# Systolic Arrays
# EE4415 Lab 2 Report

Josiah Mendes
*A0261318J* E0978323@u.nus.edu

## I. OSPE Systolic Array RTL Implementation

The following subsections detail the RTL implementation for the OSPE Systolic Array from the bottom up.

### A. *OSPE.v*

The output stationary processing element (OSPE) has a simple RTL implementation implemented in `part2_p1/src/OSPE.v`. The multiplication and accumulation operation is implemented as combinatorial logic using a wire to store the result as shown in Listing 1. The registered outputs are updated on positive edges of the clock and is reset using the active low `rstnPipe` and `rstnPsum` signals.

```
// multiplication and accumulation
// (combinational logic)
wire [31:0] opC_wire = ipA * ipB + opC;

// update opC
// (sequential logic, reset with rstnPsum)
always @(posedge clk) begin
    if (!rstnPsum) opC <= 32'b0;
    else opC <= opC_wire;
end
```

Listing 1: Implementation of OSPE MAC

### B. *OSPEArray.v*

The `OSPEArray` instantiates 16 OSPE modules and connects them together to form the systolic array. The module outputs 32-bit integers from each processing element and takes in the clock signal, system reset signal, per-element reset signals and the inputs for the first row and column of the array.

To connect each OSPE, 32 32-bit wires are instantiated to direct the two outputs of each PE to the next PE.

Each OSPE has a one cycle delay, and hence for the inputs to be synchronised, the inputs to the first row and the first column In order to multiply and accumulate the correct values, the inputs to the first row and the first row need to be delayed. This is implementing using synchronous logic and 12 32-bit registers to form a triangular shaped array of height 3 on each side of the systolic array.

The implementation code is shown in Listing 2.

```
reg [31:0] ipA3_d1, ipA3_d2, ipA3_d3;
reg [31:0] ipA2_d1, ipA2_d2;
...
always @(posedge clk) begin
    if (!rstnPipe) begin
        ipA1_d1 <= 32'b0;
        ...
    end else begin
        ipA1_d1 <= ipA1;
        ...
        ipB3_d1 <= ipB3;

        ipA2_d2 <= ipA2_d1;
        ...
        ipB3_d2 <= ipB3_d1;

        ipA3_d3 <= ipA3_d2;
        ipB3_d3 <= ipB3_d2;
    end
end
```

Listing 2: Implementation of OSPE Array Delay

### C. *DATA.v*

The `DATA` module is responsible for the output synchronisation of the OSPE array, and selecting the inputs that go into the OSPE array.

*1) Input Data Multiplexer:* This module takes in 8 256-bit inputs for $A_0$ to $A_3$ and $B_0$ to $B_3$ and a selection signal `latCnt`. Depending on the value of `latCnt`, different 32-bits of the inputs are passed to the `OSPEArray` module as inputs `ipA0, ipA1...`. When the latCnt is 0, the lowest 32 bits (31:0) are chosen, and when latCnt is 1, the next 32 bits are chosen (63:32) etc. This multiplexer was implemented combinatorially using a **`case`** statement.

As there was a lot of repetition in each case statement with the same signals being assigned to and from, with just varying chosen bits, a `` `define` `` macro was created to make the code less verbose, shown below in Listing 3, simplifying each case statement to just `` 4'd1: `SET_IP(63, 32). ``

```
`define SET_IP(BIT1, BIT2) begin     \
    ipA0 = MemOutputA0[BIT1:BIT2];  \
    ipA1 = MemOutputA1[BIT1:BIT2];  \
    ...                             \
    ipB3 = MemOutputB3[BIT1:BIT2];  \
end
```

Listing 3: Define Macro for Input Data Multiplexer

*2) Output Synchronisation:* As each PE completes their output at different times, with the last PE completing 6 cycles after the first, a similar approach to input synchronisation as described in the previous section is taken. With a triangle of registers created so that the outputs appear at the output of the DATA module at the same time.

### D. CTRL.v

The CTRL module implements the control for the systolic array and the DATA module. The control is done by taking input signals from the TOP module which change a state machine that determines the output of the module together with a counter.

*1) State Machine:* The finite state machine has 4 states, INIT, CAL, LOAD_NEXT, LOAD_NEXT_IDLE. Transitions from INIT only happen when the input signal startSys goes high causing the state to transition to CAL, otherwise the state remains the same. When in CAL, the next stage (LOAD_NEXT) is only transitioned to when the counter latCnt matches the parameterised systolic latency (number of cycles needed for the calculation of the entire array to complete). LOAD_NEXT is not affected by input signals except for reset, and immediately transitions to LOAD_NEXT_IDLE on the next cycle, which then immediately transistions to CAL on the following cycle. When the rstnSys active low reset signal is low, every state transitions to INIT. This was implemented in Listing 4.

```
reg [1:0] currentState, nextState;

always @ (posedge clk) begin
    if (!rstnSys) currentState <= INIT;
    else currentState <= nextState;
end

always @ (*) begin
  case (currentState)
    INIT: begin
        if (startSys) nextState = CAL;
        else nextState = INIT;
    end
    CAL: begin
        if (latCnt == SYSTOLIC_LATENCY)
             nextState = LOAD_NEXT;
        else nextState = CAL;
    end
    LOAD_NEXT: nextState = LOAD_NEXT_IDLE;
    LOAD_NEXT_IDLE: nextState = CAL;
  endcase
end
```

Listing 4: Implementation of OSPE Array State Machine

*2) Output Generation:* The CTRL module's single bit output signals are generated by the current state of the processor, which control the DATA module and let the TOP module know the current calculation state. The counter latCnt's output is simply implemented as a connection to the internal counter used to keep track of state. rstnPsum's implementation is more complex, as the output signal depends on the value of the counter, determining which processing elements should be reset. A case statement on the current counter value together with the set outputs is used to implement this signal.

### E. TOP.v

This module's implementation is extremely simple, instantiating the CTRL and DATA modules and the wires needed to connect the control signals together so that the units function.

## II. WSPE Systolic Array RTL Implementation

As the CTRL and TOP modules are essentially identical for both arrays, they are not further detailed in this section. The main difference to note is that the CTRL has a different SYSTOLIC_LATENCY value as WSPE systolic arrays produce a new result on every cycle, rather than all together at the end as was the case with the OSPE array.

### A. WSPE.v

The weight stationary processing element operates in a slightly different manner to the OSPE as it takes in one extra input and outputs one less value. But its implementation is similarly simple as shown in part2_p2/src/WSPE.v. The same MAC is applied on different inputs - opC is replaced with input ipPsum as shown in Listing 5. As the reset is synchronous, the **always** block is only triggered on clock edge, rather than in the case of an asynchronous reset, where both clock edge and reset edge are used to trigger the execution of the always block.

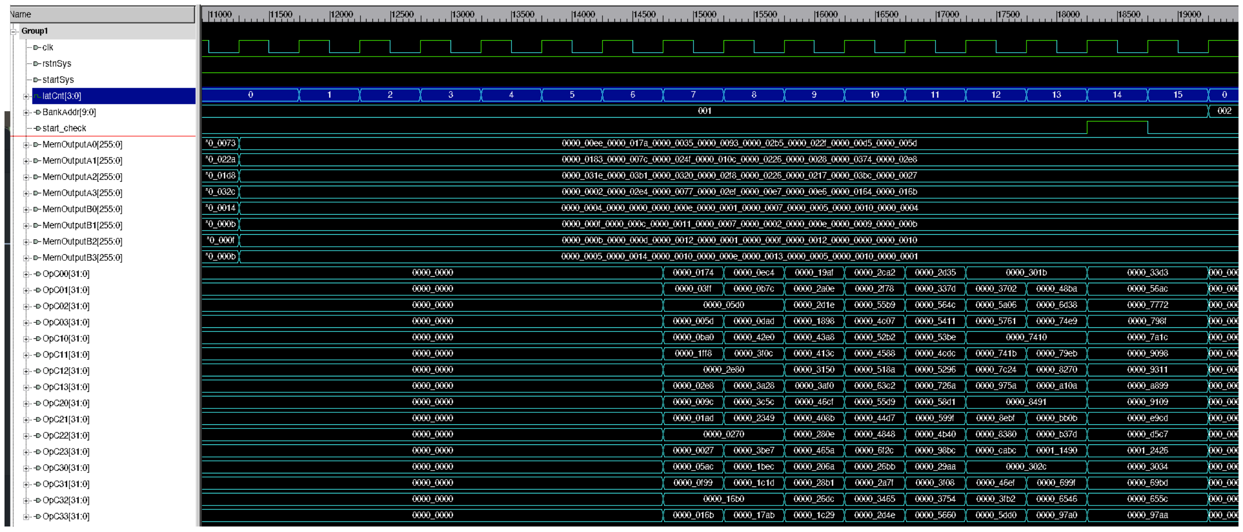
```
// multiplication and accumulation
// (combinational logic)
wire [31:0] opPsum_wire = ipA * ipB + ipPsum;

// update opC
// (sequential logic, reset with rstnPsum)
always @(posedge clk) begin
    if (rstnPsum == 1'b0) opPsum <= 32'b0;
    else opPsum <= opPsum_wire;
end
```

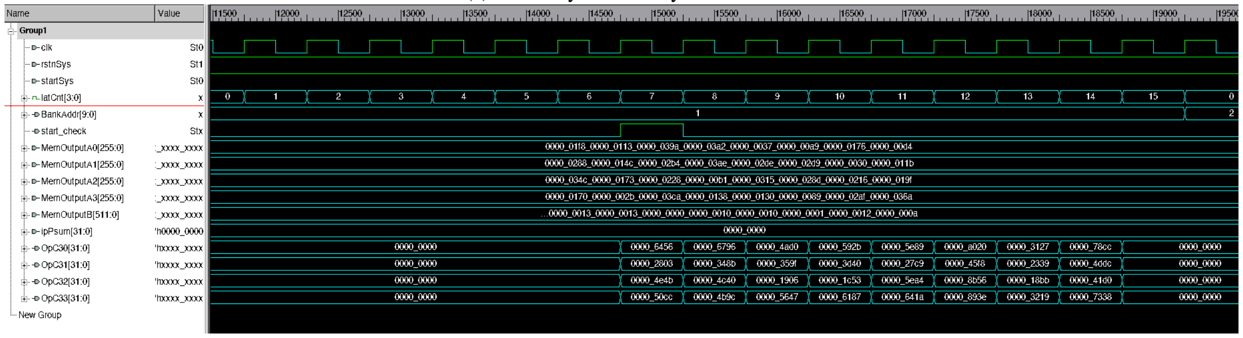
Listing 5: Implementation of WSPE MAC

### B. WSPEArray.v

The WSPEArray is similar to OSPEArray, with the same 16 WSPE module instances and connection between each component using 32 bit wires. The inputs along the side both require delays, so similarly two sets of triangular of registers are created so that the correct inputs are applied to the systolic array at the right time. The WSPEArray also takes in 16 32-bit inputs that are used as stationary weights for each WSPE element. This remains fixed for the entire calculation period and is obtained from the DATA.v module.



(a) OSPE Systolic Array Waveform



(b) WSPE Systolic Array Waveform

### C. DATA.v

This module is similar to its OSPE counterpart with a few differences. As there are fewer inputs and outputs to the systolic array, there are fewer required registers for input and output synchronisation. The `` `define `` statement is used similarly to select the correct $A$ input, while as there are only 4 outputs from the WSPE Array, the number of registers required for output synchronisation goes down from 48 to 6.

An additional mapping is needed to transform the 512 bit `MemOutputB` into the weight mappings for each WSPE array element. This multiplexer on lines 82-99 of `part2_p2/src/DATA.v`, splits the input from memory into 16 32-bit wires for input into WSPEArray module.

## III. RTL Compilation and Testing

The provided makefile and testbench is used for testing. The `compile` command calls the Synopsys design compiler to compile the testbench module into a simulation executable for functional verification with the given timescale, by including all the source files and the top level test bench as arguments for the compiler to find modules.

The command then calls `./simv` to run the simulation, following the commands within the testbench module. The testbench first loads the input data from files into memory, initialises and resets the system under test, and then provides the data to the system. When the DUT sets the `start_check` signal to high, the results are compared to the correct outputs and any errors are recorded.

```
compile:
    vcs -full64 +lint=all \
        -debug_all -timescale=1ns/10ps \
        ../src/* ../tb/tb_TOP.v
    ./simv
plot:
    dve -vpd waveform.vpd &
clean:
    @echo "Remove auto-generated files"
    rm *.txt *.vpd
```

The `plot` command can be run after the `compile` command is run to visualise the produced waveform database and view the signals within the testbench that was produced during the simulation run and saved inside `waveform.vpd`.

The `clean` command removes simulation generated files.

Running the compile command showed that the current implementation for both OSPE and WSPE passed 1024 out of the 1024 provided testcases. Figure 1a shows the successful waveform for the OSPE array with results being produced once every 14 cycles. Figure 1b shows the successful waveform with results being produced every cycle.

## IV. System Synthesis

The system was constrained with the provided constraints shown in Table I. The equivalent commands for Design Vision are shown in Listing 6.

TABLE I: Provided System Synthesis Constraints

| Constraint Name | Constraint |
|---|---|
| Clock Period | 5ns |
| Clock Uncertainty | 0.1ns |
| Input Transition | 0.1ns |
| Input Delay | 0.2ns |
| Output Delay | 0.2ns |
| Load Capacitance | 5fF |
| Driving Gate | 4X Inverter |
| Max Area | 0 |

```
# Create Clock with period 5, ± 0.1
create_clock -period 5.0
    \ [get_ports clk]
set_clock_uncertainty 0.1
    \ [get_clocks clk]

# Set input signal properties
set_input_transition 0.1
    \ [remove_from_collection [all_inputs]
    \ [get_ports clk]]
set_input_delay 0.2 -max -clock clk
    \ [remove_from_collection [all_inputs]
    \ [get_ports {clk rstnSys}]]

# Set output delay and output capacitance
set_output_delay 0.2 -max -clock clk
    \ [all_outputs]
set_load 5 [all_outputs]

# Set Driving Cell
set_driving_cell -lib_cell INVX4_RVT
    \ [all_inputs]

set_max_area 0
```

Listing 6: Design Vision System Constraint Commands

This `constraints.tcl` was used together with `dc-syn.tcl` to synthesise each system. The `dc-syn.tcl` file contained the commands for the Design Vision compiler to first load the design by clearing any existing design, setting the top module and adding the `src` folders for the compiler to locate the module definitions before then reading the design by using `elaborate`. It then sources the constraint file to set the system constraints, before calling `set_fix_multiple_port_nets` which inserts buffers to isolate input ports from output ports, buffer logic constants and prevent cell driver pins from driving multiple outputs. `link` is then called to locate and connect all designs and library components used in the referenced design. The `compile` command then performs logic-level and gate-level synthesis based on the system constraints and provided implementation code. Ports that are unused are then removed at this stage, before the design files (`sdc`, `ddc`, `sdf` and synthesised design file) are written to the output folder. The design is then checked, and area + timing + quality of report (Qor) reports for each design are generated.

The synthesised design file is also a Verilog file, but each module's implementation is replaced with standard cells comprising of basic logic functions such as AND, INV, BUFF and flip flops such as DFF from the chosen library. There are also more complex gates such as FADD which implements a full adder.

### A. *OSPE Array Synthesis*

The OSPE Array synthesis area report is reproduced in Listing 7. The report shows that the large number of registers needed for synchronisation of the outputs results in 21 % of area being used for non-combinational logic, while as multipliers are typically area heavy, combinatorial logic takes up around 60% of total area.

```
Number of ports:                      2574
Number of nets:                       2597
Number of cells:                         2
Number of combinational cells:           0
Number of sequential cells:              0
Number of macros/black boxes:            0
Number of buf/inv:                       0
Number of references:                    2

Combinational area:          71839.900927
Buf/Inv area:                 3751.419727
Noncombinational area:       25583.914339
Macro/Black Box area:            0.000000
Net Interconnect area:       22479.775811
Total cell area:             97423.815266
Total area:                 119903.591077
```

Listing 7: OSPE Array Area Report

For timing, the OSPE array has no paths that violate timing constraints, but it was observed that there are multiple paths paths with low slack (0.01ns) and thus in the real design other introduced uncertainties may lead to unstable performance. The critical path is from input A of OSPE array[0][0], through the multiplier and adder.

```
  Timing Path Group 'clk'
-----------------------------------
Levels of Logic:                46.00
Critical Path Length:            4.89
Critical Path Slack:             0.02
Critical Path Clk Period:        5.00
Total Negative Slack:            0.00
No. of Violating Paths:          1.00
Worst Hold Violation:            0.00
Total Hold Violation:            0.00
No. of Hold Violations:          0.00
```

Listing 8: OSPE Array Timing Report Summary

## B. WSPE Array Synthesis

The area report for the WSPE Array is shown in Listing 7. The combinatorial logic takes up 70% of the total design area but compared to the OSPE array, the total area for combinatorial logic is lower. This is even though each individual OSPE and WSPE have a similar amount of combinatorial logic (1 multiplier + 1 adder), but this will be explored in the next section. The non-combinatorial logic takes up 11% of total area, but it halves the amount needed by the OSPE array when compared directly.

As a result, when the total overall area is compared to the OSPE array, the WSPE array is much more area efficient.

```
Number of ports:                        1710
Number of nets:                         1733
Number of cells:                           2
Number of combinational cells:             0
Number of sequential cells:                0
Number of macros/black boxes:              0
Number of buf/inv:                         0
Number of references:                      2

Combinational area:             65813.892421
Buf/Inv area:                    2095.925672
Noncombinational area:          10937.595653
Macro/Black Box area:               0.000000
Net Interconnect area:          18009.917975
Total cell area:                76751.488074
Total area:                     94761.406049
```

Listing 9: WSPE Array Area Report

The timing summary report is reproduced in Listing 10. There are no timing violations for this design, but there is also very little slack so the circuit may perform differently under real world conditions. The critical timing report showed the worst case critical path to also be on the top left element of the array, with the path from the input A through the multiplier and adder.

```
  Timing Path Group 'clk'
  -----------------------------------
  Levels of Logic:              38.00
  Critical Path Length:          4.87
  Critical Path Slack:           0.00
  Critical Path Clk Period:      5.00
  Total Negative Slack:          0.00
  No. of Violating Paths:        0.00
  Worst Hold Violation:          0.00
  Total Hold Violation:          0.00
  No. of Hold Violations:        0.00
```

Listing 10: WSPE Array Timing Report Summary

Both of the timing analysis show that the main problem with both PEs is the long critical path within an element that connects the multiplier and the adder, so it should be something that is broken up into multiple stages either by register insertion or retiming. The rest of the design comprises of little combinatorial logic and there are many delay stages, so it may be possible to retime the entire design for lower clock frequency.

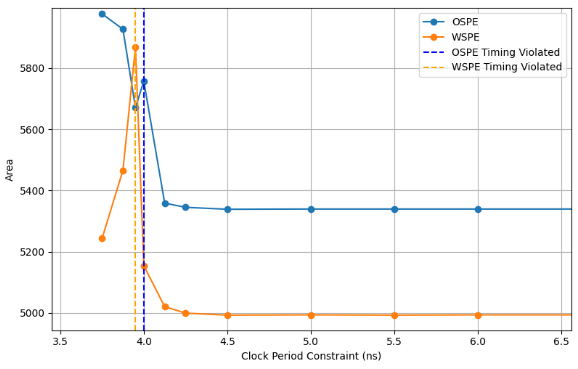


Fig. 2: OSPE vs WSPE Area-Timing Trade-off Curve

## V. Timing and Area Optimisation

In the following sub-sections, the individual OSPE and WSPE elements will be analysed for timing and for area, with analysis of the timing area tradeoff, and techniques applied to make them more efficient.

### A. Initial Area and Timing

*1) OSPE:* Under the provided constraints and the baseline implementation of the OSPE array element, it has a total area of 5340 units, with combinatorial taking up 73% and non-combinatorial taking up 11% and the rest being used for interconnect.

For timing, the baseline implementation has no violated timing paths, and the critical path has 0.77ns slack between the data required time and arrival time.

*2) WSPE:* For the WSPE array element, its total area is 4993 units with the combinatorial using 77% (almost the same as the OSPE), and the non-combinatorial using 8.5%.

On the timing side, there is a slack of 0.77ns for the critical path and no violated paths.

### B. Area vs Timing Trade-off Curve

The area vs timing trade-off curve for the baseline implementation is shown in Figure 2. Curves for both WSPE and OSPE are both plotted on the figure. It can be observed that initially when the clock period constraint is lax, the area remains constant without much change. However, as the timing constraint is reduced to an increasingly smaller value, in an effort to meet the constraint, the synthesis tool increases the area of the circuit by increasing transistor sizing and adding buffers. This increase in area is most significant between around 4.0ns and 4.25ns in the graph above. But once the timing constraints can no longer be met, the area behaviour can vary with the OSPE area still remaining high, while the WSPE area takes a sharp drop.

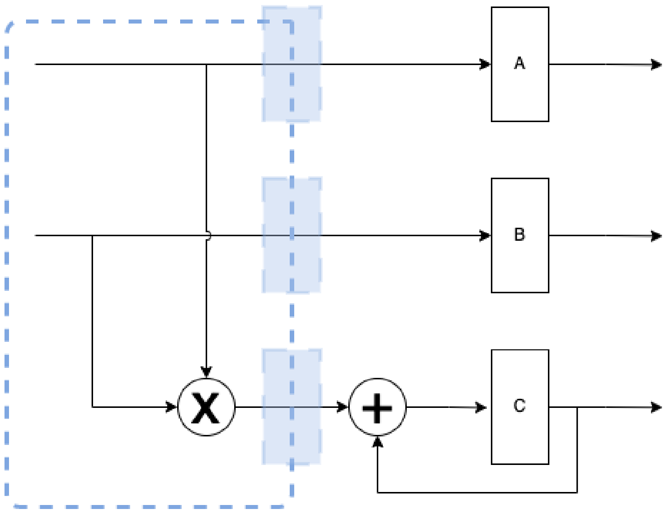


Fig. 3: OSPE Register Insertion through Feedforward Cutset Register Insertion

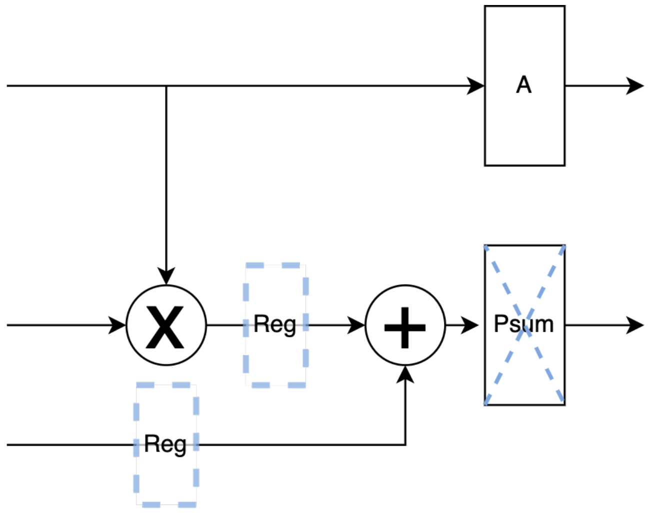


Fig. 4: WSPE Retiming

### C. OSPE Optimisation

*1) Option A:* One way that the critical path on the OSPE element could be broken is by inserting a register in between the multiplier and the adder, which would break the critical path, leaving the worst case delay to just be the propagation delay of the multiplier. By identifying a feed-forward cutset and inserting registers, the functionality of the element can be maintained, as shown in Figure 3.

When re-synthesised with this configuration, the worst case timing path is now as expected between input A through the multiplier to the intermediate register, and with a 5ns clock period constraint has a slack value of 0.91ns. The addition of the extra 3 registers leads to an increase in area from 5340 units to 6246 units, and non-combinatorial taking up 20% of total area. However, when combined with the rest of the circuit, as each element takes more than one cycle to process an element now, it would require a redesign of the circuit to include more synchronisation and delay logic.

*2) Option B:* Another way that the design can be optimised is by performing retiming using the synthesis tool. Adding the following command in Listing 11 tells the synthesis tool to find the critical loop and retime the sequential cells to meet the timing constraints.

```
optimize_registers  -print_critical_loop \
                    -minimum_period_only
```

Listing 11: Synthesis Tool Command to Retime Registers on Mapped Gatelist

This option was tested on the single element and reduced the critical path length from 4.87ns to 2.09ns, increasing the slack value to 2.58ns. However, it also leads to the area increasing, causing the total area to increase to 6553 units, due to the addition of more sequential cells and registers in design, similar to Option A.

### D. WSPE Optimisation

As the WSPE does not have a loop within the circuit, it is a bit easier to optimise. From the naive implementation, the output register for `opPsum` can be pushed backwards through the adder, adding a register between the multiplier and the adder, and an additional register on `ipPsum` for synchronisation. This change is shown in Figure 4. As this change does not add any extra registers, the overall functionality of the entire circuit remains the same and it continues to function, passing all 1024 testcases.

The individual element timing report shows an improvement as the critical path is reduced to 3.76ns, allowing a slack of 0.91ns. The element area also increases due to the additional register, going up to 5409 units.

However, this change does mean that when put into the array, as each adder is chained together, there still remains a long critical path in the design. This was shown when synthesising the entire system with the updated design, as the critical path length was 4.87ns, with 0ns slack and no timing paths violated. However, to meet that target, the total area was increased to 101998 units with significant increases in non-combinatorial units.

## VI. Open-Ended Improvement

There are multiple potential places for improvement in the design. As they are time intensive and require redesigning the entire circuit, they were not implemented, but rather described.

### A. Reducing Calculation Precision

Currently, each multiply operation within each processing element has to do a 32-bit multiply and accumulate operation. The 32-bit multiply is an expensive operation, as seen in the previous sections. It is possible that given a knowledge and analysis of the data inputs to the systolic array, that not the full range of 32-bit numbers is being used which would allow a smaller datatype and hence a less-expensive multiply operation that has a lower propagation delay.

### *B. Custom MAC operator*

The multiply and accumulate operation is currently implemented using a Verilog multiply symbol and a Verilog add, which allows the compiler to determine what the best implementation is. The synthesis tool creates an instance a full 32-bit multiply instance producing a 64-bit result, cascaded with a 32-bit full adder.

Previous research has shown that it is possible to create a merged MAC operator that optimises for the particular use case that would allow a reduction in area and an increase in speed.