ECE 564: Hardware Accelerator for a Simple Neural Network

Project Responsibilities: All modules, control and datapath, and verification of the modules and the total design. ID #: 200205676 Name: Kevin Volkel

Delay (ns to run provided provided example). Clock period: 15 ns # cycles: 572	Logic Area: (um^2) 42582.08 Memory: N/A	1/(delay.area) ($ns^{-1}.um^{-2}$) $2.74 * 10^{-9}$
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

A hardware accelerator was designed in order to accelerate the calculations for two layers of a simplified version of a Convolutional Neural Network. The hardware that was designed reads from two different SRAMs and calculates two different sets of data, an intermediate set and an output set. The intermediate values are results from the first layer calculations. During the second layer calculations, intermediate values and filter vectors are operated upon to produce the output data. One of the SRAMs holds a 12x12 input array, and the other holds 2 sets of filter vectors, where 1 set is used for the first layer and the 2^{nd} set is used for the 2^{nd} layer. During the design of the accelerator, the fastest possible design was striven for while considering how the area would be affected by different design choices that increase the speed of the design. Some key innovations that were employed to achieve a fast design were to load each first step vector into its own register file, and to load each intermediate value into a register file. This allows for the hardware to exploit parallelism that can be found in the algorithm since loading into register files circumvents limited SRAM bandwidth and allows for multiple calculations to be performed at once. The designed hardware was also synthesized in order to obtain the area and the clock period of the design.

Introduction:

Hardware was designed and implemented in order to perform computations for 2 layers of a simplified Convolutional Neural Network. The designed hardware reads from two SRAMs, performs 2 steps of computations using the contents, and outputs the final results to an output SRAM. The first step of computation is the calculation of intermediate values that are obtained from applying a dot product between 4 different filter vectors and each of the four 3x3 sub-quadrants of the four 6x6 quadrants of the 12x12 input array. The intermediate values are arranged into a 64 element vector. Dot products are then performed with the 64 element intermediate array and 8 more 64 element filter vectors. The 8 results from these dot products are the final outputs of the simple Convolutional Neural Network. Because of bandwidth restrictions on the SRAMs (only one value can be either read or written to an SRAM in one clock cycle), innovations were made in order leverage parallelism that is in the algorithm. These innovations include pre-loading the first step filter vectors into their own register files, and writing intermediate values to their own register file instead of writing them back to the scratchpad space inside the input SRAM memory. As will be seen later in this report, pre-loading the first step vectors allows for all 4 dot products for each 3x3 sub-quadrant to be calculated in parallel, and also frees up the bandwidth of the vector memory SRAM so that step 2 calculations can be done while the step 1 calculations are still completing. Putting intermediate results into their own register file also makes it so that intermediate results do not use up input memory bandwidth to write back to the scratch pad space in the input memory. In this report, the hardware algorithm will be discussed and the high level architecture of the system will be presented with a short discussion about the data flow in the design. Also, the interface of the design will be discussed along with the meaning of the signals at the interface. Technical Implementation will be presented after the interface, and will go into more detail about the hardware in the lower levels of the design hierarchy. Finally, verification will be discussed and results for area, clock period, and number of cycles to completion will be presented.

Micro-Architecture:
Below is pseudo code that captures the algorithm that is being designed for. This pseudo code is not at a level where hardware can be designed, but instead, is presented in order demonstrate how the target algorithm works. After the pseudo code, the algorithm used by the hardware is presented.

```
//Y is a 2x2x4 array (row x column x layer)
Y = [2x2x4]
Z=[4x4x4]
L=[64x1]
O=[8x1]
P=[9x1]
//Step 1, (O,N) is quadrant position in 12x12 array. (0,0) top left, (1,1) bottom right,
//(1,0) bottom left, (0,1) top right
for each 6x6  quadrant Q in the 12x12 input array with position (O,N)
        //(U,W)=(0,0) for top left (0,1) for top right (1,0) for bottom left and (1,1) for
        //bottom right 3x3 quadrant
        for each 3x3 quadrant T with position (U,W) in Q
                P = T arranged in row major order
                for each filter vector b
                        Y[U,W,b]=Dot(P,b)
                        //calculate min
                        Y[U,W,b]=min(Y[U,W,b])
        //place min value into Z
        //O,N is the quadrant position of Q
        Z[O,N,:]=Y
//Step 2
L = Z arranged in row major and layer order
for each filter vector m
        O[m]=Dot(L,m)
        O[m]=min(O[m])
```

- Read the vector SRAM and preload *b* vectors into their own register files.
- Once *b* vectors are pre-loaded, begin calculations on the 12x12 input array by performing dot products (multiply accumulates) between each *b* vector and each 3x3 sub-quadrant, in the quadrant order of (0,0), (0,1), (1,0), (1,1). The 3x3 sub-quadrants are read from the input memory SRAM in the following positional order (0,0), (0,1), (1,0), (1,1). Where (x,y) represents the sub-quadrant position in a 6x6 quadrant.
- After reading a sub-quadrant element, the calculations for that sub-quadrant element will be done since all four dot products for a single sub-quadrant are done at the same time. The 4 values that correspond to these 4 dot product results are intermediate results and are transferred to the intermediate value register file on the same cycle that the first operation of the dot products for the next 3x3 sub-quadrant is being performed.

- The process of calculating 4 dot product values is done continuously until all input data values are covered.
- After the first 4 values are transferred to the intermediate register file, the second step calculations are started and continued until all *m* vector dot products are finished.
- To perform step 2 calculations, the *m* vector elements are read from the vector SRAM, and the intermediate values are read from the intermediate register file.
- Because each intermediate value needs to be multiplied and accumulated with 8 different corresponding *m vector* components, it takes 8 cycles to complete all calculations for 1 intermediate value since the *m* vector components are being read 1 at a time out of the SRAM.
- After the 8 cycles for each intermediate value, the next intermediate value is read from the register file and the next 8 corresponding *m* vector components are fetched and operated upon. This process is repeated until the step 2 calculations are done. When done, the hardware signifies that all calculations are complete after the output data is written to the output SRAM.

From the hardware algorithm, the two tricks to get higher performance are mentioned. These tricks are pre-loading the *b* vectors into a register file, and storing the intermediate values into a intermediate value register file. There are two benefits from pre-loading *b* vectors. One benefit is that now the filter vector SRAM is free, allowing for *m* vector elements to be fetched and used while the first step is still completing calculations. Another benefit is the opportunity for parallelism because now all 4 dot products for each sub quadrant can be performed in parallel. For example, for an input array element, all 4 corresponding *b* vector elements can be read from their respective register file and all operations related to an input array element and all 4 corresponding *b* vector elements are completed in 1 cycle. Writing the intermediate values to a register file allows for parallelism amongst the 2 steps in the calculations. For example, instead of trying to write intermediate values back to the input array SRAM and interrupting step 1 calculations, the intermediate values can be written to a register file where they can be used for step 2 calculations while step 1 calculations are completing. The intermediate value register file is also a little different than the *b* vector register files because the intermediate value register file has 2 data ports and 2 address ports, one for writing, and one for reading. This simplifies the control for the register file and also allows the step 2 calculations to keep processing while new values are being written into the register file. Also, because the *m* vectors are not used in parallel, there does not need to be 8 units that perform dot products. Instead, one multiply and one adder with a multiplexer to select which dot product is being calculated can be used to reduce the overall area of the design.

Below in Figure 1 is a high level architectural drawing showing the top level of the design and Figure 2 is a high level drawing showing the main modules in the design and will be used to show how data flows into and out of the design. To reduce complexity, not all signals between the modules are shown in Figure 2. All needed signals for each module are shown in the technical implementation section of the report.

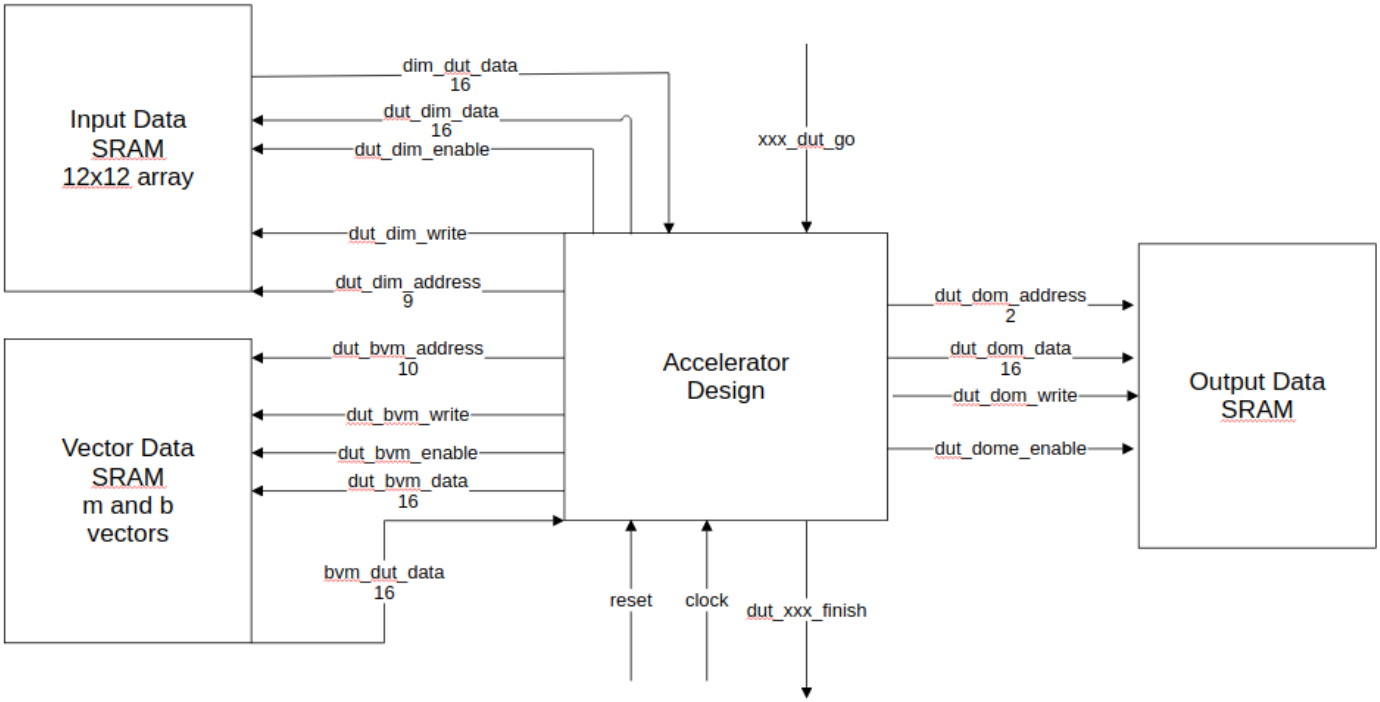

Figure 1: Architectural drawing showing the design and the interfaces to the input, vector, and output SRAMs.

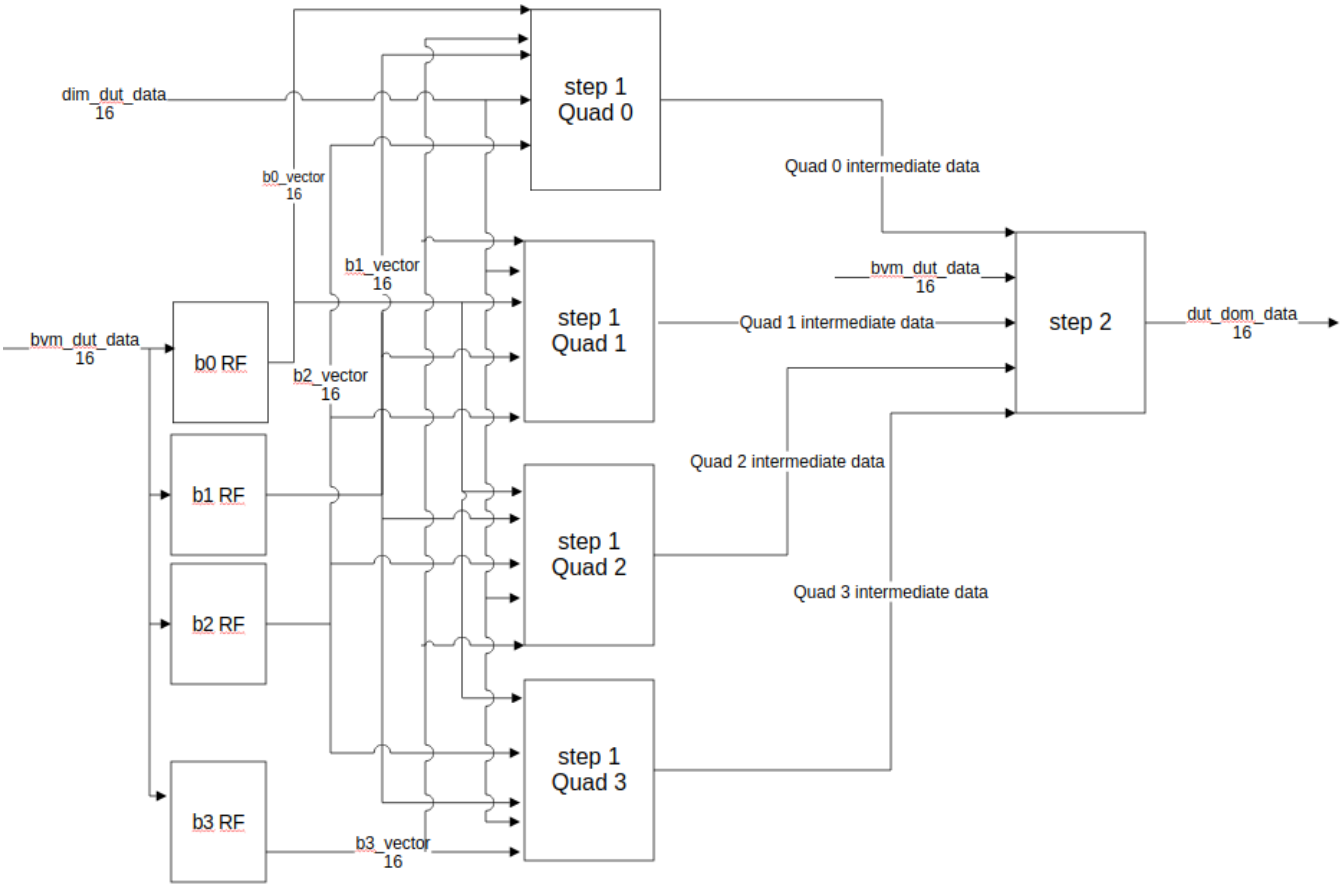

Figure 2: Main modules in the data path, used to describe how data flows from the inputs to the outputs.

In Figure 2, the data path is shown. The step 1 module instances handle the step 1 calculations for their respective quadrants. The outputs of these step 1 modules are inputs to the step 2 module, which handles calculations for step 2. Once calculations are finished, step 2 writes these values to the output SRAM. The blocks labeled bX RF are the register files for the *b* vectors, and each output of these register files are inputs to each step 1 instance along with the input data coming from the input SRAM. Although there are 4 step 1 instances, each instance does not do anything while one is performing its respective quadrant calculations. This is because it was determined to be not worthwhile to add another register file to pre-load the input data in order to perform all quadrant calculations in parallel.

Interface Specification:
This section provides a specification of the interface to the the hardware accelerator. The 3 structures that the design must interface to are the 3 different SRAMs. For this project it was specified that the inputs to the SRAMs must be registered, that is, the address and data ports to the SRAM must be driven by flip flops. This means that when the design's controller generates an address, the resulting data is not received until 2 cycles later. The finish output was also registered in order to make sure that all outputs are driven by flip flops. The names of the wires and buses that connect the design to outside components are shown in Figure 1, and a table below (Table 1) is provided to give a description of the purpose of each signal. When a signal is described as "not used" this just means that the signal is not asserted, if it is a data line thats unused then a constant value is just driven to it.

Table 1: Table showing the interfacing signals along with their respective direction, width and function.

Signal Name	Input or Output	Width	Description
dim_dut_data	input	16	Data from the input array SRAM.
dut_dim_data	output	16	Data to the input SRAM. Not used, scratch pad not used.
dut_dim_enable	output	1	Enable signal to the input SRAM. Must be high to read.
dut_dim_write	output	1	Write signal to input SRAM, not used (constantly 0).
dut_dim_address	output	9	Address for the input array SRAM.
bvm_dut_data	input	16	Data from the vector SRAM.
dut_bvm_data	output	16	Data from design to vector SRAM. Not used.
dut_bvm_enable	output	1	Vector SRAM enable. Must be high to read.
dut_bvm_address	output	10	Vector SRAM address.
dut_bvm_write	output	1	Write signal for vector SRAM. Not used.
dut_bom_data	output	16	Write data to the output SRAM.
dut_bom_enable	output	1	Enable signal to output SRAM. Must be high to write.
dut_bom_address	output	3	Address to the output SRAM

dut_bom_write	output	1	Write signal to the output SRAM. 1 indicates a write.
Reset	input	1	Global reset to the design. Puts design into a known state.
go	input	1	Signal that tells the design to begin calculations.
finish	output	1	Flag that indicates if the design can do another calculation. 0 for not ready, 1 for ready.

Technical Implementation:

This section presents some of the lower levels in the hierarchy of the design. Detailed schematics are presented for the Step 1, Step 2, and controller modules used for this design. In the Step 1 modules, there are other modules called dot modules which provide the functionality of multiply accumulation needed to implement the dot products, a schematic is also provided for this module. Also, because the controller module is so large, multiple figures are used to present the logic within the controller. Because the hardware follows a set of specific steps for each run, a master counter was used to implement the core of the controller. Based on this master counter, the control signals were derived. For some signals this involves just a simple logical function or decoding of the master counter, but for some other signals (such as address lines) their repetitive patterns are more easily implemented by using sets of counters that are regulated by the master counter. Tables 2 and 3 below show tables of the control signals and when they are supposed to occur for a certain master count number. It should be noted that this table was constructed for a system that assumed no registers in between the design and the SRAMs in order to easily see the patterns in the signals. With the addition of registers, all signals that are not signals that interface to the SRAMs need to be started at a master count that is 2 higher than the ones displayed in the table because it takes 2 cycles for data to arrive from the SRAM once an address is generated. More discussion for specific modules is provided under the following figures.

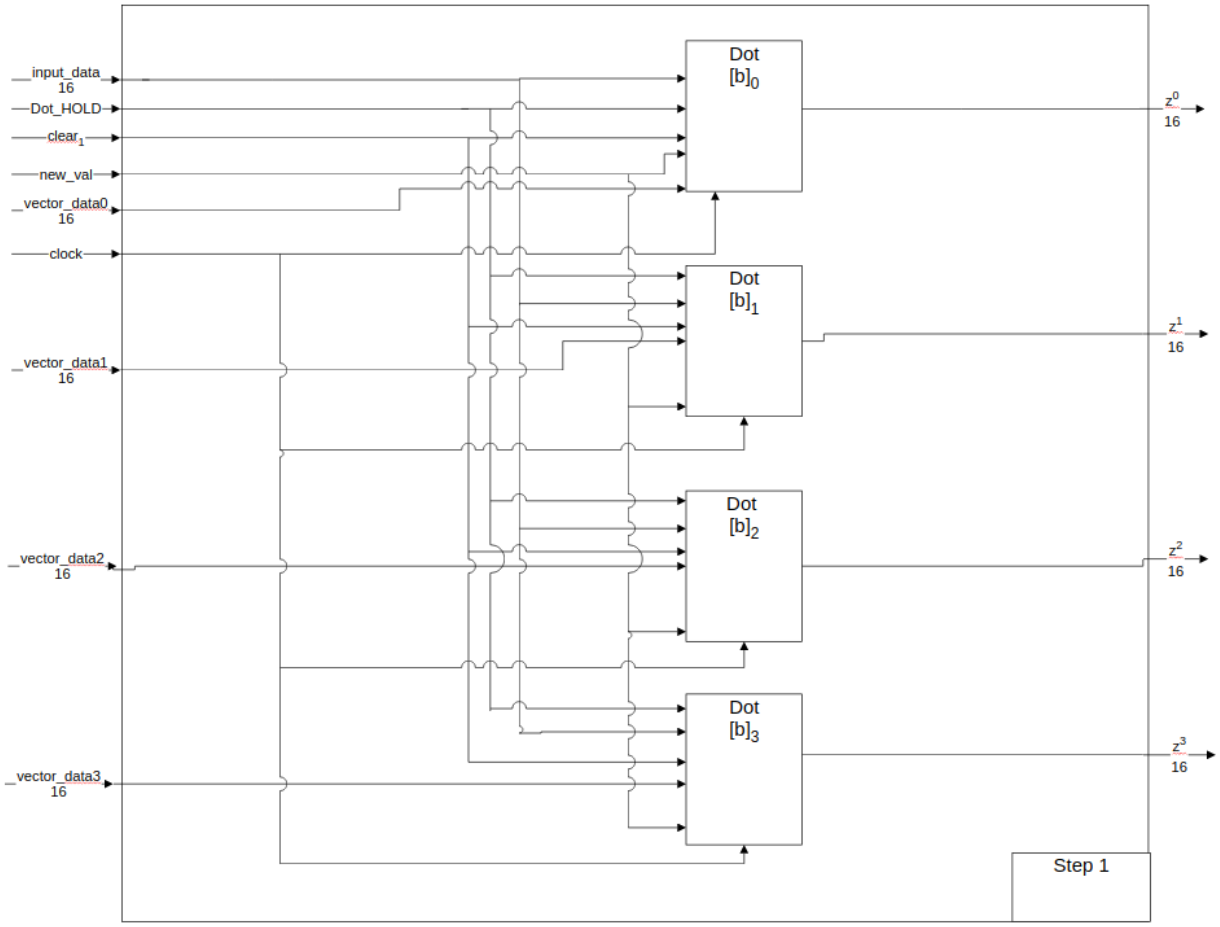

Figure 3: Figure showing the hierarchy of the Step 1 module, arrows on the left and right side of the drawing indicate the inputs and outputs of the module.

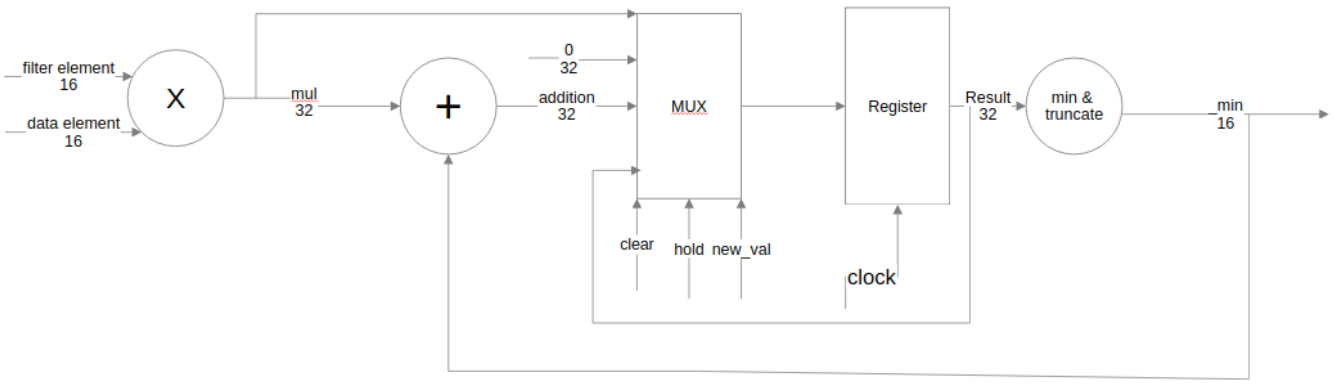

Figure 4: RTL design of the Dot modules used in the step 1 modules.

Figures 3 and 4 show the hierarchy of the step 1 module and show the detailed implementation of the Dot module. As seen from the figures, the step 1 module is made up of 4 instances of the Dot module. All of the Dot module instances share the same control lines and input array data that is inputted to the step 1 modules, and each Dot instance gets a filter element that corresponds to the dot product that the instantiated Dot unit is designed to calculate. Each Dot module is composed of a multiplier, adder, accumulation register, and min/truncate logic. When a new dot product is starting, the *new_val* signal needs to be asserted in order to load in just the first multiplication of the new dot product rather than the accumulation of the multiply and the result of the last calculated dot product. The *clear* signal clears the accumulation register to 0. The min & truncate combinational logic checks the most significant bit of the acculturation register named *Result*, if it is a 1 then the output *_min* is 0, if it is 0 then the output is the *Result* signal shifted 16 bits to the right.

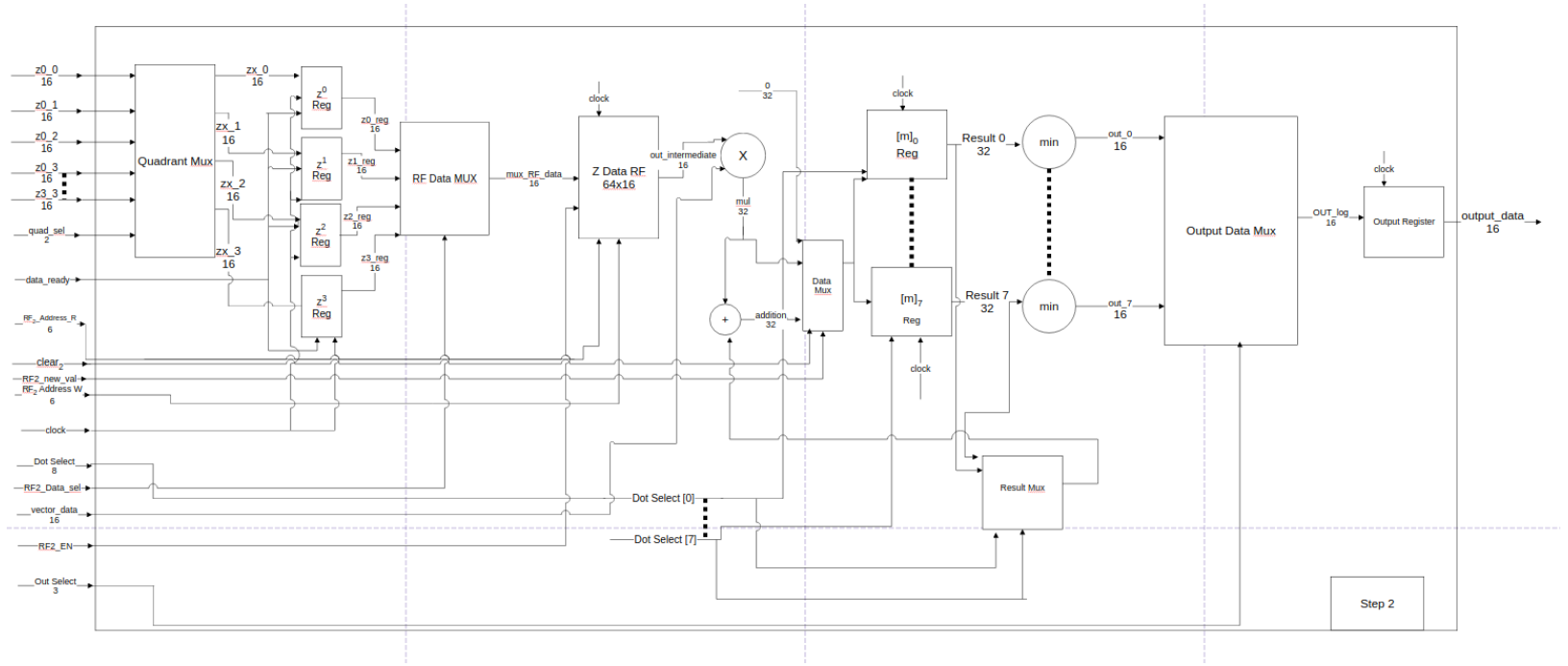

Figure 5: Detailed implementation of the step 2 module, arrows on the outside of the box surrounding the RTL diagram represent inputs and outputs of the module. All signals named inside the module correspond to named signals in the Verilog code for this module.

Figure 5 shows the detailed implementation for the Step 2 module. This module takes in the intermediate values calculated by the four step 1 instances, and selects which data it should load in to the Z Data RF. The selection line for the Quadrant Mux is generated in the controller and selects the input data based on which quadrant is being operated upon. The data output from the Quadrant Mux is only loaded into the register file when the *data_ready* signal is asserted which signals that a 3x3 sub-quadrant has been finished. When the *data_ready* signal is asserted, the Quadrant Mux outputs are latched into 4 registers labeled z^0...z^3, and these registers latch the values that need to be written to the register file. The purpose of these registers is to keep the register file input data constant

as the next 3x3 sub-quadrant is being operated on in the step 1 modules. The outputs of these registers are inputs to the RF Data Mux which selects between which of the values should be presented to the register file. The select lines for this mux are also generated by the controller and start by first selecting the z^0 data. After writing the z^0 data, the z^1, z^2, and z^3 data are written to the register file in that order. To prevent overwriting, the mux control lines and register file write address are held constant to the z^3 data and address until the next set of data is brought into the module. While data is being brought into the module and written to the register file, the step 2 calculations are being completed. This is possible because the register file has a read address port and a write address port, allowing for data to be read out from the file while data is being written in. This kind of register file simplifies the control because the step 2 calculations do not need to be halted every time new data needs to be written into the register file. Also, this kind of register file allows for easier address generation since there are now only two addresses that follow simple patterns, rather than one address that follows a complex pattern in order to juggle write and read addresses.

The step 2 module performs the step 2 calculations by reading an intermediate value from the register file. This intermediate value and a corresponding filter vector data element from an *m* vector are multiplied together, and this multiplication is added with one of 8 registers with outputs labeled *Result0, Result1, Result2….Result7*. The registers with these labels are the registers that accumulate the dot product for each of the 8 *m* filter vectors. For example *Result0* corresponds to the filter vector m_0. The *Result* signal chosen must be the signal that corresponds to the *m* vector that the *vector_data* input is an element of. It should be noted that it takes 8 clock cycles to complete all computations for a single intermediate value. This is because there are 8 filter vectors, and each filter vector has 1 element that corresponds to an intermediate value. Once all 8 calculations for an intermediate value are finished, the next intermediate value is read from the register file and the 8 calculations for that intermediate value are completed. This process is done until all 64 intermediate values are processed. It should be noted that instead of multiplexing the *Result* signals, 8 instances of the Dot module could be used. Although this would make control simpler, this wastes a lot of area considering only a part of 1 dot product is being calculated each cycle. This results in the other 7 Dot instances being unused each cycle, wasting a lot of area.

After the calculations are completed for step 2, the output data needs to be written to the output SRAM. This is done by multiplexing the outputs of the *min & truncate* blocks for each of the *Result* registers. The output of the mux is the input of the register that interfaces to the output SRAM. The control signals and output SRAM address signals are generated by the controller which will be discussed next.

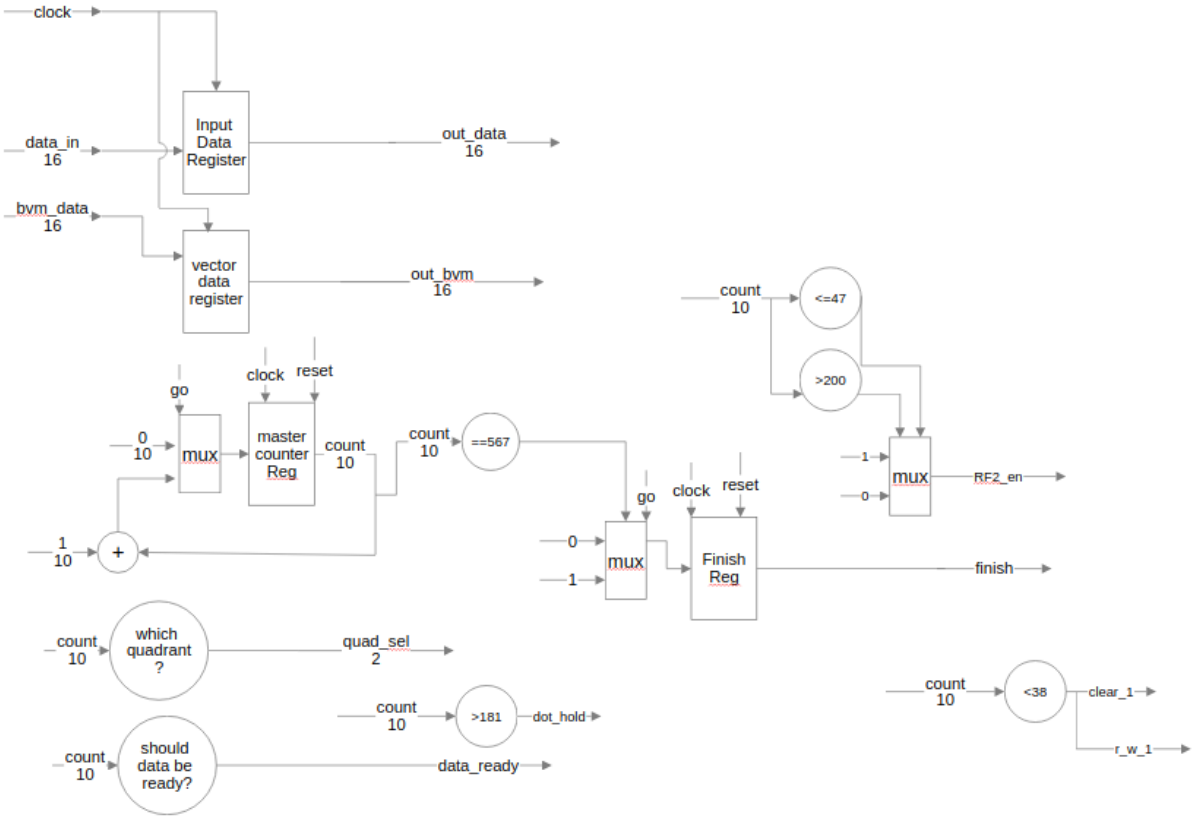

Figure 6: Block diagram for the data coming from the input array SRAM and the vector array SRAM. The registers used to register this data were put in the controller because most of the interfacing to these SRAMs is already inside the controller module. This figure also includes the quad_sel, dot_hold, data_ready, finish, RF2_en, clear_1, and r_w_1 signals.

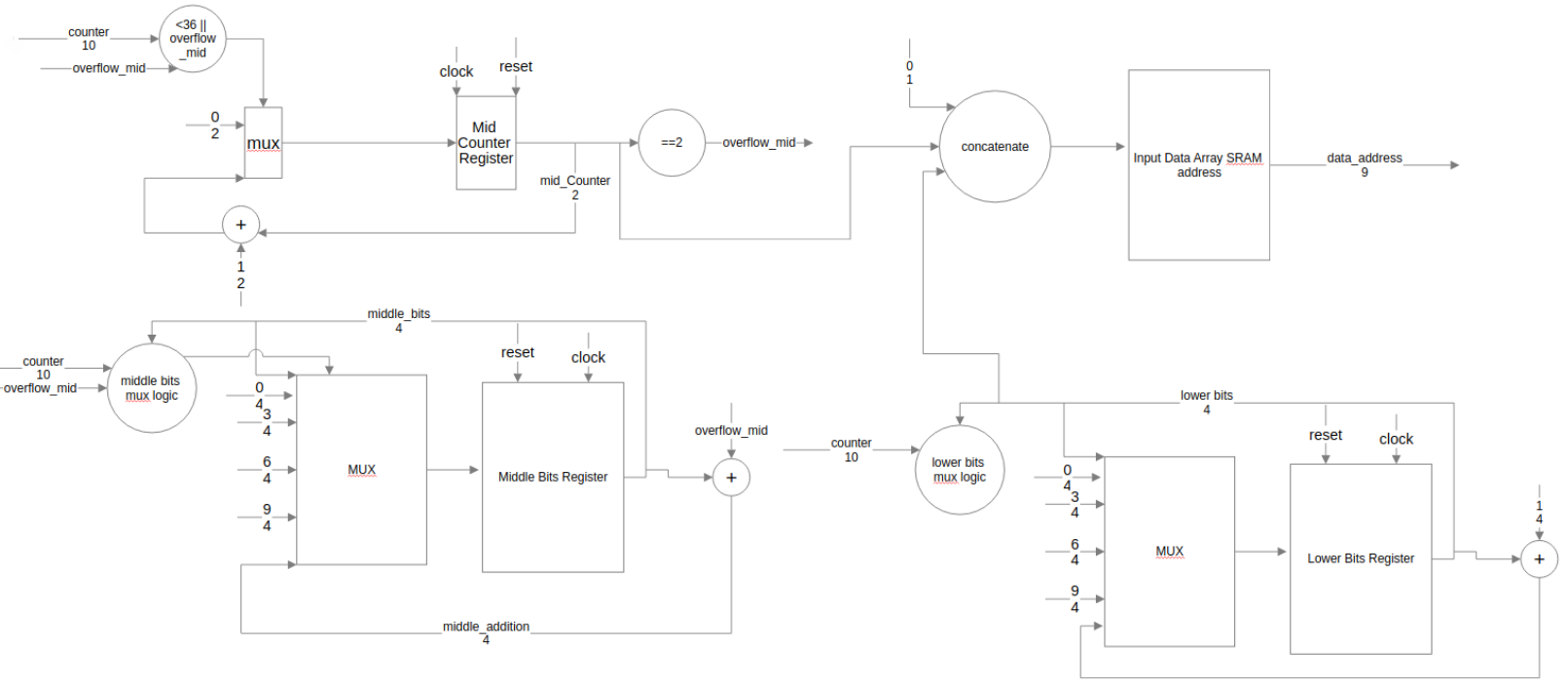

Figure 7: Diagram of the circuit used to calculate the addresses for the input SRAM. When the master counter reaches a certain value, this circuit starts calculating addresses for the input SRAM each cycle, starting with the (0,0) quadrant and ending with addresses for the (1,1) quadrant. Once started, a new address is calculated each cycle until the input SRAM is completely traversed.

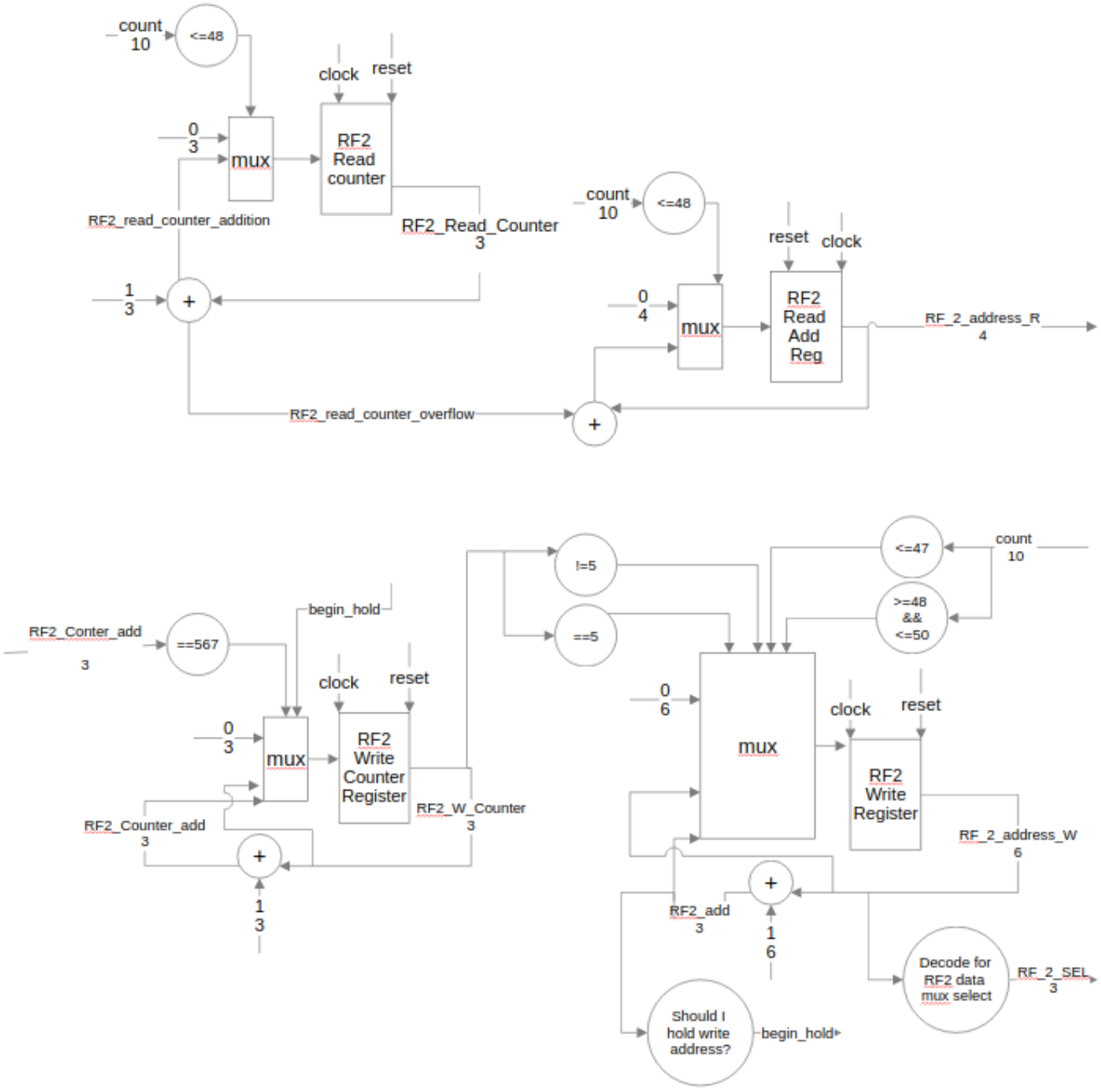

Figure 8: Circuit used to calculate the write addresses and the read addresses for the second register file in the step 2 module. The read address increments every 8 cycles once the master counter reaches a point at which the first intermediate values have been loaded into the register file. The read address starts from 0 and increments every 8 cycles up to 63. The increment has to wait 8 cycles because 8 filter vector elements need to be read in each cycle for each RF2 entry. The write address also starts from 0 and increases to 63, and increments every cycle, unless there are no values ready to be written. In this case the RF2 write address is held constant.

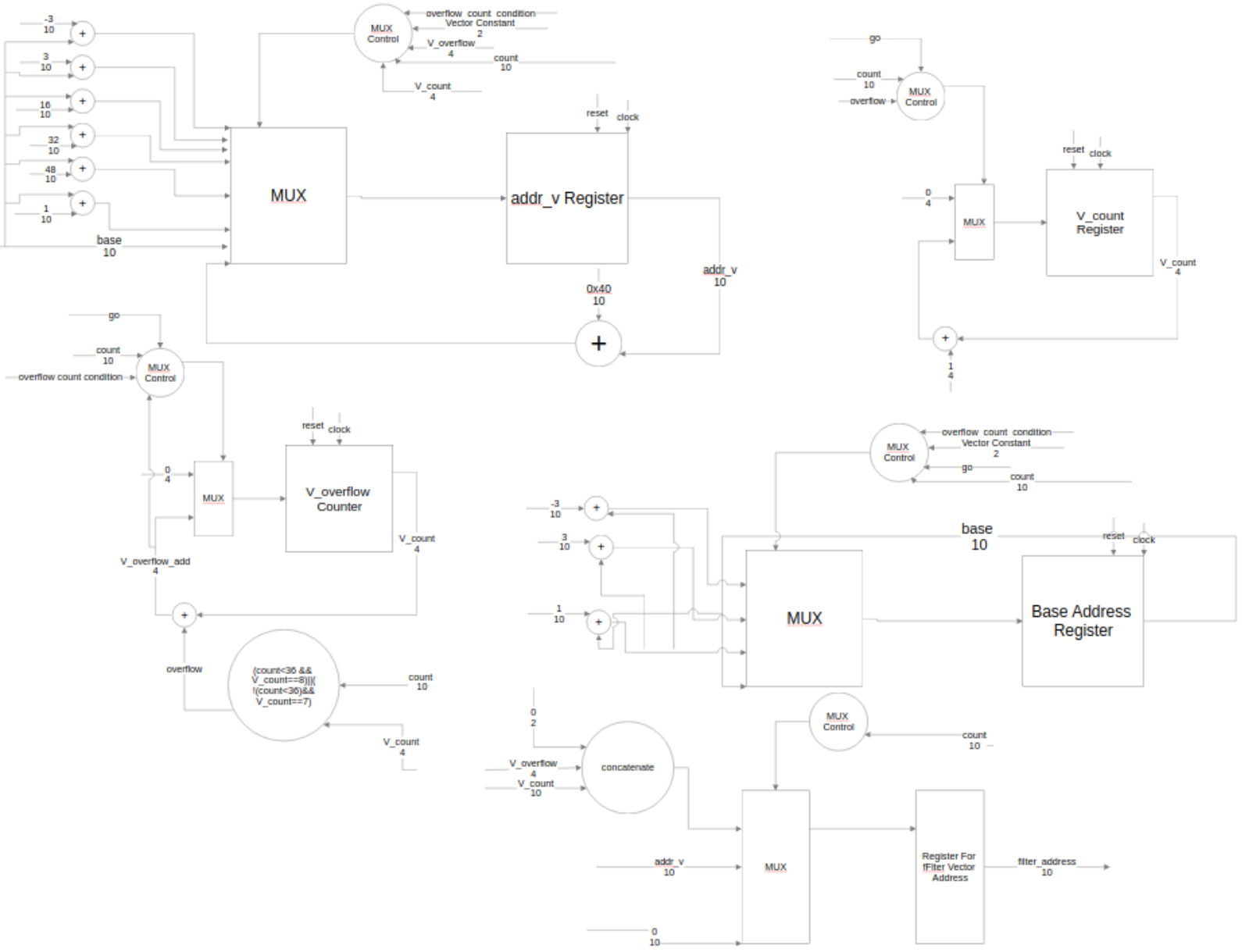

Figure 9: Circuit used to calculate the filter vector addresses for the filter vector SRAM. This address calculation is the most complex in the design, and the pattern will be discussed further.

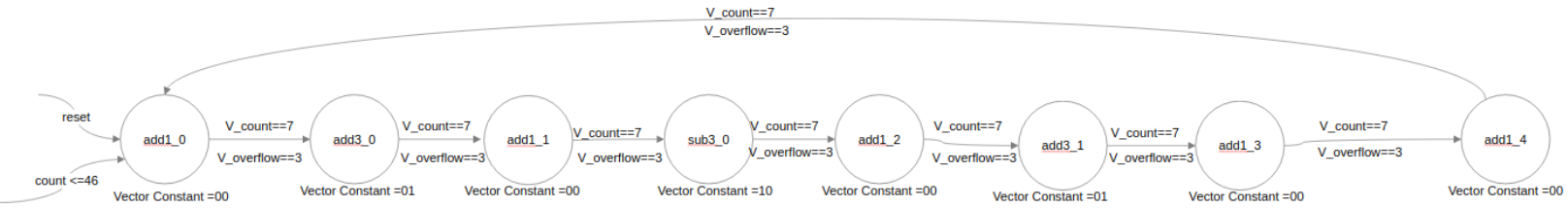

Figure 10: Moore FSM used to aid in the calculation of the filter vector SRAM addresses.

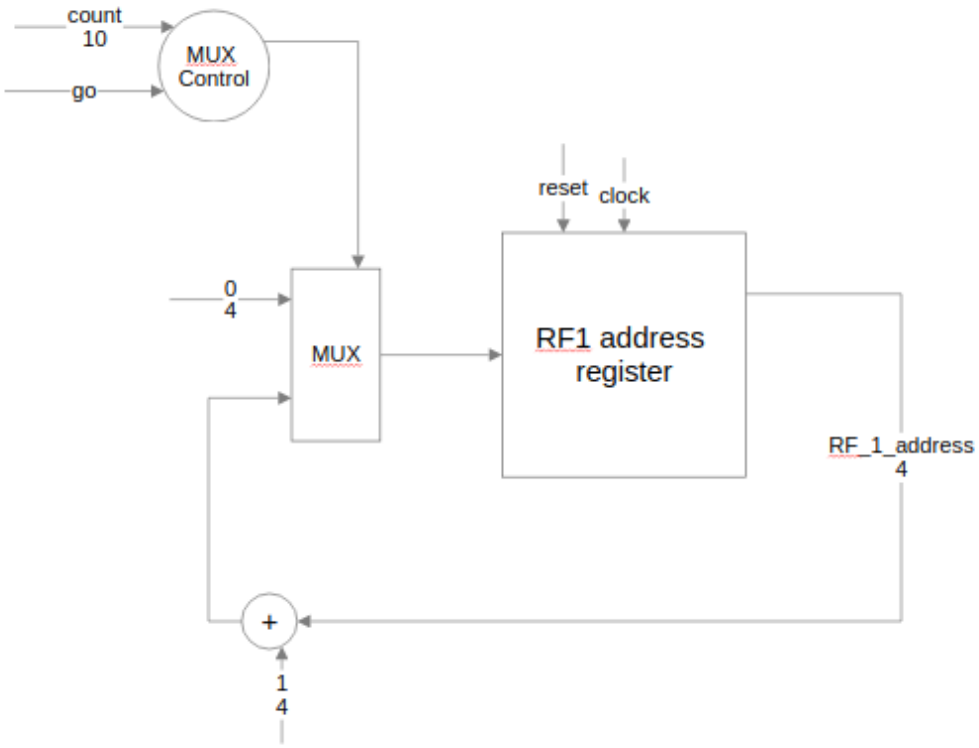

Figure 11: Circuit used to generate the RF_1_address. This address is generated by a simple counter that increments each cycle until it reaches a value of 8. When the counter reaches 8, the counter overflow and starts back at 0.

Table 2: Table used to derive the logic needed to decode the master counter for the step 1 control signals. To compress the table, signals are defined in a range of counter values. The Count Lower column is the count value for a lower part of a range and the Count Upper column is the upper count value of a range.

Count lower		Count Upper	Data_address	Vector_ Address	clear1	Dot Hold	RF1 Add	RF W Sel	R/w1	data_ready
0	loading vectors	8	x	0x00-0x08	1	0	0-8	0	1 (write)	
9	loading vectors	17	x	0x10-0x18	1	0	0-8	1	1	0
18	loading vectors	26	x	0x20-0x28	1	0	0-8	10	1	0
27	loading vectors	35	x	0x30-0x38	1	0	0-8	11	1	0
36	//start calculating quad 1 quad_sel=00	38	0x000-0x002	x	0	0	0-2	x	0 (read)	0
39		41	0x010-0x012	x	0	0	3-5	x	0	0
42		44	0x020-0x022	x	0	0	6-8	x	0	1(44)
45		47	0x003-0x005		0	0	0-2	x	0	0
48		50	0x013-0x015		0	0	3-5	x	0	0
51		53	0x023-0x025		0	0	6-8	x	0	1 (53)
54		56	0x030-0x032		0	0	0-2	x	0	0
57		59	0x040-0x042		0	0	3-5	x	0	0
60		62	0x050-0x052		0	0	6-8	x	0	1 (62)
63		65	0x033-0x035		0	0	0-2	x	0	0
66		68	0x043-0x045		0	0	3-5	x	0	0
69		71	0x053-0x055		0	0	6-8	x	0	1 (71)
72	//calculate quad 2 Quad sel=01	74	0x006-0x008		0	0	0-2	x	0	0
75		77	0x016-0x018		0	0	3-5	x	0	0
78		80	0x026-0x028		0		6-8	x	0	1 (80)
81		83	0x009-0x00B		0		0-2	x	0	0
84		86	0x019-0x01B		0		3-5	x	0	0
87		89	0x029-0x02B		0		6-8	x	0	1 (89)
90		92	0x036-0x038		0		0-2	x	0	0
93		95	0x046-0x048		0		3-5	x	0	0
96		98	0x056-0x058		0		6-8	x	0	1 (98)
99		101	0x039-0x03B		0		0-2	x	0	0
102		104	0x049-0x04B		0		3-5	x	0	0
105		107	0x059-0x05B		0		6-8	x	0	1 (107)
108	//calculate quad 3 Quad sel=10	110	0x060-0x062		0		0-2	x	0	0
111		113	0x070-0x072		0		3-5	x	0	0
114		116	0x080-0x082		0		6-8	x	0	1 (116)
117		119	0x063-0x065		0		0-2	x	0	0
120		122	0x073-0x075		0		3-5	x	0	0
123		125	0x083-0x085		0		6-8	x	0	1(125)
126		128	0x090-0x092		0		0-2	x	0	0
129		131	0x0A0-0x0A2		0		3-5	x	0	0
132		134	0x0B0-0x0B2		0		6-8	x	0	1 (134)
135		137	0x093-0x095		0		0-2	x	0	0
138		140	0x0A3-0x0A5		0		3-5	x	0	0
141		143	0x0B3-0x0B5		0		6-8	x	0	1 (143)
144	//calculate quad 4 Quad sel=11	146	0x066-0x068		0		0-2	x	0	0
147		149	0x076-0x078		0		3-5	x	0	0
150		152	0x086-0x088		0		6-8	x	0	1 (152)
153		155	0x069-0x06B		0		0-2	x	0	0
156		158	0x079-0x07B		0		3-5	x	0	0
159		161	0x089-0x08B		0		6-8	x	0	1 (161)
162		164	0x096-0x098		0		0-2	x	0	0
165		167	0x0A6-0x0A8		0		3-5	x	0	0
168		170	0x0B6-0x0B8		0		6-8	x	0	1 (170)
171		173	0x099-0x09B		0		0-2	x	0	0
174		176	0x0A9-0x0AB		0		3-5	x	0	0
177		179	0x0B9-0x0BB		0		6-8	x	0	1 (179)
	>179		X (disable)		0	1	0	x	0	

Table 3: Table used to derive the logic needed to decode the master counter for step 2 control signals.

Count	Clear 2	Out select	Dot select (which dot unit to be active)	RF2 W add	RF2 R add	Vector mem address	Out address	Out Mem EN	RF2 Data Select
44	1	x	x	x	x	Step 1 Table	x	0	x
45	1	x	x	0	x	x	x	0	0
46	1	x	x	1	x	x	x	0	1
47	0	x	0	2	0	0x40	x	0	2
48	0	x	1	3	0	0x40+0x40	x	0	3
49	0	x	2	3	0	0x40+2(0x40)	x	0	3
50	0	x	3	3	0	0x40+3(0x40)	x	0	3
51	0	x	4	3	0	0x40+4(0x40)	x	0	3
52	0	x	5	3	0	0x40+5(0x40)	x	0	3
53	0	x	6	3	0	0x40+6(0x40)	x	0	3
54	0	x	7	4	0	0x40+7(0x40)	x	0	0
55	0	x	0	5	1	0x40+16	x	0	1
56	0	x	1	6	1	(0x40+16)+0x40	x	0	2
57	0	x	2	6	1	(0x40+16)+2(0x40)	x	0	3
58	0	x	3	6	1	(0x40+16)+3(0x40)	x	0	3
59	0	x	4	6	1	(0x40+16)+4(0x40)	x	0	3
60	0	x	5	6	1	(0x40+16)+5(0x40)	x	0	3
61	0	x	6	6	1	(0x40+16)+6(0x40)	x	0	3
62	0	x	7	6	1	(0x40+16)+7(0x40)	x	0	3
63	0	x	0	7	2	(0x40+32)	x	0	0
64	0	x	1	8	2	(0x40+32)+0x40	x	0	1
65	0	x	2	9	2	(0x40+32)+2(0x40)	x	0	2
66	0	x	3	10	2	(0x40+32)+2(0x40)	x	0	3
513	0	x	0	x	0	(0x1C0+15)	x	0	3
557-564	0	0-7	//need to have a hold on dots	x	x	x	0-7	1 (558-565)	
567								0	

From Tables 2 and 3, it can be seen that many control signals can be simply generated by doing simple comparison operations on the count value. Some signals are not as simple and require counters, like all of the addresses needed for the register files and the SRAMs. Some of these counters follow simple patterns, like the register files and the input SRAM, but the filter SRAM requires a complex pattern in order to get the correct filter vector values for the intermediate values.

When the first 4 (z^0, z^1, z^2, z^3) intermediate values are ready to be used, 8 filter vector values need to be fetched for each one. For the very first intermediate value, the corresponding element for each filter vector is the first element. The first element for the first filter vector resides at address 0x40, and because the filter vectors are 64 elements long, the next filter vector's first element resides at 0x40+0x40. This pattern continues for each filter vector. When the next intermediate value needs to be handled, the corresponding filter vector element is the position of the filter element used for the last intermediate value plus 16. This is because if the intermediate values are arranged in a 4x4x4 matrix like the algorithm dictates, and then this 4x4x4 matrix is arranged in row major, then it will be seen that each 3x3 sub-quadrant intermediate value is 16 positions away from the last intermediate value for the sub-quadrant. For example, for the first 4 intermediate values loaded into the register file, the position of the first filter vector's element for the second intermediate entry will be 0x40+16, and the previously discussed pattern of adding 0x40 to get to each filter vector is applied. Both of these patterns are documented in Table 3, but the last pattern in generating the filter vector SRAM addresses is not shown in this table. Table 4 below represents the first layer of the 4x4x4 array, and each number in a cell represents the position in the row major 64 element vector.

Table 4: First layer of the 4x4x4 quadrant.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

When the first set of 4 intermediate values are done processing, the next set is processed. This set corresponds to the (0,1) sub-quadrant in the (0,0) quadrant, and the first intermediate value for this set corresponds to position 1 in Table 4 above. Thus the corresponding filter vector element is 0x40+1 for the first filter vector, and the previously discussed patterns of adding 0x40 and 16 to do calculations for each of the 8 filter vectors and each of the 3 remaining intermediate values are followed. When the third set of intermediate values arrives (values associated with the (1,0) sub-quadrant in the (0,0) quadrant), the first intermediate value in this set corresponds to position of 4 in the table. Because of this, the position of the first filter vector's corresponding element is 0x40+4. The complete pattern of the first filter vector's element that corresponds to the first value in the set of 4 intermediate values is shown in Table 5. As can be seen from the table, the pattern across sets of intermediate values follows +1, +3, +1, -3, +1,+3, +1…. . To implement this pattern, the FSM in Figure 10 was employed. The FSM determines what should be done (+1, +3, or -3) when a new set of intermediate values is starting to be worked on.

Table 5: Pattern for the filter vector SRAM addresses.

Set #	**First corresponding element in the first filter vector**
0	0x40
1	0x40+1
2	0x40+(4)
3	0x40+(5)
4	0x40+(2)
5	0x40+(3)
6	0x40+(6)
7	0x40+(7)

Verification:

Verification was completed by using a provided test bench. This provided test bench could be initiated with a System Verilog seed which generated random values for the filter vectors and input array SRAMs. The test bench also calculates all of the correct expected values and compares these expected values to values that are written to the output memory. If correct values are written, then a pass message is outputted, if not, then a fail message is presented. When the design is done calculating and sets the finish flag high, the test bench makes the design go through another calculation, and this is repeated again after this calculation is done for a total of 3 consecutive runs. When all 3 runs are done, an output file for the results is created and shows the passes and fails of the design. The test bench is in the corresponding folder for the Verilog files. All verification for each module was completed using this test bench. The simulation output file displaying the correct functioning of the design is presented in the result folder under the name *transcript*.

Results:

Table 6 below presents notable results in tabular form. Because the files for these results are rather large, the main results are presented in a table and more detail can be obtained in the specified file name in the results folder. Other files that may be of interest in the results folder include the results from the read command in Synopsys, showing that the design can be read into synthesis cleanly with no errors or unintentional latches. A file was also added to show the final results of the *check_design* command, named *check_final.txt.* This file shows the warnings given about the final synthesized design. These warnings do not indicate that the design is not synthesizable, but they are just indicating that some things in the design were optimized to constant 0 or 1 logic, and some designware adders are receiving constant values on their *CI* input, which is okay.

Table 6: Table showing key results from simulation and synthesis. Corresponding files can be found with the specified file name in the results folder.

Timing Results Clock Period Achieved = 15 ns		**Area Results**	**Cycles**
Setup Timing	**Hold Timing**	**Area (um^2)**	**# Cylces:**
Slack = 0.008 ns	Slack = 0.0135 ns	42582.08	572
File Name	**File Name**	**File Name**	**File Name:**
timing_max_slow_holdfixed_project.rpt	timing_min_fast_holdcheck_project.rpt	area_report.txt	transcript

Conclusion:

In this report, a design for a hardware accelerator for two layers of a Convolutional Neural Network was discussed, and results from synthesis and simulation were presented. Key results include a clock cycle time of 15 ns, a total design area of 42582.08 um^2, and 572 cycles to completion. Although some strategies were employed to make the design faster, more strategies can be used to make it even faster, especially when it comes to the cycle time of the design. One more strategy that can be applied is pipelining the execution of the multiply accumulate so that the add and multiply happen in their own stages. This would reduce the critical path of the design with the trade off of adding more area for registers. Also, the area of the reported design may be artificially increased due to meeting the requirements specified for the project, such as using 4 separate step 1 units for step 1 calculations. One single unit could actually be used for all the quadrants, and it would achieve the same performance because in the current design only one of the step 1 module's outputs are used at a time. To leverage 4 step 1 modules to do all quadrant calculations in parallel, then the input array values would need to be loaded into a register file to circumvent bandwidth limitations. This would not provide that much performance for the added area because each input array value still needs to be read out one at a time to the register file and then processed after loading the register file, while the design with 1 module can do all calculations for each read in 1 cycle rather than reading to load data and reading data out of the register file. Besides these issues, the current design works properly and synthesizes with no warnings that indicate poorly specified RTL code.