

Systolic Array Toy Implementation

The Systolic Array

In order to multiply the matrices a *weight stationary systolic array* is used where the weights (first matrix) are loaded into the array, and the features (second matrix) are pipelined through allowing for parallelism in operations.

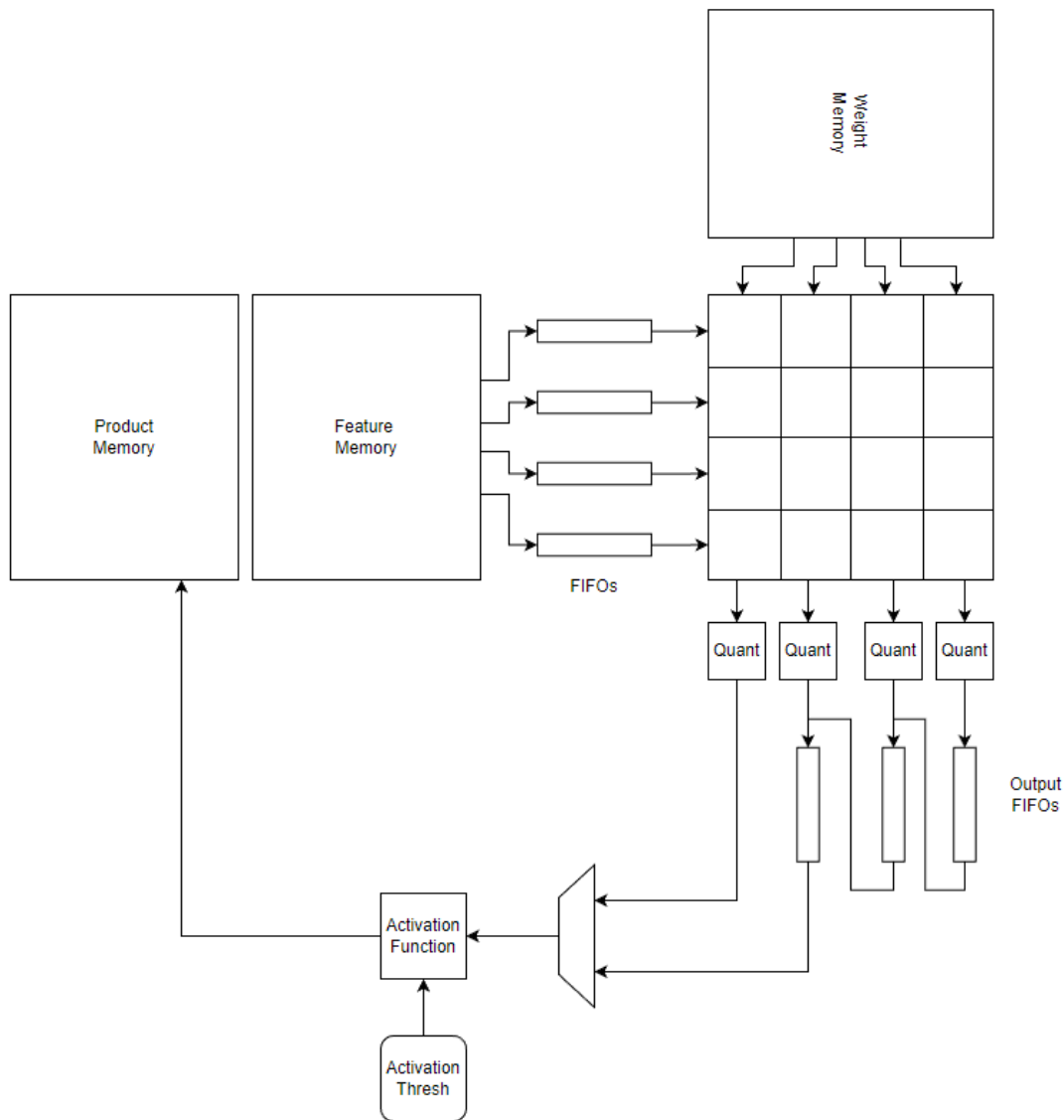
Take the example product here:

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Below is a table of all the operations over each clock cycle. It will take 4 cycles for the first output to reach the final row which make sense as each MAC unit completes one part of the dot product requiring four MAC operations. Each column corresponds to a row in the product matrix. At that point the output row will look like below where each new clock cycle will add new outputs:

Clock Cycle	Column Output 1	Column Output 2	Column Output 3	Column Output 4
4	$w_{11} * a_{11} + w_{12} * a_{21} + w_{13} * a_{31} + w_{14} * a_{41}$			
5	$w_{11} * a_{12} + w_{12} * a_{22} + w_{13} * a_{32} + w_{14} * a_{42}$	$w_{21} * a_{11} + w_{22} * a_{21} + w_{23} * a_{31} + w_{24} * a_{41}$		
6	$w_{11} * a_{13} + w_{12} * a_{23} + w_{13} * a_{33} + w_{14} * a_{43}$	$w_{21} * a_{12} + w_{22} * a_{22} + w_{23} * a_{32} + w_{24} * a_{42}$	$w_{31} * a_{11} + w_{32} * a_{21} + w_{33} * a_{31} + w_{34} * a_{41}$	
7	$w_{11} * a_{14} + w_{12} * a_{24} + w_{13} * a_{34} + w_{14} * a_{44}$	$w_{21} * a_{13} + w_{22} * a_{23} + w_{23} * a_{33} + w_{24} * a_{43}$	$w_{31} * a_{12} + w_{32} * a_{22} + w_{33} * a_{32} + w_{34} * a_{42}$	$w_{41} * a_{11} + w_{42} * a_{21} + w_{43} * a_{31} + w_{44} * a_{41}$
8		$w_{21} * a_{14} + w_{22} * a_{24} + w_{23} * a_{34} + w_{24} * a_{44}$	$w_{31} * a_{13} + w_{32} * a_{23} + w_{33} * a_{33} + w_{34} * a_{43}$	$w_{41} * a_{12} + w_{42} * a_{22} + w_{43} * a_{32} + w_{44} * a_{42}$
9			$w_{31} * a_{14} + w_{32} * a_{24} + w_{33} * a_{34} + w_{34} * a_{44}$	$w_{41} * a_{13} + w_{42} * a_{23} + w_{43} * a_{33} + w_{44} * a_{43}$
10				$w_{41} * a_{14} + w_{42} * a_{24} + w_{43} * a_{34} + w_{44} * a_{44}$

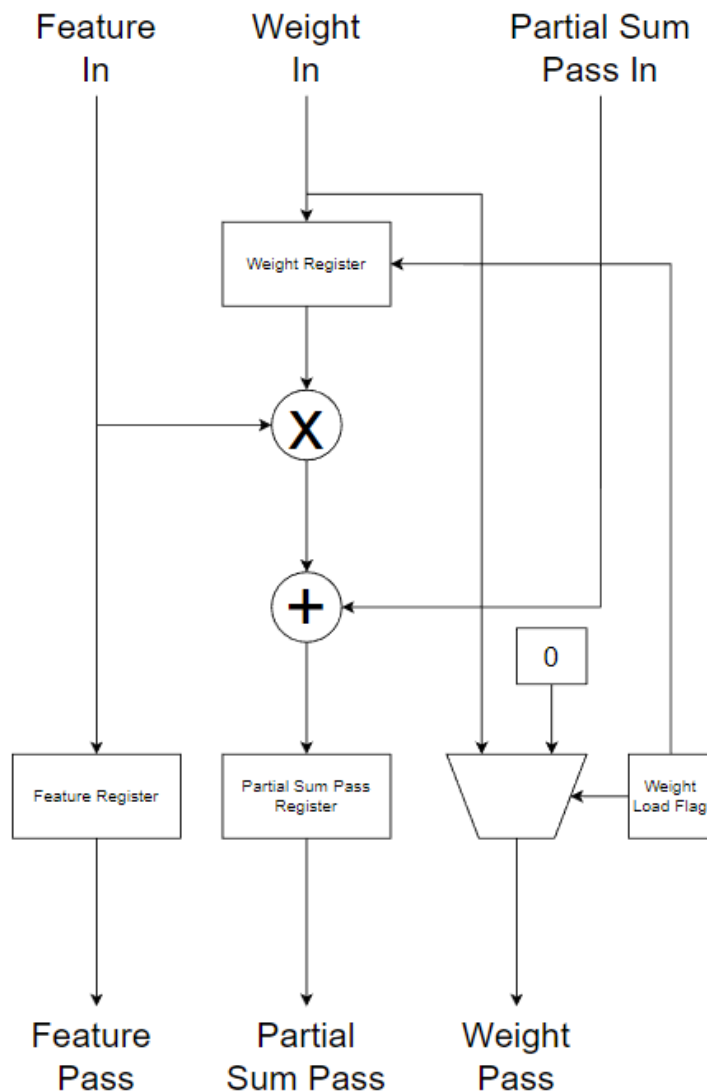
The Design:



Here is a general block diagram of the multiplier design. The center is the 4x4 Systolic Array combined with a weight memory, feature memory, and a product memory. Moreover, the quantization units cap the partial sums to 255, and the activation unit with a programmable threshold floors outputs to zero if they are below the set threshold. These pieces reflect the machine learning applications found in a Google TPU architecture.

The memory units are 4x4 with single byte words. Using a write-enable pin, and the address and data busses, the memory can be written too. The address goes across the row and down the column start from the first index to the last. Due to the needs of the Systolic Array, the memory outputs 32 bits of a column chosen by the first two bits of the address input.

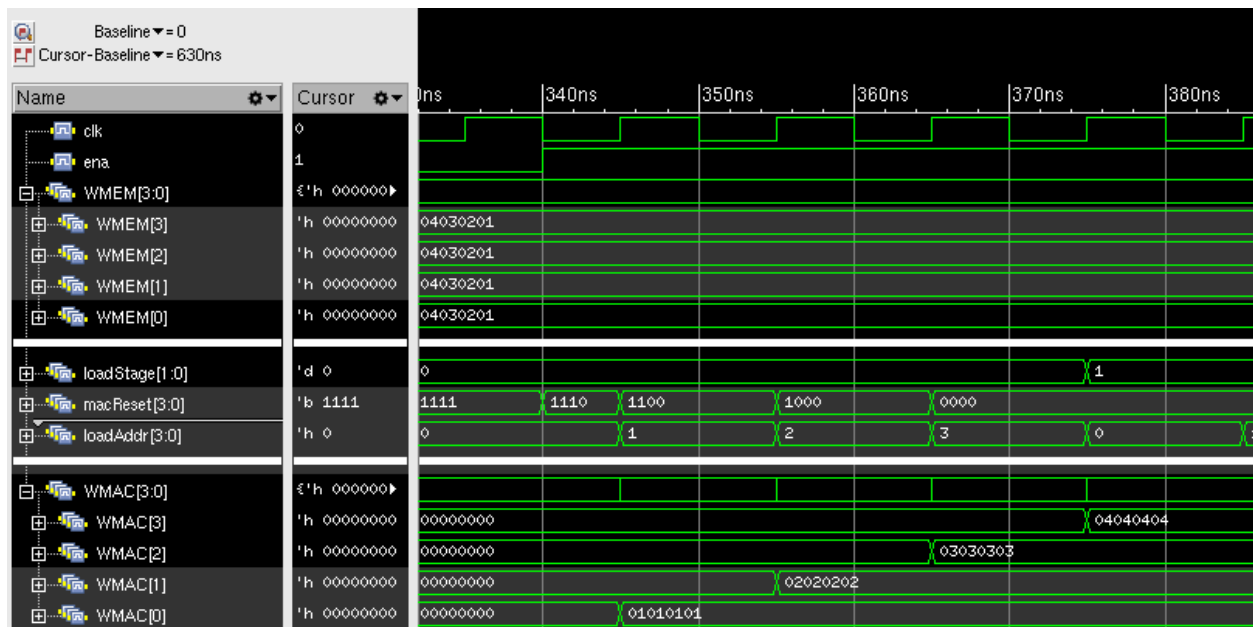
To facilitate the multiplication efficiently the weights should be loaded in 4 cycles and the multiplication should be completed in 10 cycles. Let's take a look at the MAC design to understand the Systolic Array logic.



The MAC needs to do the arithmetic as well as pass on data. To avoid delays, we accept features and previous partial sums as direct inputs to a combinatorial multiply-accumulate block which has its output registered for passing on. The feature value is still registered on that clock cycle to pass on to the next column's MAC. Before doing any multiplication the weights are loaded one row at a time from weight memory.

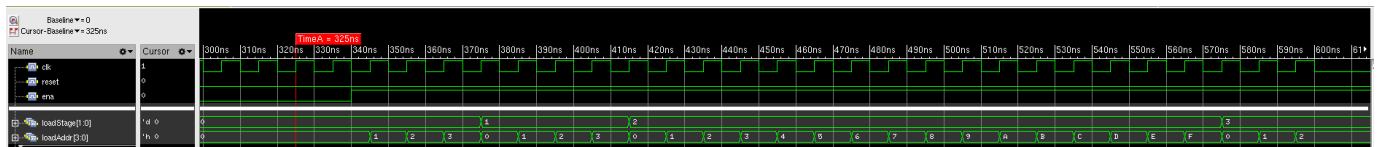
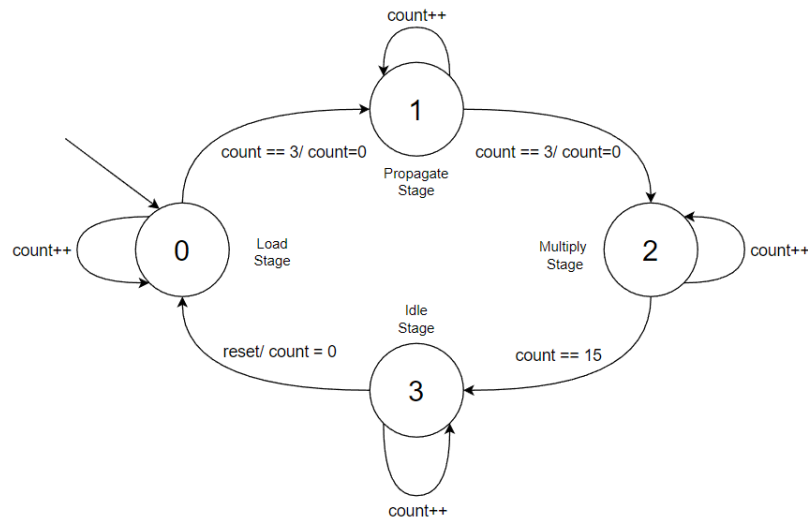
The MAC is designed to always load a weight on its first clock cycle, and then feed forward its weight input on to the next MAC after that. This is so weights can be loaded from top to bottom from weight memory starting at address zero. The same address is used to load the feature FIFO's. It's important to note that the MAC will load any weight input on the first clock cycle.

Therefore, each row of MAC's that are not loading a weight is held in their reset state. Without doing this all but the first row would load a zero and never update again. This is not the only way to do it, but much of this control hardware is reused later and it keeps the number of inputs/wires to a MAC unit small. So to summarize, the first row of the Systolic Array is awaiting a clock edge to register the weights from the weight memory. Each clock cycle after, the reset signals are pulled low in a wave until all weights are loaded. The reset signal is simply a shift register with a binary 1111. Each cycle it shifts in zeros until all the rows are loaded and ready. This is demonstrated in the *macReset* signal below. Notice the *WMAC* (MAC Weights) flow in row by row. Also, it's important to note that the memories are upside down as Cadence displays the 0th row at the bottom. This continues for all memory blocks.

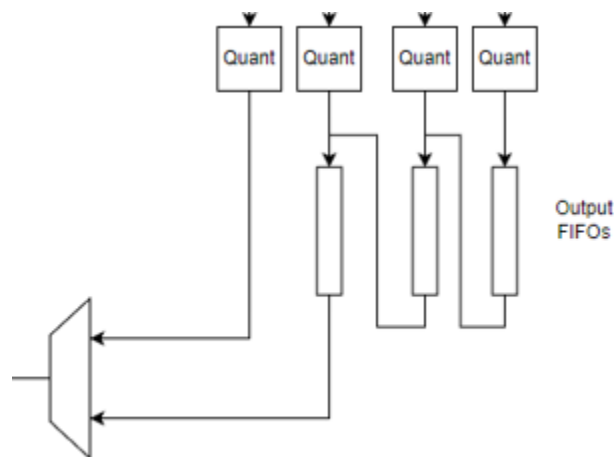


Four clock cycles are already needed to load the weights so this time is used to load feature memory into feature FIFO's. These simply right shift and load feature memory outputs as the address increases. This is demonstrated below.

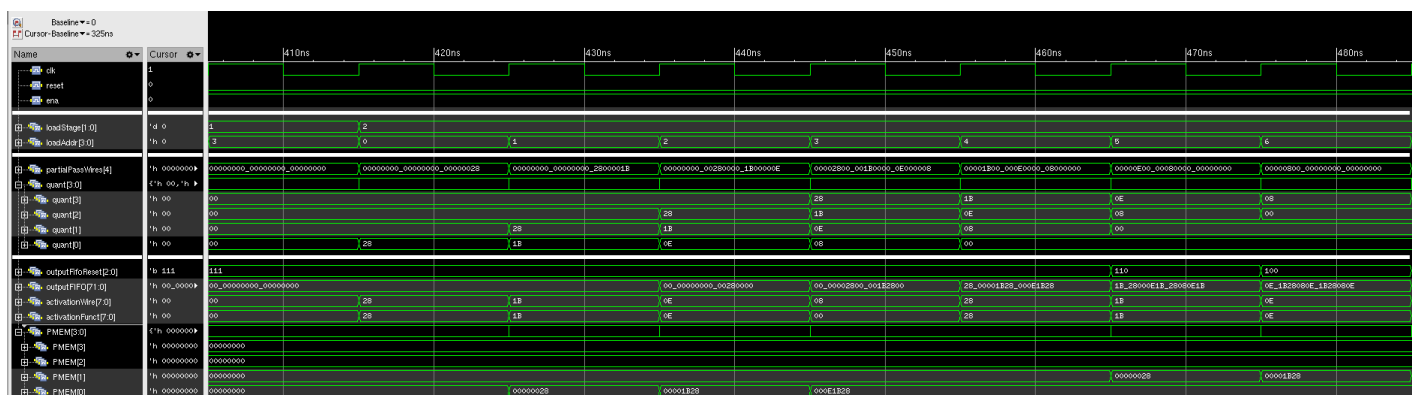
row. This also takes four cycles. The multiplication stage is where the product is finally written back data to memory. The multiplication itself takes ten cycles, but including the write back, it is longer. Finally, the idle stage is just there to stop the system and wait for a reset. The counter of this state machine doubles as the required address for loading weights/feature, or writing back data.



Above are the outputs from the state machine. The *loadAddr* signal is the count and load address used to either load weights/features or write back outputs. The *loadStage* controls the stages mentioned earlier using a case statement.



The output system is made up of the quantization units, output FIFO, activation unit, and product memory. Each of the Systolic array column outputs has its own quantization unit. This extra hardware is useful as the outputs need to be buffered since only one byte is written back at a time. The quantization units cut each output's length from three to one bytes which massively reduces the output FIFO's length. If one output can be written back per cycle then at most only nine bytes must be buffered. The first output columns outputs can be immediately written back so no FIFO is needed there. Using the state machine count a MUX is controlled to take the first output column in the first four clock cycles and take from the FIFO buffer after. As shown above, the output FIFO behaves like three separate FIFO's that feed into each other. In the HDL one FIFO is made and new values are inserted at the right index (2, 5, and 8). Each cycle the FIFO will left shift.



You can see above the column outputs in *partialPassWires[4]* which is nine bytes where each three bytes output is concatenated. Below that signals are quantized which are only one byte as expected (First row being *quant[0]*). Below that is the *outputFifo* signal which buffers all the outputs except the first column. You can see the FIFO shifts over and then registers in new outputs in the right place. Using an activation threshold of ten only the 0x08 outputs get changed to zero. Finally, the contents of the product memory are shown. You can see the first row fill up before moving to the second row.

Near the end of the multiply operation, some columns will have no meaningful output. We don't want to store those in the output FIFO, so similar to the MAC reset and feature FIFO enable register, we have a disable shift register that activates at the right state and disables the storing of garbage outputs. This is represented in the waveforms above as the signal called *outputFifoReset*. Furthermore, the outputs will go through the activation unit and are written back to memory using the state counter as the address (*loadAddr* above). The activation unit was made using dataflow comparators as to be combinatorial and save another clock cycle. Depending on timing constraints of the real synthesis this may need to be synchronous logic but that would only require decrementing the load address for write back.

Analysis:

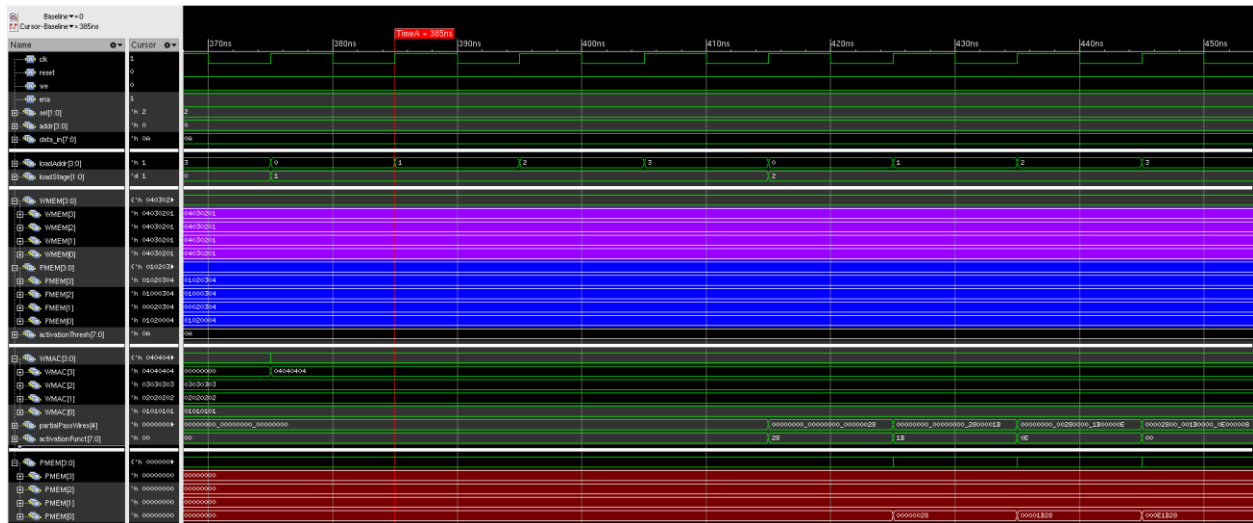
[illegible]

The timing diagram displays various digital signals over a time interval from 340ns to 380ns. The signals are organized into several groups:

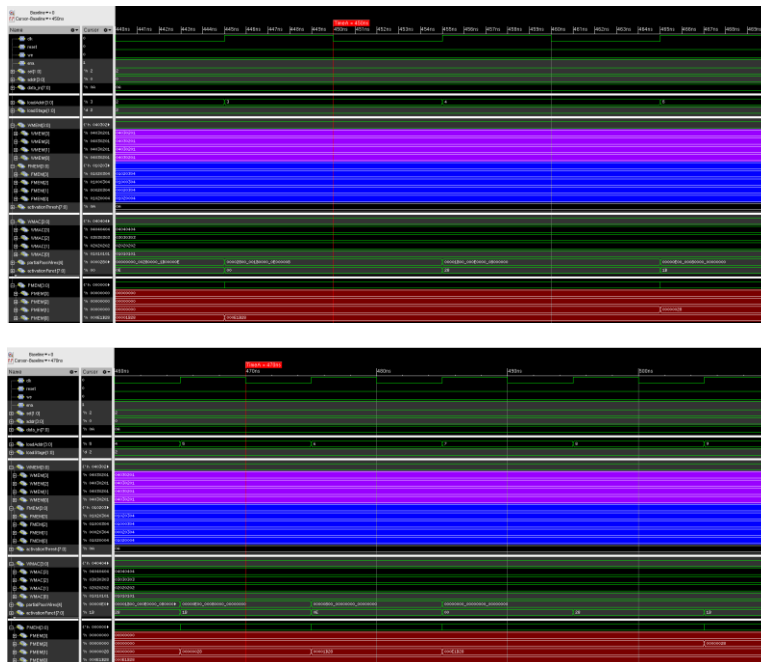
- Control Signals:**
 - ck:** A clock signal that transitions from low to high at approximately 340ns.
 - reset:** A signal that is high initially and then transitions to low.
 - we:** A signal that is high initially and then transitions to low.
 - ena:** A signal that is high initially and then transitions to low.
 - sel[1:0]:** A 2-bit signal that transitions from 00 to 01 at approximately 340ns.
 - addr[3:0]:** A 4-bit signal that transitions from 0A to 0B at approximately 340ns.
 - data_in[7:0]:** An 8-bit signal that transitions from 0A to 0B at approximately 340ns.
 - loadAddr[3:0]:** A 4-bit signal that transitions from 0 to 1 at approximately 340ns.
 - loadStage[1:0]:** A 2-bit signal that transitions from 0 to 1 at approximately 340ns.
- Memory Signals:**
 - VMEM[3:0]:** A 4-bit signal that transitions from 0 to 1 at approximately 340ns.
 - FMEM[3:0]:** A 4-bit signal that transitions from 0 to 1 at approximately 340ns.
- Other Signals:**
 - WMAC[3:0]:** A 4-bit signal that transitions from 0 to 1 at approximately 340ns.
 - partialPassWires[4]:** A 4-bit signal that transitions from 0 to 1 at approximately 340ns.

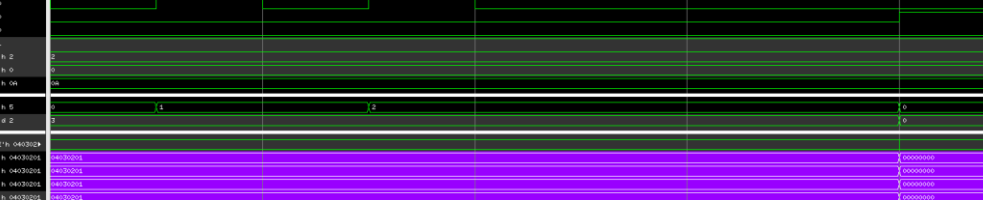
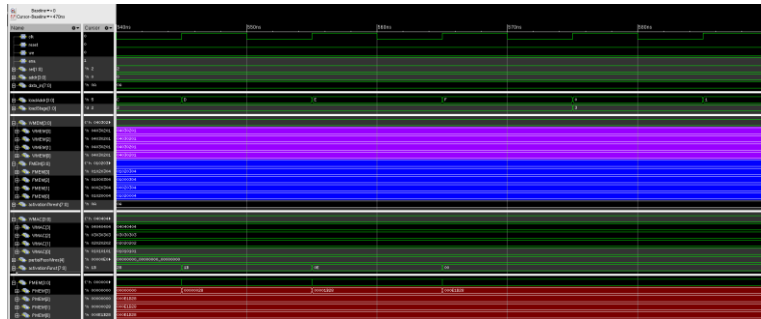
The time axis is marked from 340ns to 380ns, with a red vertical line at 385ns labeled "TimeA = 385ns".

Next, the write-enable is set low and the *ena* is set high signaling to the system that multiplication should start. The weights and feature FIFOs being loaded.



A transition from loading to multiplying occurs after 4 cycles. Notice the four cycle gap until outputs become present. In effect, the loadStage changes from one to two, and the *loadAddr* is reset to zero. The output of all the columns can be seen in *partialPassWires[4]*. Below that is the activation function output which only effects the 0x08 outputs. Finally, at the bottom the product memory is filling up based on the *loadAddr* addressing signal.





The screenshot displays the WinBox debugger's memory dump window. The top bar shows the current address (00000000) and the size of the selected memory (4 bytes). The main area is a table with columns for memory addresses and rows for different data types. The 'test' variable is located at address 00000000 and contains the value 00000000. The dump also shows the memory layout of the 'test' variable, including its size (4 bytes) and the address of the next variable.

Address	Size	Value
00000000	4	00000000
00000004	4	00000000
00000008	4	00000000
0000000C	4	00000000
00000010	4	00000000
00000014	4	00000000
00000018	4	00000000
0000001C	4	00000000
00000020	4	00000000
00000024	4	00000000
00000028	4	00000000
0000002C	4	00000000
00000030	4	00000000
00000034	4	00000000
00000038	4	00000000
0000003C	4	00000000
00000040	4	00000000
00000044	4	00000000
00000048	4	00000000
0000004C	4	00000000
00000050	4	00000000
00000054	4	00000000
00000058	4	00000000
0000005C	4	00000000
00000060	4	00000000
00000064	4	00000000
00000068	4	00000000
0000006C	4	00000000
00000070	4	00000000
00000074	4	00000000
00000078	4	00000000
0000007C	4	00000000
00000080	4	00000000
00000084	4	00000000
00000088	4	00000000
0000008C	4	00000000
00000090	4	00000000
00000094	4	00000000
00000098	4	00000000
0000009C	4	00000000
000000A0	4	00000000
000000A4	4	00000000
000000A8	4	00000000
000000AC	4	00000000
000000B0	4	00000000
000000B4	4	00000000
000000B8	4	00000000
000000BC	4	00000000
000000C0	4	00000000
000000C4	4	00000000
000000C8	4	00000000
000000CC	4	00000000
000000D0	4	00000000
000000D4	4	00000000
000000D8	4	00000000
000000DC	4	00000000
000000E0	4	00000000
000000E4	4	00000000
000000E8	4	00000000
000000EC	4	00000000
000000F0	4	00000000
000000F4	4	00000000
000000F8	4	00000000
000000FC	4	00000000

Finally, the product memory is filled and the systems idles. That concludes a load, multiply, and write back operation.