Final Project Report First Page. Hardware Convolutional Neural Network

Name: DJ Hansen
Unityid: djhanse2
StudentID:. 200097121

Delay (ns to run provided provided example).
Clock period: 3.9
cycles": 1

Logic Area: (um^2)
524.552

Memory: N/A

1/(delay.area) ($ns^{-1}.um^{-2}$)
1/(524.552*3.9) = 4.888e-4

Delay (TA provided example. TA to complete)

1/(delay.area) (TA)

Abstract

This project implements a fixed size single stage of a binary convolutional neural network in hardware. The task was successfully achieved by implementing a well-known strategy of convolution utilizing XNOR gates and a bit counter to output a resultant matrix from an input and weight matrix. This hardware utilizes SRAM to store and retrieve matrices for its calculations. Verification of the functionality of the hardware was done by way of simulation and synthesis. The achieved metrics are adequate for the purposes of demonstrating functionality, but further improvements could be made to improve timing and cell area.

Hardware Convolutional Neural Network

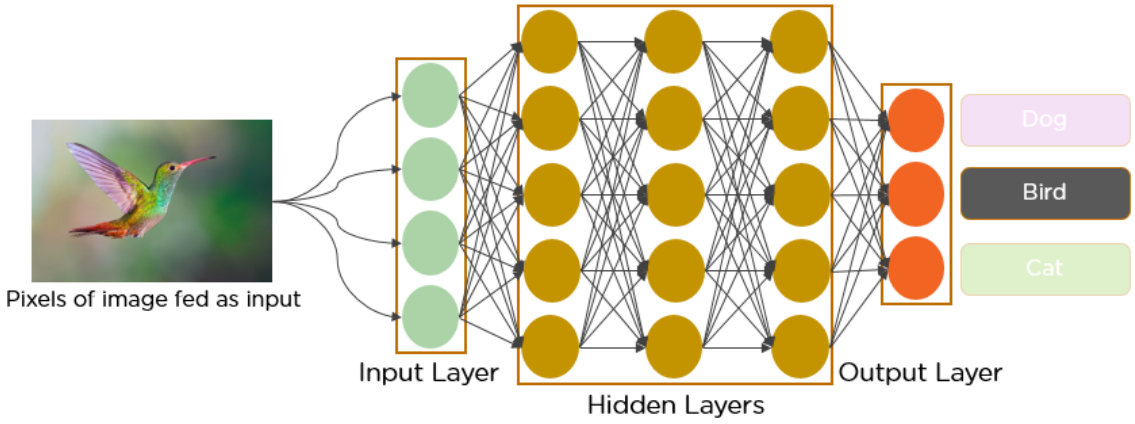
DJ Hansen

Abstract

This project implements a fixed size single stage of a binary convolutional neural network in hardware. The task was successfully achieved by implementing a well-known strategy of convolution utilizing XNOR gates and a bit counter to output a resultant matrix from an input and weight matrix. This hardware utilizes SRAM to store and retrieve matrices for its calculations. Verification of the functionality of the hardware was done by way of simulation and synthesis. The achieved metrics are adequate for the purposes of demonstrating functionality, but further improvements could be made to improve timing and cell area.

Introduction

The purpose of this project is to implement a fixed size single stage of a binary convolutional neural network in hardware. A convolutional neural network is a network where nodes are divided into groups that form various layers and are connected to all nodes in adjacent layers. Convolution is performed in order to connect the layers by using convolution on the input matrix and a rounded weight matrix. This is often used in visual imagery processing as seen in the following image.



1https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

The scope of this project is only intended to implement a single stage of this process, and only accommodates a fixed sized input and weight matrix. This is designed and simulated using Verilog with the aim of synthesizing the design to meet timing and area specifications. The following key results were achieved:

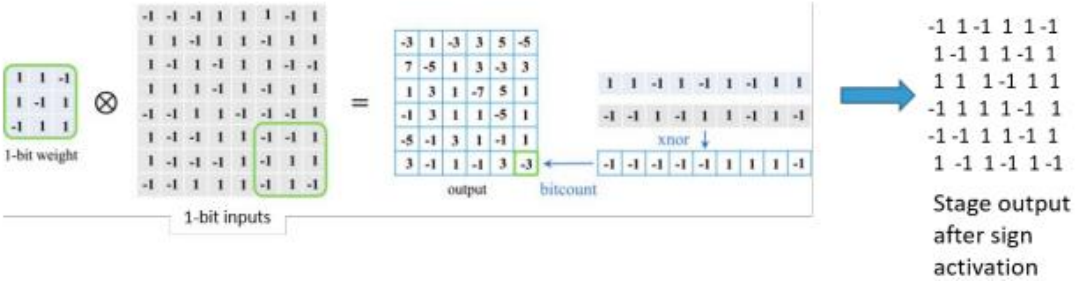
- Compute Cycle: 1
- Clock Period: 3.9
- Total Cell Area: 524.552

Methodologies and implementation will be discussed further in the report, as well as more timing information.

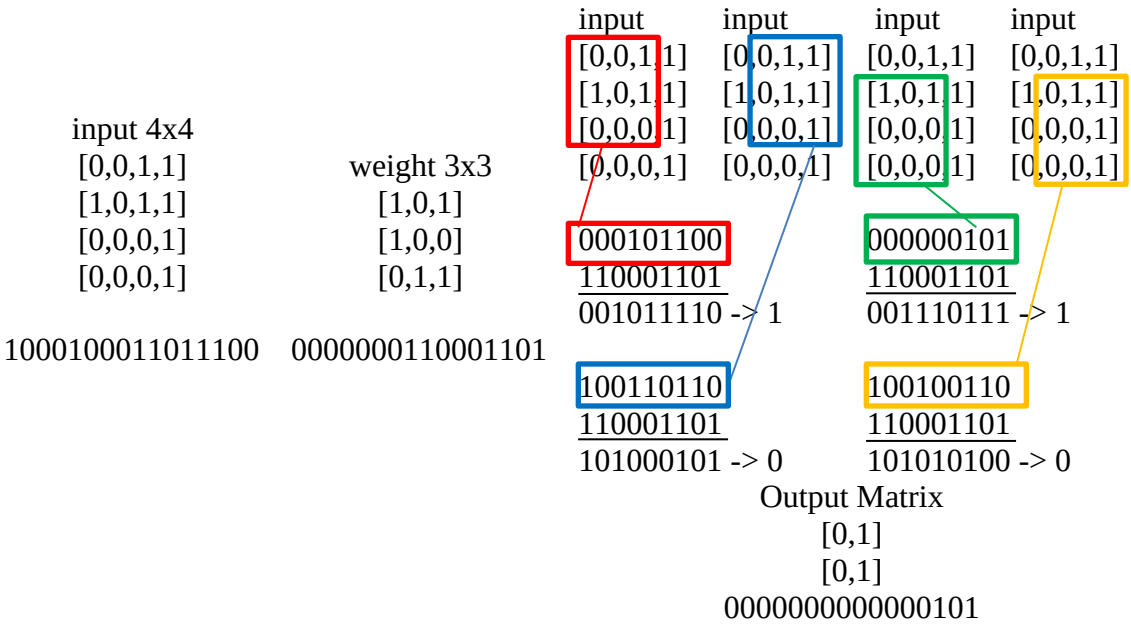
Micro-Architecture

- Hardware “algorithmic” approach used.
- High level architecture drawing, and description of data flow
- Details on claimed innovations

In order to accomplish this task, the first decision to be made was how to perform convolution. Knowing that the input matrix was fixed at 4x4, and the weight matrix was fixed at 3x3 helped because it meant that my output matrix was going to be 2x2. Instead of performing convolution by multiplying and adding, I chose to use a method that implemented XNOR's and bit counting to perform the task. The following diagram illustrates this methodology well. For the purposes of this project, -1 was treated as 0, and 1 kept its value.



Essentially, the weight matrix is overlayed on top of the input matrix incrementally until it covers the entire matrix. The values that correspond are XNOR'd and then negated to form a new matrix. The number of 1's and the number of 0's are then compared, if the matrix has more 1's than 0's than the output is 0, and if there are more 0's than 1's than the output is 1. The following is an example of this calculation, including how each matrix is packed in memory.

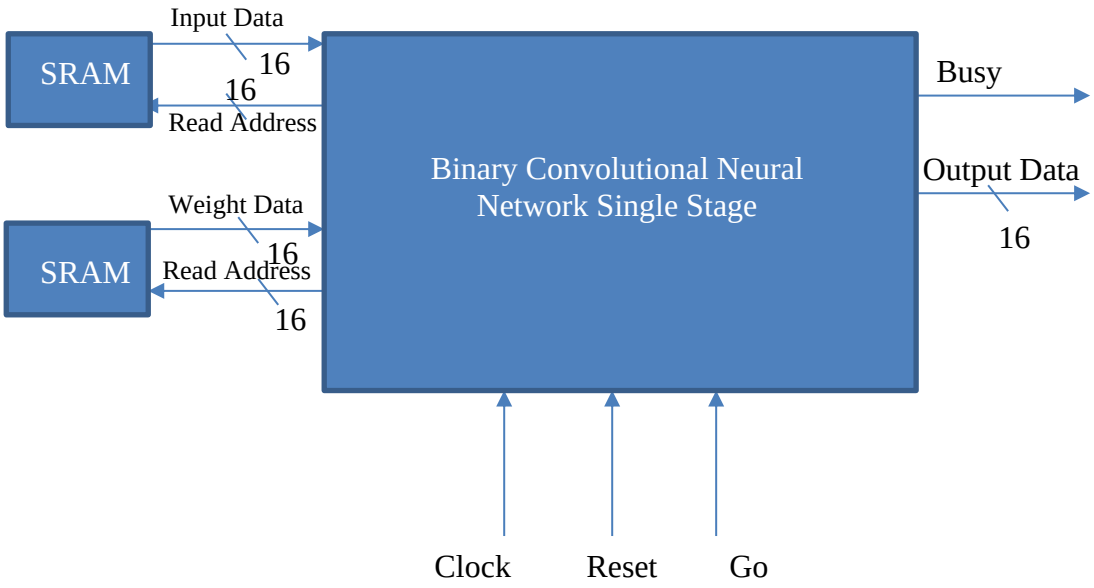


Knowing that the input matrix was 4x4, there would only be four calculations performed with the weight matrix, so the input was split into four different matrices. The convolution and bit counting was performed as described previously and put into an output matrix ready to be written to the SRAM.

This report will discuss each of the modules in the design, their signals, and design challenges, and then will discuss verification and synthesis with respect to the design in total.

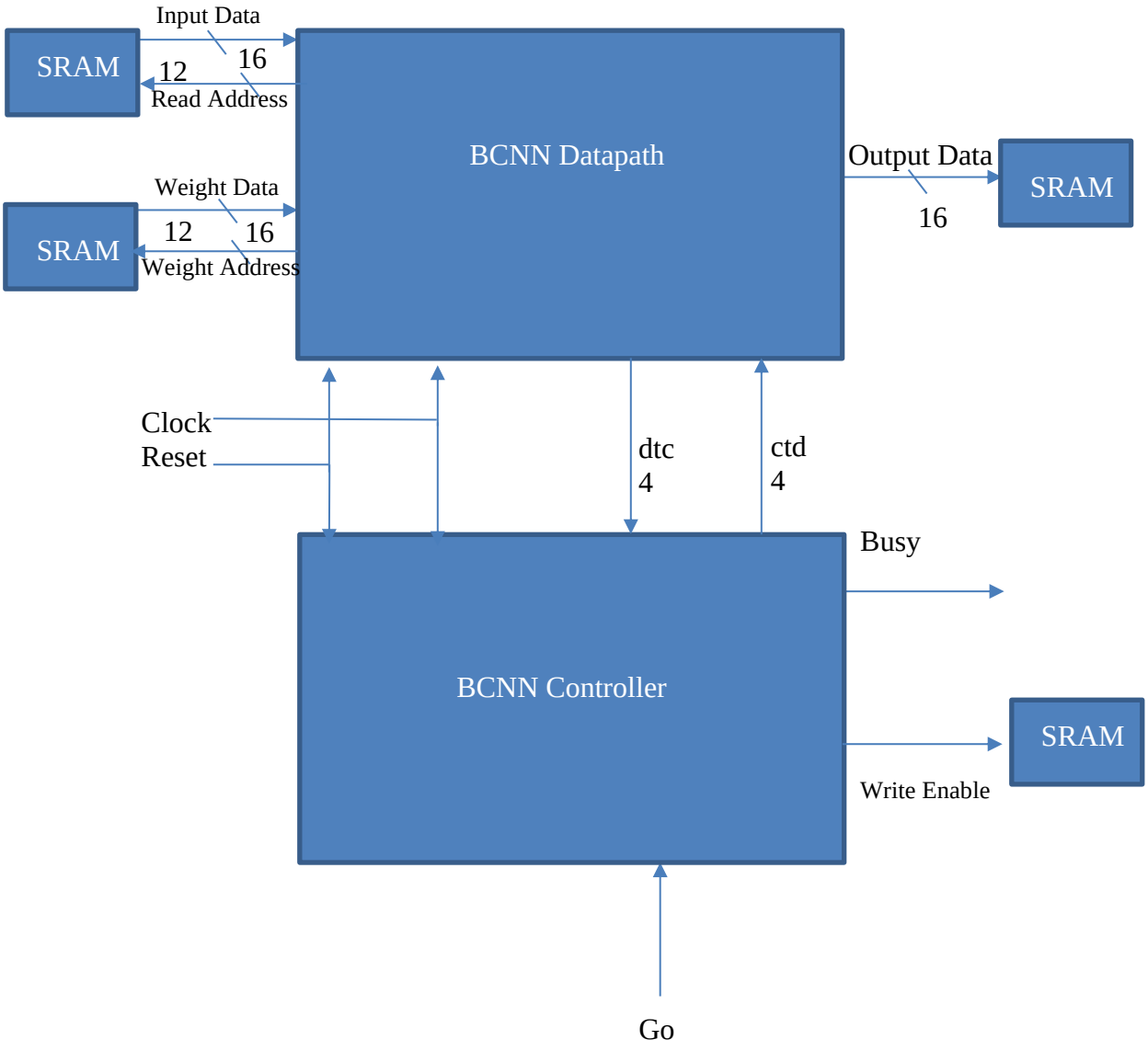
High Level

At the highest level, upon assertion of the Go signal, we have an interface that reads the input and weight matrix according to the address sent to the SRAM. It asserts a Busy signal and performs all calculations and outputs the desired matrix to the SRAM, flipping the Busy signal off. In order to best separate the datapath and controller found in this design, I wanted to keep the top level as simple as possible.



Signal Name	I/O/Internal	Weight (bits)	Interfaced
dut_run	Input	1	Controller
dut_busy	Output	1	Controller
reset_b	Input	1	Both
clk	Input	1	Both
dut_sram_write_address	Output	12	Datapath
dut_sram_write_data	Output	16	Datapath
dut_sram_write_enable	Output	1	Controller
dut_sram_read_address	Output	12	Datapath
sram_dut_read_data	Input	16	Datapath
dut_wmem_read_address	Output	12	Datapath
wmem_dut_read_data	Input	16	Datapath
ctd	Internal	4	Both
dtc	Internal	4	Both

Here it becomes obvious where each of the signals are intended to go as well as their respective size. In order to keep things simple, I only have two Signals between the controller and the datapath, intended to relay information as to which state we are in and when the datapath is done manipulating data.



While a more in-depth description of the datapath and controller will follow, this is a high level overview of what each aims to accomplish. The signals dtc and ctd signify controller to datapath and datapath to controller, and are simply control signals that allow for state and data manipulation. All other signals are defined on the previous page. The

following code is my high level module that connects the datapath and the controller together.

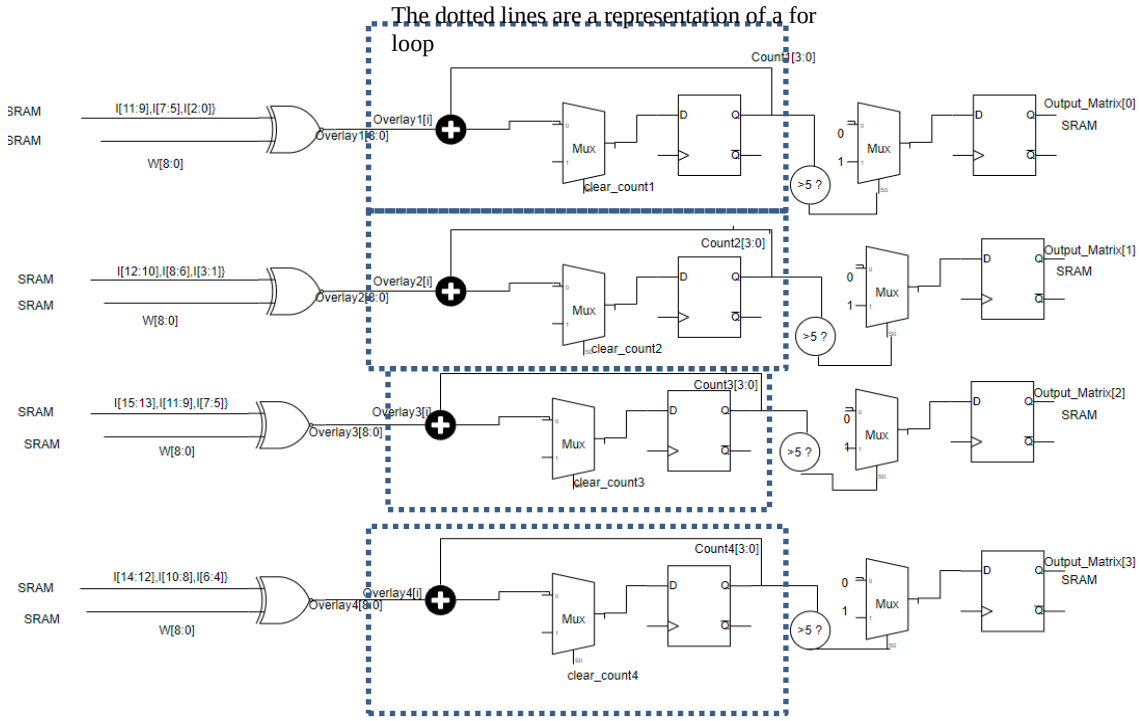
```
module MyDesign
(dut_run,dut_busy,reset_b,clk,dut_sram_write_address,dut_sram_write_dat
a,

dut_sram_write_enable,dut_sram_read_address,sram_dut_read_data,
               dut_wmem_read_address,wmem_dut_read_data);
input dut_run;
output dut_busy;
input reset_b;
input clk;
output [11:0] dut_sram_write_address;
output [15:0] dut_sram_write_data;
output dut_sram_write_enable;
output [11:0] dut_sram_read_address;
input [15:0] sram_dut_read_data;
output [11:0] dut_wmem_read_address;
input [15:0] wmem_dut_read_data;

 wire [3:0] dtc;
 wire [3:0] ctd;
datapath u1(
      .clk(clk),
      .reset(reset_b),
      .input_data(sram_dut_read_data),
      .weight_data(wmem_dut_read_data),
      .input_address(dut_sram_read_address),
      .weight_address(dut_wmem_read_address),
      .write_data(dut_sram_write_data),
      .write_address(dut_sram_write_address),
      .sig_in(ctd),
      .sig_out(dtc)
);

controller u2(
      .clk(clk),
      .reset(reset_b),
      .go(dut_run),
      .busy(dut_busy),
      .write_enable(dut_sram_write_enable),
      .ctrl_in(dtc),
      .ctrl_out(ctd)
);
```

```
endmodule
```

Datapath

In this design, the datapath is intended to manipulate all data from the SRAM and also output the requested data. It handles all the interaction with SRAM other than a write enable signal which is performed in the controller. The following is the logic behind my datapath.

Signal Name	I/O/Internal	Weight (bits)
clk	Input	1
reset	Input	1
input_data	Input	16
weight_data	Input	16
input_address	Output	12
weight_address	Output	12
write_address	Output	12
write_data	Output	16
sig_in	Input	4
sig_out	Output	4
overlay1	Internal wire	9
overlay2	Internal wire	9
overlay3	Internal wire	9
overlay4	Internal wire	9
count1	Internal reg	4
count2	Internal reg	4
count3	Internal reg	4
count4	Internal reg	4



The trickiest part to the data manipulation was determining how to count the number of 1's in a register while keeping timing and total cell area low. After much deliberation, I determined that a for loop was the best way to do this. The datapath consists of both combinational and clocked logic.
The overlays as described in the introduction of this report were calculated using combinational logic. This works because we know we will not be getting any more input as long as the busy signal is set high.

```
assign overlay1 = ~({input_data[10:8],input_data[6:4],input_data[2:0]} ^ weight_data[8:0]);
assign overlay2 = ~({input_data[11:9],input_data[7:5],input_data[3:1]} ^ weight_data[8:0]);
assign overlay3 = ~({input_data[14:12],input_data[10:8],input_data[6:4]} ^ weight_data[8:0]);
assign overlay4 = ~({input_data[15:13],input_data[11:9],input_data[7:5]} ^ weight_data[8:0]);
```

We have one always @ (posedge clk) block that determines what data we need to manipulating at that time. We are either getting the address from SRAM, getting the data from SRAM, counting the bits in our overlay calculations, or we are writing to output SRAM.

```
module datapath( // I/O from top module and SRAM
                          clk, reset, input_data, weight_data, input_address,
                          weight_address, write_address, write_data,
                          // I/O from controller<->datapath
                          sig_in, sig_out
                          );

// signals from SRAM
input clk;
input reset;
input [15:0] input_data; // data from SRAM in matrix form, need all bits
input [15:0] weight_data; // data from SRAM in matrix form, only need 9 LSB
output reg [15:0] write_data;
output reg [11:0] input_address;
output reg [11:0] weight_address;
output reg [11:0] write_address;

input [3:0] sig_in; // [3] = get_address, [2] = get_data, [1] = count_ready,
[0] = write_ready
output reg [3:0] sig_out; // [3] = got_data, [2] = ready_to_receive, [1] =
count_done, [0] = written_success


wire [8:0] overlay1; // result of xnor calculation
wire [8:0] overlay2; // result of xnor calculation
wire [8:0] overlay3; // result of xnor calculation
wire [8:0] overlay4; // result of xnor calculation
reg [3:0] count1;
reg [3:0] count2;
reg [3:0] count3;
reg [3:0] count4;
```

```
integer N;

always @ (posedge clk) begin
        if(sig_in[3]) begin
              input_address <= 12'b0;
              weight_address <= 12'b0;
              sig_out <= 4'b0100;
              end
       else if(sig_in[2]) begin
                     sig_out <= 4'b1000;
                     count1 = 3'b0;
                     count2 = 3'b0;
                     count3 = 3'b0;
                     count4 = 3'b0;
                     end
       else if(sig_in[1]) begin
                      for(N = 0; N < 9 ; N = N + 1) begin
                            count1 = count1 + overlay1[N];
                            count2 = count2 + overlay2[N];
                            count3 = count3 + overlay3[N];
                            count4 = count4 + overlay4[N];
                     end
                     write_data[0] <= (count1 >= 5) ? 1'b1 : 1'b0;
                     write_data[1] <= (count2 >= 5) ? 1'b1 : 1'b0;
                     write_data[2] <= (count3 >= 5) ? 1'b1 : 1'b0;
                     write_data[3] <= (count4 >= 5) ? 1'b1 : 1'b0;
                     sig_out <= 4'b0010;
                     end
       else if (sig_in[0]) begin
                     sig_out <= 4'b0001;
                     write_data <= {11'b0,write_data[3:0]};
       end else begin
                     sig_out <= 4'b0000;
       end
end
assign overlay1 = ~({input_data[10:8],input_data[6:4],input_data[2:0]} ^
weight_data[8:0]);
assign overlay2 = ~({input_data[11:9],input_data[7:5],input_data[3:1]} ^
weight_data[8:0]);
assign overlay3 = ~({input_data[14:12],input_data[10:8],input_data[6:4]} ^
weight_data[8:0]);
assign overlay4 = ~({input_data[15:13],input_data[11:9],input_data[7:5]} ^
weight_data[8:0]);
endmodule
```

Controller

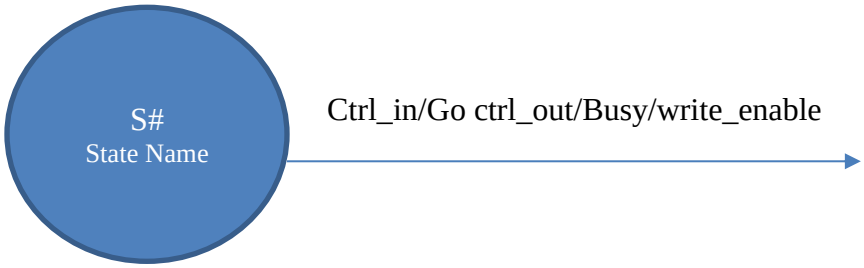
The controller is designed as state machine that will changed based on various inputs from the datapath. It is only interfaced through the SRAM through the write_enable signal. There are 6 states that it will cycle through.

Signal Name	I/O/Internal	Weight (bits)
clk	Input	1
reset	Input	1
go	Input	1

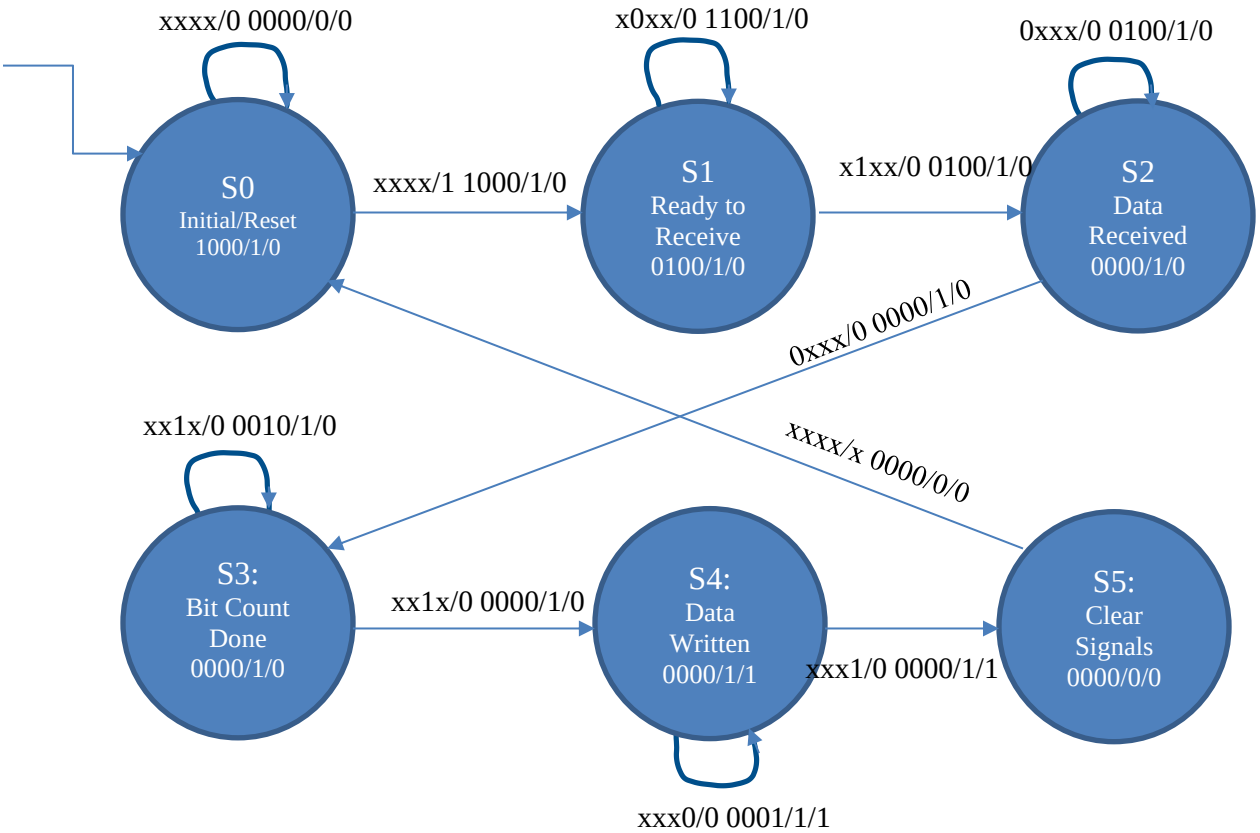
busy	Output	1
write_enable	Output	1
ctrl_in	Input	4
ctrl_out	Output	4
current_state	Internal reg	3
next_state	Internal reg	3

The input to the state machine is ctrl_in which is 4 bits and go
The output to the state machine is ctrl_out which is 4 bits, Busy, and write_enable.

Example:



State Machine:



```
module controller( // signals from top module and SRAM
                                    clk, reset, go, busy, write_enable,
                                    // controller <-> datapath
                                    ctrl_in, ctrl_out
);
input clk;
input reset;
input go;
output reg busy;
output reg write_enable;

input [3:0] ctrl_in; // [3] = got_data, [2] = ready_to_receive, [1] = count_done, [0] =
written_success
output reg [3:0] ctrl_out; //  [3] = get_address, [2] = get_data, [1] = count_ready,
[0] = write_ready

reg [2:0] current_state, next_state;

parameter
       s0 = 3'b000,
       s1 = 3'b001,
       s2 = 3'b010,
       s3 = 3'b011,
       s4 = 3'b100,
       s5 = 3'b101;

always @ (posedge clk)
       begin
       if(!reset) begin
              current_state <= s0;
              end
       else begin
              current_state <= next_state;
              end
       end
always @ *
       begin
       case (current_state)
       s0: begin
              if(go) begin
                     write_enable <= 1'b0;
                     busy <= 1'b1;
                     ctrl_out <= 4'b1000;
                     next_state <= s1;
              end
              else begin
                     write_enable <= 1'b0;
                     busy <= 1'b0;
                     ctrl_out <= 4'b0000;
                     next_state <= s0;
              end
       end
       s1: begin
              if(ctrl_in[2]) begin
                     write_enable <= 1'b0;
                     busy <= 1'b1;
                     ctrl_out <= 4'b0100;
                     next_state <= s2;
              end
              else begin
                     write_enable <= 1'b0;
```

```
                busy <= 1'b1;
                ctrl_out <= 4'b1100;
                next_state <= s1;
        end
    end
    s2: begin
        if(ctrl_in[3]) begin
                write_enable <= 1'b0;
                busy <= 1'b1;
                ctrl_out <= 4'b0000;
                next_state <= s3;
        end
        else begin
                write_enable <= 1'b0;
                busy <= 1'b1;
                ctrl_out <= 4'b0100;
                next_state <= s2;
        end
    end
    s3: begin

        if( ctrl_in[1]) begin
                write_enable <= 1'b0;
                busy <= 1'b1;
                ctrl_out <= 4'b0000;
                next_state <= s4;
        end
        else begin
                write_enable <= 1'b0;
                busy <= 1'b1;
                ctrl_out <= 4'b0010;
                next_state <= s3;
        end
    end
    s4: begin
        if(ctrl_in[0]) begin
                write_enable <= 1'b1;
                busy <= 1'b1;
                ctrl_out <= 4'b0000;
                next_state <= s5;
        end
        else begin
                write_enable <= 1'b1;
                busy <= 1'b1;
                ctrl_out <= 4'b0001;
                next_state <= s4;
        end
        end
    s5: begin
                write_enable <= 1'b0;
                busy <= 1'b0;
                ctrl_out <= 4'b0000;
                next_state <= s0;
        end
    default: begin
                write_enable <= 1'b0;
                busy <= 1'b0;
                ctrl_out <= 4'b0000;
                next_state <= s0;
        end
endcase
```

```
end
endmodule
```

Verification

There were two methods used to verify that all of the parts of the design functioned correctly. The original part of the verification plan was to verify that my design functions properly, plan to create a more exhaustive test bench with hand verified examples of different input matrices and verify them according to a golden output data file. Plan to perform timing verification and design check using the different synthesis tools we have used in class. As well as hand verify a timing diagram and compare with the output waveform. Using modelsim and the testbench file found within this ZIP file, we were able to verify that our design matched the golden output file.
The following commands were used to run our design:

```
[djhanse2@grendel25 test_project]$ vlog *.v
Model Technology ModelSim SE-64 vlog 2019.2 Compiler 2019.04 Apr 16 2019
Start time: 19:08:43 on Nov 17,2021
vlog controller.v datapath.v MyDesign_final.v MyDesign_init.v MyDesign.v
-- Compiling module controller
-- Compiling module datapath
-- Compiling module MyDesign
Top level modules:
MyDesign
End time: 19:08:44 on Nov 17,2021, Elapsed time: 0:00:01
Errors: 0, Warnings: 2
[djhanse2@grendel25 test_project]$ vlog testbench.sv
Model Technology ModelSim SE-64 vlog 2019.2 Compiler 2019.04 Apr 16 2019
Start time: 19:08:48 on Nov 17,2021
vlog testbench.sv
-- Compiling module sram
-- Compiling module tb_top
Top level modules:
tb_top
End time: 19:08:48 on Nov 17,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
[djhanse2@grendel25 test_project]$ vsim -voptargs=+acc tb_top
Reading pref.tcl
```

After running the simulation, the following indicated that we had correctly calculated the results.

```
# ------------------------------start_simulation------------------------------
# ------------------------------Round 0 start---------------------------------

# --------------------------------Round 0 check start--------------------------------
#
# --------------------------------store results to g_result.dat-------------------------
------
#
# --------------------------------load results to output_array-------------------------
-----
#
# --------------------------------load results to golden_output_array------------------
------------
```

```
#
# ------------------------------Round 0 start compare ------------------------------
#
# ------------------------------Round 0 Your report------------------------------
#
# Check 1 : Correct g results = 1/1
# computeCycle=1
# ----------------------------------------------------------------------------------
#
# ** Note: $finish    : testbench.sv(232)
#    Time: 775 ns  Iteration: 1  Instance: /tb_top
# 1
# Break in Module tb_top at testbench.sv line 232
```

After we had verified with multiple different matrices by updating the .DAT files found in our input_0 directory, we moved on to synthesizing our design using design vision.
The following commands were run in Design Vision, for the sake of space, not all the results will be posted here, but a few important notes will be made.

```
Add synopsys2019
Source setup.tcl
Source read.tcl
Check_design
Source Constraints.tcl
Source CompileAnalyze.tcl
Report_timing
Report_area
```

After running the setup.tcl file, the desired clock period was confirmed, which was set to 3.9. Then, setup.tcl was ran. Initially, there were quite a few errors that signified that there were some design errors. This was fixed by fixing inferred logic in the design. Eventually, read.tcl ran with no errors and showed we had all flip-flops. Check design was ran, and there were a few ignorable warnings. Constraints.tcl also gave a few ignorable warnings. CompileAnalyze.tcl gave a lot of warnings, all of which were ignorable. Eventually report_timing and report_area were ran, which showed that the Slack time was met, and the area was not out of control. All this synthesis confirmed that our design worked and was buildable in hardware.

Results Achieved

There were 3 main results that needed to be recorded and achieved. The compute cycle, the clock period, and the area.
With the compute Cycle, we can see we achieved this in 1 cycle from the output of our modelsim simulation.

```
# ------------------------------Round 0 Your report------------------------------
#
# Check 1 : Correct g results = 1/1
# computeCycle=1
# ----------------------------------------------------------------------------------
```

For the clock period, we set it to be 3.9 in our setup of our synthesis. After we synthesized, we can see that we achieved this clock period. This is found in timing_max_slow_holdfixed_tut1.rpt.

```
clock clk (rise edge)                    3.9000     3.9000
clock network delay (ideal)              0.0000     3.9000
clock uncertainty                       -0.0500     3.8500
u1/write_data_reg[2]/CK (DFF_X2)         0.0000     3.8500 r
library setup time                      -0.3152     3.5348
data required time                                  3.5348
-----------------------------------------------------------
data required time                                  3.5348
data arrival time                                  -3.5339
-----------------------------------------------------------
slack (MET)                                         0.0009
```

From this report we can also see that we met our slack time.
We can also verify our area to be 524.552 from our cell_report_final.rpt.

```
Total 415 cells                                        524.5520
```

Conclusions

This project was successful by completing the initial goal of implementing a fixed size single stage of a binary convolutional neural network in hardware. Overall, improvements in the cell area could be made, as well as being able to reduce the clock period without violating the slack time.