_fx	[0, IC1_FY_FX-1] ic1_fy_fx_not_last_cycle		[0, IC1_FY_FX-1] ic1_fy_fx_not_last_cycle	>0 ic1_fy_fx_not_zero_cycle	[1, IC1_FY_FX-1]	> 1	[0,IC1_FY_FX-1] ic1_fy_fx_not_last_cycle	>0 ic1_fy_fx_not_zero_cycle

For weight_double_buffer module:

Control signal	ren	weight_read_addr_gen_en
On cycle : oy0_ox0	[0, IC0-1]	
Condition: ic1_fy_fx	[0, IC1_FY_FX-1] ic1_fy_fx_not_last_cycle	

For ifmap_double_buffer module:

Control signal	ren	ifmap_read_addr_gen_en

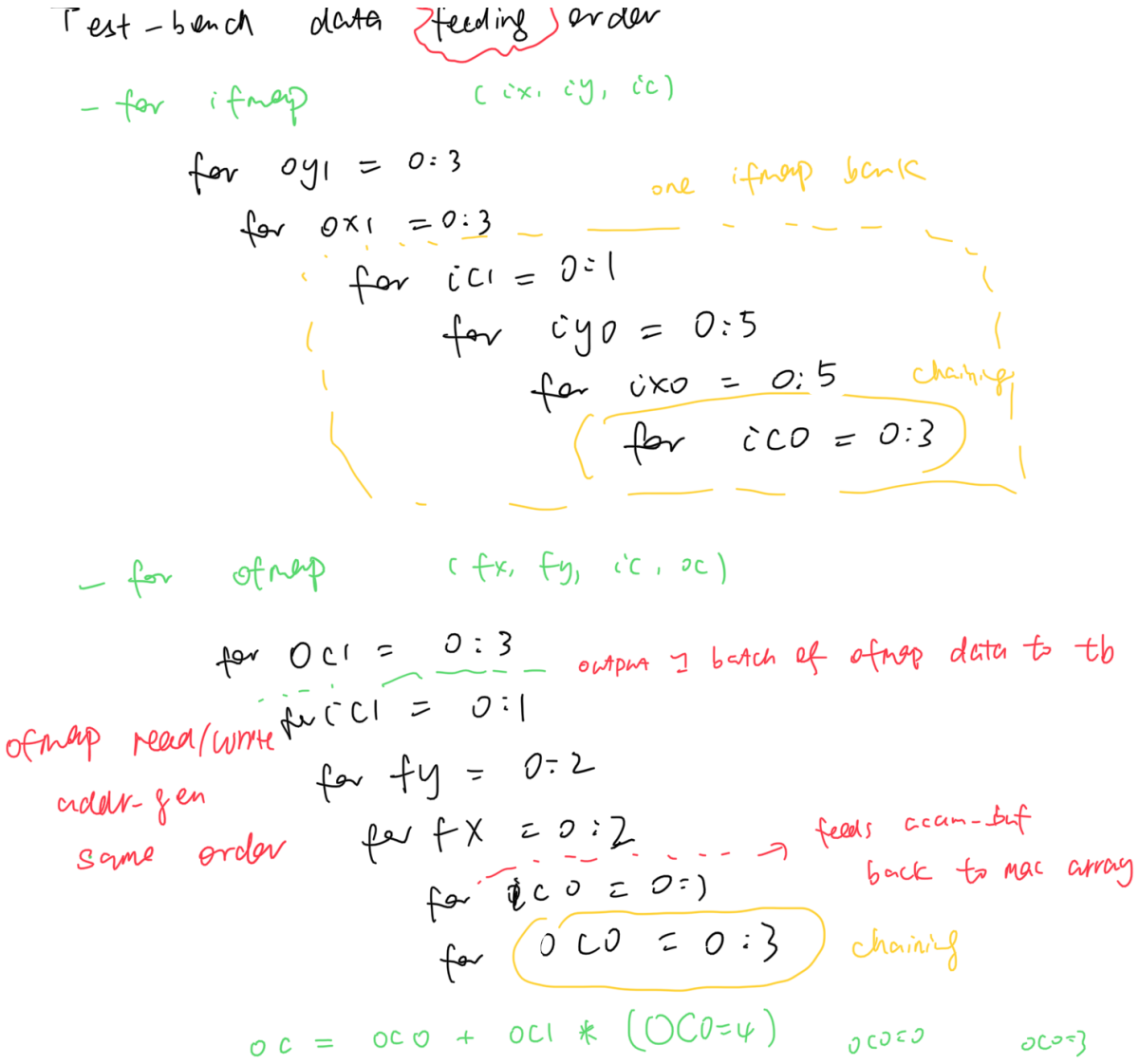
On cycle : oy0_ox0	[1, OY0_OX0] oy0_ox0_not_zero_cycle
Condition: ic1_fy_fx	[0, IC1_FY_FX-1] ic1_fy_fx_not_last_cycle

For accum_double_buffer module:

Control signal	ren	wen	
On cycle : oy0_ox0	[1, OY0_OX0] oy0_ox0_not_zero_cycle	[IC0+OC0+1, OY0_OX0]	[0, IC0+OC0-1]
Condition: ic1_fy_fx	[**1**, IC1_FY_FX-1] ic1_fy_fx_not_zero_or_last_cycle	[0, IC1_FY_FX-1] ic1_fy_fx_not_last_cycle	[1, IC1_FY_FX] ic1_fy_fx_not_zero_cycle

- assign accum_read_addr_gen_en = ren;
- assign accum_write_addr_gen_en = wen;

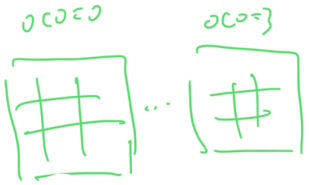
testbench ifmap/weight/ofmap data organization (*needs correction)



oc = oc0 + oc1 * (OC0=4)

- for accum_buffer (ox, oy, oc)

```
for oy0 = 0:2
    for ox0 = 0:2
```



- to re-organize ofmap at tb: (ox, oy, oc)

```
for oy1 = 0:3
    for ox1 = 0:3
        for oc1 = 0:3
            for oy0 = 0:2
                for ox0 = 0:2
                    for oc0 = 0:3
```

ox = ox0 + ox1 x (Ox0 = 3)

Tools

Synopsys VCS for compilation and simulation
Cadence Verdi for waveform visualization and debugging
Synopsys DVE (deprecated by Synopsys during the project ….)

Commands

module unload vcs
module load vcs/O-2018.09-SP

Project Files

Verilog CNN implementation files

Top level

- conv_tiled.v
  - My CNN implementation with MAC array

- conv.sv
  - pseudo CNN implementation
  - directly reads “gold_ofmap” data from file and assigns it to ofmap array

ifmap related

- input_write_addr_gen.v
  - serial port sends data to double layer buffer
  - the addr increases from 0 to IC1*IX0*IY0, because **the data for IC0 = 4 channels are chained together**
  - Each bank saves IC1*IX0*IY0*IC0 elements of ifmap data
  - **input sequence order** implemented by the test bench:

```
for oy1 in range(0,4):
    for ox1 in range(0,4):
        for ic1 in range(0,2):
            for iy0 in range(0,5):
                for ix0 in range(0,5):
                    for ic0 in range(0,4):

                        ix = ix0 + ox1 * FX;
                        iy = iy0 + oy1 * FY;
                        ic = ic1 * ic0
                        ifmap( ix, iy, ic)
```

■

- input_read_addr_gen.v
    - it generates the read address for the input double buffer → **MAC array reads data from the ifmap double buffer**
    - it implements the big for loop in the double buffer level and outputs addr in that order for (IC1, FY, FX, OY0, OX0)
        - this is the iteration required for sum accumulation
        - We **still need to iterate through OC2** for blocking
    - **The double layer buffer saves ifmap (IC1, IX0, IY0), and outputs it for OC1 = 4 times for one block**

- input_chaining.v
    - serial-in-parallel-out
    - chains ifmap data over IC0 = 4, before inputting to the ifmap_double_buffer

- ifmap_input_FSM.v
    - controls the timing for input_chaining and writing operations for the ifmap_double_buffer
    - Outputs one_write_bank_done flag to the MAIN FSM
    - takes an input start_new_write_bank from the MAIN FSM to start the next bank

- ifmap_input_controller.v
    - connects ifmap_chaining, ifmap_double_buffer, input_write_addr_gen and other small logic blocks

Weights related

- weights_chaining.v
    - serial-in-parallel-out
    - chains weights data over OC0 = 4, before inputting to the weight_double_buffer
- weight_addr_gen.v
    - addr generation for the read/write operation of weight double buffer
- weight_input_controller.v
    - connects weight_input_chaining, weight_double_buffer, weight_write_addr_gen and other small logic blocks

Ofmap related

- ofmap_PISO.v
    - unchains ofmap data over OC0 = 4, before outputting to the serial bus of the test bench.

- accum_addr_gen.v
    - generates read/write address for the accum_double_buffer
    - goes from 0 to (OY0*OX0 - 1)

- accum_double_buffer.v
    - Double buffer for the accumulated sum and ofmap data
    - Different from the general double buffer used for ifmap and weights, this buffer has 1 write port but 2 read ports.
    - One read port and one write port uses the same bank to save the partial accum sum. The other port is used to read ofmap data output for the testbench.

MAC related

- mac.v
    - the mac cell
- mac_array.v
    - mac array for IC0 = 4 and OC0 = 4

- en_weight_shifter.v
    - derives the en_weight[ic0][oc0] signal for the mac_array from a single en_weight signal for the 1st mac cell [icc = 0][oc0 =0].

Memory related

- double_buffer.v
    - used in two instances to store ifmap and weights locally
    - in each cycle, can perform one read, one write in parallel

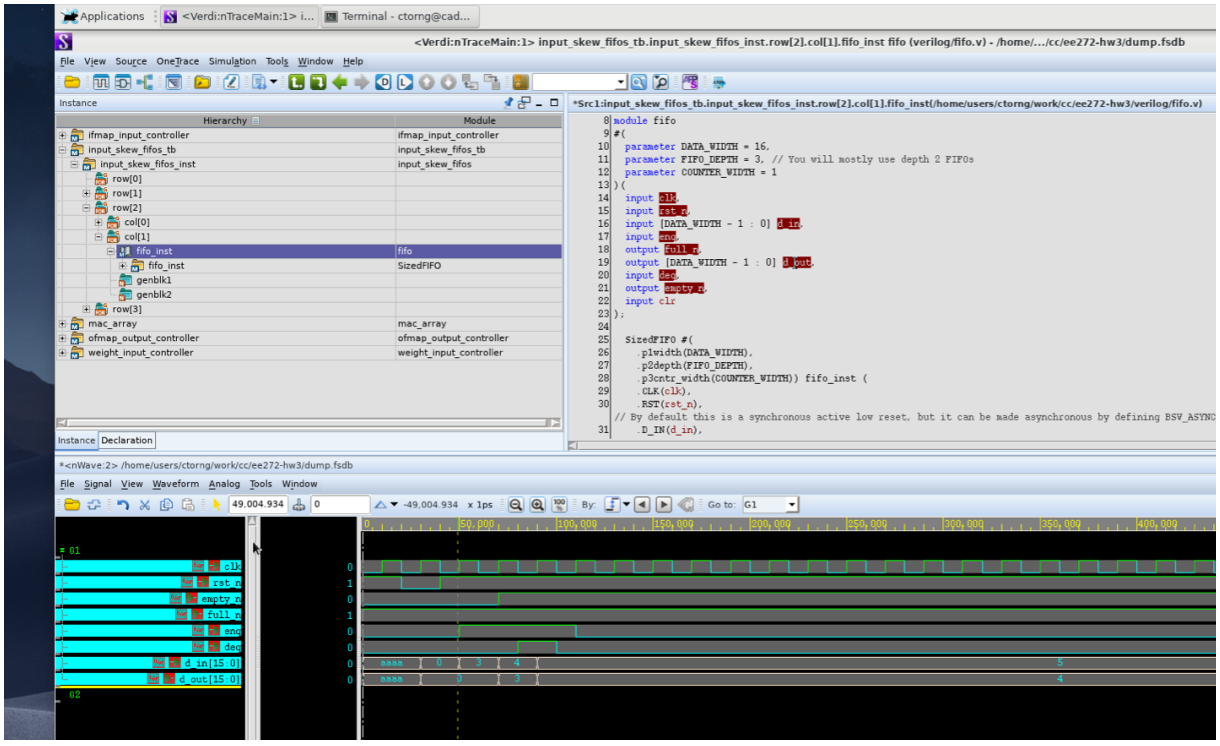
- ram_sync_1r1w.v
    - can read and/or write one word each clock cycle
    - read data becomes available in the next cycle

- wrapped_skew_fifo.v

    - **When en = 1, each row dequeues 1 data until the fifo becomes empty**
    - add an en signal: enqueue and dequeue stays 0 unless en = 1;
    - one enq signal for the entire module: input arrives at the same time from the double buffer
    - Remove deq[i] as an output port: deq[i] will be derived from en and empty_n_w_[i] signals

- accum_out_skew_fifo.v
    - **When en = 1, each row dequeues 1 data until the fifo becomes empty**
    - only 1 en signal for the n-th fifo

- input_skew_fifos.v
    - each row contains a chain of fifos
    - **normally, each cycle, the data moves from the n-th fifo to the n+1 th**
    - 1 pos clk edge after *enque[1]* goes high, data appears at d_w[1][1] -- the output of the first fifo in that row
    - empty_n and full_n signals are c**ombinational**, changes within the same cycle
    - empty_n[row] = empty_n[row][last] : if the last fifo is NOT empty, we can dequeue from the chained fifo
    - full_n[row] = full_n[row][0]: if the first fifo is not full, we can enqueue to it
    - **The fifo output stays unchanged**, w/o active “dequeue” signal



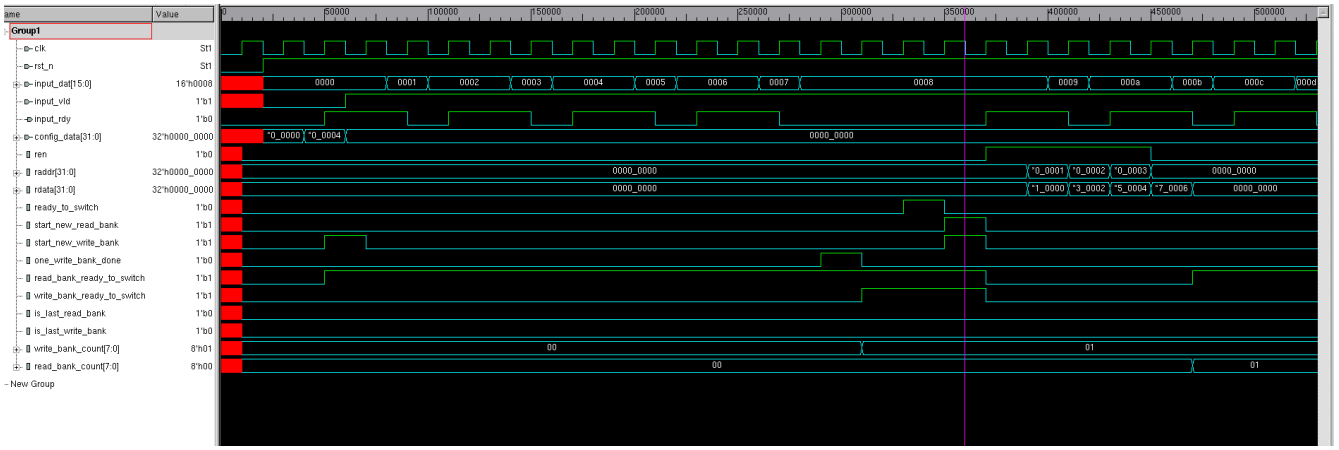
- fifo.v
    - The queue only holds **1D data**.

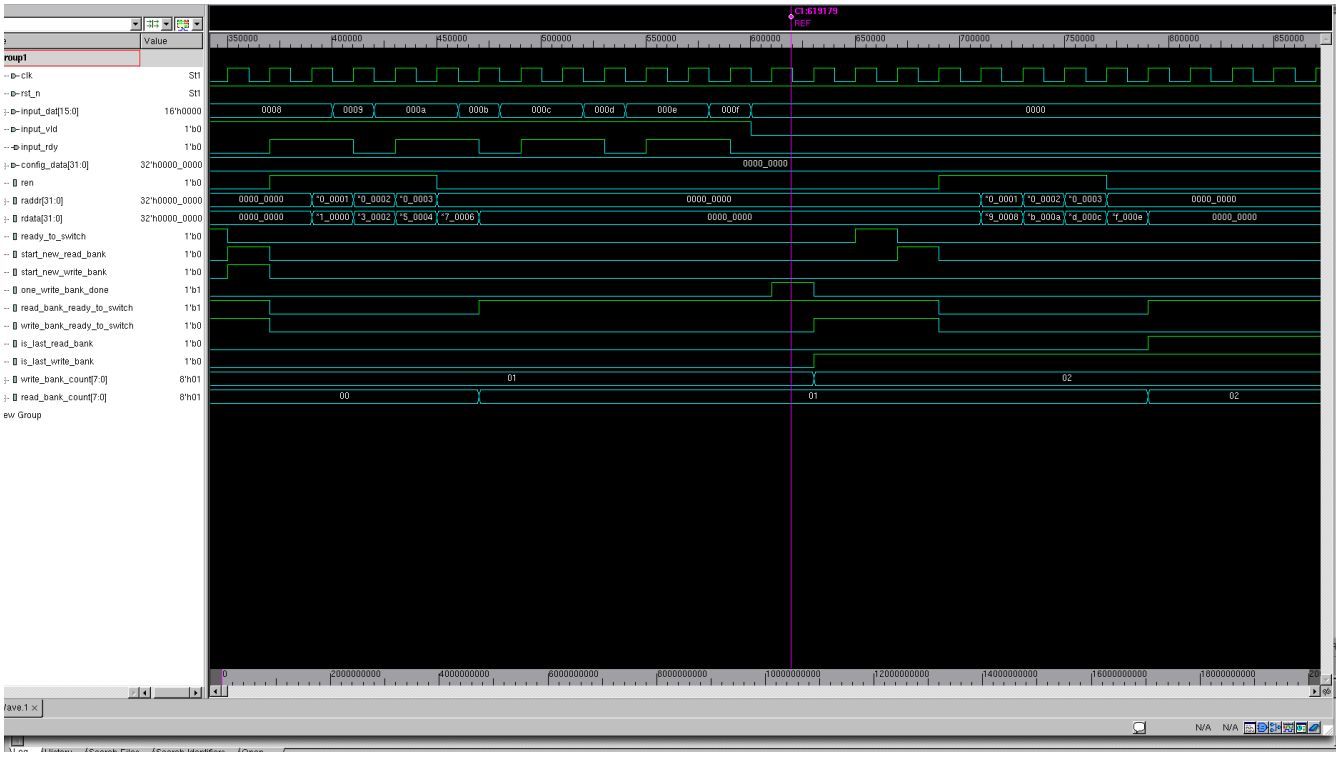
    - What's the COUNTER_WIDTH? The # of elements the queue can save
        - log2(depth-1)
    - it creates a fifo module using the given SizedFIFO.

- SizedFIFO.v
    - it's given to us with copyright to Bluespec, Inc.
    - what's p2depth?
        - The number of elements that can be saved in the FIFO.
        - the FIFO array size is actually (p2depth - 1), the one extra element floats
    - p3cntr_width is the binary counter's width to hold the index of the FIFO elements.
    - head points to the element to be dequeued next
        - can hold (p2depth-1) number of integers, it's like an index
    - tail points to the element that was (just enqueued
    - array saves a number of (p2depth-1) data with p1width bits each.
    - Do we need to seet BSV_RESET_FIFO_ARRAY?
        - this clears the array internal values upon rst_n signal.
    - Do we need to set BSV_RESET_FIFO_HEAD?
        - this sets D_OUT to be zero upon reset
        - I think we need this, because rn D_out defaults to aaaa….

Simple main controller

- simple_read_FSM.v
- simple_read_controller.v
    - contains simple_read_FSM, read_addr_gen, read_bank_counter modules
    - relays control signals between the read_FSM and the main_FSM
- simple_main_FSM.v
- simple_main_controller.v
    - write banks of data to the double buffer, and read all banks out in the same order
    - contains ifmap_input_controller, simple_read_controller, and simple_main_FSM.
- simple_main_controller_tb.sv
    - write all banks and then read all banks in the same addr order, for IC0 = 2 (chaining length), IC1*IY0*IX0 = 4 (# of data per bank), OY1*OX1 = 2 (total # of banks)
    - see the simulation waveform below
        - Due to chaining, writing one bank takes longer than reading one.
        - In the CNN, we can read one bank multiple times (i.e. the ifmap data) before switching and writing the next bank.





Testbench files

C files

- conv_gold_sv.cpp
    - the **gold model** that can be called by system verilog modules.

    - the callable function is “run_cov_gold()”
        - This file calls “conv_gold.cpp”.
    - **Define CNN parameters** here, needs to match with the SV testbench

- conv_gold.cpp
    - C++ version implementation of the “conv” module
    - is a parameterized function

- conv_gold_test.cpp
    - **Testbench** for “conv_gold.cpp”.
    - It runs 3xx CNN test cases
    - It contains a function “run_layer()” that runs a specific layer using “conv_gold.cpp”, and then compares the results to pre-computed correct values.

Verilog files

- simple_tb_1.sv
    - test bench for **conv_tiled.sv**
    - only 1 ifmap/weight/ofmap bank
    - data in ifmap/weight inputs are positive integers <10.
    - IC0 = OC0 = 2

simple_tb_1.sv

```
// Description: testbench for conv_tiled.v
//              - only 1 ifmap/weight/ofmap bank
//              - just directly feeds the ifmap/weight data
//              - OY0 = OX0 = 3, OC0 = IC0 = 2, FY = FX = 2
// Author: Cheryl (Yingqiu) Cao
// Date: 2022-07-24

module simple_tb_1;

// local parameters ++
  localparam PARAM_NUM = 6;
  localparam PARAM_WID = 16;
  localparam BANK_ADDR_WIDTH = 32;       // width needed to save IC1*IX0*IY0, OY1_OX1
  localparam BUFFER_MEM_DEPTH = 256;     // capacity of the memory, larger than IC1*Ix0*IY0

//  cnn parameters
  localparam OY0 = 3;
  localparam OX0 = 3;
  localparam OC0 = 2;
  localparam IC0 = 2;
  localparam Stride = 1;

// parameters on: input data dimensions
  localparam OY = 3;
  localparam OX = 3;
  localparam OC = 2;
  localparam IC = 2;
  localparam FY = 2;
  localparam FX = 2;

// derived parameters
```

- simple_tb_2.sv
    - test bench for **conv_tiled.sv**
    - similar to tb_1, but contains negative numbers in ifmap/weight inputs

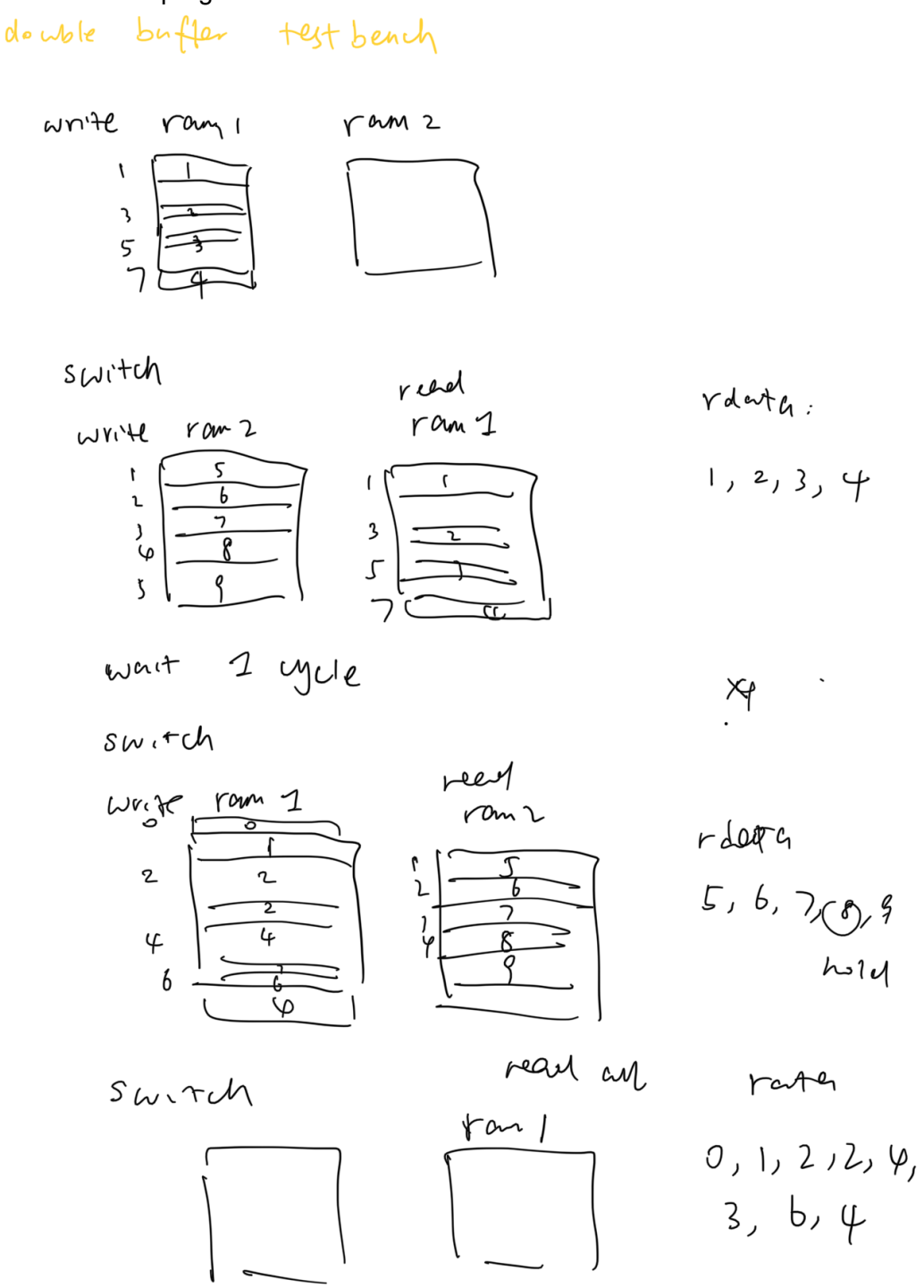
- conv_tb.sv
    - test bench for **conv.sv**
    - conv.sv module computes ofmap data, which will be saved in the transaction item
    - Scorebox calls “run_conv_gold()” to compute **gold_ofmap**, and compared ofmap data with the correct values.
    - The test signal was applied through “test” class => **“apply_sim” task**

```
+1 conv.v | 4 ifmap_input_controller.v | 5 layer_params.svh | 6 conv_tb.sv | 7 conv_gold.cpp | 8 conv_gold_test.cpp | 11 conv_
310
311
312 module conv_tb;
313
314     reg clk;
315
316     always #10 clk =~clk;
317
318     conv_if _if (clk);
319
320     conv #(.IFMAP_SIZE(`LAYER_IFMAP_SIZE), .WEIGHTS_SIZE(`LAYER_WEIGHTS_SIZE), .OFMAP_SIZE(`LAYER_OFMAP_SIZE))
321         dut (
322             .clk(_if.clk),
323             .rst_n(_if.rst_n),
324             .ifmap_dat(_if.ifmap_dat),
325             .ifmap_rdy(_if.ifmap_rdy),
326             .ifmap_vld(_if.ifmap_vld),
327             .weights_dat(_if.weights_dat),
328             .weights_rdy(_if.weights_rdy),
329             .weights_vld(_if.weights_vld),
330             .ofmap_dat(_if.ofmap_dat),
331             .ofmap_rdy(_if.ofmap_rdy),
332             .ofmap_vld(_if.ofmap_vld),
333             .layer_params_dat(_if.layer_params_dat),
334             .layer_params_rdy(_if.layer_params_rdy),
335             .layer_params_vld(_if.layer_params_vld)
336     );
337
338     initial begin
339         test t0;
340
341         clk <= 0;
342         _if.rst_n <= 0;
343         #20 _if.rst_n <= 1;
344
```

- input_read_addr_gen_tb.v
    - test bench for input_read_addr_gen.v
    - sets the blocking parameters for one example and verifies the generated addr by the dut.

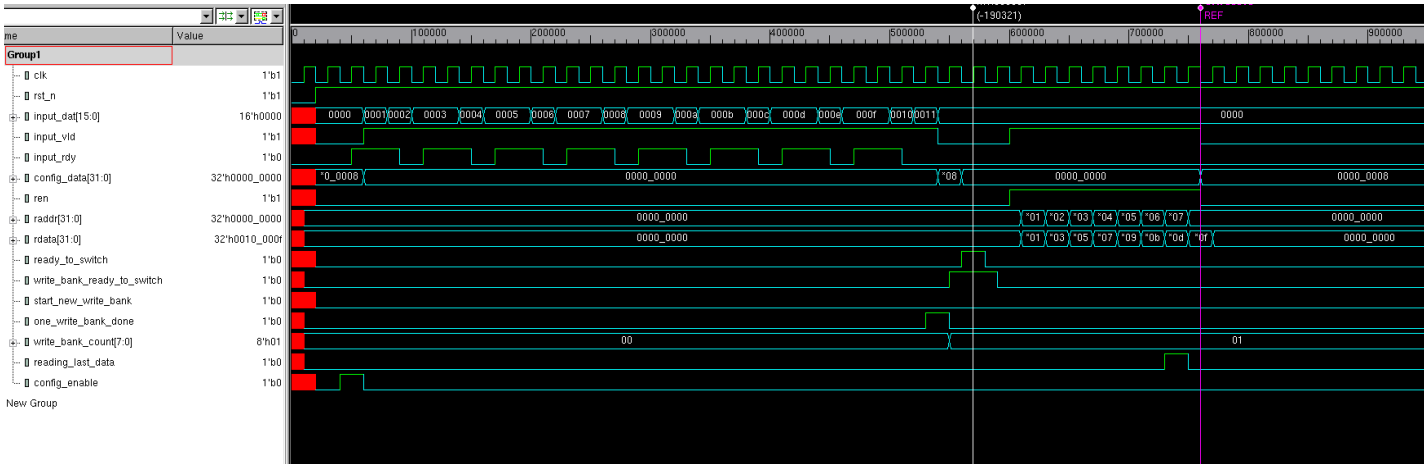
- double_buffer_tb.v
    - test read/write functions for the double buffer
    - see the figure below for functionality

- see the progress notes for 2021-11-28 for the simulation waveforms

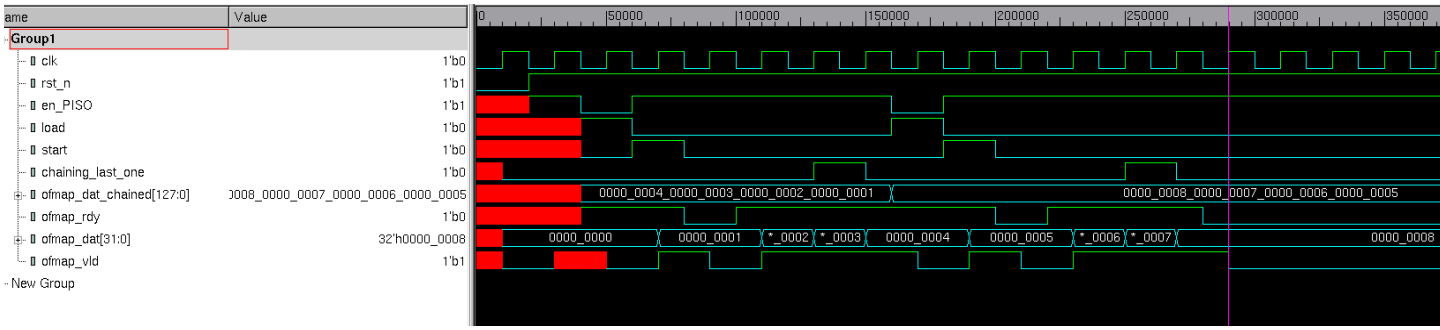


- ifmap_input_controller_tb1.sv
  - Tests the ifmap FSM, write/read address_gen, and double buffer
  - Writes one ifmap bank with IC0 = 2, IC1*IX0*IY0 = 2*2*2 = 8, and then read it for once

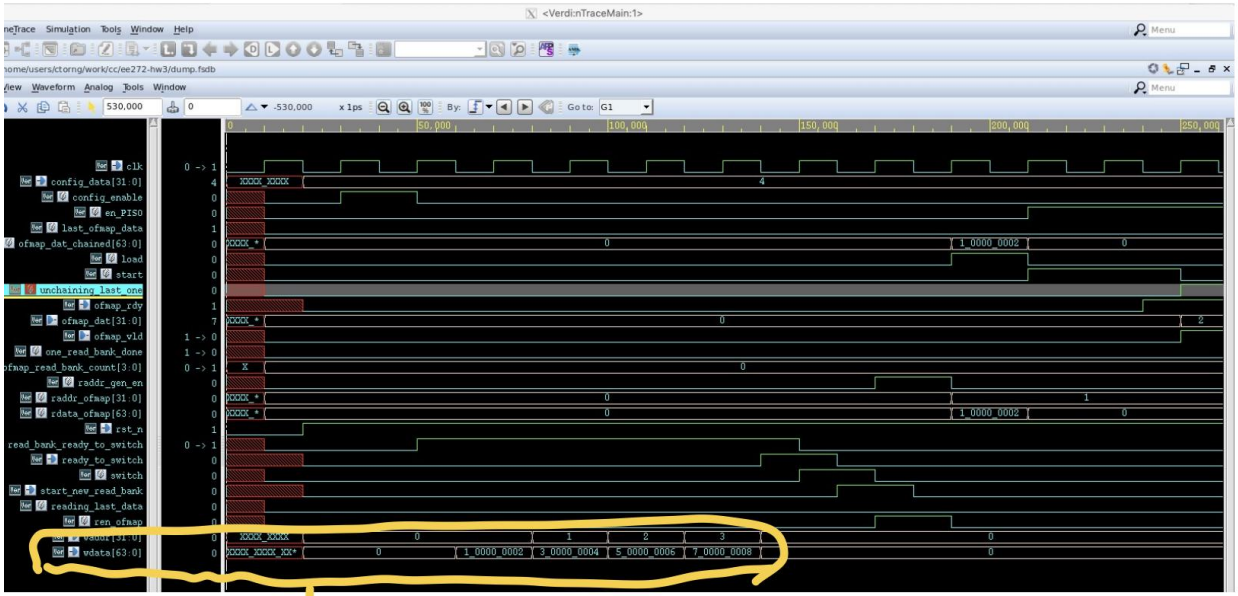
See the simulated waveform as below.

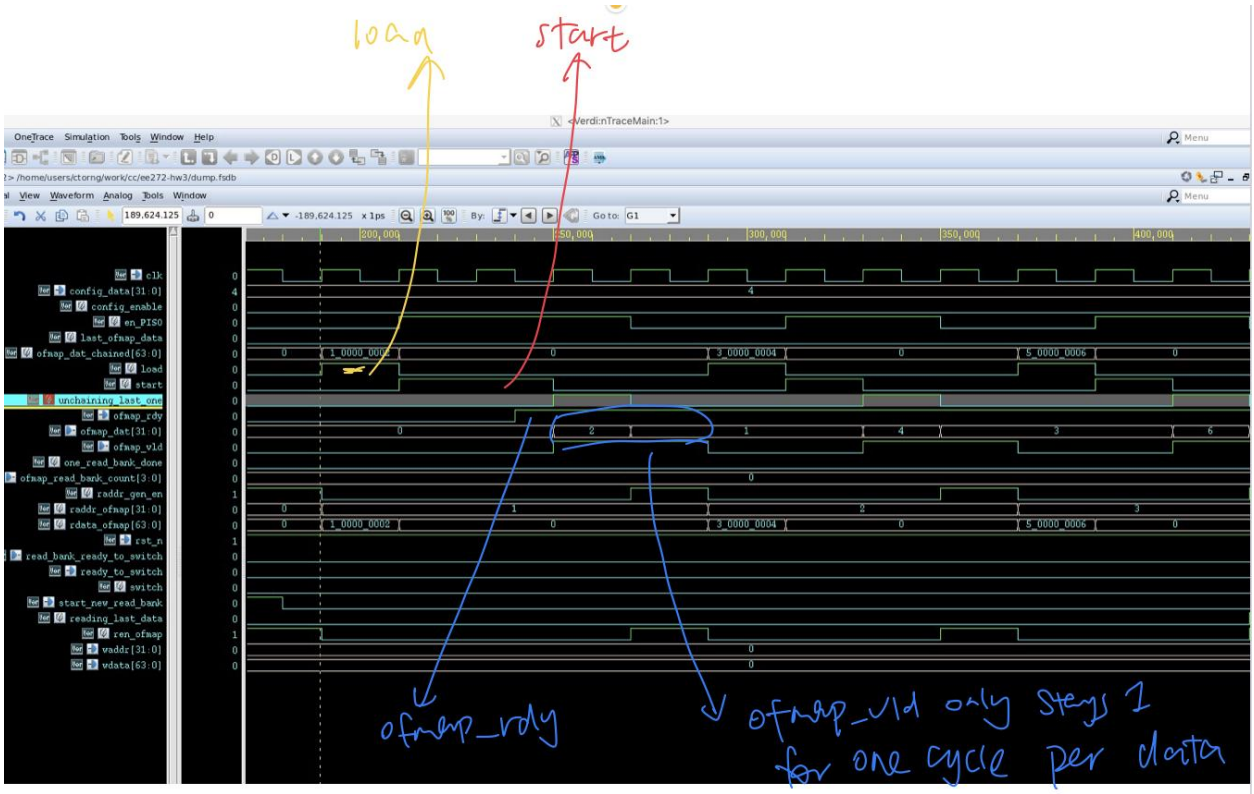


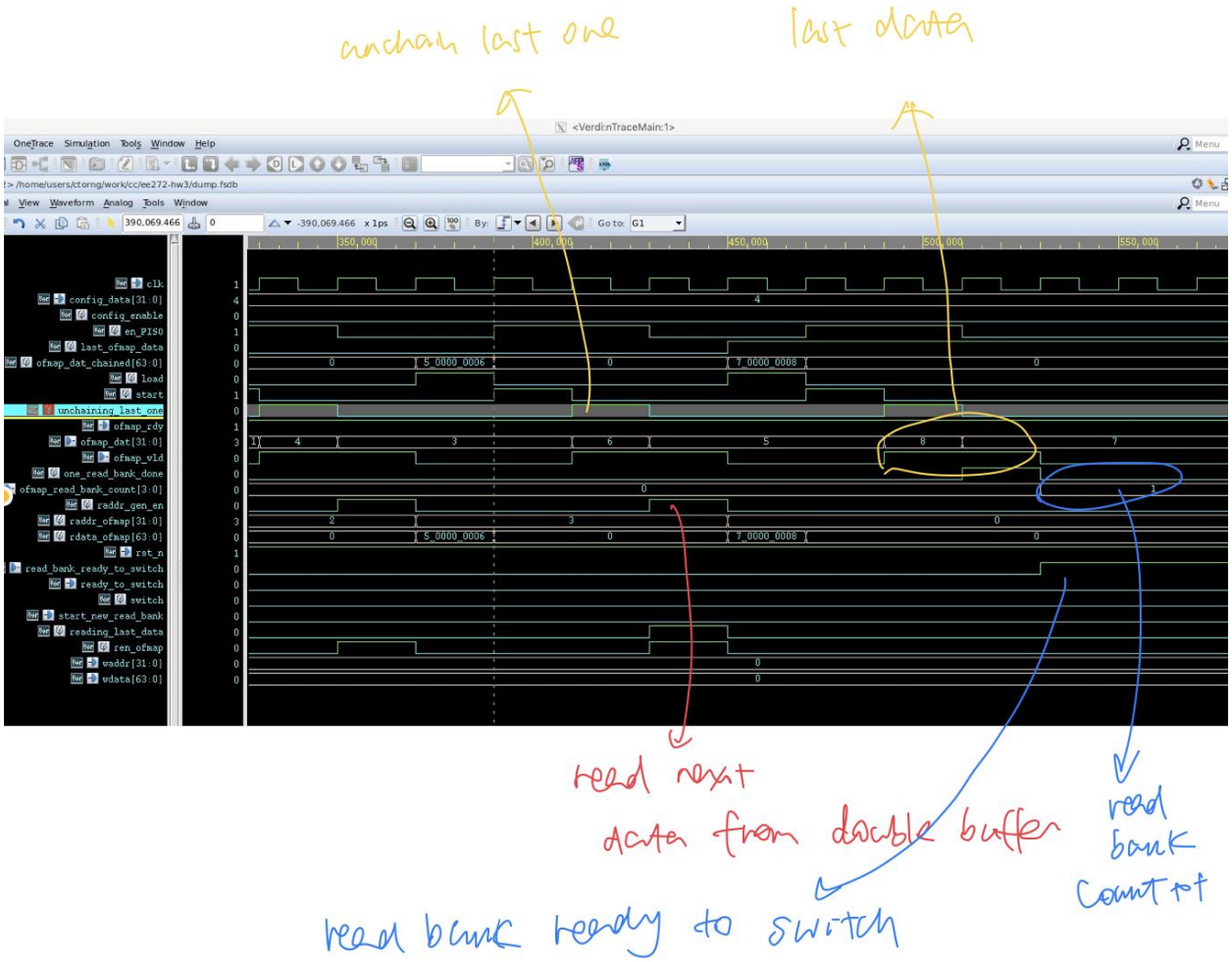
- ofmap_PISO_tb.sv
  - OC0 = 4, 2 unchain cycles
  - unchains 4_3_2_1 and 8_7_6_5



- ofmap_output_controller_tb1.sv
  - OY0_OX0 = 2*2 = 4
  - OC0 = 2 for unchaining
  - run for one ofmap read bank
    - white data 1_2, 3_4, 5_6, 7_8 to accum_double_buffer
    - switch banks
    - let ofmap_output_controller run to read ofmap data out and unchain them
  - monitor ofmap_data, we should get 2,1, 4,3, …, 8, 7.

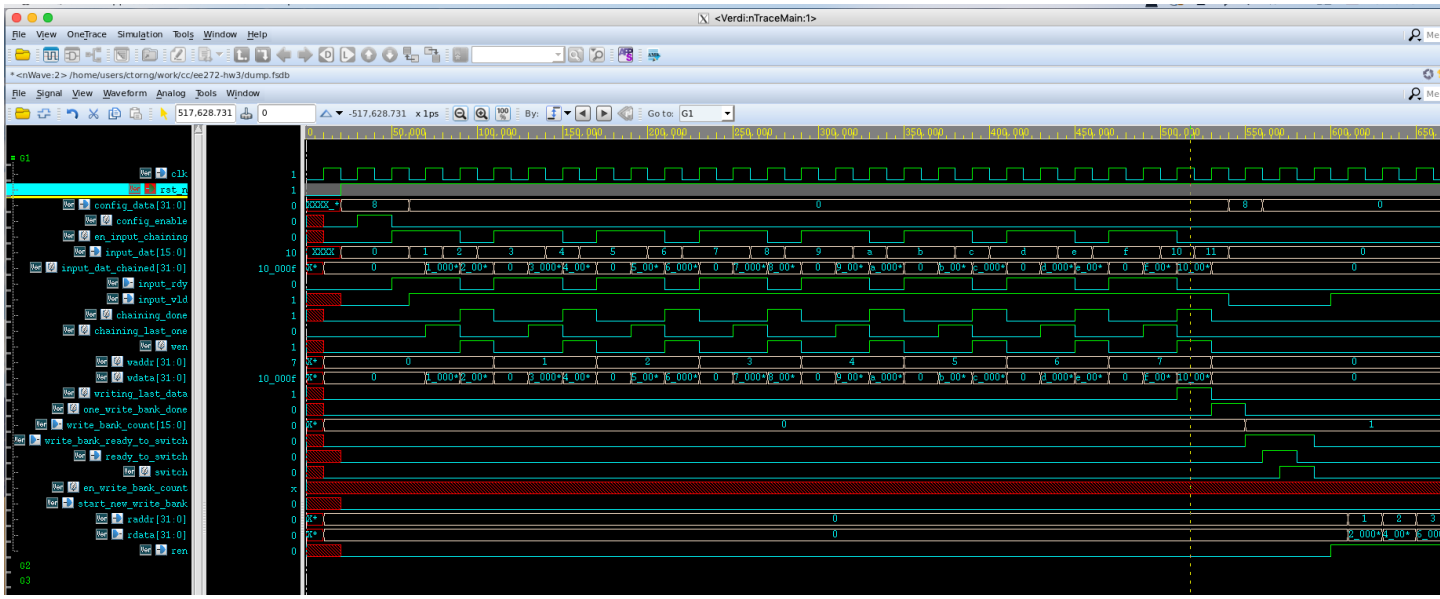

write ofmap data to accum_double_buffer


load
start
ofmap_rdy
ofmap_vld only stays 1 for one cycle per data



- weight_input_controller_tb1.sv
    - Writes one weight bank with OC0 = 2, OC1*IC1*Fy*Fx*IC0 = 2*2*2 = 8, and then read it for once

      See the simulated waveform as below.



- mac_tb.sv
  - ctorng@caddy01: ~/work/c

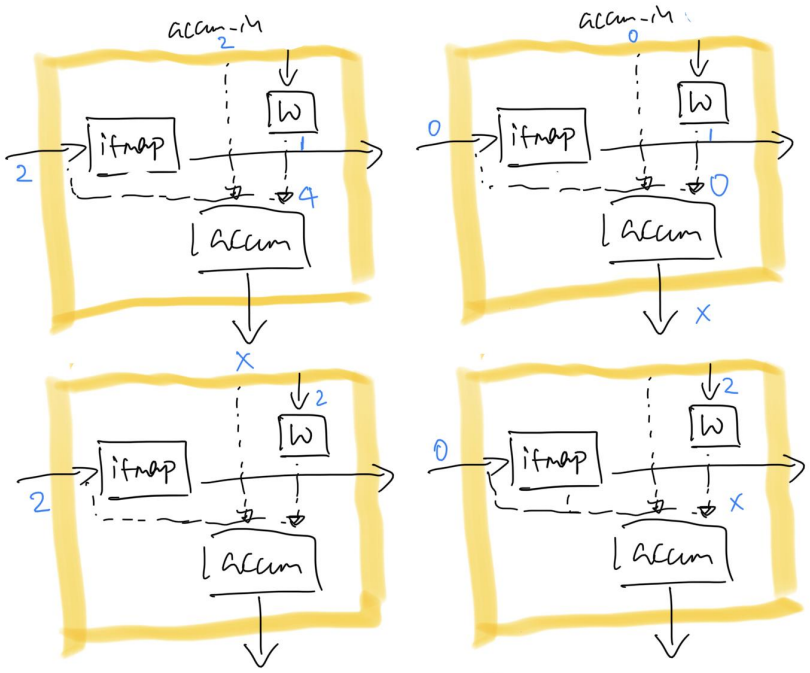
```
+1 mac_tb.sv | 3 mac.v | 4 ofmap_output_controller_tb1.sv
 1 // Description:
 2 //              ifmap_in   weight_in  accum_in   en_weight  |  accum_out
 3 //                x          1          x          1        |     x
 4 //                1          1          1          0        |     2
 5 //                2          2          2          1        |     4
 6 //                3          3          3          0        |     9
 7 //                4          4          4          1        |     12
 8 //                5          5          5          0        |     25
 9 //    weight_in needs to be ready 1 cycle earlier than ifmap_in and
10 //    accum_in
11 // Author: Cheryl (Yingqiu) Cao
12 // Date: 2022-04-2
13
14
```

- mac_array_tb.sv

- 1 mac_array_tb.sv | 3 Makefile

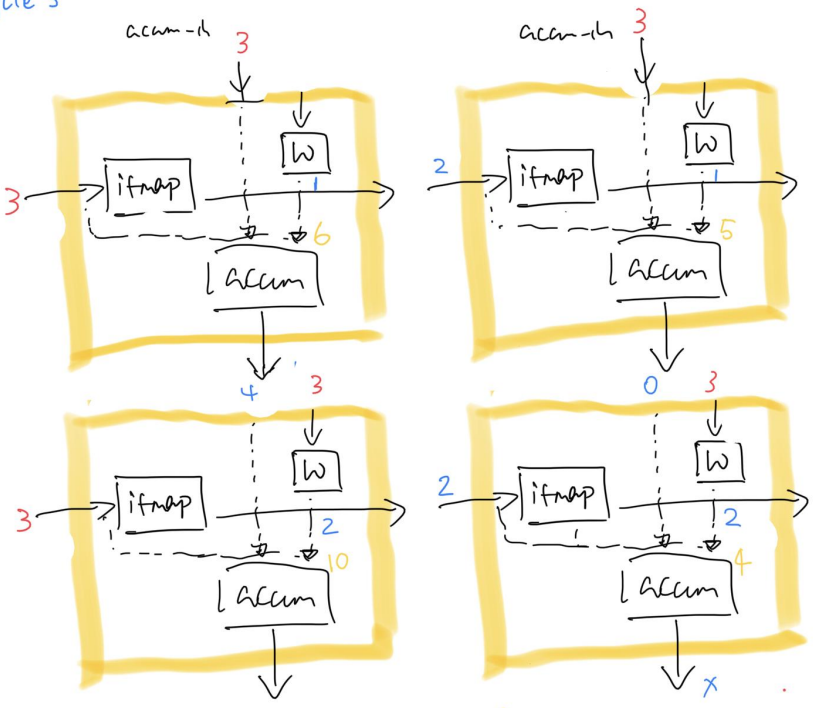
```
 1 // Description:
 2 //  IC0 = OC0 = 4
 3 //  en_weight goes high for each row one by one (2 cycles) and then turn
 4 //  off
 5 //  chaining goes MSB -> LSB
 6 //  en_weight weight_chained  ifmap_chained  accum_in |  accum_out
 7 //      1          1_1          0_0            0_0   |
 8 //      1          2_2          2_2            0_2   |
 9 //      0          3_3          3_3            3_3   |
10 //      0          4_4          4_4            4_4   |   4_10
11 //      0          0            0              0     |   11_14
12 //     cycle 6                                        |   15_8
13 //     cycle 7                                        |   4_0
14 //     cycle 8                                        |   0_4
15 // Author: Cheryl (Yingqiu) Cao
16 // Date: 2022-04-03
17
18
```

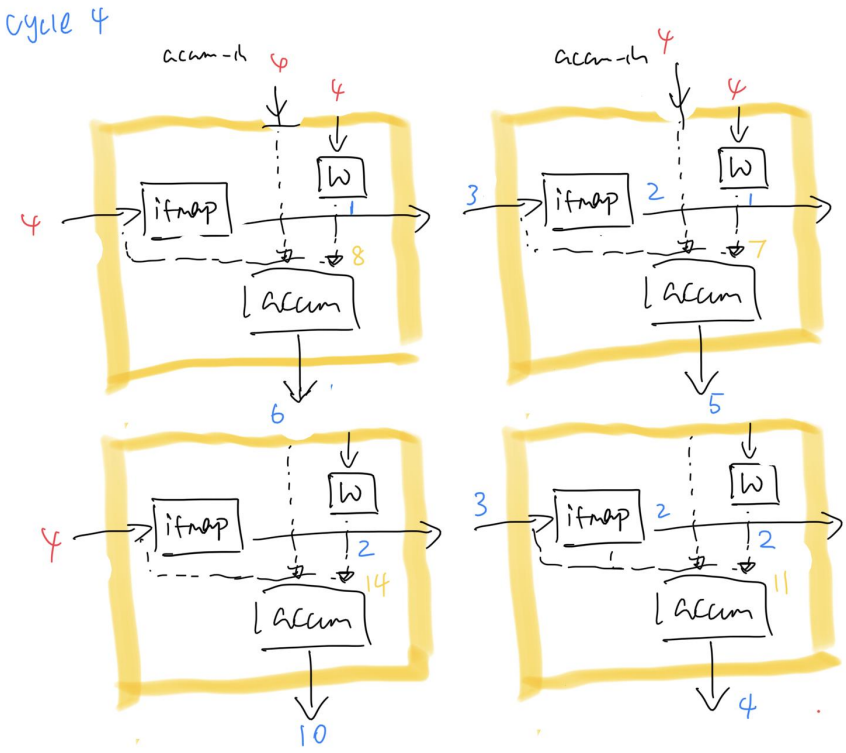
Cycle 2



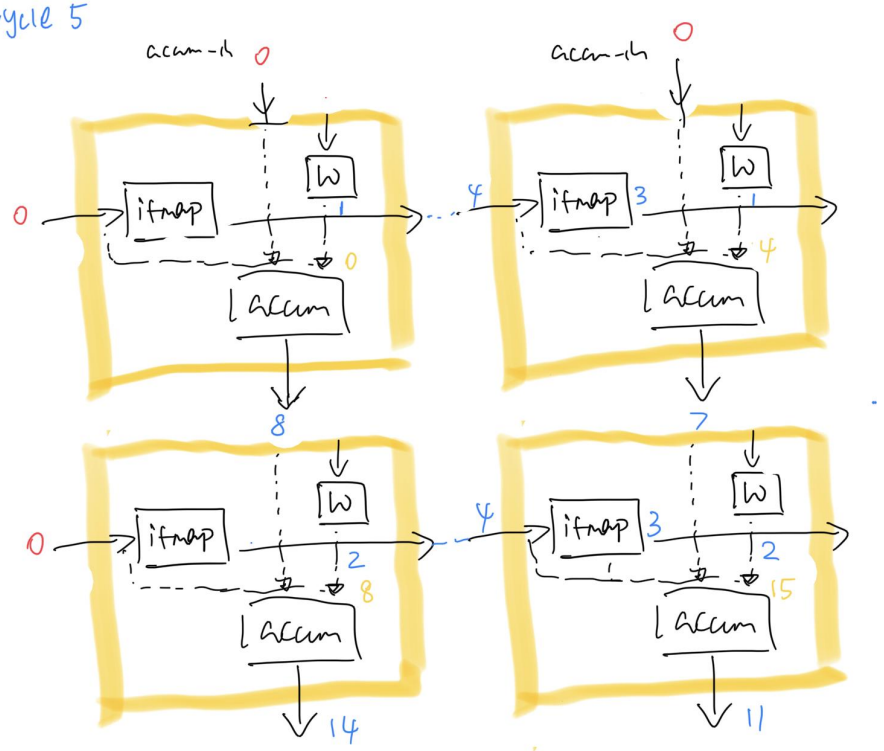
-

Cycle 3

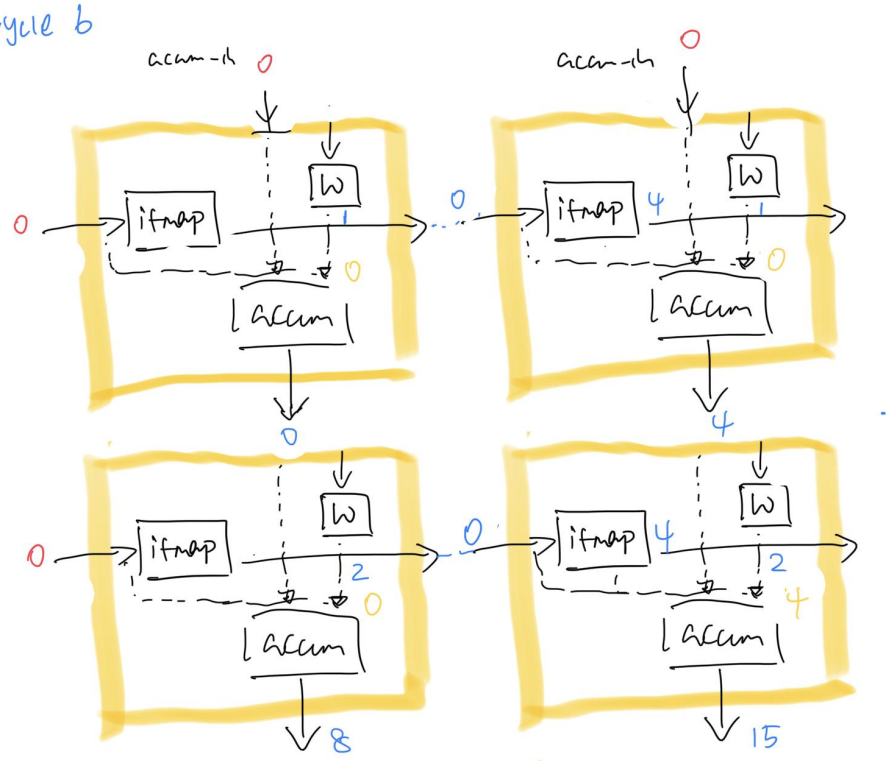


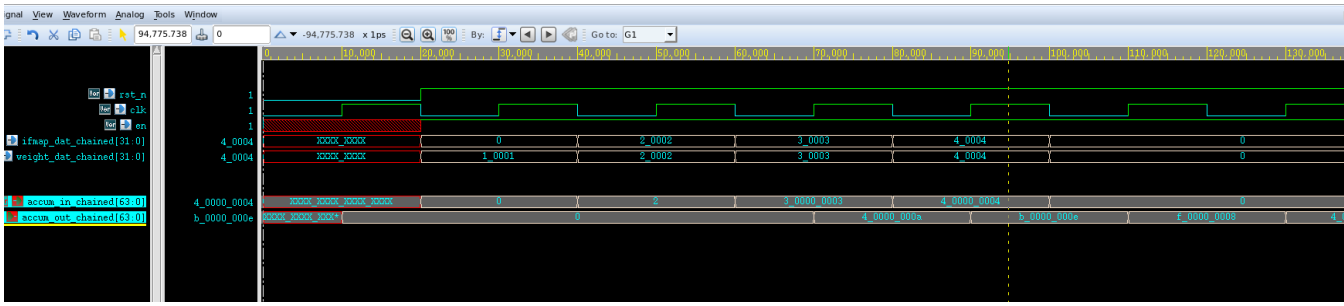

cycle 4
accum-ch
4
4
W
ifmap
4
1
8
Accum
6
accum-ch
4
4
W
ifmap
3
2
1
7
Accum
5
ifmap
4
W
2
14
Accum
10
ifmap
3
2
W
2
11
Accum
4

cycle 5



cycle 6





Typedef files

- layer_params.svh
    - defines user type “layer_params_t”

Verilog Notes

- Need to use non-blocking assignment "**<=**" inside of always blocks
  - <= means to evaluate the expression AT the clock edge
  - the exception is always(*), which is essentially combinational, in which case it does not matter.

- "Casez" allows you to check for multiple bits at the same time:

```
casez ({CLR, DEQ, ENQ, hasodata, ring_empty})
  // Clear operation
  5'b1????: begin
      head          <= `BSV_ASSIGNMENT_DELAY {p3cntr_width {1'b0}} ;
      tail          <= `BSV_ASSIGNMENT_DELAY {p3cntr_width {1'b0}} ;
      ring_empty    <= `BSV_ASSIGNMENT_DELAY 1'b1;
      not_ring_full <= `BSV_ASSIGNMENT_DELAY 1'b1;
      hasodata      <= `BSV_ASSIGNMENT_DELAY 1'b0;
  end
  // ----------------------
  // DEQ && ENQ case -- change head and tail if added to ring
  5'b011?0: begin
      tail          <= `BSV_ASSIGNMENT_DELAY next_tail;
      head          <= `BSV_ASSIGNMENT_DELAY next_head;
  end
```

- Head <= delay next_head assigns the signal with delay

It's equivalent to head <= #5 next_head

- {} is for concatenation. The following means to repeat the inner bracket for p3cntr number of times

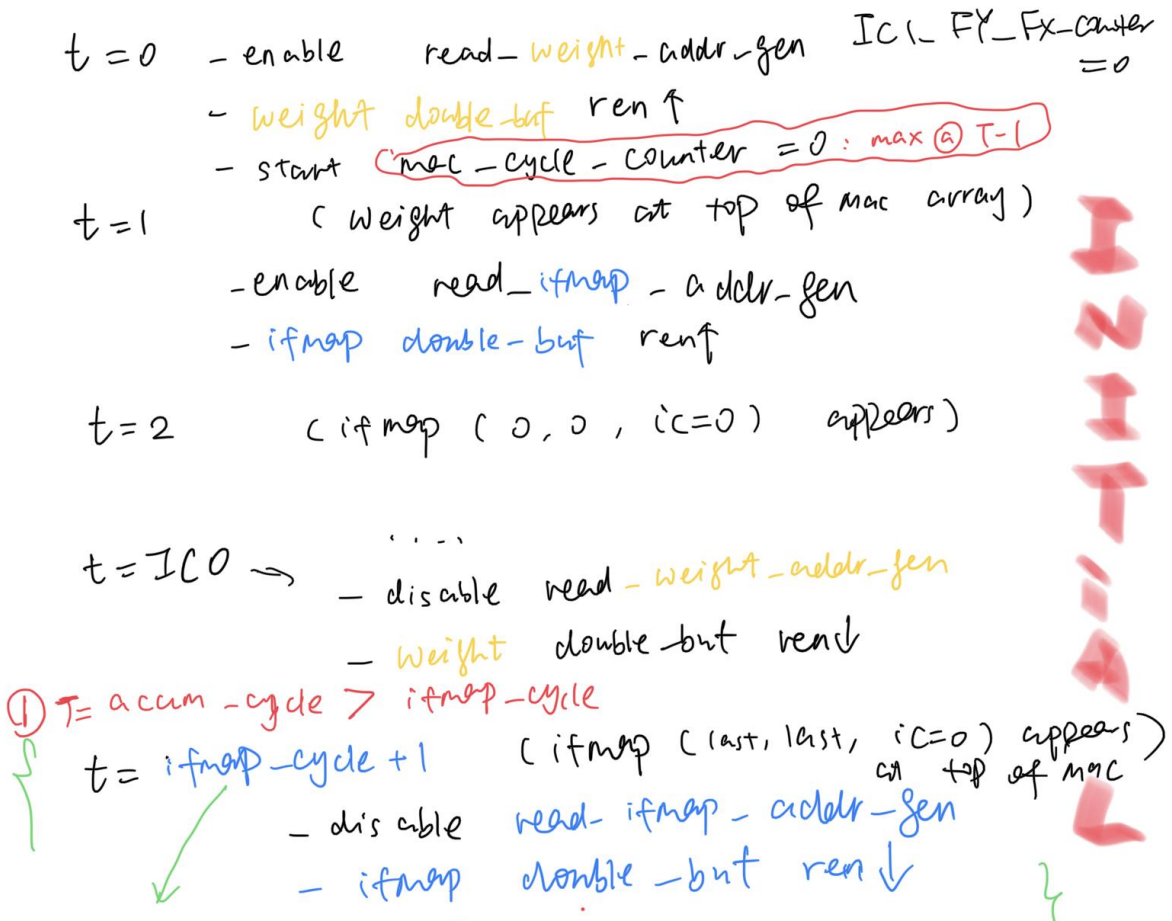
```
assign    next_head = (head == depthLess2 ) ? {p3cntr_width{1'b0}} : incr_head ;
assign    next_tail = (tail == depthLess2 ) ? {p3cntr_width{1'b0}} : incr_tail ;
```

- Macro assertion
  -- here if the macro BSV_RESET… is not set, the compiler will ignore the lines between `ifdef and `endif

```
  // Update the fast data out register
  always @(posedge CLK `BSV_ARESET_EDGE_HEAD)
    begin
`ifdef  BSV_RESET_FIFO_HEAD
      if (RST == `BSV_RESET_VALUE)
        begin
          D_OUT    <= `BSV_ASSIGNMENT_DELAY {p1width {1'b0}} ;
        end // if (RST == `BSV_RESET_VALUE)
      else
`endif
      begin
          casez ({CLR, DEQ, ENQ, hasodata, ring_empty})
            // DEQ && ENQ cases
            5'b011?0: begin D_OUT <= `BSV_ASSIGNMENT_DELAY arr[head]; end
            5'b011?1: begin D_OUT <= `BSV_ASSIGNMENT_DELAY D_IN; end
            // DEQ only and data is in ring
```

Appendix

- Synchronization case 1: accum_cycle > ifmap_cycle
  - It takes longer for the 1st partial sum ofmap(0,0, IC = 0:3) to get out of the MAC array than to iterate through the ifmap elements for weight (Fx = 0, Fy = 0)
  - need to pause ifmap double buffer input



t = T+2 (partial sum ofmap (0,0, ic=0:3) appears @ top of mac array.
ifmap (1,0, ic=0) for Fx=1, Fy=0 appears)

...

t = Ico same as Initial

t = ifmap_cycle (patial sum ofmap (last, last, ic=0:3) ready to de queue)
- disable write_accum_addr_gen
- accum_buf wen ↓
- plus operations in Initial

normal

t = ifmap_cycle + 1
- disable read_accum_addr_gen
- accum_buf ren ↓
- plus operations in Initial

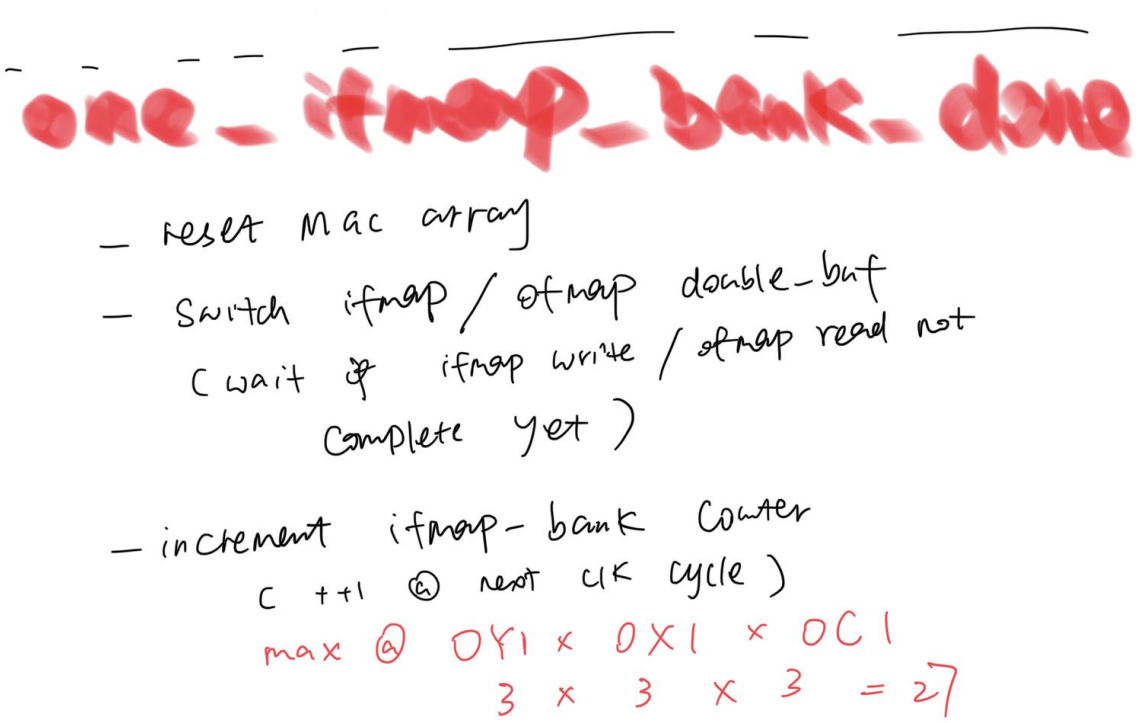
Final

Ic1_Fy_Fx counter
== Ic1 x Fy x Fx
2 x 3 x 3 = 18

t = 0 instead of the accum_buffer
write to the ofmap_double_buf
- enable write_accum_addr_gen
- ofmap_double_buf wen ↑

...

t = ifmap_cycle (ofmap write complete)
- disable write_accum_addr_gen
- ofmap_double_buf wen ↓



- Synchronization case 2: ifmap_cycle > accum_cycle
  - It takes longer to iterate through the ifmap elements for weight (Fx = 0, Fy = 0), than for the 1st partial sum ofmap(0,0, IC = 0:3) to get out of the MAC array
  - need to wait to feed back the accum_buf to MAC array

② T = ifmap_cycle ≥ accum_cycle

" OYo · OXo = 9 " OCo + ICo + 1 = 9 (2 + 2 + 1 = 5)

$t=0$ - enable read_weight_addr_gen IC1_FY_FX_counter = 0

- weight double_buf ren↑
- start mac_cycle_counter = 0 : max @ T-1

$t=1$ (weight appears at top of mac array)

- enable read_ifmap - addr_gen
- ifmap double-buf ren↑

$t=2$ (ifmap (0, 0, ic=0) appears)

$t=ICO \rightarrow$ ...

- disable read_weight_addr_gen
- weight double_buf ren↓

$t=$ accum_buf_cycle (1st partial sum ofmap(0, 0, ic=0:3) ready to enqueue)

- enable write_accum_addrr_gen
- accum_buf wen↑

$t=$ ifmap_cycle / 0 (read weight (FX=1, FY=0)

- enable read_weight_addr_gen
- weight double-buf ren↑

- mac_cycle_counter rolls over to 0
- increment IC1-FY-FX counter

$t=$ ifmap_cycle + 1 (ifmap (last, last, ic=2) appears @ top of Mac array)

(dequeue ofmap(0, 0, ic=0:3)

- enable read_accum_addr_gen
- accum_buf ren↑

INITIAL

t = 2 (- ifmap (0, 0, ic=0) for Fx=1, Fy=0 appears
- ofmap (0,0, ic=0 :)) ⑤ top of mac
- weight Fx=1, Fy=0 ready)

t = Ico same as (Initial) **Normal**

t = accm-buf-cycle
same as (Initial)

Pre-final

- Other cycles same as (normal)

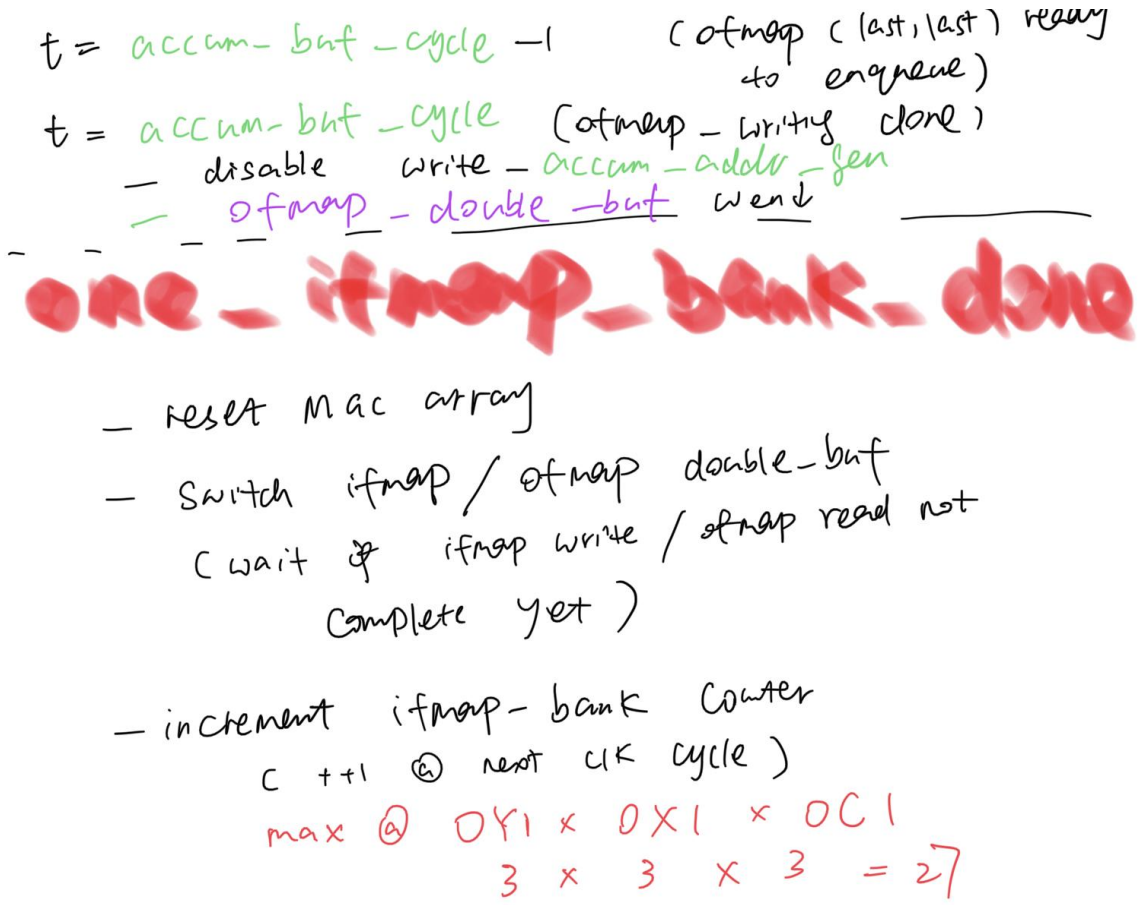
t = accm-buf-cycle (1st partial sum ofmap(0,0,..) ready to enqueue)
write to ofmap-double-buf
- enable write-accm-addr-gen
- ofmap-double-buf wen↑

Final

Ic1-Fy-Fx counter
== Ic1 × Fy × Fx
2 × 3 × 3 = 18

t=0 stop reading weights
- disable read-weights-addr-gen
- weights-buf ren↓

t=1 stop reading ifmap (ifmap(last)(last) appears ⑤ top of mac)
- disable read-ifmap-addr-gen
- ifmap-buf ren↓



- Synchronization case 3: ifmap_cycle == accum_cycle
  - It takes the same # of cycles to iterate through the ifmap elements for weight (Fx = 0, Fy = 0), as for the 1st partial sum ofmap(0,0, IC = 0:3) to get out of the MAC array.
  - A special case of Case 2.

③ T = ofmap_cycle = accum_cycle

ofmap_cycle = OY0 · OX0 = 9

accum_cycle = OC0 + IC0 + 1 = 9 (2 + 2 + 1 = 5)

t = 0 - enable read_weight_addr_gen IC1_FY_FX_counter = 0
- weight double_buf ren ↑
- start mac_cycle_counter = 0 : max ⓐ T-1

t = 1 (weight appears at top of mac array)
- enable read_ifmap_addr_gen
- ifmap double_buf ren ↑

t = 2 (ifmap (0, 0, ic=0) appears)

t = IC0 → - disable read_weight_addr_gen
- weight double_buf ren ↓

INITIAL

t = accum_buf_cycle = ifmap_cycle / 0 (1st partial sum ofmap(0, 0, ic=0:3) ready to enqueue)
- enable write_accum_addr_gen
- accum_buf wen ↑

(read weight (FX=1, FY=0)
- mac_cycle_counter rolls over to 0
- increment IC1-FY-FX counter

- enable read_weight_addr_gen
- weight double_buf ren ↑

t = ifmap_cycle + 1 (ifmap (last, last, ic=0) appears ⓐ top of mac array)
(dequeue ofmap(0, 0, ic=0:3))
- enable read_accum_addr_gen
- accum_buf ren ↑

t = 2 (- ifmap (0, 0, ic=0) for Fx=1, Fy=0 appears
- ofmap (0,0, ic=0 : :) ⑤ top of mac
- weight Fx=1, Fy=0 ready)

t = Ic0 same as (Initial) Normal

Final Ic1 - Fy - Fx counter
= = Ic1 × Fy × Fx
2 × 3 × 3 = 18

0
||
t = accum_buf_cycle (1st partial sum ofmap(0,0,..)
ready to enqueue)
write to ofmap_double_buf instead
- enable write_accum_addr_gen
- ofmap_double_buf wen↑
- accum_buf wen↓

△ stop reading weights
- disable read_weights_addr_gen
- weights_buf ren↓

t = 1 △ stop reading ifmap (ifmap(last, last) appears
⑤ top of mac)
- disable read_ifmap_addr_gen
- ifmap_buf ren↓
△ stop feeding accum to MAC
- disable read_accum_addr_gen
- accum_buf ren↓

t = accum_buf_cycle - 1 (ofmap (last, last) ready
to enqueue)

one_ifmap_bank_done

t = accum_buf_cycle (ofmap_writing done)
- disable write - accum_addr_gen
- ofmap_double_buf wen↓

- reset mac array
- switch ifmap / ofmap double_buf
  (wait if ifmap write / ofmap read not complete yet)
- increment ifmap_bank counter
  (+1 @ next clk cycle)
  max @ 0x1 x 0x1 x 0c1
  3 x 3 x 3 = 27