

ECE 327/627

Digital Hardware Systems

Lecture 4: Design Case Studies

Andrew Boutros

andrew.boutros@uwaterloo.ca

Some Logistics

- Don't forget to register your lab team on LEARN by tomorrow!
 - If you are an ECE 627 student working alone on the labs, you need to register a team of 1 on LEARN so we can create the GitLab repo for you
- Lab 1 deadline is in next Friday
 - Worth **5%** of the total course grade for ECE 327
 - Worth **3%** of the total course grade for ECE 627
 - Deadline **January 23 @ 11:59 pm**

In the Previous Lecture ...

- More SystemVerilog basics
 - Blocking (=) vs. non-blocking (<=) assignments
 - Blocking → “sequential” description of the behavior
 - Non-blocking → “concurrent” description of the behavior
 - Conditions (if-else or case) in procedural blocks
 - Translate to multiplexers in hardware
 - For loops in procedural blocks
 - Describe the **behavior** of a circuit using loops
 - Generate for loops
 - **Instantiate** many sub-components of a module
 - Writing SystemVerilog testbenches for simulation
 - How does the simulator work behind the scenes?
 - Event-driven simulation → evaluate & update events
 - Blocking assignments (=) – update right after evaluate
 - Non-blocking assignments (<=) – update later at the end of time step

Machine Learning (ML) Hardware is Everywhere

- Many ML hardware startups

groq™

cerebras

SambaNova®
SYSTEMS



d-Matrix

etched

Tenstorrent

Machine Learning (ML) Hardware is Everywhere

- Many ML hardware startups

groq™

cerebras

SambaNova®
SYSTEMS



d-Matrix

etched

Tenstorrent

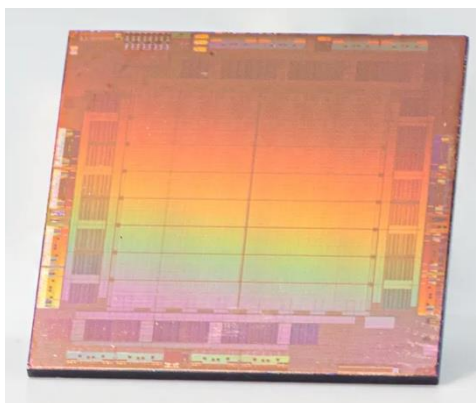
- Many large software companies building ML hardware

Microsoft Maia



Source: theverge.com

Meta MTIA



Source: zdnet.com

Google TPU



Source: forbes.com

Amazon Trainium



Source: medium.com

Two Case Studies

- Use some of what we learned about SystemVerilog & digital hardware design so far to design parts of commercial ML accelerators:
 - Nvidia Tensor Cores
 - Google Tensor Processing Unit (TPU)

Nvidia Tensor Cores

- Early on, Nvidia realized that ML compute will need more than just conventional GPU architecture
- Introduced new “Tensor Cores” in their Volta architecture in 2017
 - Functional units dedicated only for matrix-matrix multiplication
 - Most important operation in almost all ML workloads
 - One 4x4 matrix multiply-add / cycle

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} + \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

Source: Nvidia



Source: Nvidia

Nvidia Tensor Cores

- Early on, Nvidia realized that ML compute will need more than just conventional GPU architecture
- Introduced new “Tensor Cores” in their Volta architecture in 2017
 - Functional units dedicated only for matrix-matrix multiplication
 - Most important operation in almost all ML workloads
 - One 4x4 matrix multiply-add / cycle

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} + \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

Source: Nvidia

Let's try and implement one of those tensor cores!



Source: Nvidia

Tensor Core Computation

$$\begin{bmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} + \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix}$$

- Computation can be formulated using this pseudo code:

```
for (int i = 0; i < 4; i++) {  
    for (int j = 0; j < 4; j++) {  
        D[i,j] = C[i,j];  
        for (int k = 0; k < 4; k++) {  
            D[i,j] += A[i,k] * B[k,j];  
        }  
    }  
}
```

Tensor Core Computation

$$\begin{bmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} + \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix}$$

- Computation can be formulated using this pseudo code:

```
for (int i = 0; i < 4; i++) {  
    for (int j = 0; j < 4; j++) {  
        D[i,j] = C[i,j];  
        for (int k = 0; k < 4; k++) {  
            D[i,j] += A[i,k] * B[k,j];  
        }  
    }  
}
```

For each one of the 4x4 output elements,

Tensor Core Computation

$$\begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

- Computation can be formulated using this pseudo code:

```

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        D[i,j] = C[i,j];
        for (int k = 0; k < 4; k++) {
            D[i,j] += A[i,k] * B[k,j];
        }
    }
}

```

For each one of the 4x4 output elements,

Dot product between a row of A and a column of B

Tensor Core Computation

$$\begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

- Computation can be formulated using this pseudo code:

```

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        D[i,j] = C[i,j];
        for (int k = 0; k < 4; k++) {
            D[i,j] += A[i,k] * B[k,j];
        }
    }
}

```

For each one of the 4x4 output elements,

Add a scalar to the result of the dot product

Dot product between a row of A and a column of B

Tensor Core Computation

$$\begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

- Computation can be formulated using this pseudo code:

```

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        D[i,j] = C[i,j];
        for (int k = 0; k < 4; k++) {
            D[i,j] += A[i,k] * B[k,j];
        }
    }
}

```

For each one of the 4x4 output elements,

Add a scalar to the result of the dot product

Dot product between a row of A and a column of B

For simplicity, we will assume A & B are 8b matrices and C & D are 32b matrices!

Tensor Core Computation

$$\begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

- Computation can be formulated using this pseudo code:

```

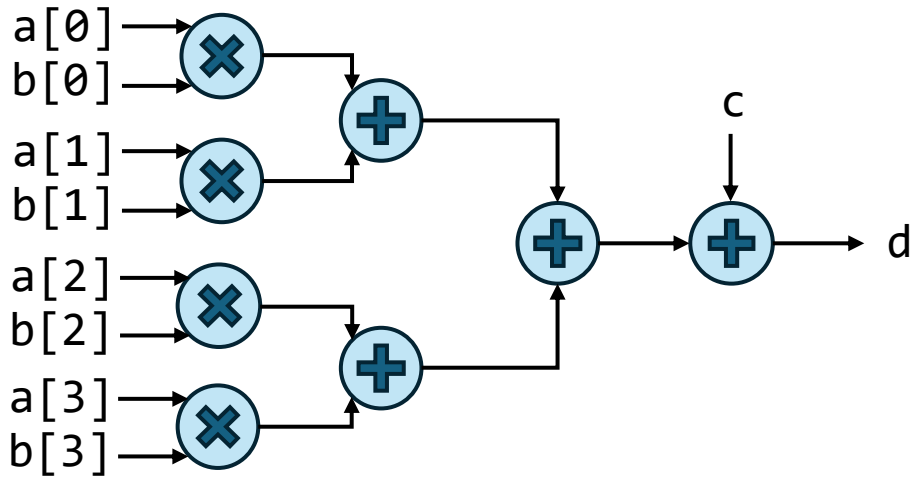
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        D[i,j] = C[i,j];
        for (int k = 0; k < 4; k++) {
            D[i,j] += A[i,k] * B[k,j];
        }
    }
}

```

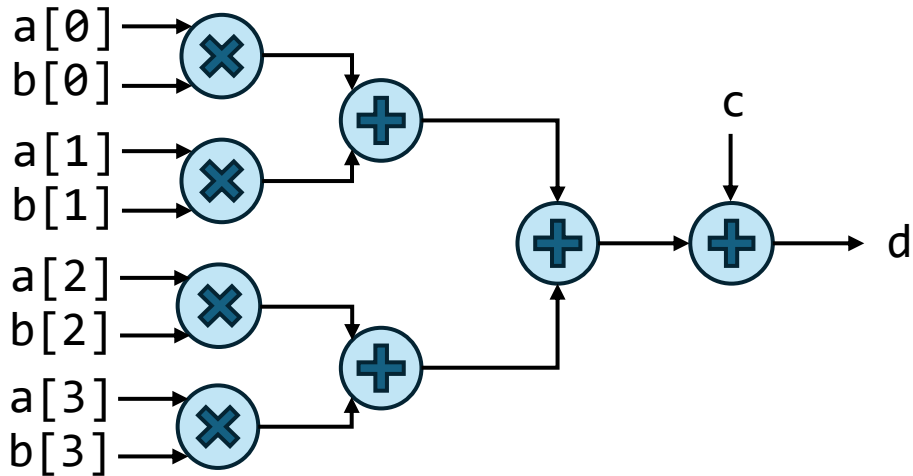
For each one of the 4x4 output elements,
Add a scalar to the result of the dot product
Dot product between a row of A and a column of B

For simplicity, we will assume A & B are 8b matrices and C & D are 32b matrices!

Dot Product Module

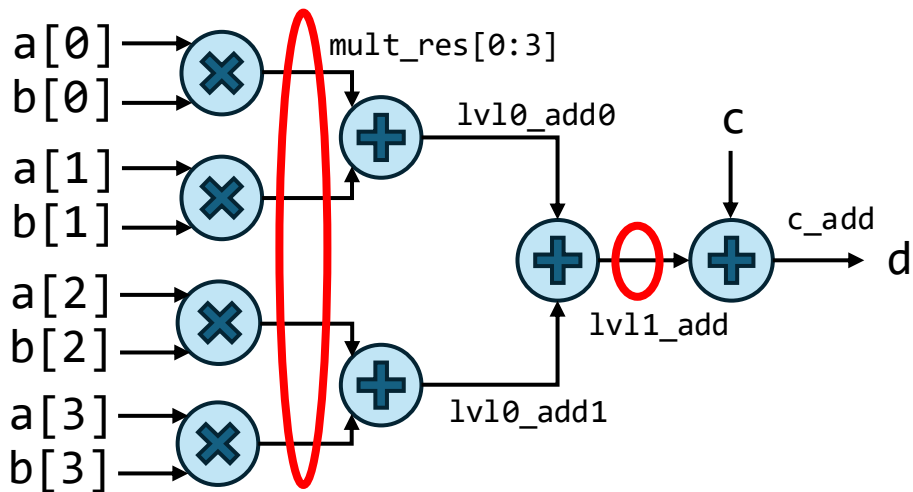


Dot Product Module



```
module dot (  
    input signed [7:0] a [0:3],  
    input signed [7:0] b [0:3],  
    input signed [31:0] c,  
    output signed [31:0] d  
);  
  
assign d = (a[0] * b[0]) +  
           (a[1] * b[1]) +  
           (a[2] * b[2]) +  
           (a[3] * b[3]) + c;  
  
endmodule
```

Dot Product Module



```

module dot (
    input signed [7:0] a [0:3],
    input signed [7:0] b [0:3],
    input signed [31:0] c,
    output signed [31:0] d
);

assign d = (a[0] * b[0]) +
           (a[1] * b[1]) +
           (a[2] * b[2]) +
           (a[3] * b[3]) + c;

endmodule

```

=

```

module dot (
    input signed [7:0] a [0:3],
    input signed [7:0] b [0:3],
    input signed [31:0] c,
    output signed [31:0] d
);

logic signed [15:0] mult_res [0:3];
logic signed [16:0] lv10_add0, lv10_add1;
logic signed [17:0] lv11_add;
logic signed [31:0] c_add;

always_comb begin
    mult_res[0] = a[0] * b[0];
    mult_res[1] = a[1] * b[1];
    mult_res[2] = a[2] * b[2];
    mult_res[3] = a[3] * b[3];

    lv10_add0 = mult_res[0] + mult_res[1];
    lv10_add1 = mult_res[2] + mult_res[3];

    lv11_add = lv10_add0 + lv10_add1;

    c_add = lv11_add + c;

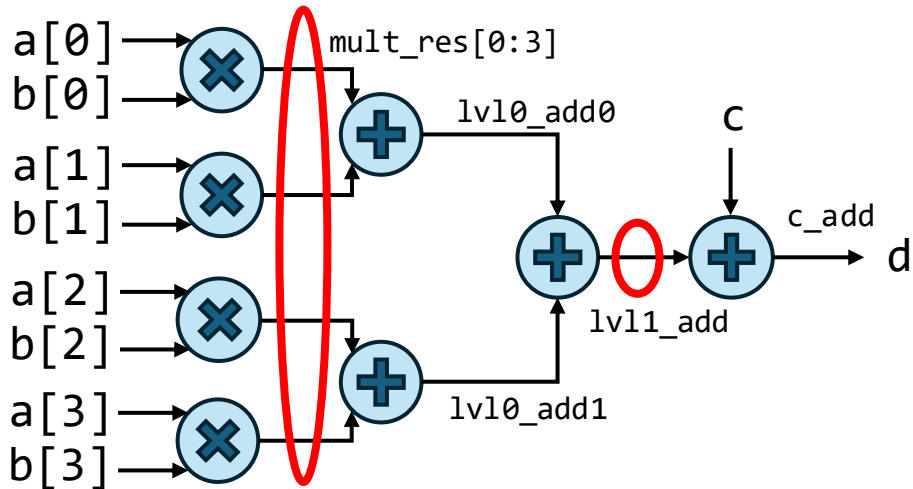
end

assign d = c_add;

endmodule

```

Dot Product Module



```

module dot (
    input signed [7:0] a [0:3],
    input signed [7:0] b [0:3],
    input signed [31:0] c,
    output signed [31:0] d
);

assign d = (a[0] * b[0]) +
           (a[1] * b[1]) +
           (a[2] * b[2]) +
           (a[3] * b[3]) + c;

endmodule

```

=

```

module dot (
    input signed [7:0] a [0:3],
    input signed [7:0] b [0:3],
    input signed [31:0] c,
    output signed [31:0] d
);

    Be careful with precisions
    logic signed [15:0] mult_res [0:3];
    logic signed [16:0] lv10_add0, lv10_add1;
    logic signed [17:0] lv11_add;
    logic signed [31:0] c_add;

    always_comb begin
        mult_res[0] = a[0] * b[0];
        mult_res[1] = a[1] * b[1];
        mult_res[2] = a[2] * b[2];
        mult_res[3] = a[3] * b[3];

        lv10_add0 = mult_res[0] + mult_res[1];
        lv10_add1 = mult_res[2] + mult_res[3];

        lv11_add = lv10_add0 + lv10_add1;

        c_add = lv11_add + c;

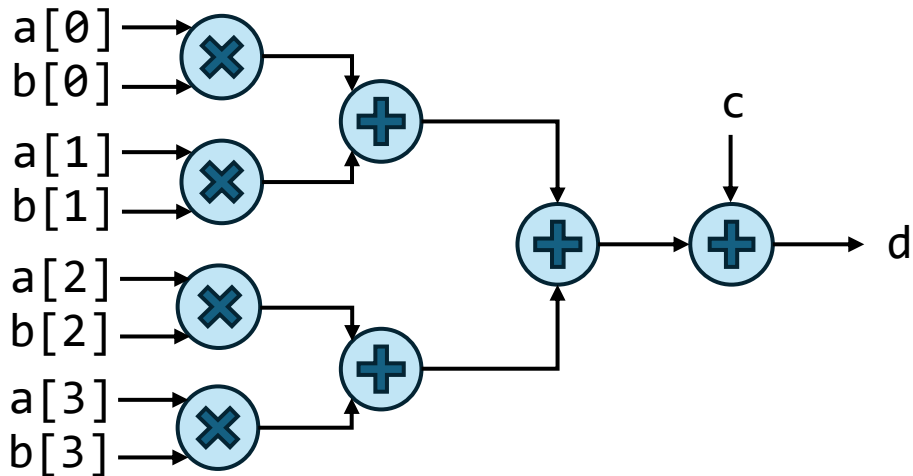
    end

    assign d = c_add;

endmodule

```

Dot Product Module



```

module dot (
    input signed [7:0] a [0:3],
    input signed [7:0] b [0:3],
    input signed [31:0] c,
    output signed [31:0] d
);

assign d = (a[0] * b[0]) +
           (a[1] * b[1]) +
           (a[2] * b[2]) +
           (a[3] * b[3]) + c;

endmodule

```

=

Can write a fancier implementation in case in next generation we want to change the tensor core numerical precision or size!

```

module dot # (
    parameter IDATAW = 8,
    parameter ODATAW = 32,
    parameter VECLLEN = 4
)(
    input signed [IDATAW-1:0] a [0:VECLLEN-1],
    input signed [IDATAW-1:0] b [0:VECLLEN-1],
    input signed [ODATAW-1:0] c,
    output signed [ODATAW-1:0] d
);

logic signed [ODATAW-1:0] result;
integer i;

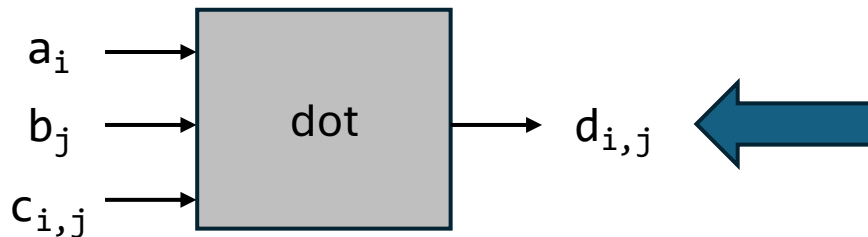
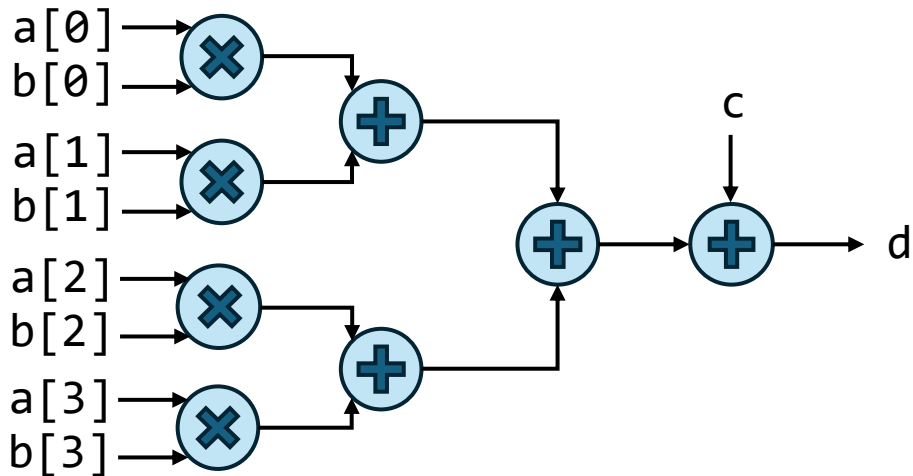
always_comb begin
    result = 'd0;
    for (i = 0; i < VECLLEN; i = i + 1) begin
        result = result + (a[i] * b[i]);
    end
    result = result + c;
end

assign d = result;

endmodule

```

Dot Product Module



Can write a fancier implementation in case in next generation we want to change the tensor core numerical precision or size!

```

module dot # (
    parameter IDATAW = 8,
    parameter ODATAW = 32,
    parameter VECLEN = 4
) (
    input signed [IDATAW-1:0] a [0:VECLEN-1],
    input signed [IDATAW-1:0] b [0:VECLEN-1],
    input signed [ODATAW-1:0] c,
    output signed [ODATAW-1:0] d
);

logic signed [ODATAW-1:0] result;
integer i;

always_comb begin
    result = 'd0;
    for (i = 0; i < VECLEN; i = i + 1) begin
        result = result + (a[i] * b[i]);
    end
    result = result + c;
end

assign d = result;

endmodule
  
```

Tensor Core Computation

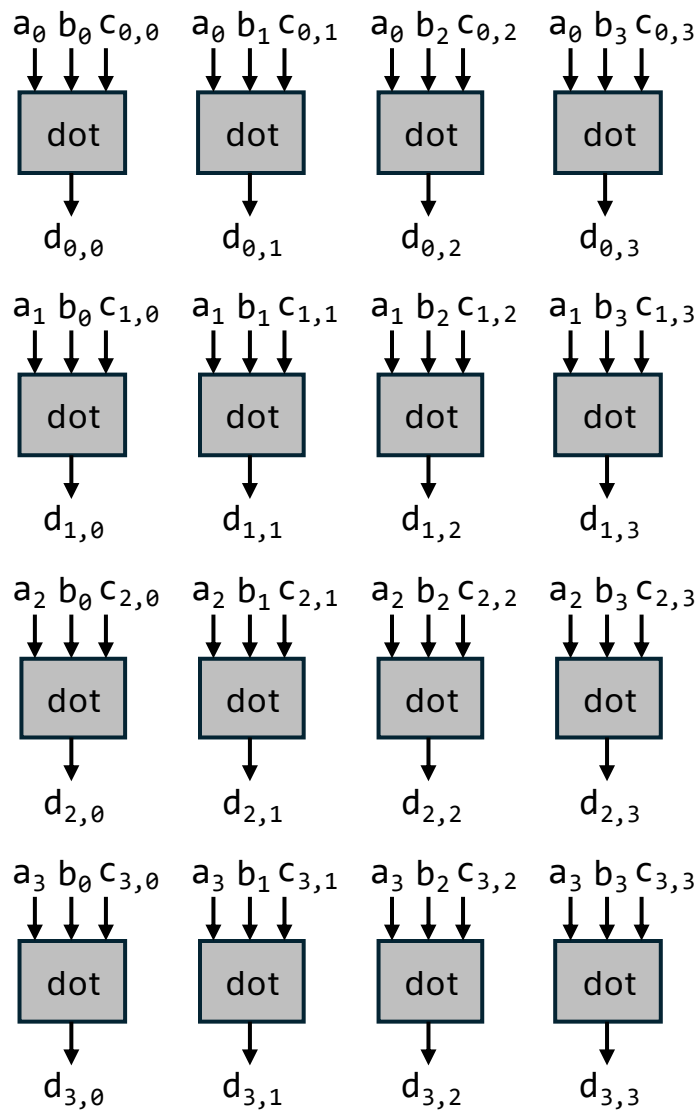
$$\begin{bmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} + \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix}$$

- Computation can be formulated using this pseudo code:

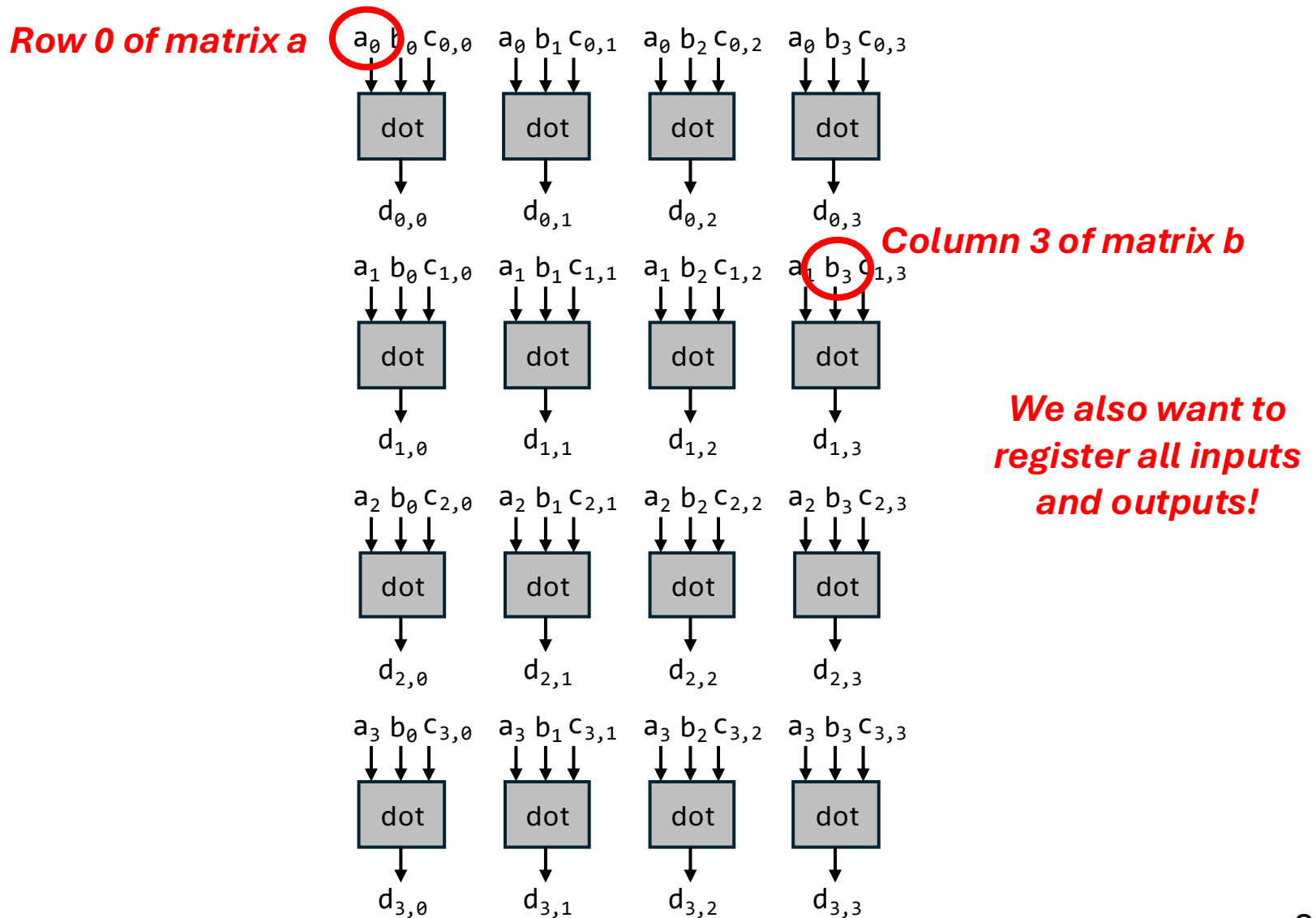
```
for (int i = 0; i < 4; i++) {  
  for (int j = 0; j < 4; j++) {  
    D[i,j] = C[i,j];  
    for (int k = 0; k < 4; k++) {  
      D[i,j] += A[i,k] * B[k,j];  
    }  
  }  
}
```



Tensor Core Module



Tensor Core Module



```
module tensor_core (  
    input clk,  
    input rst,  
    input signed [7:0] a [0:3][0:3],  
    input signed [7:0] b [0:3][0:3],  
    input signed [31:0] c [0:3][0:3],  
    output logic signed [31:0] d [0:3][0:3]  
);
```

```
endmodule
```

Define the interfaces of the tensor core:

- 3 input matrices (a, b, c)
- 1 output matrix (d)
- Clock and Reset signal

```

module tensor_core (
    input  clk,
    input  rst,
    input  signed [7:0] a [0:3][0:3],
    input  signed [7:0] b [0:3][0:3],
    input  signed [31:0] c [0:3][0:3],
    output logic signed [31:0] d [0:3][0:3]
);

logic signed [7:0] a_reg [0:3][0:3];
logic signed [7:0] b_reg [0:3][0:3];
logic signed [31:0] c_reg [0:3][0:3];
logic signed [31:0] d_wire [0:3][0:3];

integer k, l;
always_ff @ (posedge clk) begin
    for (k = 0; k < 4; k = k+1) begin
        for (l = 0; l < 4; l = l+1) begin
            if (rst) begin
                a_reg[k][l] <= 'd0;
                b_reg[k][l] <= 'd0;
                c_reg[k][l] <= 'd0;
                d[k][l] <= 'd0;
            end else begin
                a_reg[k][l] <= a[k][l];
                b_reg[k][l] <= b[l][k];
                c_reg[k][l] <= c[k][l];
                d[k][l] <= d_wire[k][l];
            end
        end
    end
end
end
end

```

```
endmodule
```

Register the inputs and outputs of the tensor core in an always_ff block

```

module tensor_core (
    input  clk,
    input  rst,
    input  signed [7:0] a [0:3][0:3],
    input  signed [7:0] b [0:3][0:3],
    input  signed [31:0] c [0:3][0:3],
    output logic signed [31:0] d [0:3][0:3]
);

```

```

    logic signed [7:0] a_reg [0:3][0:3];
    logic signed [7:0] b_reg [0:3][0:3];
    logic signed [31:0] c_reg [0:3][0:3];
    logic signed [31:0] d_wire [0:3][0:3];

```

```

    integer k, l;

```

```

    always_ff @ (posedge clk) begin

```

```

        for (k = 0; k < 4; k = k+1) begin

```

```

            for (l = 0; l < 4; l = l+1) begin

```

```

                if (rst) begin

```

```

                    a_reg[k][l] <= 'd0;

```

```

                    b_reg[k][l] <= 'd0;

```

```

                    c_reg[k][l] <= 'd0;

```

```

                    d[k][l] <= 'd0;

```

```

                end else begin

```

```

                    a_reg[k][l] <= a[k][l];

```

```

                    b_reg[k][l] <= b[l][k];

```

```

                    c_reg[k][l] <= c[k][l];

```

```

                    d[k][l] <= d_wire[k][l];

```

```

                end

```

```

            end

```

```

        end

```

```

    end

```

**Flip indices
for b such
that $b_reg[i]$
would hold
the i -th
column of b**

```

endmodule

```

Register the inputs and outputs of the tensor core in an always_ff block

```

module tensor_core (
    input  clk,
    input  rst,
    input  signed  [7:0] a [0:3][0:3],
    input  signed  [7:0] b [0:3][0:3],
    input  signed [31:0] c [0:3][0:3],
    output logic signed [31:0] d [0:3][0:3]
);

logic signed  [7:0] a_reg [0:3][0:3];
logic signed  [7:0] b_reg [0:3][0:3];
logic signed [31:0] c_reg [0:3][0:3];
logic signed [31:0] d_wire [0:3][0:3];

integer k, l;
always_ff @ (posedge clk) begin
    for (k = 0; k < 4; k = k+1) begin
        for (l = 0; l < 4; l = l+1) begin
            if (rst) begin
                a_reg[k][l] <= 'd0;
                b_reg[k][l] <= 'd0;
                c_reg[k][l] <= 'd0;
                d[k][l] <= 'd0;
            end else begin
                a_reg[k][l] <= a[k][l];
                b_reg[k][l] <= b[l][k];
                c_reg[k][l] <= c[k][l];
                d[k][l] <= d_wire[k][l];
            end
        end
    end
end
end
end

```

```

genvar i, j;
generate
for (i = 0; i < 4; i = i+1) begin: gen_row
    for (j = 0; j < 4; j = j+1) begin: gen_col

        end
    end
endgenerate

endmodule

```

Use a generate for loop to instantiate all 16 dot modules instead of doing it manually

```

module tensor_core (
    input  clk,
    input  rst,
    input  signed  [7:0] a [0:3][0:3],
    input  signed  [7:0] b [0:3][0:3],
    input  signed [31:0] c [0:3][0:3],
    output logic signed [31:0] d [0:3][0:3]
);

logic signed  [7:0] a_reg [0:3][0:3];
logic signed  [7:0] b_reg [0:3][0:3];
logic signed [31:0] c_reg [0:3][0:3];
logic signed [31:0] d_wire [0:3][0:3];

integer k, l;
always_ff @ (posedge clk) begin
    for (k = 0; k < 4; k = k+1) begin
        for (l = 0; l < 4; l = l+1) begin
            if (rst) begin
                a_reg[k][l] <= 'd0;
                b_reg[k][l] <= 'd0;
                c_reg[k][l] <= 'd0;
                d[k][l] <= 'd0;
            end else begin
                a_reg[k][l] <= a[k][l];
                b_reg[k][l] <= b[l][k];
                c_reg[k][l] <= c[k][l];
                d[k][l] <= d_wire[k][l];
            end
        end
    end
end
end
end

```

```

genvar i, j;
generate
for (i = 0; i < 4; i = i+1) begin: gen_row
    for (j = 0; j < 4; j = j+1) begin: gen_col
        dot #(
            .IDATAW(8),
            .ODATAW(32),
            .VECLEN(4)
        ) dot_inst (
            .a(a_reg[i]),
            .b(b_reg[j]),
            .c(c_reg[i][j]),
            .d(d_wire[i][j])
        );
    end
end
endgenerate

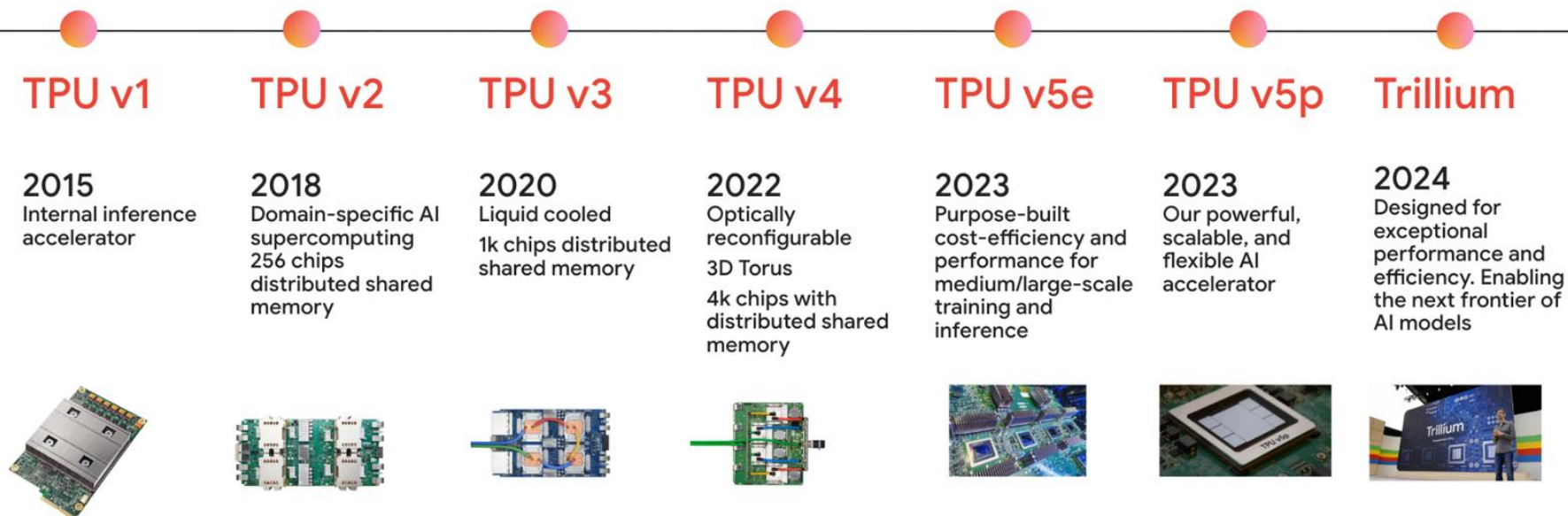
endmodule

```

Instantiate and connect the dot modules

Google Tensor Processing Unit (TPU)

- In 2015, Google expected that if each user utilized ML-based voice search for only 3 mins per day, they would need to double their datacenter CPUs (billions of \$ every year)
- That's when they decided to build a specialized chip that can execute only ML workloads very efficiently



TPUv4 Systems

- A TPU pod is 64 racks x 8 units x 2 boards x 4 chips = 4096 chips working together on one job
- Connections between racks optically-switched
- 7 nm chip with 22B transistors packaged with 4 stacks of high-bandwidth memory (HBM)

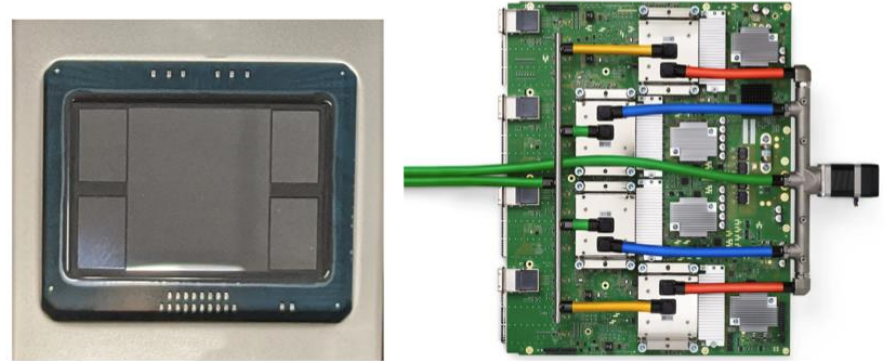


Figure 2: The TPU v4 package (ASIC in center plus 4 HBM stacks) and printed circuit board with 4 liquid-cooled packages. The board's front panel has 4 top-side PCIe connectors and 16 bottom-side OSFP connectors for inter-tray ICI links.



Figure 3: Eight of 64 racks for one 4096-chip supercomputer.

N. Jouppi, et al., “TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings”, ISCA 2023
Nice talk about TPUv4: <https://youtu.be/osurjQmKrys?si=Hd67GB-8YiSdfvPg>

TPUv4 Systems

- A TPU pod is 64 racks x 8 units x 2 boards x 4 chips = 4096 chips working together on one job
- Connections between racks optically-switched
- 7 nm chip with 22B transistors packaged with 4 stacks of high-bandwidth memory (HBM)

Let's look inside the chip!

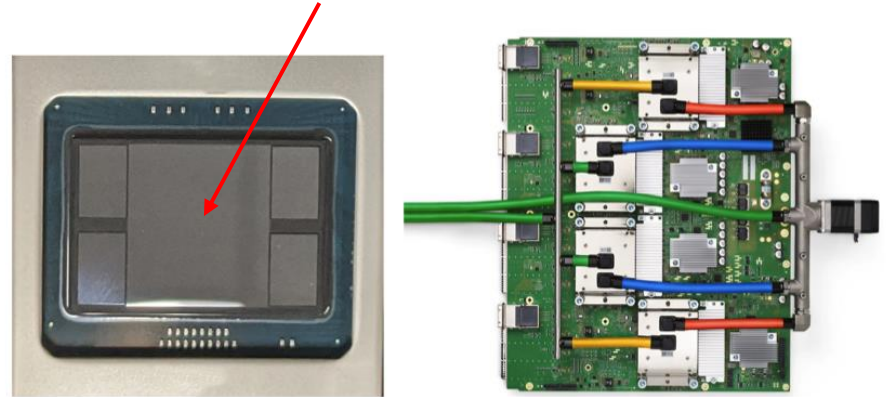


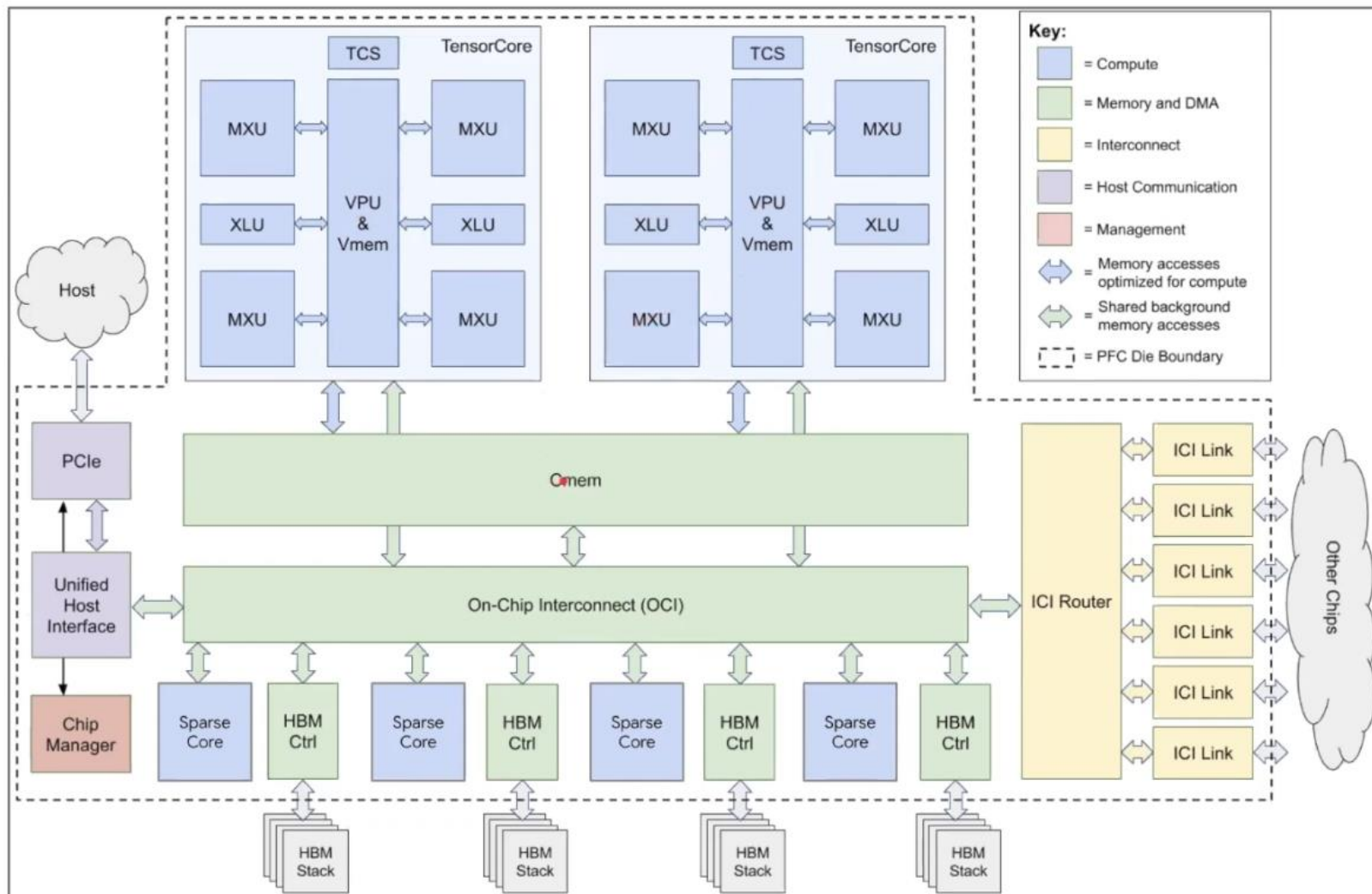
Figure 2: The TPU v4 package (ASIC in center plus 4 HBM stacks) and printed circuit board with 4 liquid-cooled packages. The board's front panel has 4 top-side PCIe connectors and 16 bottom-side OSFP connectors for inter-tray ICI links.



Figure 3: Eight of 64 racks for one 4096-chip supercomputer.

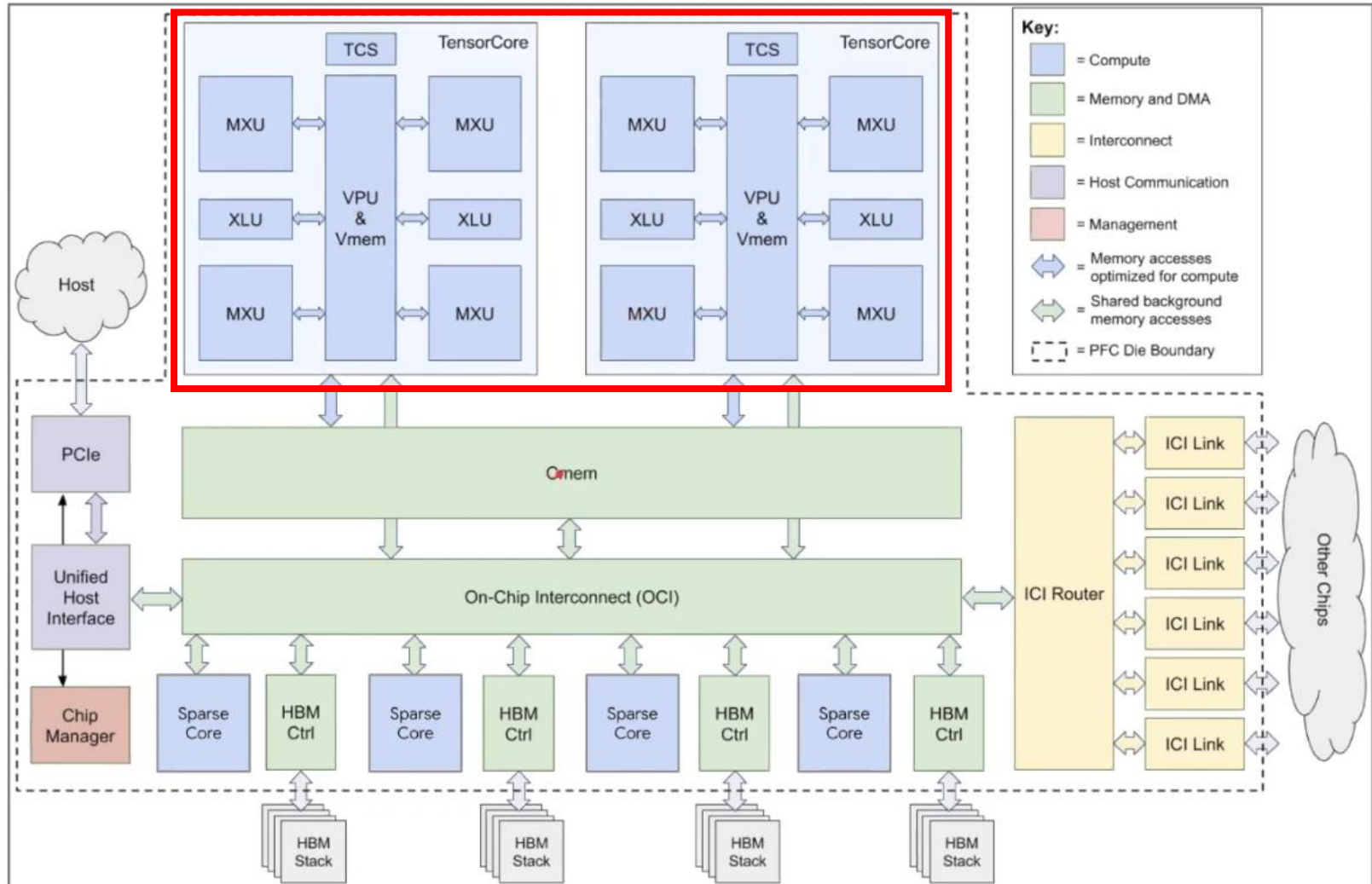
N. Jouppi, et al., “TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings”, ISCA 2023
Nice talk about TPUv4: <https://youtu.be/osurjQmKrys?si=Hd67GB-8YiSdfvPg>

TPUv4 Chip Architecture



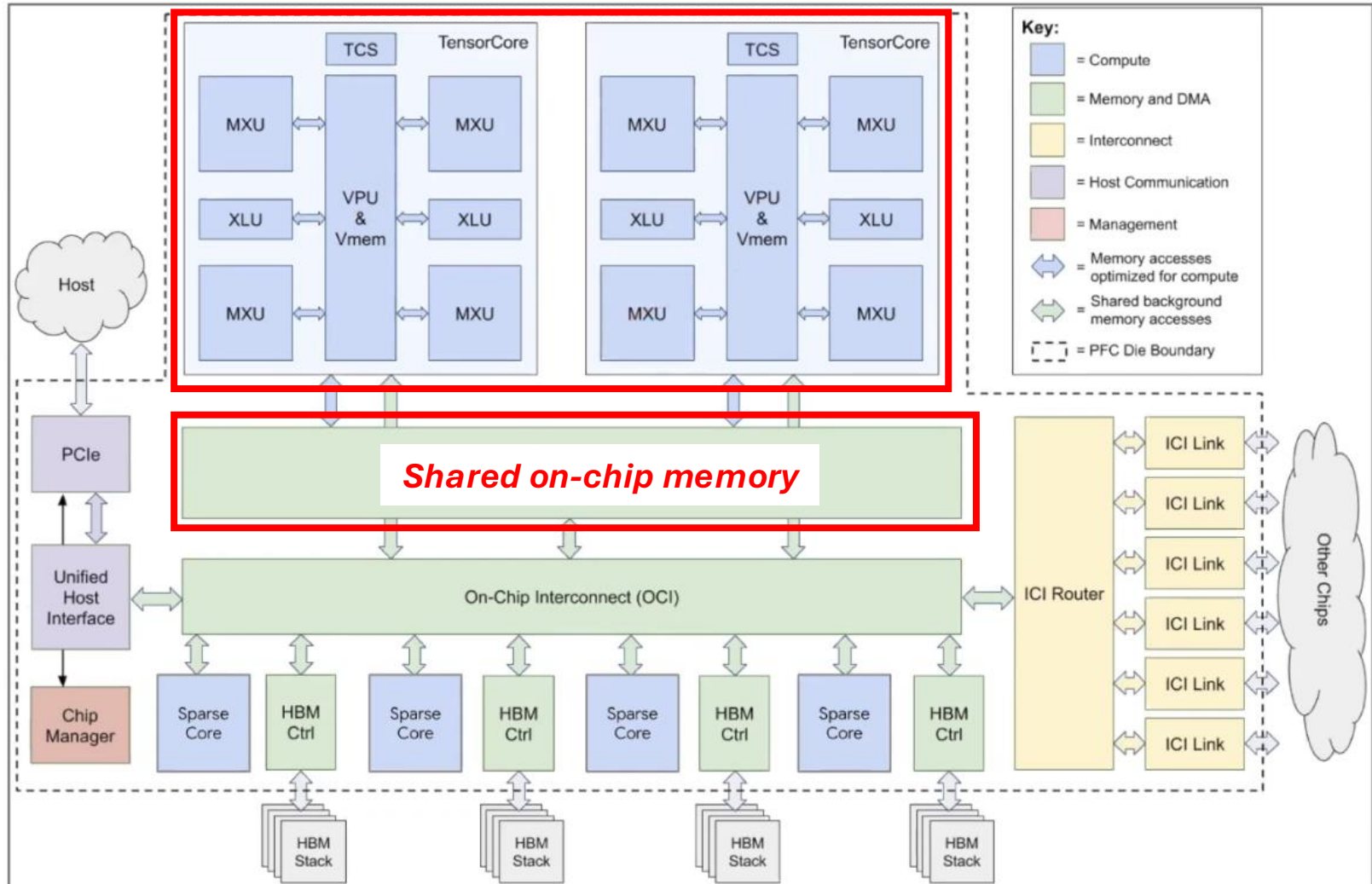
TPUv4 Chip Architecture

*Two cores where the compute happens
(matrix mult, transpose, vector ops)
Each core has four 128x128 systolic
matrix-matrix multipliers (MXUs)*



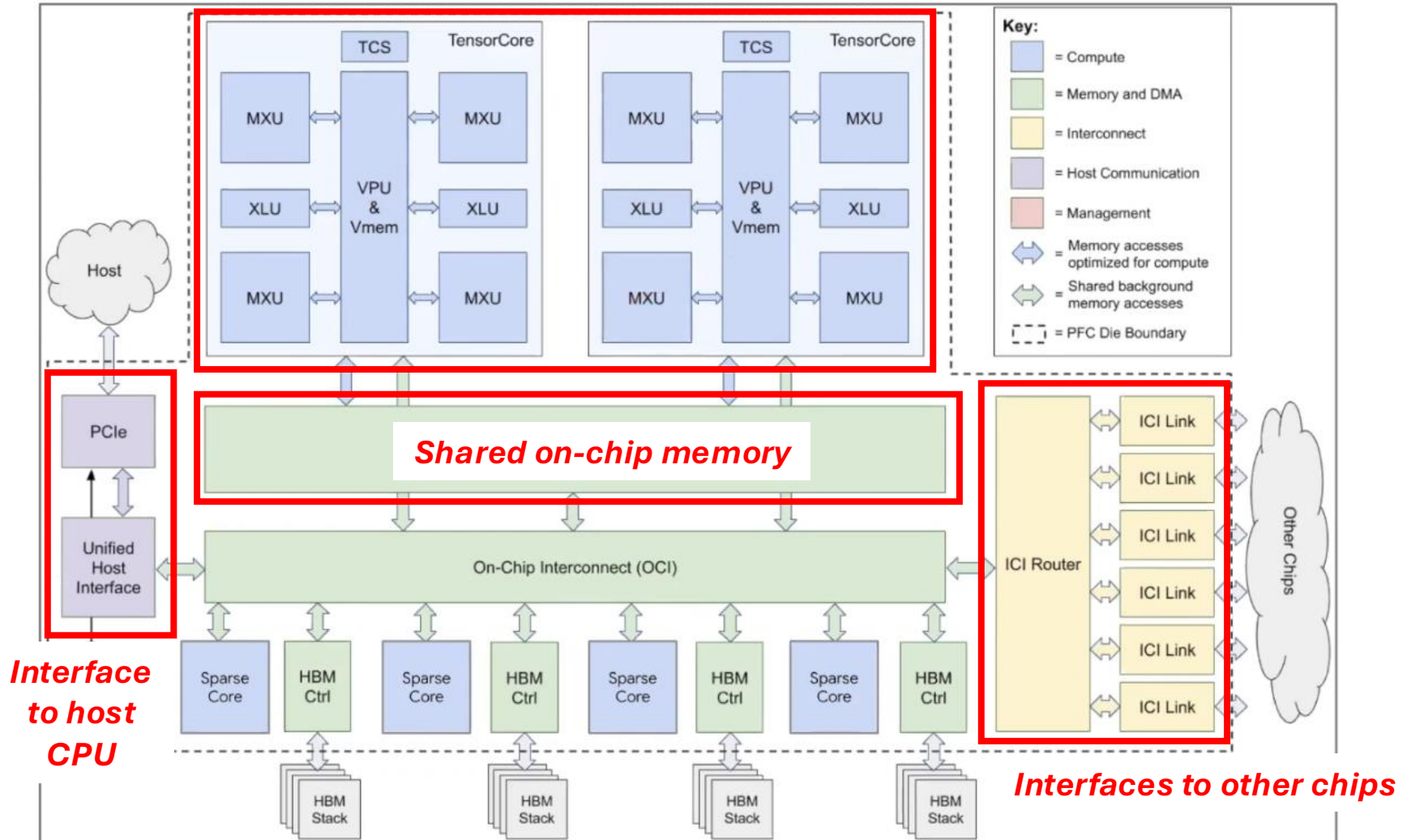
TPUv4 Chip Architecture

*Two cores where the compute happens
(matrix mult, transpose, vector ops)
Each core has four 128x128 systolic
matrix-matrix multipliers (MXUs)*

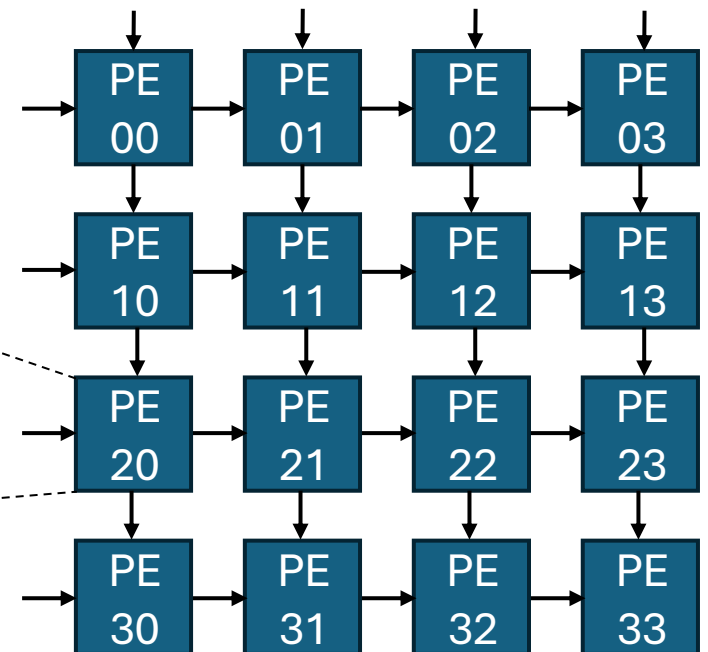
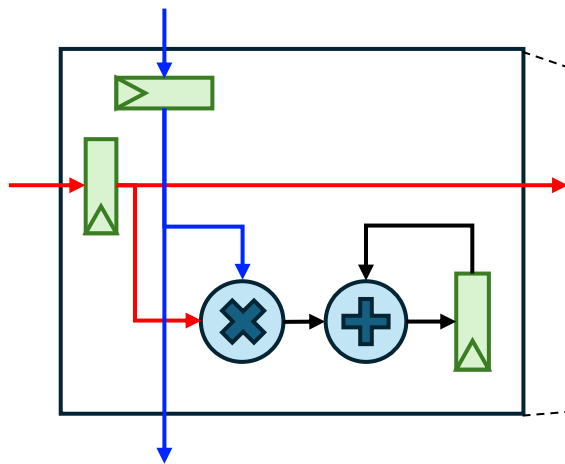


TPUv4 Chip Architecture

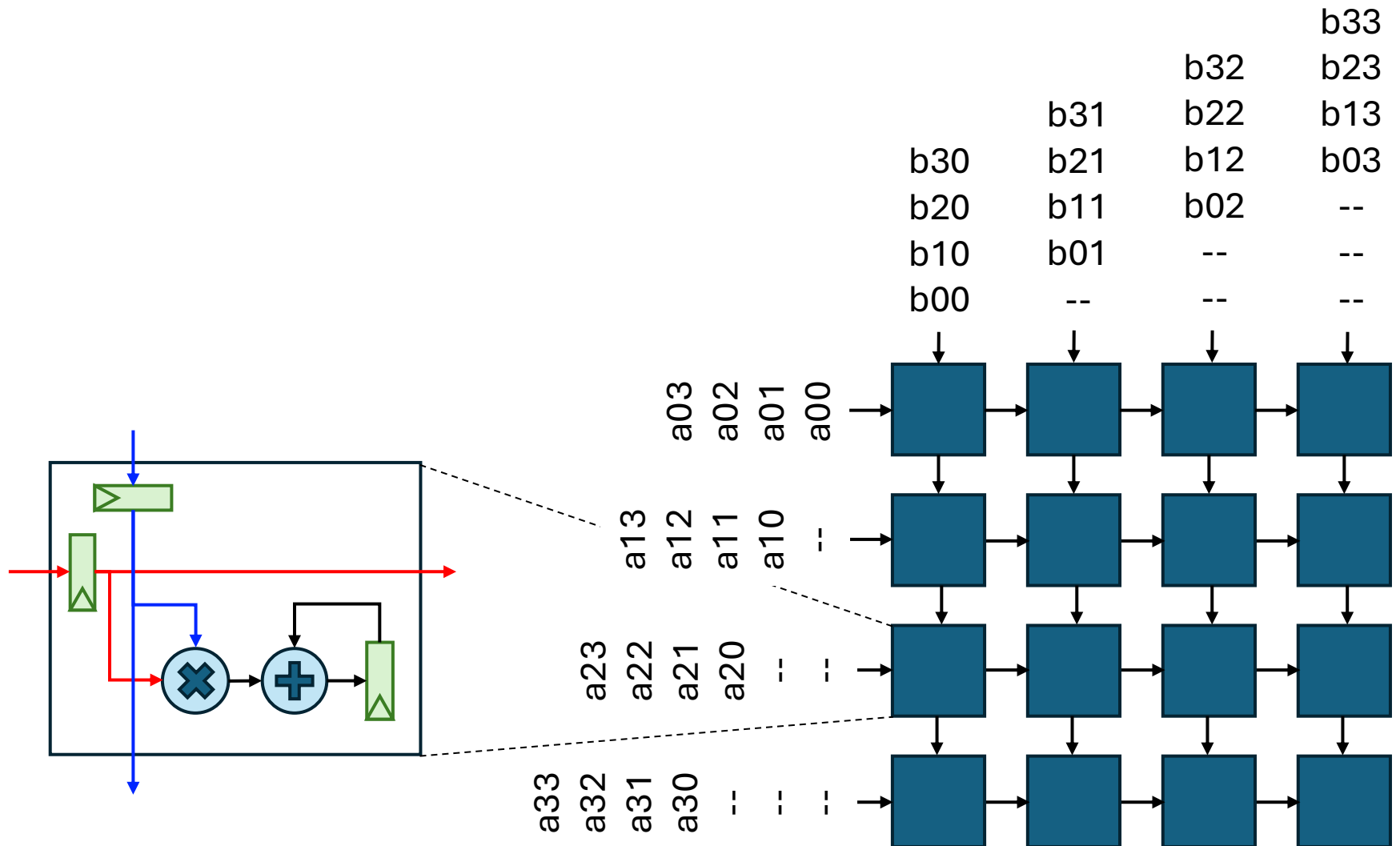
*Two cores where the compute happens
(matrix mult, transpose, vector ops)
Each core has four 128x128 systolic
matrix-matrix multipliers (MXUs)*



4x4 Systolic Matrix Multiplier



4x4 Systolic Matrix Multiplier

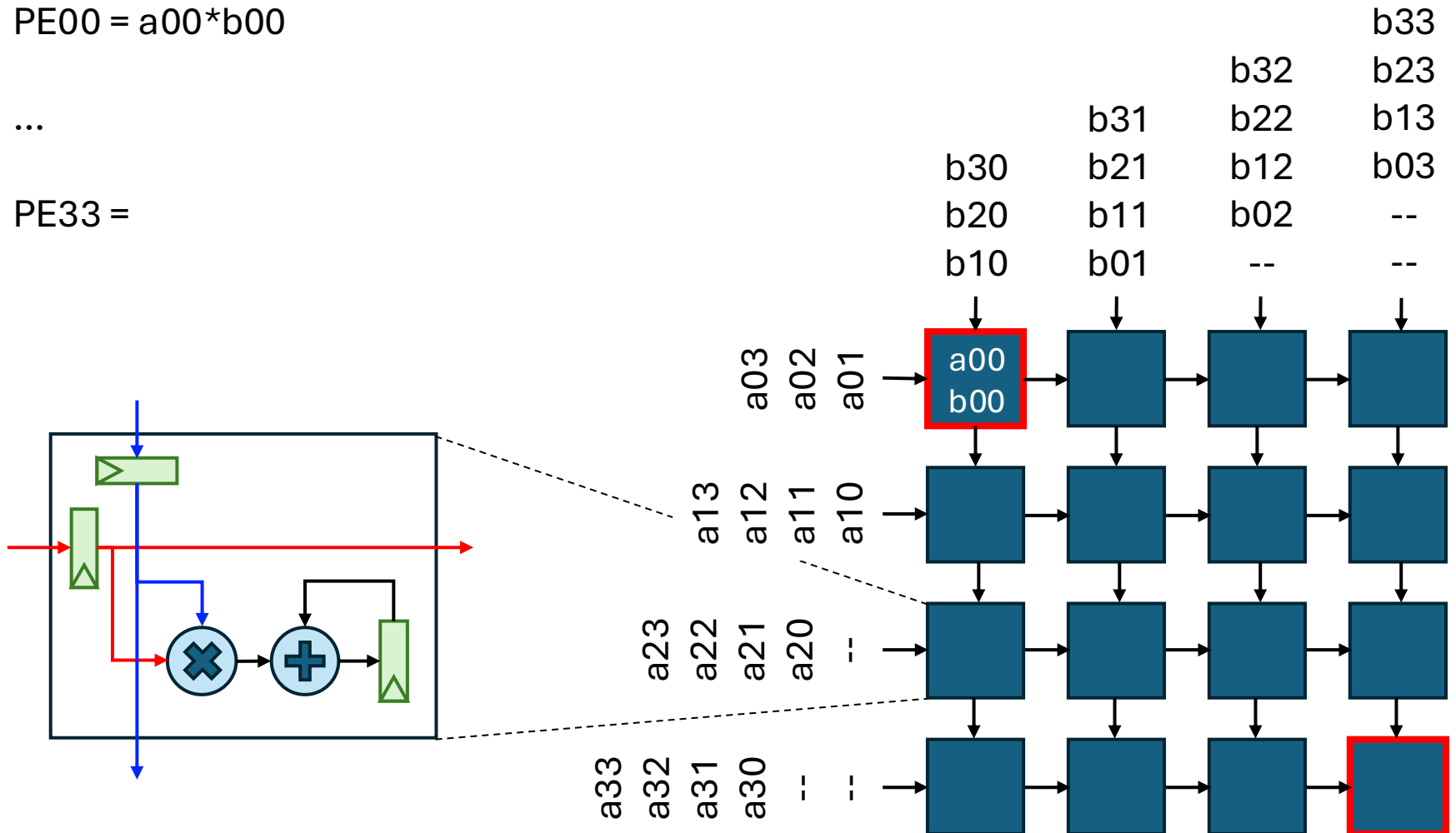


4x4 Systolic Matrix Multiplier

$$PE_{00} = a_{00} * b_{00}$$

...

$$PE_{33} =$$

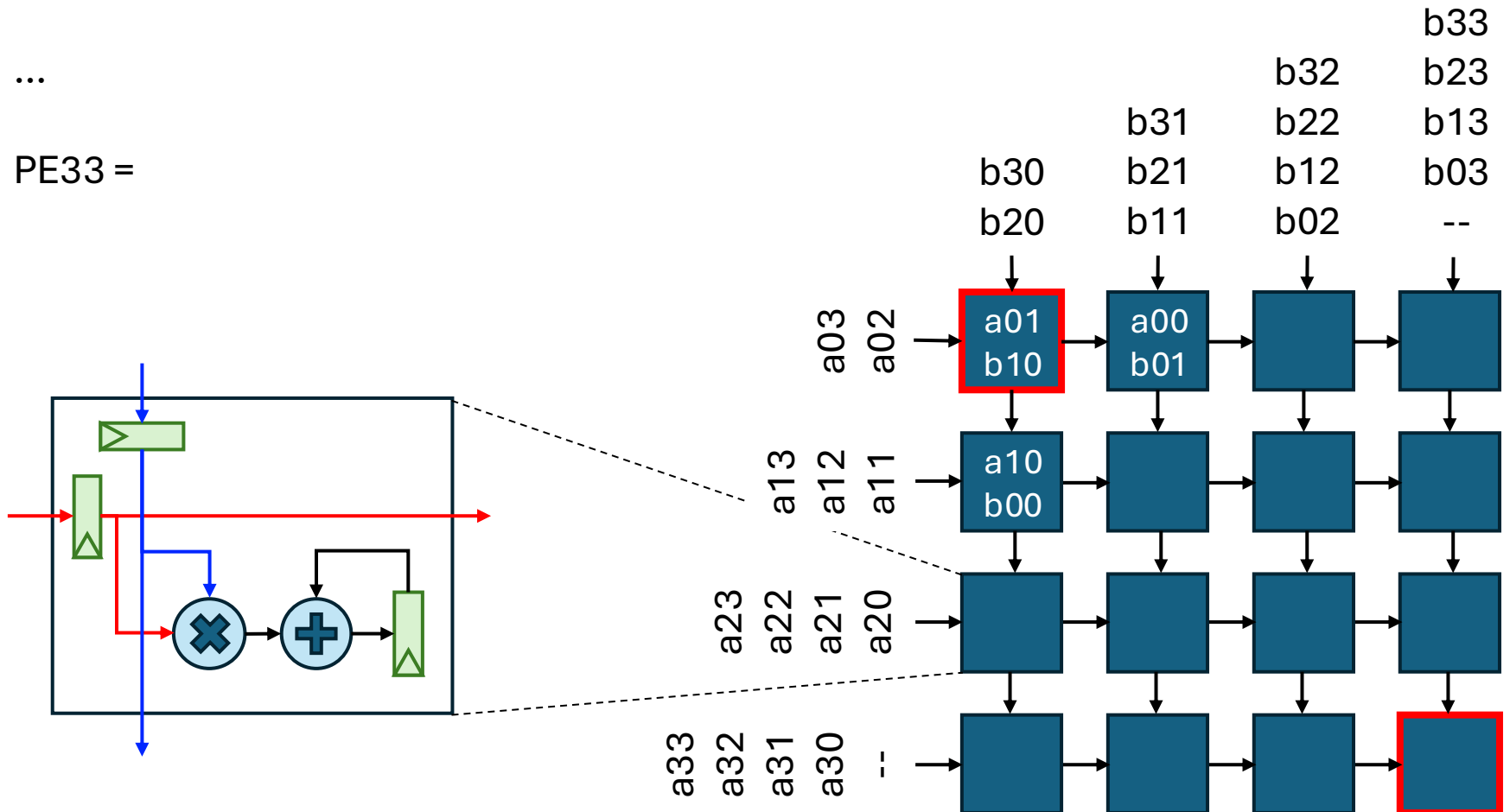


4x4 Systolic Matrix Multiplier

$$PE_{00} = a_{00} * b_{00} + a_{01} * b_{10}$$

...

$$PE_{33} =$$

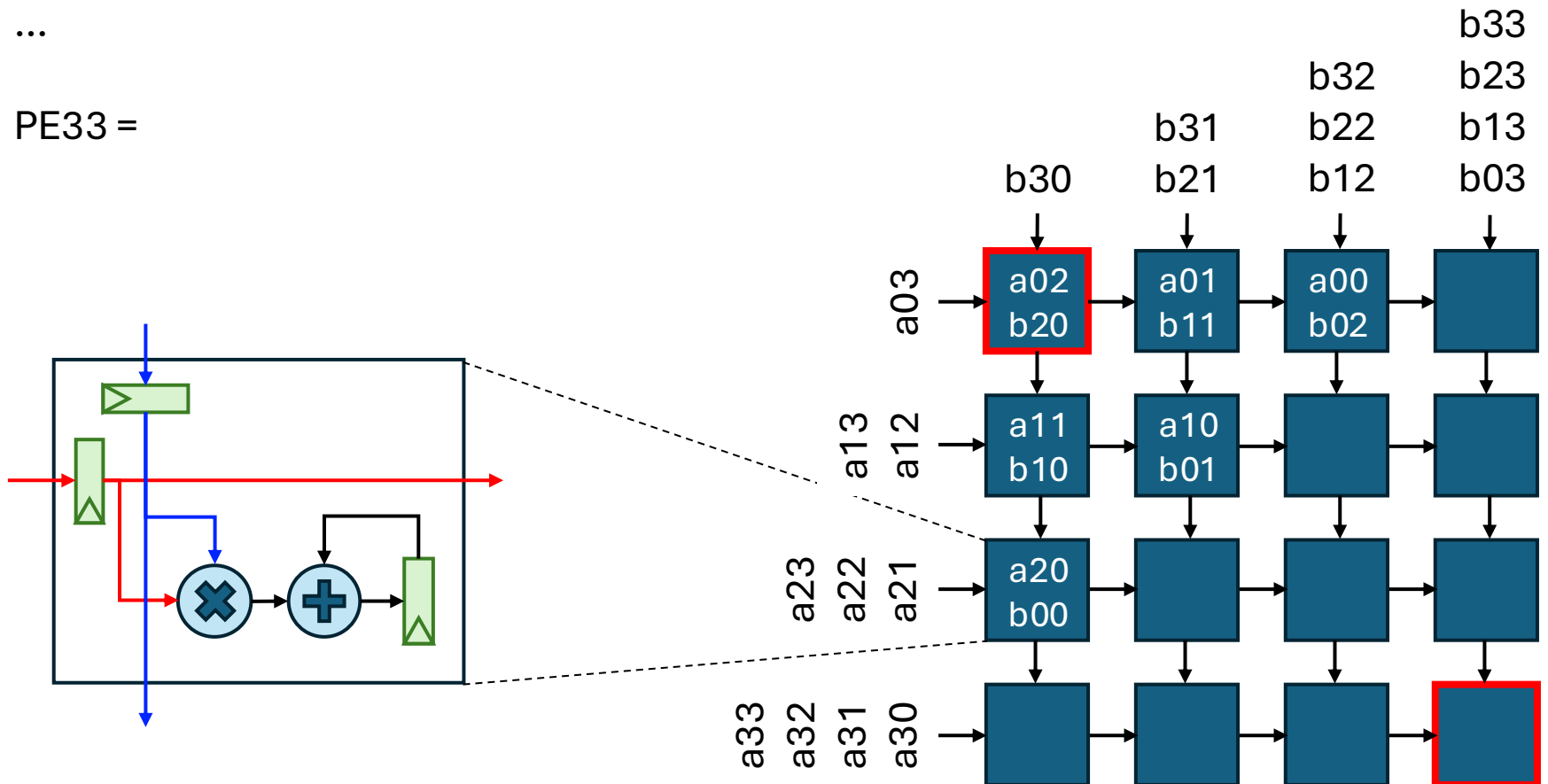


4x4 Systolic Matrix Multiplier

$$PE00 = a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20}$$

...

$$PE33 =$$

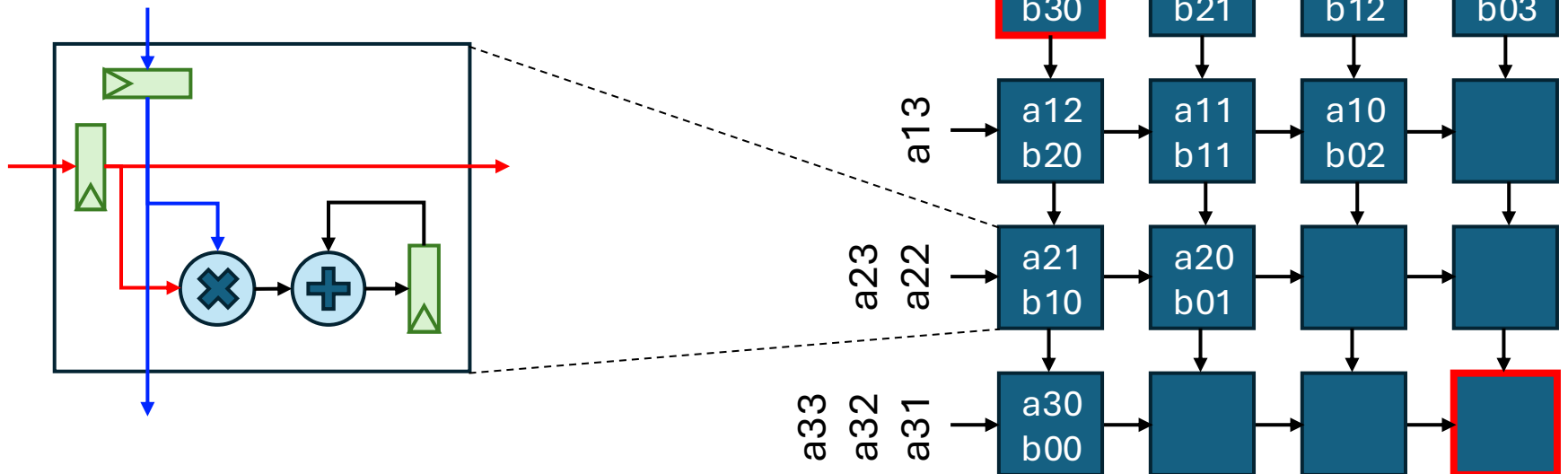


4x4 Systolic Matrix Multiplier

$$PE00 = a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20} + a_{03} \cdot b_{30}$$

...

$$PE33 =$$

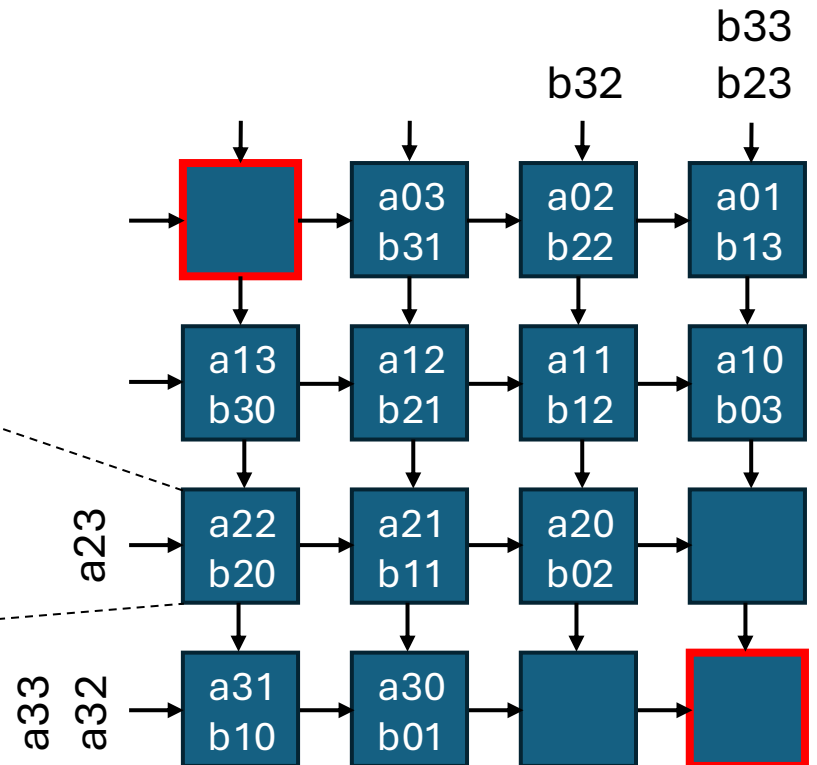
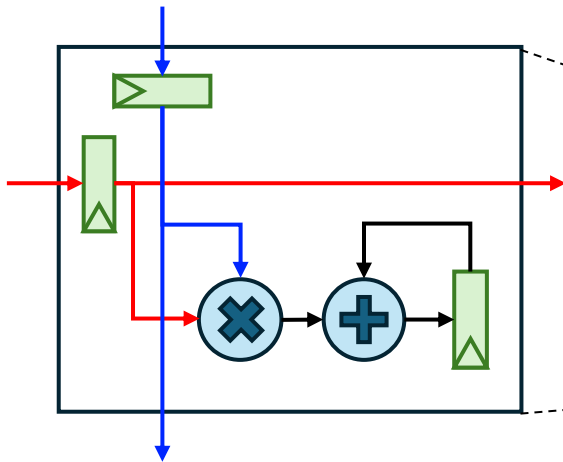


4x4 Systolic Matrix Multiplier

$$PE_{00} = a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20} + a_{03} * b_{30}$$

...

$$PE_{33} =$$

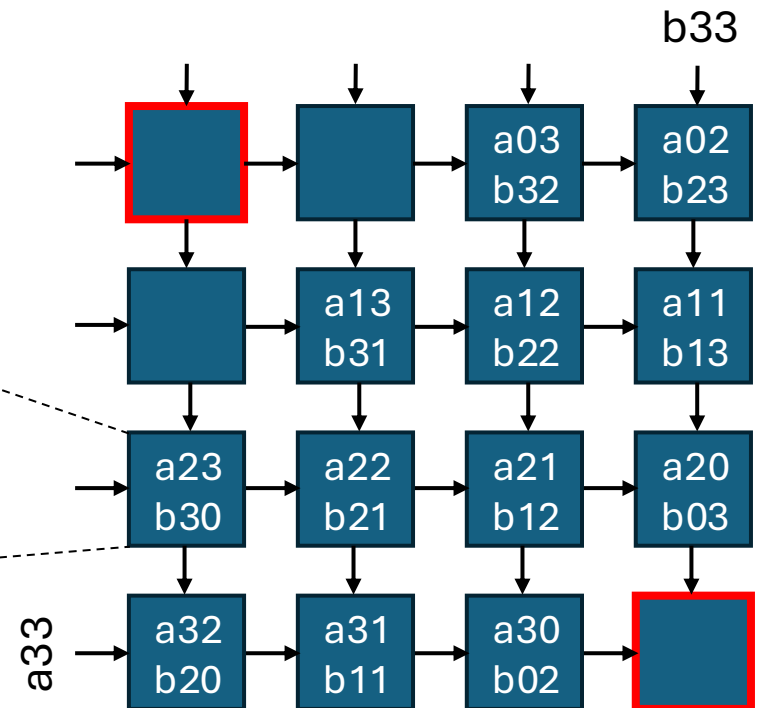
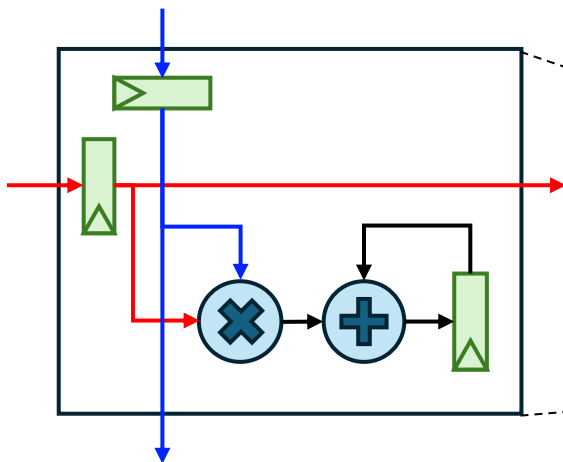


4x4 Systolic Matrix Multiplier

$$PE_{00} = a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20} + a_{03} * b_{30}$$

...

$$PE_{33} =$$

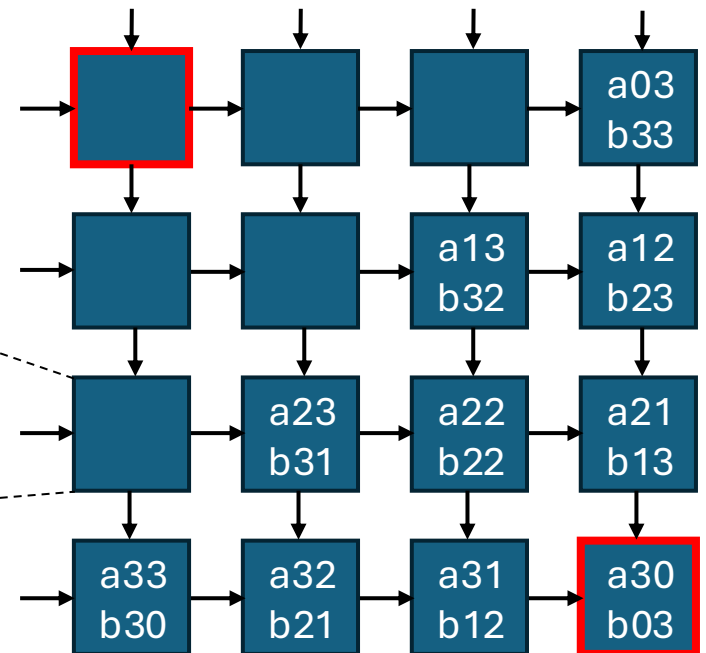
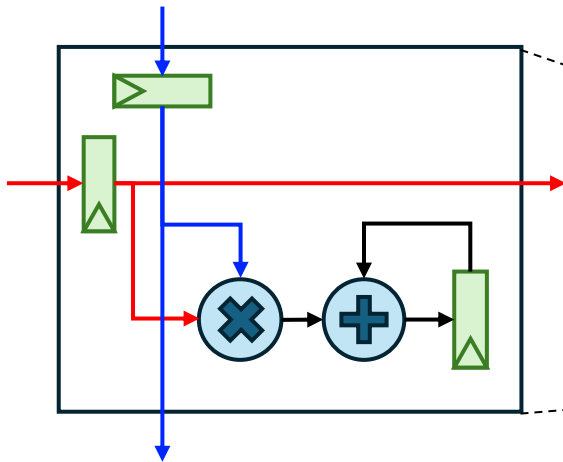


4x4 Systolic Matrix Multiplier

$$PE00 = a00*b00 + a01*b10 + a02*b20 + a03*b30$$

...

$$PE33 = a30*b03$$

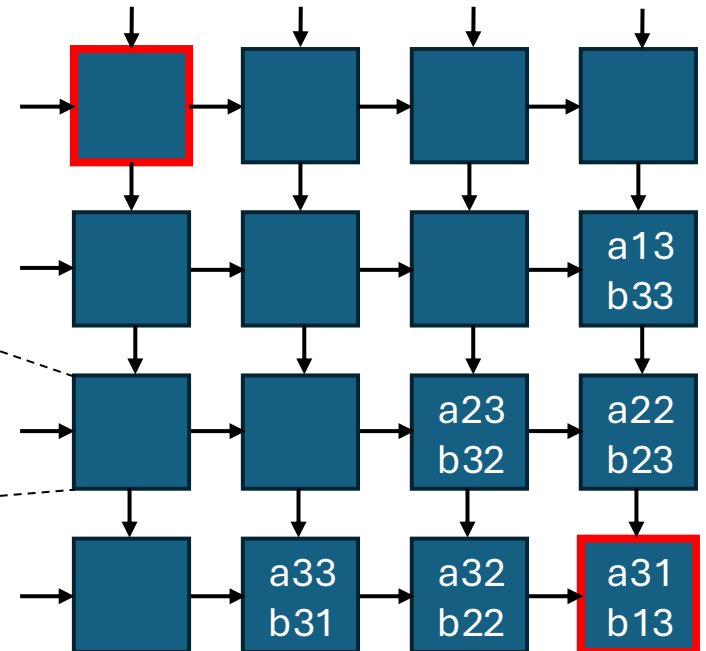
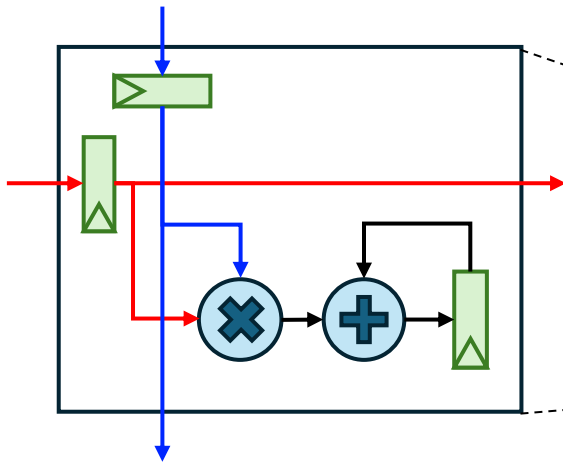


4x4 Systolic Matrix Multiplier

$$PE00 = a00*b00 + a01*b10 + a02*b20 + a03*b30$$

...

$$PE33 = a30*b03 + a31*b13$$

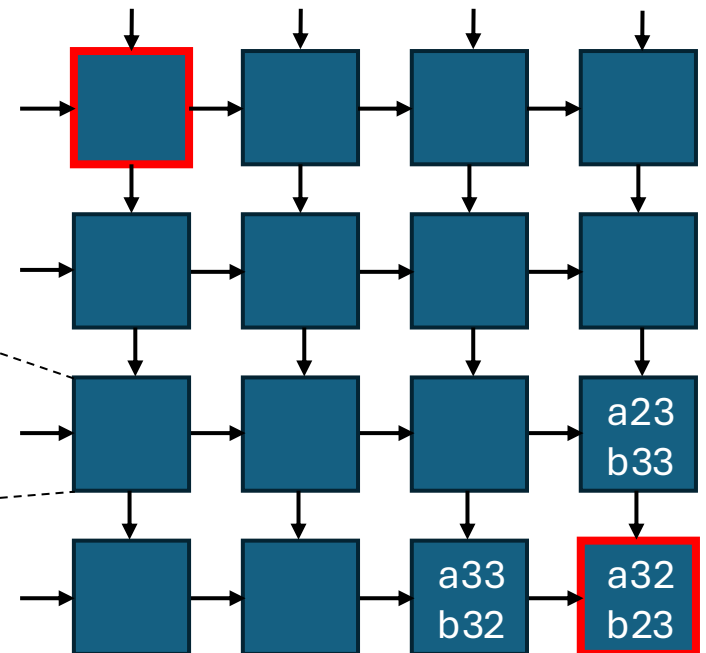
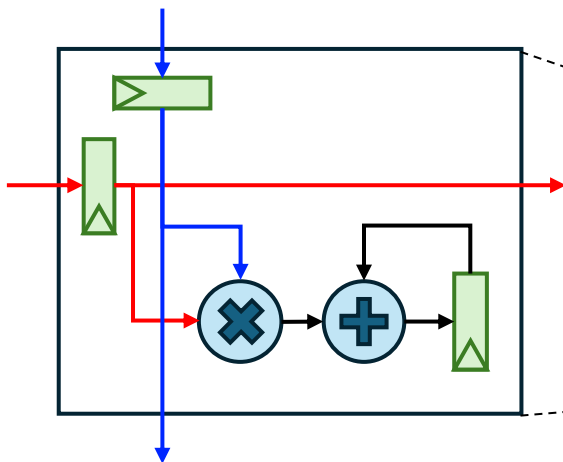


4x4 Systolic Matrix Multiplier

$$PE_{00} = a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20} + a_{03} * b_{30}$$

...

$$PE_{33} = a_{30} * b_{03} + a_{31} * b_{13} + a_{32} * b_{23} + a_{33} * b_{33}$$

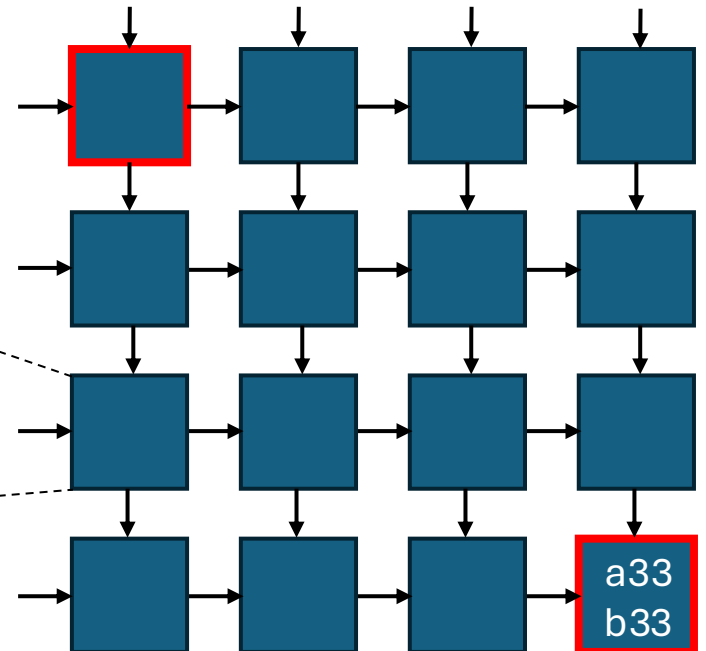
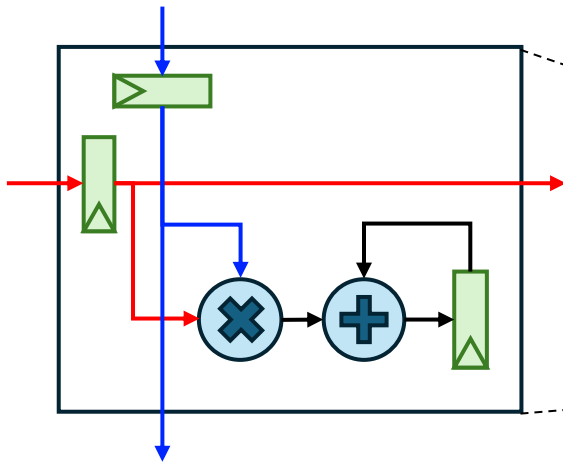


4x4 Systolic Matrix Multiplier

$$PE_{00} = a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20} + a_{03} * b_{30}$$

...

$$PE_{33} = a_{30} * b_{03} + a_{31} * b_{13} + a_{32} * b_{23} + a_{33} * b_{33}$$

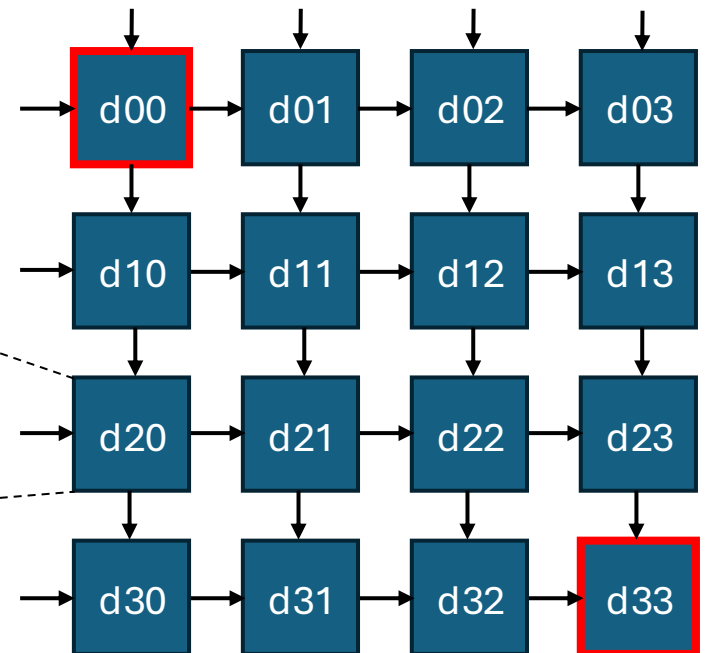
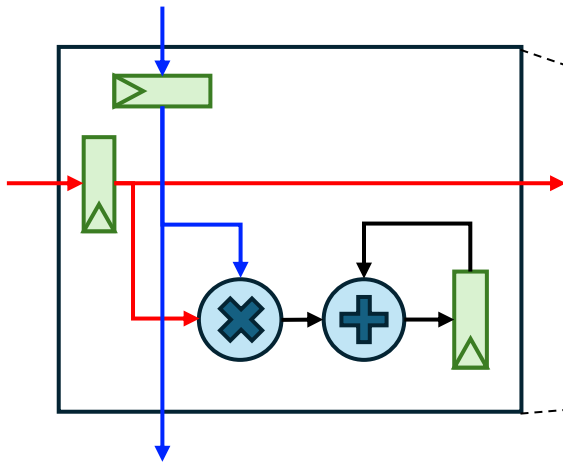


4x4 Systolic Matrix Multiplier

$$PE00 = a00*b00 + a01*b10 + a02*b20 + a03*b30$$

...

$$PE33 = a30*b03 + a31*b13 + a32*b23 + a33*b33$$



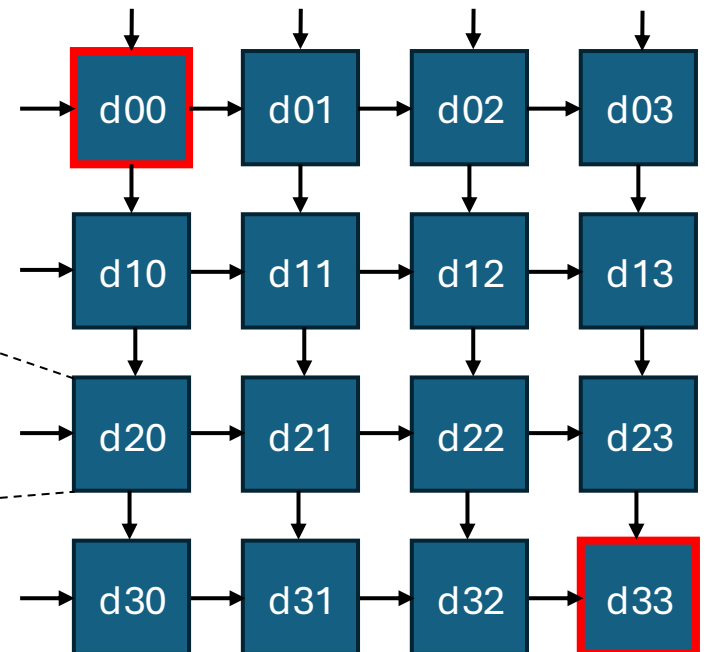
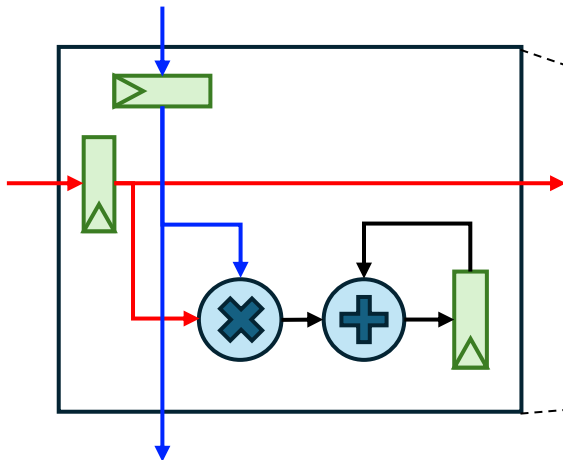
4x4 Systolic Matrix Multiplier

$$PE00 = a00*b00 + a01*b10 + a02*b20 + a03*b30$$

...

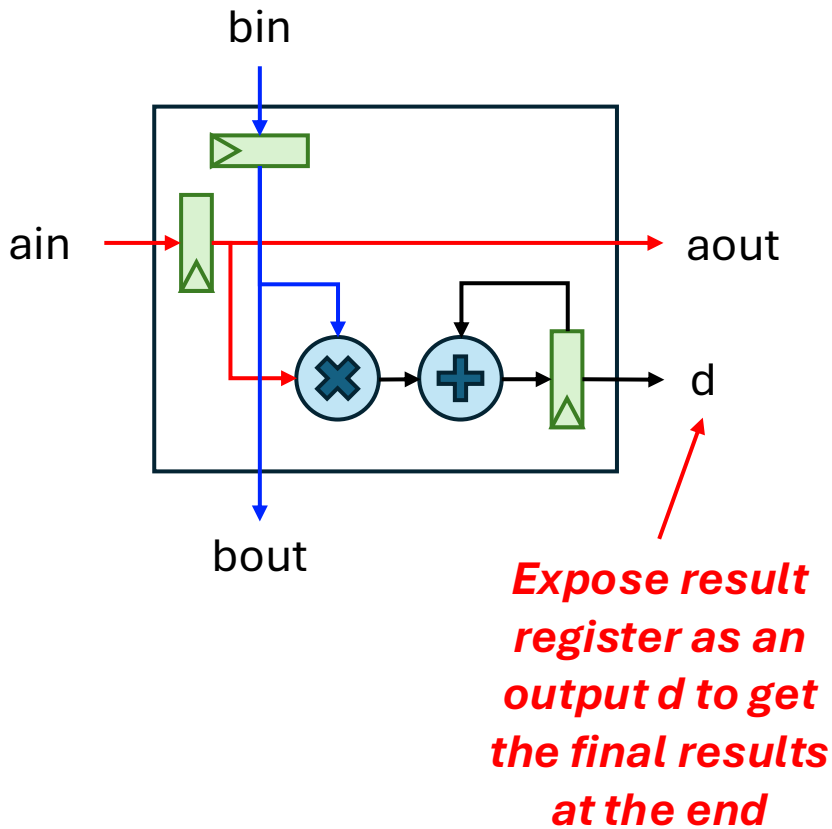
$$PE33 = a30*b03 + a31*b13 + a32*b23 + a33*b33$$

*Google's TPUv4 implements eight
128x128 systolic arrays per chip!*



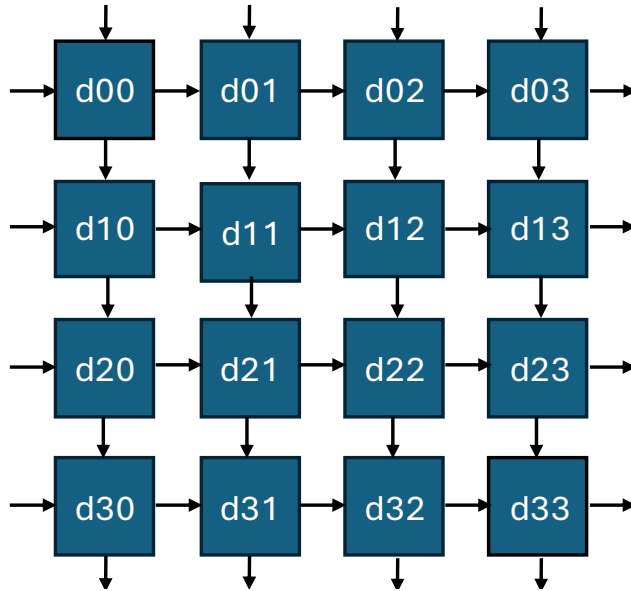
Processing Element (PE)

For simplicity, we will assume input operands are 8b & results are 32b integers



```
module pe # (  
    parameter IDATAW = 8,  
    parameter ODATAW = 32  
)(  
    input  clk,  
    input  rst,  
    input  signed [IDATAW-1:0] ain,  
    input  signed [IDATAW-1:0] bin,  
    output signed [IDATAW-1:0] aout,  
    output signed [IDATAW-1:0] bout,  
    output signed [ODATAW-1:0] d  
);  
  
logic signed [IDATAW-1:0] a_reg, b_reg;  
logic signed [ODATAW-1:0] d_reg;  
  
always_ff @ (posedge clk) begin  
    if (rst) begin  
        a_reg <= 0; b_reg <= 0; d_reg <= 0;  
    end else begin  
        a_reg <= ain; b_reg <= bin;  
        d_reg <= d_reg + (a_reg * b_reg);  
    end  
end  
  
assign aout = a_reg;  
assign bout = b_reg;  
assign d = d_reg;  
  
endmodule
```

Systolic Array

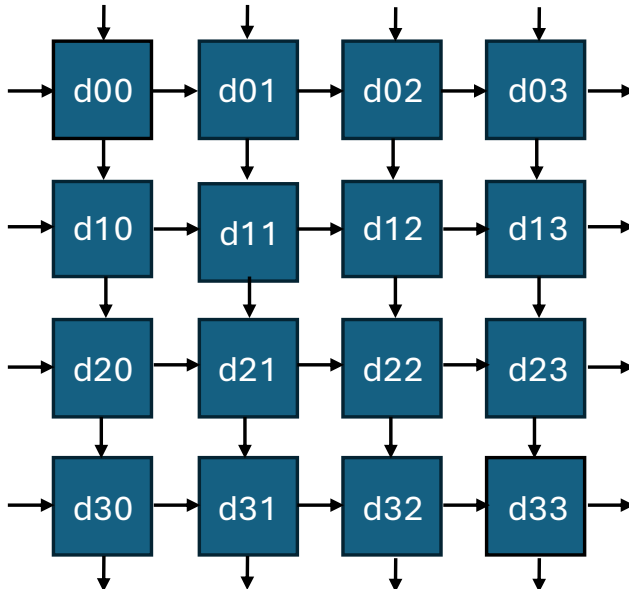


Define the interfaces of the systolic array:

- 4 elements of a matrix (right)
- 4 elements of b matrix (top)
- All 16 results

```
module mxu # (  
    parameter IDATAW = 8,  
    parameter ODATAW = 32,  
    parameter SIZE = 4  
)(  
    input clk,  
    input rst,  
    input signed [IDATAW-1:0] ain [0:SIZE-1],  
    input signed [IDATAW-1:0] bin [0:SIZE-1],  
    output signed [ODATAW-1:0] d [0:SIZE-1] [0:SIZE-1]  
);
```

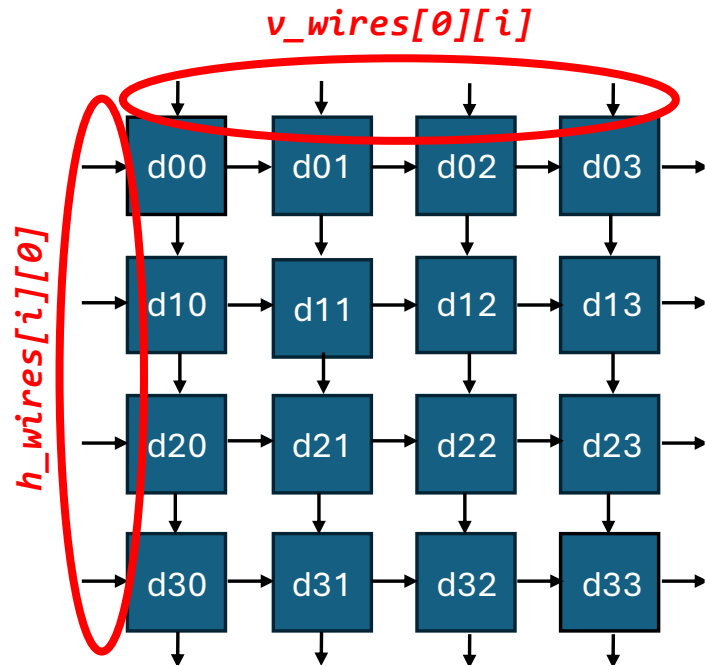
Systolic Array



Declare wires to connect the PEs
of the systolic array

```
module mxu # (  
    parameter IDATAW = 8,  
    parameter ODATAW = 32,  
    parameter SIZE = 4  
)(  
    input clk,  
    input rst,  
    input signed [IDATAW-1:0] ain [0:SIZE-1],  
    input signed [IDATAW-1:0] bin [0:SIZE-1],  
    output signed [ODATAW-1:0] d [0:SIZE-1] [0:SIZE-1]  
);  
  
logic signed [IDATAW-1:0] h_wires [0:SIZE-1][0:SIZE];  
logic signed [IDATAW-1:0] v_wires [0:SIZE][0:SIZE-1];
```

Systolic Array



Tie the first column of horizontal wires to a inputs and first row of vertical wires to b inputs

```

module mxu # (
    parameter IDATAW = 8,
    parameter ODATAW = 32,
    parameter SIZE = 4
)(
    input clk,
    input rst,
    input signed [IDATAW-1:0] ain [0:SIZE-1],
    input signed [IDATAW-1:0] bin [0:SIZE-1],
    output signed [ODATAW-1:0] d [0:SIZE-1] [0:SIZE-1]
);

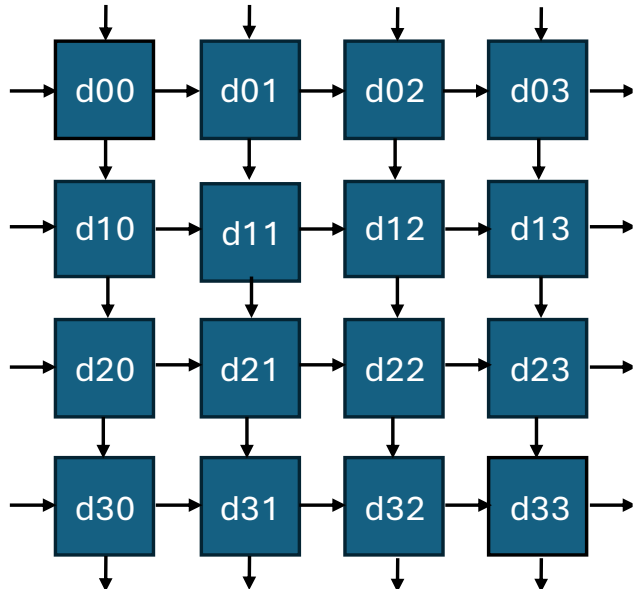
    logic signed [IDATAW-1:0] h_wires [0:SIZE-1][0:SIZE];
    logic signed [IDATAW-1:0] v_wires [0:SIZE][0:SIZE-1];

    integer i;
    always_comb begin
        for (i=0; i<SIZE; i=i+1) begin
            h_wires[i][0] = ain[i];
            v_wires[0][i] = bin[i];
        end
    end

end

...
    
```

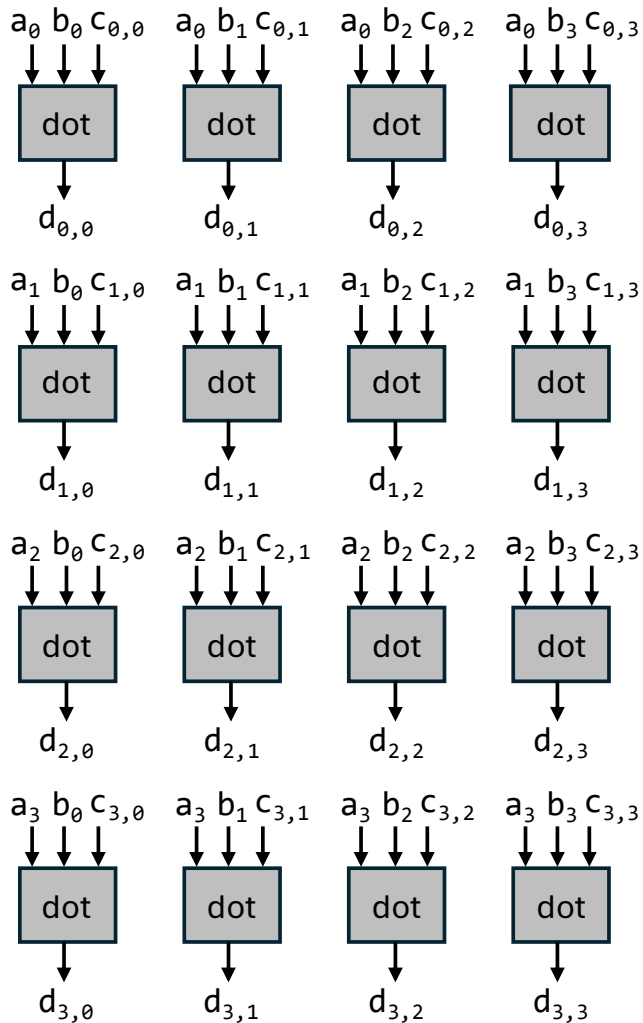
Systolic Array



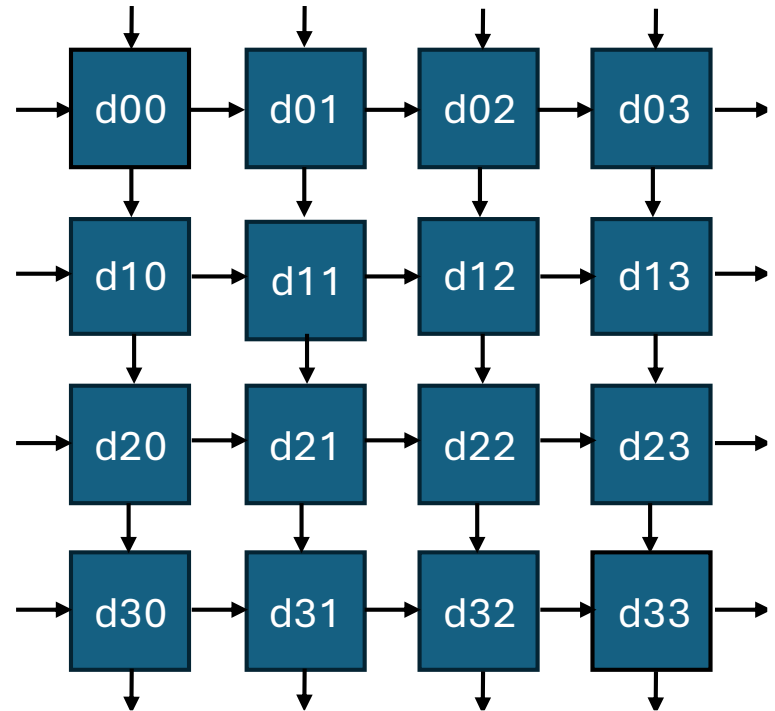
Instantiate the PEs using a generate for loop and connect them in a grid

```
...
genvar j, k;
generate
for (j=0; j<SIZE; j=j+1) begin: gen_rows
    for (k=0; k<SIZE; k=k+1) begin: gen_cols
        pe # (
            .IDATAW(IDATAW), .ODATAW(ODATAW)
        ) pe_inst (
            .clk(clk),
            .rst(rst),
            .ain(h_wires[j][k]),
            .bin(v_wires[j][k]),
            .aout(h_wires[j][k+1]),
            .bout(v_wires[j+1][k]),
            .d(d[j][k])
        );
    end
end
endgenerate
endmodule
```

What are the trade-offs?



vs.



Next Lecture

We will learn about control finite state machines (FSMs) and how to implement them in SystemVerilog