

**Introduction**

The goal of this project is to design a dual-core machine learning accelerator for attention mechanism. The purpose of the hardware designed will be to speed up computations in Natural Language Processing.

1-D Vector Processor Architecture is used as the basis for the design of each core in the design. Each core comprises the following sub-modules:

1. Computation engine (**mac\_array**) to compute partial products and sums
2. Special Function Processor (**sfp\_row**) to perform Normalization on the data from the mac\_array
3. FIFOs to support fast, multi-bit data transfer between mac\_array and the sfp\_row
4. An async FIFO is also included in the design to support data (psum) transfer between cores. Each core works on independent clock domain (async cores)
5. Memory module to store the **keys, queries** to be loaded into the mac\_array, and the **normalized psum** calculated in sfp\_row

The below features which were added on to the existing design provided initially:

1. Add support for data computation in sfp\_row through testbench
2. Integrate **sfp\_row** into the Core to compute the normalized psum
3. Write-back the computed psum into memory within the Core
4. Integrate 2 Core instances into the **fullchip** module; Asynchronous FIFO also added to support inter-clock-domain data transfer

**Design Strategy**

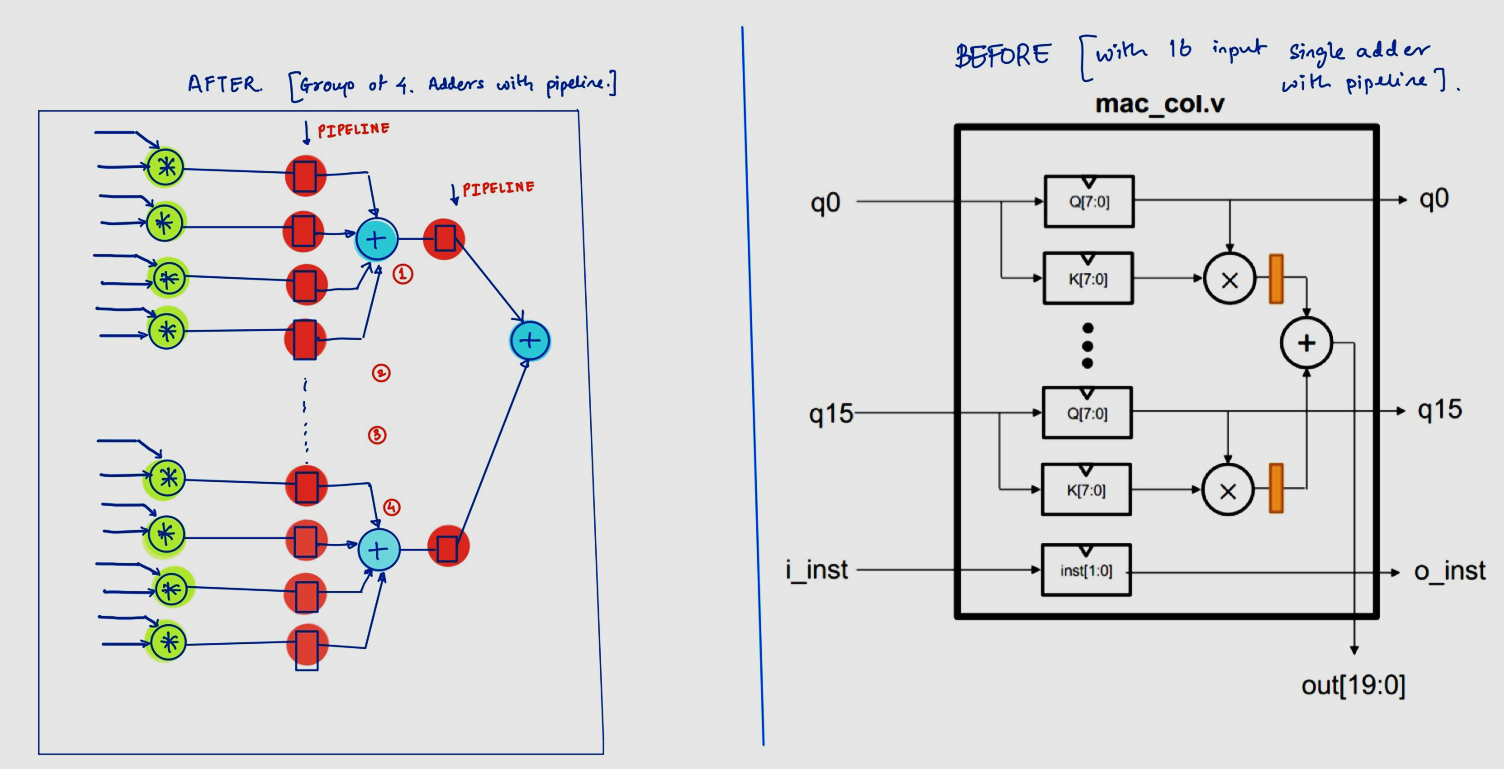
**Part 1:** **Single core synthesis & PnR**

The initial **fullchip** was synthesized to understand the critical paths that cause issue to meet timing at 1GHz clock frequency. Following components were part of the different failing paths:

1. 8-bit, 2-input multiplier in **mac\_col**
2. 16-bit, 16-input adder in **mac\_col**
3. **OFIFO read** **path** to psum\_mem instance

Following changes were made in the design to break the critical paths:

1. Pipeline addition at the **Multiplier output**
2. Split the 16-input adder into **4x 4-input adder**. Pipeline addition at the output of each adder
3. Use output from prev. stage as input to another 4-input adder to generate the matrix element (**KiQj**). Pipeline addition at the output of this adder. Pipelines are also added on the **inst** signals.



The above design has been implemented where the place and route were performed in Innovus. Data pins were placed in the layers M5 and M6 and the routes were confined to upper layers of M4-M6 too. Clock pins were placed in M7 and the routes were restricted from M4-M8 thereby, exploiting maximum resources for positive effect in delay giving advantage in skew and thus, timing. Below are the results of the PnR exercise.

| Power (vectorless) (Leakage | Total) | Reg-reg timing *(WNS | TNS | FEP)* | STD Cell Area | Utilization | DRCs |
| --- | --- | --- | --- | --- |
| 3.74mW | 264 mW | -193ps | -173ns | 1570 | 0.45mm2 | 69.72% | 5 |

Upon analysing the timing paths, it was found that critical paths were contributed by the adder data-path, which was adding 16, 16-bit numbers in one cycle. This feedback was incorporated in RTL design and the adder data-path was further simplified to carry out addition of 4, 16-bit numbers in one cycle (As shown in the image in Page 1). These optimizations were part of Part 2 RTL.

**Part 2: Output normalization**

The next step was to integrate the **sfp\_row** module to perform **normalization** on the data from **mac\_array**. The data from OFIFO is loaded into the following FIFOs:

1. **fifo\_inst\_numerator**: This FIFO will contain the matrix elements (row-wise) generated from mac\_array.
2. **fifo\_inst\_denominator**: This FIFO will contain the summation of the matrix elements (row-wise) generated from mac\_array. Prior to loading this FIFO, the **absolute value** of each matrix element is generated.

The above design creates a large combinational path 🡪 Combinational logic to calculate the absolute value and then to calculate the sum of the 8 elements (absolute value) in each row. This long path has a pipeline in the **sfp\_row** design has a pipeline stage before loading into **fifo\_inst\_denominator**.

Once all rows of the matrix and their corresponding sums are loaded into the above FIFOs, they are read out and sent to the divider to compute **normalized** value. As division logic also has a large combinational logic, the **normalized** value is computed over 3 cycles of the clock i.e., MCP of 2 is applied. The **normalized** matrix elements are now loaded into the psum\_mem\_instance within Core.   
  
The testbench is updated in order to calculate the expected values and to print the simulation result from the single core implementation.

PnR has been executed for the above mentioned RTL design. 1st run was the *vanilla run* to just check the RTL optimization results after the PnR level feedback.

| Runs | Power (vectorless) (Leakage | Total) | Reg-reg timing *(WNS | TNS | FEP)* | STD Cell Area | Utilization | DRCs |
| --- | --- | --- | --- | --- | --- |
| Vanilla Run | 3.87 mW | 253 mW | -694ps | -309ns | 320 | 0.38mm2 | 61.20% | 17 |
| PnR recipes: Max transition constraints on Clock leaf pins (60ps) and Data pins (100ps) + Max limit in Fanout (30) + Applied NDRs on clock trunks (4x) and on clock leaf(2x) | 2.8 mW | 250 mW | -137ps | -131ns | 400 | 0.38mm2 | 56% | 8 |

Upon analysing the PnR results, it was found that MCP can be applied in synthesis to manage the negative slack of 100ps. So, MCP was applied in synthesis for dual core design described below in Part3. MCP was between q\_mem and multiplier. But this MCP application is not valid but the design can be enhanced so that this MCP becomes valid and design can be closed in terms of timing at high frequencies.

Diagram

Description automatically generated**Part 3: Dual Core Implementation**

Two Core instances are added into the **fullchip** module to support parallel processing. To allow for inter-clock domain data transfer between cores, 2 asynchronous FIFOs are added in the **fullchip** module. Each Core will write its **sum** result into an async FIFO while **fifo\_inst\_denominator** within its core is also being populated. So, the **wr\_clk** for the async FIFO will be same its own core clock (source clock).   
  
Once all the sums generated in the Cores are loaded into both async FIFOs, **normalization operation** begins. Async FIFO **rd\_clk** is same as the Core reading the elements (destination clock).

During **normalization**, 1 element from the **fifo\_inst\_denominator** is added with 1 element from the **async FIFO**. The resultant sum is used as denominator for final normalization.

MCP-2 is used during the **division** operation in each core. Hence, 1 element from the FIFOs are read once every 3 clock cycles. This allows for sufficient delay to calculate the sum and to store in the memory.   
The testbench is updated in order to calculate the expected values (considering sums from both cores) and to print the simulation result from the dual core implementation. PnR have been executed for the above mentioned RTL design. 1st run was the *vanilla run* to just check the RTL optimization results after the PnR level feedback.

| Runs | Power (vectorless) (Leakage | Total) | Reg-reg timing *(WNS | TNS | FEP)* | STD Cell Area | Utilization | DRCs |
| --- | --- | --- | --- | --- | --- |
| Vanilla Run | 8.6 mW | 504 mW | -1.1ns | -2790ns | 5525 | 0.72mm2 | 52.10% | 9 |
| PnR recipes: Max transition constraints on Clock leaf pins (60ps) and Data pins (100ps) + Max limit in Fanout (30) + Applied NDRs on clock trunks (4x) and on clock leaf(2x) | 9.9 mW | 509 mW | -197ps | -238ns | 2534 | 1.38mm2 | 61.4% | 15 |
| PnR recipes + MCP between query\_mem to mac\_instance register (Only PnR trial as a prrof of concept for MCP paths) | 5.5mW | 521 mW | 0.00 | 0.00 | 0.00 | 0.73mm2 | 42.2% | 6 |

Background pattern

Description automatically generated

**Part 4: Optimization**

The following strategies are implemented to improve PPA in our design:

1. **Pipelining** to improve Fmax: Pipelines are added within the **mac\_col** and the **sfp\_row** modules to shorten the critical paths and to increase the max frequency of operation of each core.
2. **Clock gating** to reduce Dynamic power: Clock gating cells are added in each Core to turn OFF the clocks to modules which are not operating concurrently.

E.g: **Kmem** and **Qmem** modules can be clock gated when the **normalization** operation is being carried out. **sfp\_row** and **pmem** instances can be clock gated while **mac\_array** is operational.

| Runs (WC view) | Internal Power | Dynamic Power | Leakage Power | Total Power |
| --- | --- | --- | --- | --- |
| With Clock Gating (800MHz) | 47.98 | 15.91 | 10.99 | 74.88 |
| Without Clock Gating (800MHz) | 108.14 | 32.26 | 8.48 | 148.88 |

We can see from the results above that the Dynamic Power reduces by nearly 50% due to clock gating. Clock gating of **mac\_array** module provides use the highest benefit as the computation logic is the biggest component of the dual-core design.

Pointers to Work Directories:

| Data |  |  | Path |
| --- | --- | --- | --- |
| Q1 | RTL | | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/verilogQ1 |
| Synthesis | | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/singlecore\_Q1/syn |
| PnR | | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/singlecore\_Q1/pnr\_trial\_upper\_layer |
| Q2 | RTL | | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/verilogQ2 |
| Synthesis | | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/singlecore\_norm\_Q2/syn |
| PnR | | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/singlecore\_norm\_Q2/pnr\_trial |
| Q3 | RTL | Without Clock  Gating | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/verilogQ3\_clk\_mac\_pipeline\_NO\_clk\_gate |
| Synthesis | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/DualCore\_1.2GHz\_R2\_NO\_CLK\_GATE/syn |
| PnR | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/DualCore\_1.2GHz\_R2\_NO\_CLK\_GATE/pnr |
| SDF sim | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/DualCore\_1.2GHz\_R2\_NO\_CLK\_GATE/rtl\_sim |
| RTL | With Clock Gating | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/verilogQ3\_clk\_updated\_clkgate\_macpipe |
| Synthesis | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/DualCore\_1.2GHz\_R3/syn |
| PnR | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/DualCore\_1.2GHz\_R3/pnr\_no\_mcp |
|  | SDF sim | /home/linux/ieng6/ee260bwi22/sborse/ece260\_project/DualCore\_1.2GHz\_R3/rtl\_sim |

**Conclusions**

1. We can add pipelines in the division logic within the **sfp\_row** module and enable retiming to allow the tool to shorten the critical paths.
2. We can also implement a 2-D systolic array architecture instead of 1-D vector processor architecture. Although this design could take more cycles to provides computed data, critical paths will be much smaller and allows higher frequency operation.
3. For synthesis and PnR stages, we can synthesize 1 core and provide .lib files as input for **fullchip** synthesis. This will reduce the synthesis runtime as critical paths within the core can be closed easily.
4. We have implemented pipelining, clock gating in our single-core and dual-core designs. Applying features such as memory double buffering to reduce the number of cycles required for computation, and reduce the total power consumption of **fullchip**.
5. Asynchronous FIFOs are implemented for fast, multi-bit data transfer between cores. 2-phase or 4-phase handshaking protocol is much slower than asynchronous FIFOs.