Vector Norm-2 List Processor

In this endeavor we aim to design a Vector Norm-2 List Processor (VNLP). The following design is proposed, optimized and tested. The input are taken from memory of size 512 words, each word is 24-bits. The numbers’ format is fixed-point.

## 1.Necessary Memory Footprint

For our operation, the initial inputs are 24-bits: First bit is a sign bit, next 15 bits are mantissa, ant the last 8 bits are the exponent of base 2.

The first step of the operation is squaring the inputs, since the square of a number is always positive, and the addition of positive numbers will only lead to positive results, we will ignore the sign bit from here on after (there is no sign bit).

For squaring we multiply mantissas and add exponents. Hence the result will have to be 30-bits for the new mantissa, and 9-bits for the new exponent. Though we do have the option to truncate the product of squaring, we choose not to.

Next step, is the addition of the two squared numbers. For this step we have to align the mantissas to have the same weight then add. In order to preserve simplicity on the expense of some precision, we choose to add by matching the highest exponent; hence we will right shift

the mantissa with lowest exponent(and possibly lose the LSB of the mantissa).

The result of this operation, is a mantissa of 31-bits and exponent of 9-bits (same as before).

Finally we come to the final operation: accumulation.

In this operation we will repeat the same technique of the previous operation, but as we need to account for multiple addition/accumulations. Knowing the memory size is 512 slots, and each link is 4 slots and we need to compute two numbers from each link. The total number of links possible is 128 links, with ***each level of accumulation*** adding one bit to the output. So we add log2(128) = 7 bits to the input-bit-size to get the suitable output size. So the mantissa is 31 + 7 = 38 bits, and the exponent remains 9-bits. Total output size is 47 bits.

## 2.Pseudo-Code

The processor consists of four main segments: Reading memory, Squaring, Adding Pairs, Accumulating.

The pseudo code for each operation is as follows:

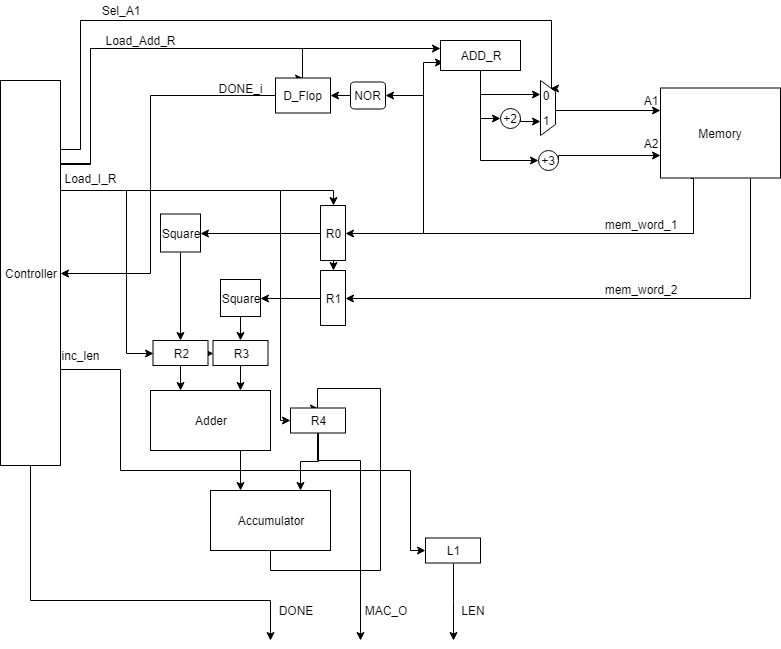
|  |
| --- |
| read\_memory(){  for(int i = 0; memory[i] != 0;)  {  x = memory[i+2];  y = memory[i+3];  //ideally here is where we do the rest of the operations  //square(x);  //square(y);  i = memory[i];  } } |

|  |
| --- |
| square(x){  return {x.mantissa^2,x.exponent\*2} } |

|  |
| --- |
| add(x,y){  int exp = max(x.exponent,y.exponent);  if (x.exponent > y.exponent)  {  y.mantissa = y.mantissa / 2^(x.exponent - y.exponent); //shifting  }  else  {  x.mantissa = x.mantissa / 2^(y.exponent - x.exponent);  }   return {x.mantissa + y.mantissa, exp} } |

|  |
| --- |
| accumulate(&x,y){  int exp = max(x.exponent,y.exponent);   if (x.exponent > y.exponent)  {  y.mantissa = y.mantissa / 2^(x.exponent - y.exponent);//shifting  }  else  {  x.mantissa = x.mantissa / 2^(y.exponent - x.exponent);  }   x = {x.mantissa + y.mantissa, exp}; } |

## 3. Block diagram of the Proposed VNLP Architecture



## 

## 4. Finite-State Machine to Control the Datapath of the VNLP.

## 

## 

## 

## 5. Performance Analysis

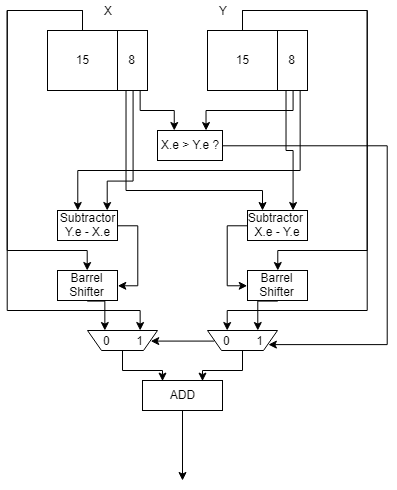
Now we’ll have a closer look into the separate stages/components and how they operate.

*Reading From Memory:* In this stage we take use of the asynchronous memory and the ability to read from two addresses at a time. First we offset the address of the link by 2 & 3 to get the values of X and Y respectively, which are loaded into registers to be processed. The offset of the address happens through adders and and a mux. On the next stage we get the address of the link (initially at 0), by selecting the normal value (not-offset) from our mux. The value readis then put in the address register (ADD\_R), ready for the next stage of reading values X and Y.

While loading the address from memory, a D-flipflop receives bitwise NOR of the address being read, this tells us if the address we read is a 0 or not, indicating we have reached the end of our linked list or not. The signal from this D-flipflop is fed into the controller to end bring the operation to a conclusion. **IT IS NOT THE *DONE* OUTPUT SIGNAL** of the VNLP**.**

*Squaring* : At this stage, we simply take the lower 8-bits representing the exponent and append a 0 to them,emulating the multiplication by 2. Then we multiply the mantissa (15-bits) with itself. We ignore the highest sign bit as the value will always be positive when squaring. As indicated above this results in a 39 bit result, that is loaded into a register. The operation is parallelized with respect to X and Y (We square both of them simultaneously).

*Adding* : To add numbers in the given format, we have to first shift our inputs to match their weights then add. In other words, we can only add if they have equal exponents.

To achieve this, we shift the number with the **lowest** exponent to the right and increment its exponent by the amount shifted to the right. The amount we shift by is the difference of the two exponents.

Of course, the highest exponent will be passed onto the output.

*Accumulating*: Is similar to adding in the previous stage. To make the input bit sizes equal (38 to 47 bits), we append 0s to the mantissa of the smaller input, and then do the same as the adder stage.

*Miscellaneous:*

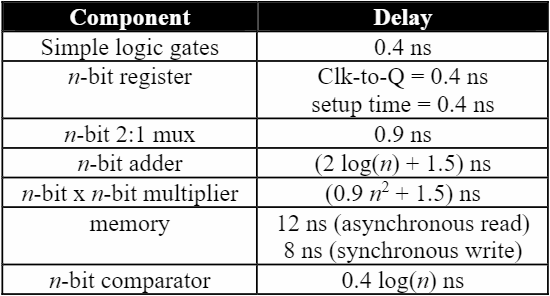
To keep count of how many links we’ve traversed, we have a 7-bit register (we have at max. 128 links) that increments on every time we update our address register.

It is to be noted that adding registers between stages (piplining) allows us to increase our clock speed to that of the longest delay, in this case the multiplier.

Due to piplining, we have to account for the processed data still in the datapath, even when the internal ‘DONE’ flag is true, and thus still operate until we push the last of the data out. Only then can we set the DONE output signal

**Timing Analysis:**

Using delay values from the component library, the worst case propagation of each stage is calculated.



First we calculate the propagation delay without any pipelining/optimization techniques, and so the critical path will consist of two Register Reads (ADD\_R and R4), one Register Write(R4), memory read, memory address adjustments(longest path passes through adder and mux), squaring, adder, and finally an accumulator.

Total delay = 0.4\*2 + 0.4 + 12 + 7.839 + 0.9 + Squaring Delay(204) + Adder Delay(12.581) + Accumulation Delay(13.362) = 251.882ns

Using pipelining/optimization techniques, the critical path is the critical path of longest segment, in this case the squaring delay(204) + read/write of registers (0.8) = 204.8ns

Hence the clock frequency possible is: 4.88 MHz

### Individual timing analysis of the segments:

Only the critical path is considered.

*Squaring* : The critical path is the multiplication of 15-bits: 204ns

*Adding* : We need 1.267ns for comparing exponent, 4.5ns for shifting, 7.839ns ns for subtracting exponents, 11.313ns for adding 30 bits, and 0.8 for Clk-to-Q and setup time for the input and output registers. Total: 25.71ns

*Accumulating*: We need 1.267ns for comparing exponents, 4.9ns for shifting, 7.839ns ns for subtracting exponents, 12.095ns for adding 38bits, and 0.8 for Clk-to-Q and setup time for the input and output registers. Total: 26.9ns

*Reading From Memory:* 0.4ns Clk-to-Q from the memory register, 0.9ns for mux and 3.408ns for address adjustment (+2), 12ns for Asynchronous memory read, and 0.4ns for NOR gate, and 0.4ns for setup time for the D\_flop. Total: 17.508ns