ECE 464 / 564 Project : Fall 2019:
Long Short Term Memory (LSTM) Cell

This project is to be conducted individually. You can collaborate on the paper version of the design, including discussion of ideas, design approach, etc. However, you are forbidden to share code. We will be running code comparison tools on your submitted code.

Long Short Term Memory (LSTM) is a general class of Recurrent Neural Network (RNN). It is commonly used in applications of a temporal nature, especially natural language applications.

ECE 464 and ECE 564-601 (EOL)

Your task is to design the matrix multiply in the g cell, i.e., $y_i = (W_g * x_i)$.

ECE 564-001

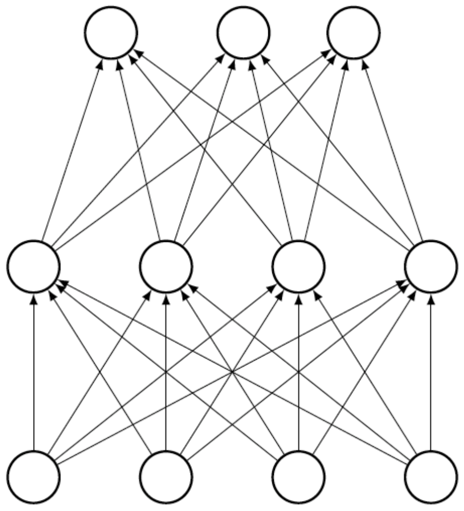
Your task is to design the g(t) gate, i.e., $y_i = tanh(W_g * x_i)$.
You can do the entire LSTM cell for some extra credit (amount TBD but not a lot - some of the performance/area points). The g(t) gate still captures the important functions of LSTM without needing a complex memory scheduler. I believe that requiring the entire LSTM cell skews the project too much in favor of those students coming in with experience.
If you turn in the project plan for tanh or the entire cell, there will be no penalty. We'll also allow late submissions for the project plan.

Note, the project is an important part of the class assessment. In general, students with a working project get at least a B in the class. Working means it passes our test benches and is synthesis clean with no errors or warnings indicating incorrect RTL. You will get a lot more points with a working g(t) gate than with a non-working LSTM cell.

This is a technical description. Separately we will also be providing the following:

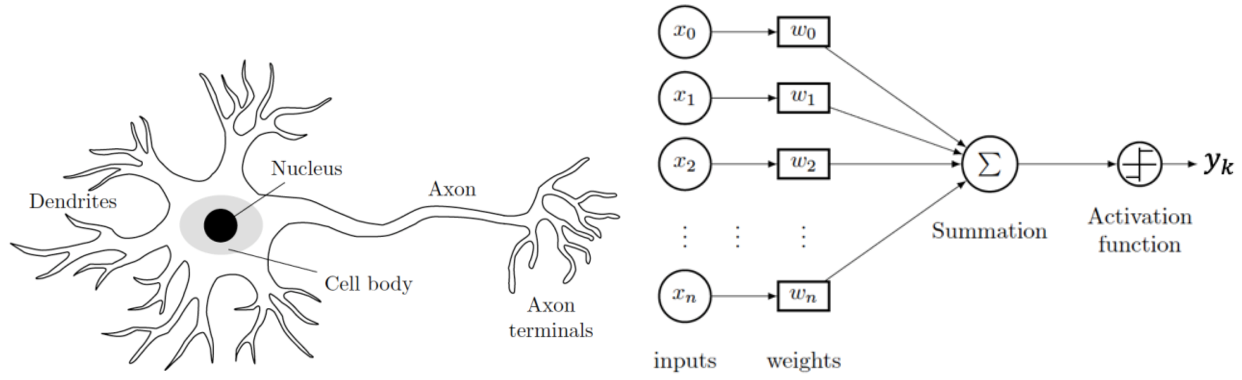
1. Sample inputs and expected outputs.
2. A test fixture precisely specifying all connections and how to feed the inputs and verify the outputs.
3. A general description of how the project will be conducted from a non-technical perspective.

A Brief Introduction to Artificial Neural Networks



Artificial neural networks are a set of algorithms inspired by the neural networks in biological system. Such algorithms are modeled base on a collection of connected perceptrons as in a biological brain. The network has to "learn" to perform tasks through a training process. During the training process, the associated weights of each connection can be adjusted to better match the desired result as training proceeds. The weight affects the strength of the connection between perceptron, as greater weight magnitude usually means the connection "conducts" more and will have more influence to the next stage.

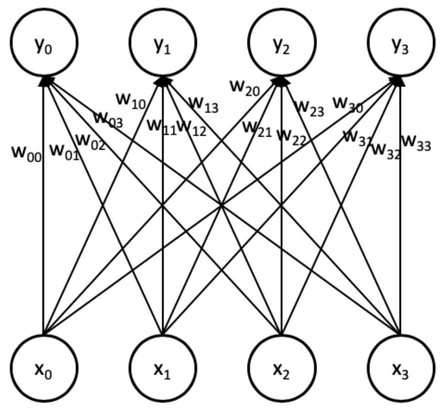
Perceptron Analogy



The artificial neuron is typically modeled as a perceptron as shown in the figure above. The output of the perceptron will be the dot products of the input vector and weight vector passing through an activation function as shown in the following equation, where $\emptyset(v_i)$ is the activation function.

$$y_k = \emptyset(w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

Weight Matrix and Connected Layer



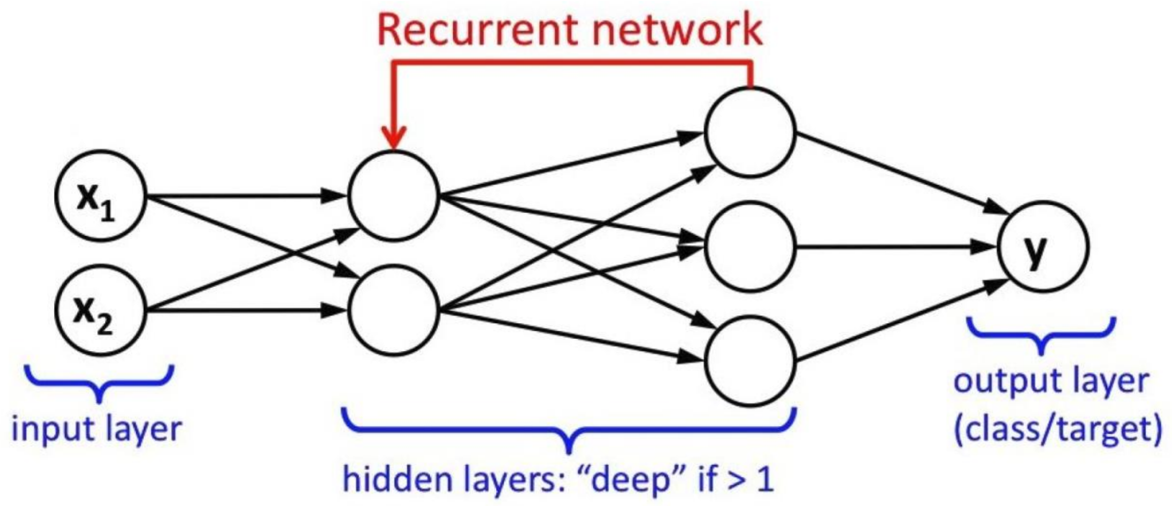
$$\begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & w_{32} & w_{33} \end{bmatrix} \mathrm{x} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$\begin{bmatrix} \emptyset(v_0) \\ \emptyset(v_1) \\ \emptyset(v_2) \\ \emptyset(v_3) \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

As we start to build up a layer of fully connected layer of the neural network, we can create a collection including all weight vectors and arrange (append and transpose) them into a matrix. With this weight matrix form, we can use matrix multiplication to write the computation as the evaluation above.

Recurrent Neural Networks

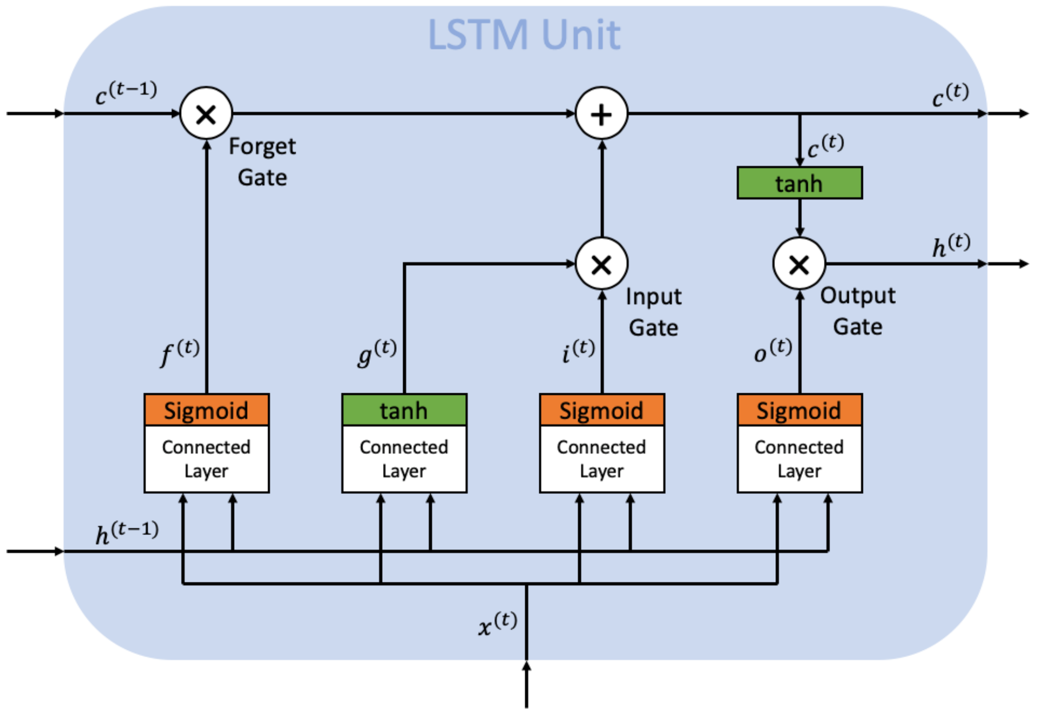
In Recurrent Neural Networks (RNN) feedback is added to the feed-forward paths normally present in a Multi-Layer Perceptron (MLP). This results in the presence of internal state, or memory, that makes RNNs potentially useful for tasks involving sequential data, such as speech, text, stock market data, video tagging or signal processing. However, RNNs are difficult to train as the feedback paths result in vanishing or exploding gradients during training.

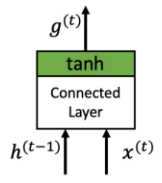


To deal with the problems of exploding and vanishing gradients, Hochreiter and Schmidhuber proposed LSTM in 1997 [Hoch97]. LSTM is highly successful, widely used in Apple, Google, etc. It constitutes 29% of Googles TPU's workload.

[Hoch97] S Hochreiter, J. Schmidhuber, "Long Short Term Memory", Neural Computation. 9(8), 1735-1780.

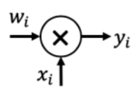
The LSTM Unit





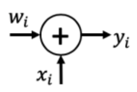
$g^{(t)} = tanh(W_g x^{(t)} + U_g h^{(t-1)} + b_g)$

where W_g and U_g are the weight matrix for $x^{(t)}$ and $h^{(t-1)}$, b_g is the bias vector of the connected layer



Hadamard product of the vector (matrix)

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_i \end{bmatrix} = \begin{bmatrix} w_0 x_0 \\ w_1 x_1 \\ w_2 x_2 \\ \vdots \\ w_i x_i \end{bmatrix}$$



Vector addition

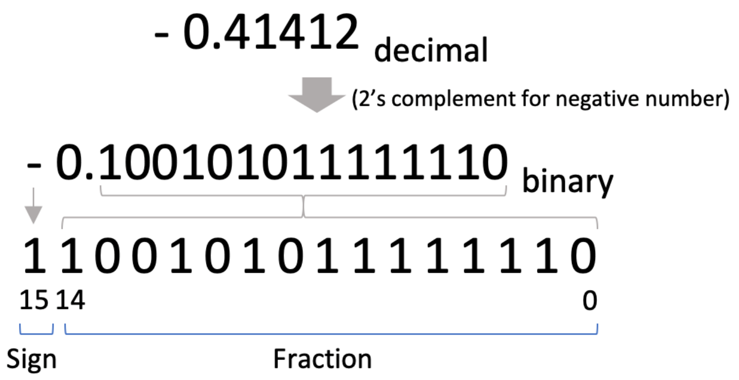
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_i \end{bmatrix} = \begin{bmatrix} w_0 + x_0 \\ w_1 + x_1 \\ w_2 + x_2 \\ \vdots \\ w_i + x_i \end{bmatrix}$$

$$f^{(t)} = \sigma(W_f x^{(t)} + U_f h^{(t-1)} + b_f)$$
$$g^{(t)} = tanh(W_g x^{(t)} + U_g h^{(t-1)} + b_g)$$
$$i^{(t)} = \sigma(W_i x^{(t)} + U_i h^{(t-1)} + b_i)$$
$$o^{(t)} = \sigma(W_o x^{(t)} + U_o h^{(t-1)} + b_o)$$
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot g^{(t)}$$
$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

$\sigma(v_i)$: sigmoid function
$\odot$: Hadamard product

The LSTM unit takes an input vector $x^{(t)}$ and the previous cell state vector $c^{(t-1)}$ and previous output vector $h^{(t-1)}$ to compute the cell state vector $c^{(t)}$ and the output vector $h^{(t)}$ for current time step. $c^{(t)}$ and $h^{(t)}$ will become the input of the next time step. $f^{(t)}$, $g^{(t)}$, $i^{(t)}$ and $o^{(t)}$ are the output vector from the corresponding connected layer.
The initial cell state, *c* and output, *h* are zeros.

Specification
All the inputs and weight matrix will be given in 16-bit fixed-point, an example is shown below.

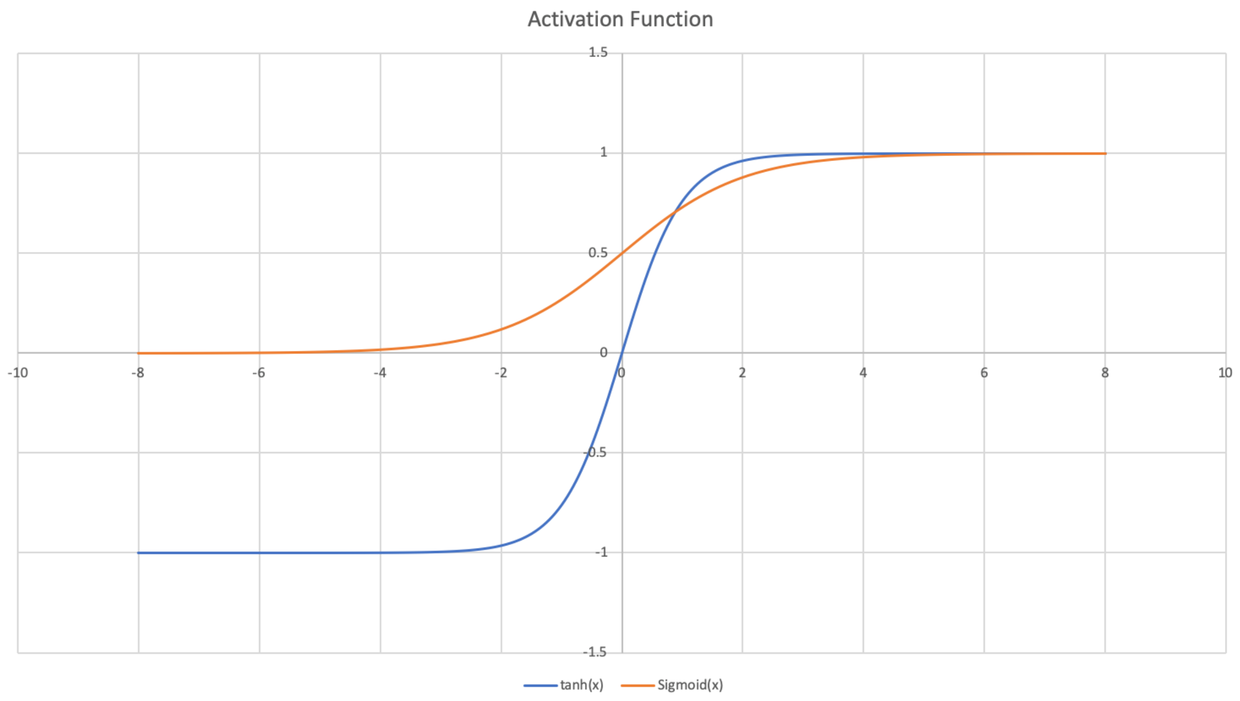


The output of the network should also be stored in the same format.

Vector size and matrix size
The following table shows the dimension of each vector and matrix.

Signal	Type	Size
$x^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
$f^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
$g^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
$i^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
$o^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
$h^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
$c^{(t)}$	Vector	$16 \times$ 16-bit fixed-point
W_f	Matrix	$16 \times 16 \times$ 16-bit fixed-point
W_g	Matrix	$16 \times 16 \times$ 16-bit fixed-point
W_i	Matrix	$16 \times 16 \times$ 16-bit fixed-point
W_o	Matrix	$16 \times 16 \times$ 16-bit fixed-point
U_f	Matrix	$16 \times 16 \times$ 16-bit fixed-point
U_g	Matrix	$16 \times 16 \times$ 16-bit fixed-point
U_i	Matrix	$16 \times 16 \times$ 16-bit fixed-point
U_o	Matrix	$16 \times 16 \times$ 16-bit fixed-point
b_f	Vector	$16 \times$ 16-bit fixed-point
b_g	Vector	$16 \times$ 16-bit fixed-point
b_i	Vector	$16 \times$ 16-bit fixed-point
b_o	Vector	$16 \times$ 16-bit fixed-point

Activation function



A hyperbolic tangent function $\tanh(x)$ and a Sigmoid function $\sigma(x)$ are used in the LSTM unit. Fortunately, the Sigmoid function can be represented as following.

$$\sigma(x) = \frac{1}{2} + \frac{1}{2} * \tanh(\frac{x}{2})$$

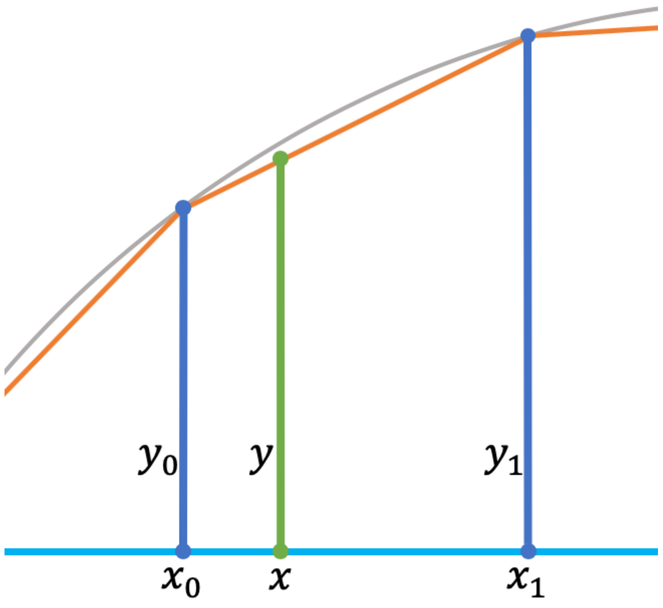
The activation function can be approximate with a piecewise linear function. A look up table for $\tanh(x)$ will be provided as sample points of the activation function, this table should be used for the computation of both $\tanh(x)$ and $\sigma(x)$. You have to interpolate the value between sample points linearly to get the result. Also, since $\tanh(x)$ is an odd function, the table will only include the range of $x = 0\ to\ 4$. The negative half of the function should be computed from the positive side of the table. The piecewise linear function in this project is defined in the following representation.

$$\begin{cases} 0.111111111111111_{binary} & , x \geq 4 \\ interpolated\ \tanh(x) & , -4 < x < 4 \\ 1.000000000000001_{binary} & , x \leq -4 \end{cases}$$

$$\begin{cases} 0.111111111111111_{binary} & , x \geq 4 \\ interpolated\ \sigma(x) & , -4 < x < 4 \\ 0.000000000000000_{binary} & , x \leq -4 \end{cases}$$

Linear interpolation

To linearly interpolate the activation function, you would have to read the closest sample points from the look up table. In the following example, when calculating output y from input x, you would have to find the neighboring sample $y_0(x_0)$ and $y_1(x_1)$, evaluate the following equation to get the output y.



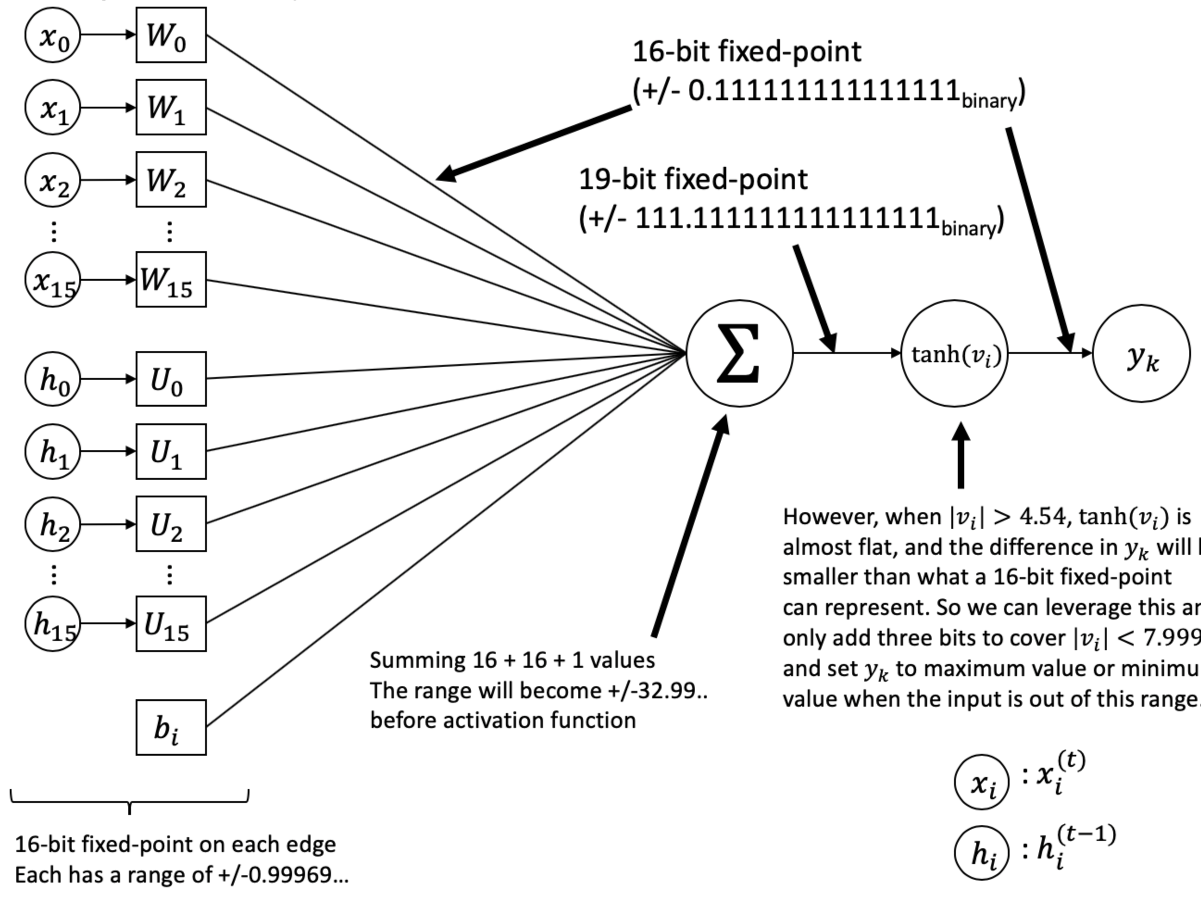
$$y = \frac{y_0(x_1 - x)}{x_1 - x_0} + \frac{y_1(x - x_0)}{x_1 - x_0}$$

In this project, the interval between samples ($x_1 - x_0$) will always be a constant of 0.000001000000000$_{\text{binary}}$ (0.015625$_{\text{decimal}}$). It is common to replace such divide by constant operation with multiply by the constant's reciprocal.

Saturate when overflow

The value of all the internal nodes (arrows in the LSTM unit) should not wrap around when overflowed. The value should be kept saturated when the sum is beyond the range of the fixed-point representation. Be careful with the carries when summing vectors.

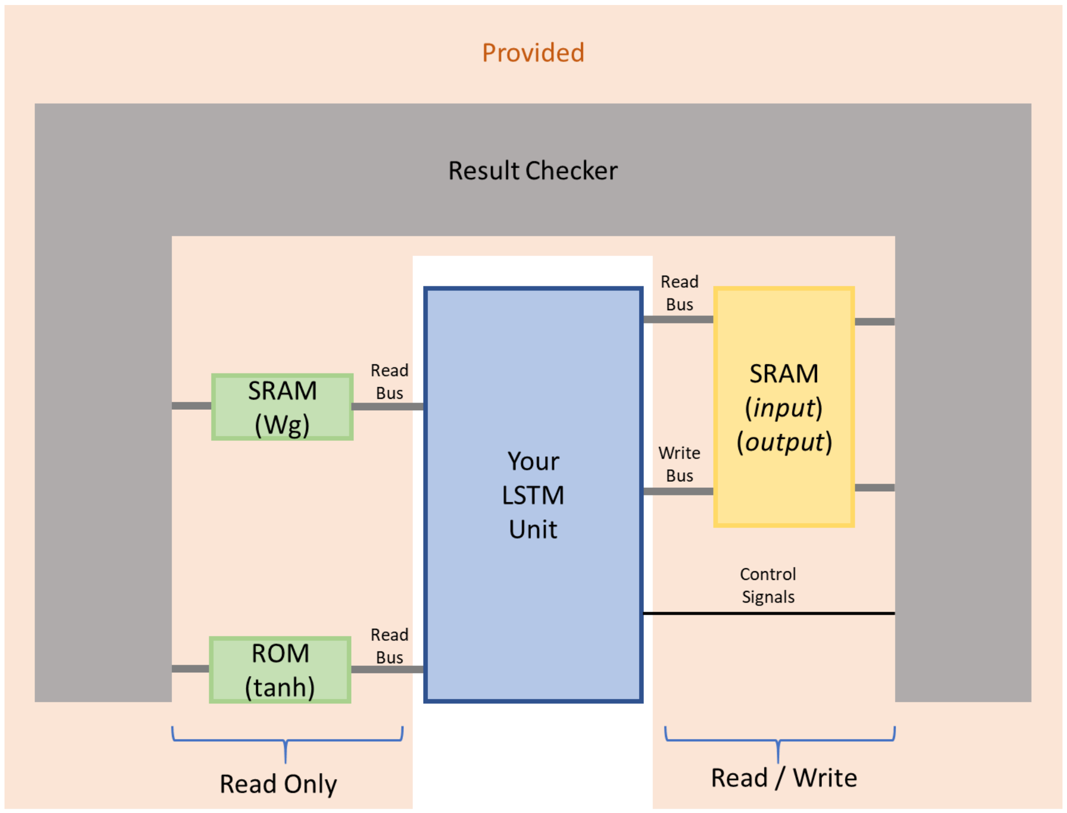
Summing connected layers



x_0 W_0
x_1 W_1
x_2 W_2
x_{15} W_{15}
h_0 U_0
h_1 U_1
h_2 U_2
h_{15} U_{15}
b_i
Σ
$\tanh(v_i)$
y_k
16-bit fixed-point (+/- $0.111111111111111_{\text{binary}}$)
19-bit fixed-point (+/- $111.111111111111111_{\text{binary}}$)
Summing 16 + 16 + 1 values
The range will become +/-32.99..
before activation function
However, when $|v_i| > 4.54$, $\tanh(v_i)$ is almost flat, and the difference in y_k will be smaller than what a 16-bit fixed-point can represent. So we can leverage this and only add three bits to cover $|v_i| < 7.999$, and set y_k to maximum value or minimum value when the input is out of this range.
x_i : $x_i^{(t)}$
h_i : $h_i^{(t-1)}$
16-bit fixed-point on each edge
Each has a range of +/-0.99969…

Since the Sigmoid function is derived from hyperbolic tangent function, the same fixed-point bit width can be used.

ECE564 Test Fixture

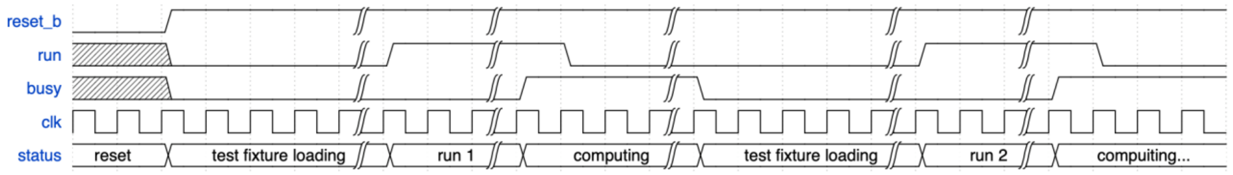


*Note that the SRAM for the weight matrices will only be loaded by the test fixture, it will have a read only memory (ROM) interface on the LSTM unit side.

Control Interface

There will be three control signals for the LSTM unit.

input	reset_b	: Active low reset signal, will clear the machine state ($h^{(t)}$ and $c^{(t)}$).
input	clk	: System clock forwarded from the test fixture.
output	busy	: The test bench will halt when busy is high, waiting for the computation.
input	run	: The test bench will set run signal high after all data has been loaded.



The cell state $c^{(t)}$and output $h^{(t)}$should be kept for future run, they can only be cleared upon reset.

Memory structure

The LSTM unit will be reading from five read only memories (ROMs), four of them will contain the weights and bias of each connected layer. One of the ROM will be storing the values for the look up table used in activation function approximation.

The unit will also be connected to one SRAM. The input vector will be written to this SRAM before the test fixture issue the run signal. The LSTM unit will then compute the result from the given input, then write the computation result back to the same SRAM (different address) before clearing the busy flag. The result will then be checked by the test fixture. The rest of the memory not used by input and output vector can be used to store any value or intermediate result (if needed).

All memory location except input vector will be set to “x” when the system is leaving reset.

Address mapping

Weight and bias memory mapping for $\left(W_f, U_f, b_f\right), \left(W_g, U_g, b_g\right), (W_i, U_i, b_i), (W_o, U_o, b_o)$

Address	Data
0x0000	$W_{0,0}$
0x0002	$W_{0,1}$
0x0004	$W_{0,2}$
0x0006	$W_{0,3}$
0x0008	$W_{0,4}$
0x000A	$W_{0,5}$
0x000C	$W_{0,6}$
……	……
0x001C	$W_{0,14}$
0x001E	$W_{0,15}$
0x0020	$W_{1,0}$
0x0022	$W_{1,1}$
0x0024	$W_{1,2}$
……	……
0x01F8	$W_{15,12}$
0x01FA	$W_{15,13}$
0x01FC	$W_{15,14}$
0x01FE	$W_{15,15}$

Address	Data
0x0200	$U_{0,0}$
0x0202	$U_{0,1}$
0x0204	$U_{0,2}$
0x0206	$U_{0,3}$
0x0208	$U_{0,4}$
0x020A	$U_{0,5}$
0x020C	$U_{0,6}$
……	……
0x021C	$U_{0,14}$
0x021E	$U_{0,15}$
0x0220	$U_{1,0}$
0x0222	$U_{1,1}$
0x0224	$U_{1,2}$
……	……
0x03F8	$U_{15,12}$
0x03FA	$U_{15,13}$
0x03FC	$U_{15,14}$
0x03FE	$U_{15,15}$

Address	Data
0x0400	b_0
0x0402	b_1
0x0404	b_2
0x0406	b_3
0x0408	b_4
0x040A	b_5
0x040C	b_6
……	……
0x041C	b_{14}
0x041E	b_{15}

Look up table mapping

Address	Data	
0x0000	tanh(+00.000000000000000$_{\text{binary}}$)	= tanh(0.000000$_{\text{decimal}}$)
0x0002	tanh(+00.000001000000000$_{\text{binary}}$)	= tanh(0.015625$_{\text{decimal}}$)
0x0004	tanh(+00.000010000000000$_{\text{binary}}$)	= tanh(0.031250$_{\text{decimal}}$)
0x0006	tanh(+00.000011000000000$_{\text{binary}}$)	= tanh(0.046875$_{\text{decimal}}$)
0x0008	tanh(+00.000100000000000$_{\text{binary}}$)	= tanh(0.062500$_{\text{decimal}}$)
0x000A	tanh(+00.000101000000000$_{\text{binary}}$)	= tanh(0.078125$_{\text{decimal}}$)
0x000C	tanh(+00.000110000000000$_{\text{binary}}$)	= tanh(0.093750$_{\text{decimal}}$)
……	……	
0x01F8	tanh(+11.111100000000000$_{\text{binary}}$)	= tanh(3.937500$_{\text{decimal}}$)
0x01FA	tanh(+11.111101000000000$_{\text{binary}}$)	= tanh(3.953125$_{\text{decimal}}$)
0x01FC	tanh(+11.111110000000000$_{\text{binary}}$)	= tanh(3.968750$_{\text{decimal}}$)
0x01FE	tanh(+11.111111000000000$_{\text{binary}}$)	= tanh(3.984375$_{\text{decimal}}$)

Input output and scratchpad memory

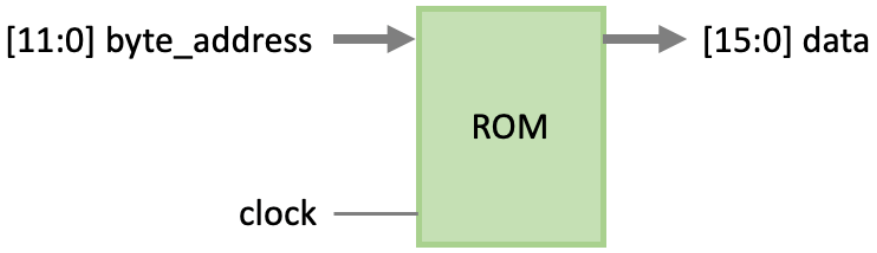
Input	
Address	Data
0x0000	$x_0^{(0)}$
0x0002	$x_1^{(0)}$
0x0004	$x_2^{(0)}$
0x0006	$x_3^{(0)}$
0x0008	$x_4^{(0)}$
0x000A	$x_5^{(0)}$
0x000C	$x_6^{(0)}$
……	……
0x001C	$x_{14}^{(0)}$
0x001E	$x_{15}^{(0)}$
0x0020	$x_0^{(1)}$
0x0022	$x_1^{(1)}$
0x0024	$x_2^{(1)}$
……	……
0x01F8	$x_{12}^{(15)}$
0x01FA	$x_{13}^{(15)}$
0x01FC	$x_{14}^{(15)}$
0x01FE	$x_{15}^{(15)}$

Output	
Address	Data
0x0200	$h_0^{(0)}$
0x0202	$h_1^{(0)}$
0x0204	$h_2^{(0)}$
0x0206	$h_3^{(0)}$
0x0208	$h_4^{(0)}$
0x020A	$h_5^{(0)}$
0x020C	$h_6^{(0)}$
……	……
0x021C	$h_{14}^{(0)}$
0x021E	$h_{15}^{(0)}$
0x0220	$h_0^{(1)}$
0x0222	$h_1^{(1)}$
0x0224	$h_2^{(1)}$
……	……
0x03F8	$h_{12}^{(15)}$
0x03FA	$h_{13}^{(15)}$
0x03FC	$h_{14}^{(15)}$
0x03FE	$h_{15}^{(15)}$

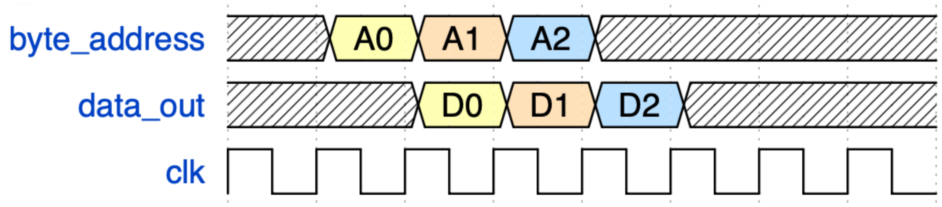
Output	
Address	Data
0x0400	$c_0^{(0)}$
0x0402	$c_1^{(0)}$
0x0404	$c_2^{(0)}$
0x0406	$c_3^{(0)}$
0x0408	$c_4^{(0)}$
0x040A	$c_5^{(0)}$
0x040C	$c_6^{(0)}$
……	……
0x041C	$c_{14}^{(0)}$
0x041E	$c_{15}^{(0)}$
0x0420	$c_0^{(1)}$
0x0422	$c_1^{(1)}$
0x0424	$c_2^{(1)}$
……	……
0x05F8	$c_{12}^{(15)}$
0x05FA	$c_{13}^{(15)}$
0x05FC	$c_{14}^{(15)}$
0x05FE	$c_{15}^{(15)}$

Available memory	
Address	Data
0x0600	Scratchpad Memory
0x0602	
0x0604	
0x0606	
0x0608	
0x060A	
0x060C	
……	
0x0FFC	
0x0FFE	

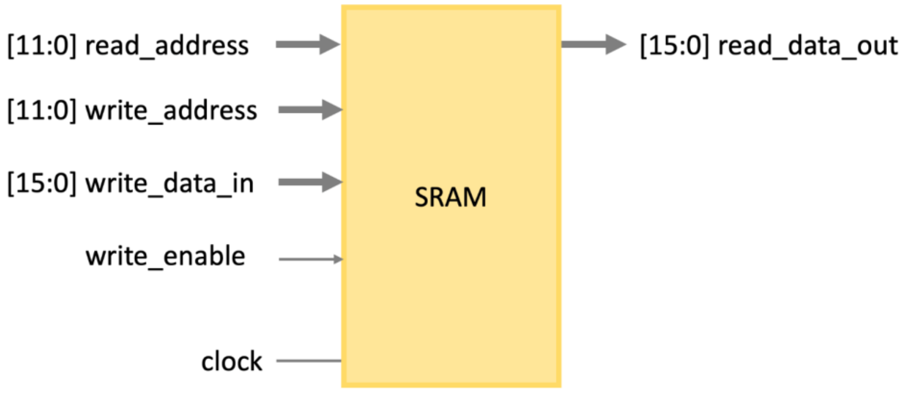
ROM interface (for weights and look up table)



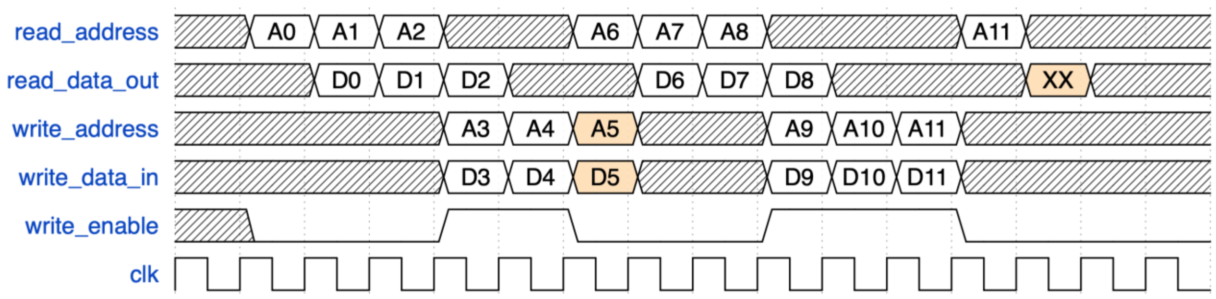
The byte addressable ROM has a 16-bit data bus, all access should be aligned with word size (for example 0x0003 are prohibited). The requested data will appear on the data bus at the next clock cycle.



SRAM interface (for input output and scratchpad)

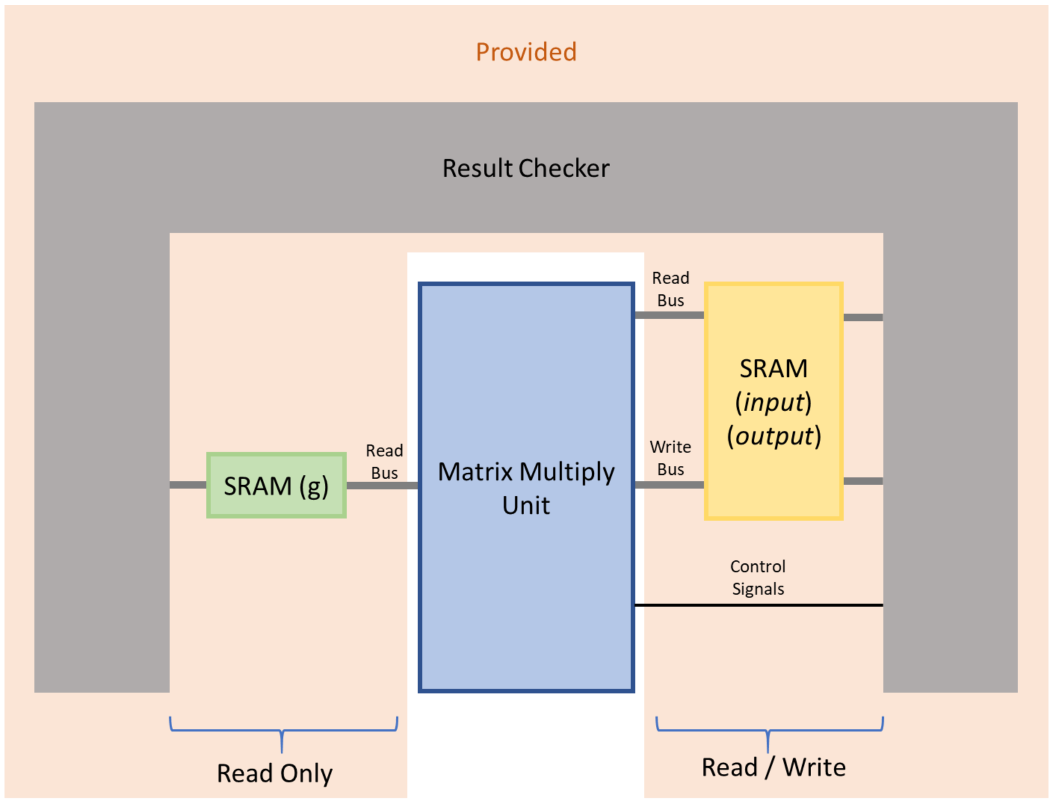


Same as the ROM, the SRAM is byte addressable and has a one cycle delay between address and data. When writing to the SRAM, you would have to set the “write_enable” to high. The SRAM will write the data in the next cycle. The “read_data_out” is valid when only “read_write_select” is set to low when requesting the data.



As shown in the example above, since "write_enable" is set to low when A5 and D5 is on the write bus, D5 will not be written to the SRAM.
Note that the SRAM cannot handle consecutive read after write (RAW) to the same address (shown as A11 and D11 in the timing diagram). You would have to either manage the timing of your access, or write the data forwarding mechanism yourself. As long as the read and write address are different, the request can be pipelined.

ECE464 Test Fixture

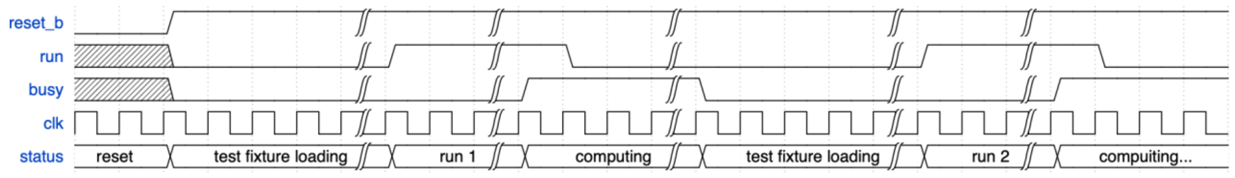


*Note that the SRAM for the weight matrices will only be loaded by the test fixture, it will have a read only memory (ROM) interface on the LSTM unit side.

Control Interface

There will be three control signals for the LSTM unit.

input	reset_b	: Active low reset signal, will clear the machine state.
input	clk	: System clock forwarded from the test fixture.
output	busy	: The test bench will halt when busy is high, waiting for the computation.
input	run	: The test bench will set run signal high after all data has been loaded.



Memory structure

The LSTM unit will be reading from five read only memories (ROMs), four of them will contain the weights and bias of each connected layer. One of the ROM will be storing the values for the look up table used in activation function approximation.

The unit will also be connected to one SRAM. The input vector will be written to this SRAM before the test fixture issue the run signal. The LSTM unit will then compute the result from the given input, then write the computation result back to the same SRAM (different address)

before clearing the busy flag. The result will then be checked by the test fixture. The rest of the memory not used by input and output vector can be used to store any value or intermediate result (if needed).
All memory location except input vector will be set to “x” when the system is leaving reset.

Address mapping for ECE464

Weight memory mapping for W_g

Address	Data
0x0000	$W_{0,0}$
0x0002	$W_{0,1}$
0x0004	$W_{0,2}$
0x0006	$W_{0,3}$
0x0008	$W_{0,4}$
0x000A	$W_{0,5}$
0x000C	$W_{0,6}$
……	……
0x001C	$W_{0,14}$
0x001E	$W_{0,15}$
0x0020	$W_{1,0}$
0x0022	$W_{1,1}$
0x0024	$W_{1,2}$
……	……
0x01F8	$W_{15,12}$
0x01FA	$W_{15,13}$
0x01FC	$W_{15,14}$
0x01FE	$W_{15,15}$

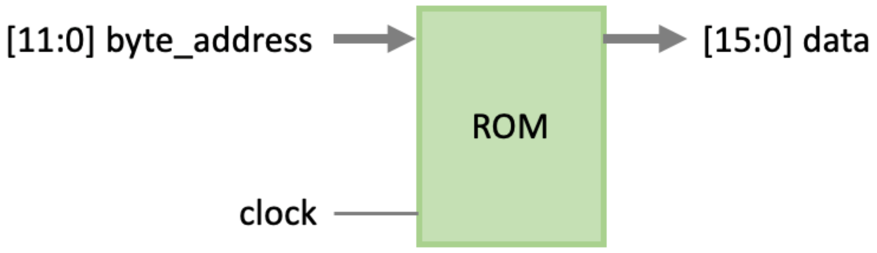
Input output and scratchpad memory for $y_i = (W_g * x_i)$

Input	
Address	Data
0x0000	x_0
0x0002	x_1
0x0004	x_2
0x0006	x_3
0x0008	x_4
0x000A	x_5
0x000C	x_6
……	……
0x01FC	x_{254}
0x01FE	x_{255}

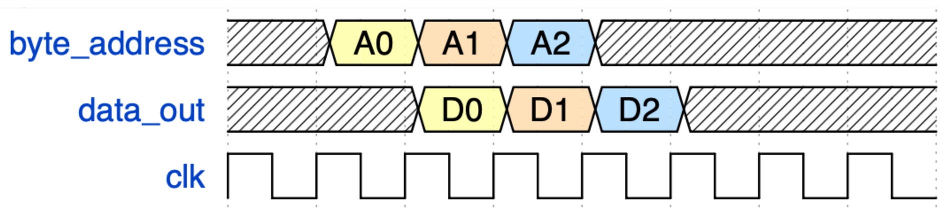
Output	
Address	Data
0x0200	y_0
0x0202	y_1
0x0204	y_2
0x0206	y_3
0x0208	y_4
0x020A	y_5
0x020C	y_6
……	……
0x03FC	y_{254}
0x03FE	y_{255}

Available memory	
Address	Data
0x0400	Scratchpad Memory
0x0402	
0x0404	
0x0406	
0x0408	
0x040A	
0x040C	
……	
0x0FFC	
0x0FFE	

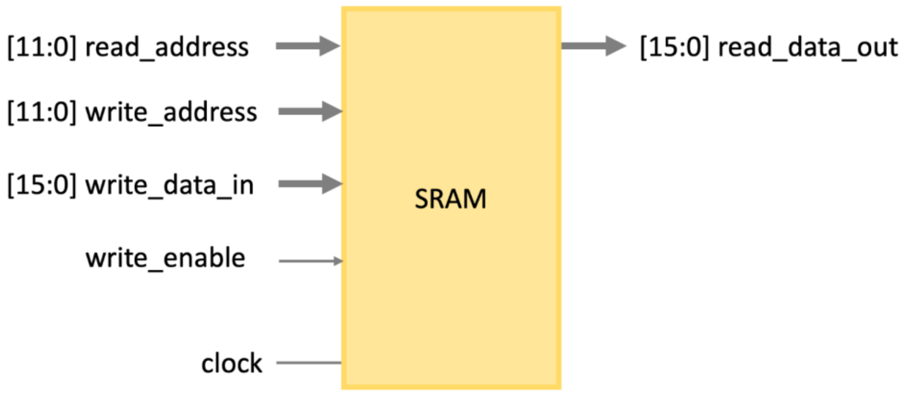
ROM interface (for weights and look up table)



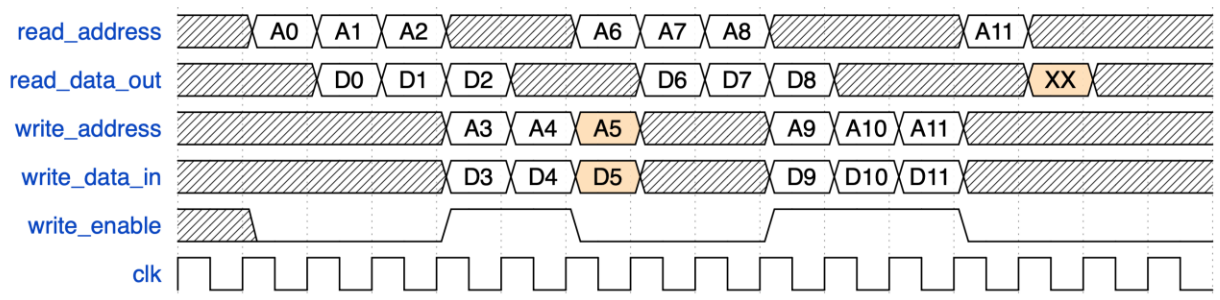
The byte addressable ROM has a 16-bit data bus, all access should be aligned with word size (for example 0x0003 are prohibited). The requested data will appear on the data bus at the next clock cycle.



SRAM interface (for input output and scratchpad)



Same as the ROM, the SRAM is byte addressable and has a one cycle delay between address and data. When writing to the SRAM, you would have to set the “write_enable” to high and the “read_write_select” signal to high. The SRAM will write the data in the next cycle. The “read_data_out” is valid when only “read_write_select” is set to low when requesting the data.



As shown in the example above, since “write_enable” is set to low when A5 and D5 is on the write bus, D5 will not be written to the SRAM.
Note that the SRAM cannot handle consecutive read after write (RAW) to the same address (shown as A11 and D11 in the timing diagram). You would have to either manage the timing of your access, or write the data forwarding mechanism yourself. As long as the read and write address are different, the request can be pipelined.