# Synthesis Examples (A top-down approach)

**Nachiket Kapre**
nachiket@uwaterloo.ca

# Outline

- NVIDIA's Tensor Cores
- Google's Tensor Processing Unit

# NVIDIA Tensor Cores

- Machine learning and AI workloads are rich in matrix arithmetic
- NVIDIA added a custom hardware unit to their GPUs called **Tensor Core**
- Each core can perform a $4 \times 4$ matrix-matrix multiplication on floating-point numbers along with a matrix-matrix addition

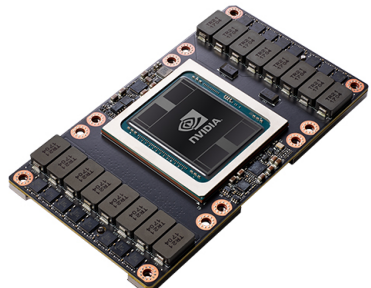



Figure: NVIDIA Volta V100 GPU!

https://www.nvidia.com/en-us/data-center/tesla-v100/

# Matrix-Matrix Multiplication and Matrix Addition

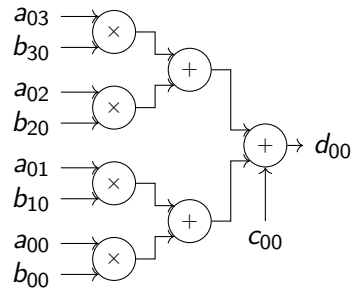
$$
\underset{D}{\begin{pmatrix} d_{00} & d_{01} & d_{02} & d_{03} \\ d_{10} & d_{11} & d_{12} & d_{13} \\ d_{20} & d_{21} & d_{22} & d_{23} \\ d_{30} & d_{31} & d_{32} & d_{33} \end{pmatrix}} = \underset{A}{\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}} \times \underset{B}{\begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix}} + \underset{C}{\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix}}
$$

- Formula: $D = A \cdot B + C$
- 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];
    }
  }
}
```

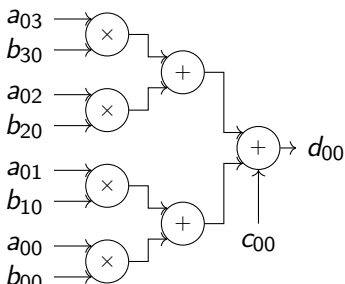
- Example $d_{00} = c_{00} + \sum_{k=0}^{k=3}(a_{0k} \cdot b_{k0})$
  - $d_{00} = c_{00} + a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20} + a_{03} \cdot b_{30}$

# Parallel Computation of $d_{00}$



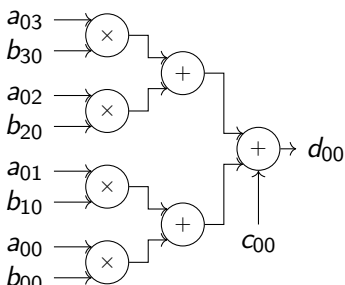

- $d_{00} = c_{00} + a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20} + a_{03} \cdot b_{30}$
  - Each product $a_{ik} \cdot b_{kj}$ can be computed in parallel
  - The sums can be added in parallel $\rightarrow$ associative and commutative properties of integers!
    - a+b=b+a, a+b+c = (a+b) + c = (a+c) + b = (b+c) +a
- NVIDIA GPU operates using floating-point numbers $\rightarrow$ associativity is **not** valid for floating-point numbers
  - NVIDIA thinks its fine/optional $\rightarrow$ ML/AI computations are probably OK with this relaxation of requirement

# RTL Code for Dot Product block



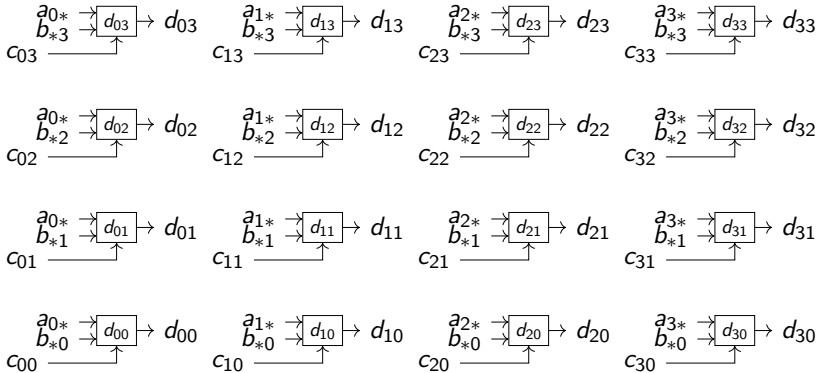

```
module dot_prod
    (input shortreal a0,
     input shortreal a1,
     input shortreal a2,
     input shortreal a3,
     input shortreal b0,
     input shortreal b1,
     input shortreal b2,
     input shortreal b3,
     input real c,
     output real d
     );
```

# RTL Code for Dot Product block



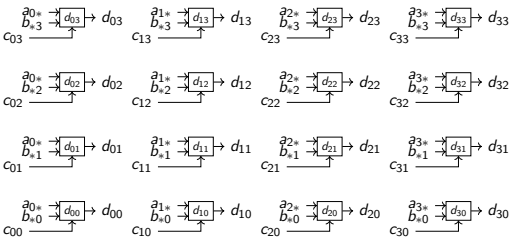

```
shortreal d1;
shortreal d2;

assign d1 = a0*b0 + a1*b1;
assign d2 = a2*b2 + a3*b3;
assign d  = (d1 + d2) + c;
```

# Parallel Computation of $D$



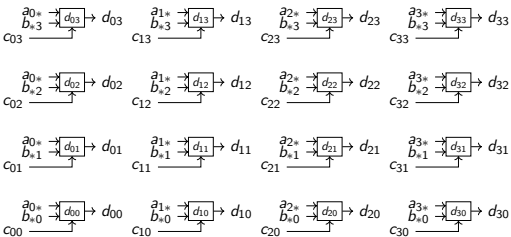

- Each $d_{ij}$ can be computed completely independent!
  - No loop dependency in the $i, j$ loops
- Final tally
  - 64 multipliers $\rightarrow$ FP16 floating-point format (16-bit)
  - 48 adders $\rightarrow$ FP32 floating-point format (32-bit)
  - $C$ and $D$ can either be FP16 or FP32

# RTL Code for Tensor Core




```
module tensor_core (
 input shortreal a [0:3][0:3],
 input shortreal b [0:3][0:3],
 input real c [0:3][0:3],
 output real d [0:3][0:3]
);
```

# RTL Code for Tensor Core




```
genvar i,j;

generate
for (i=0; i<=3; i=i+1) begin : row
  for (j=0; j<=3; j=j+1) begin : col
    dot_prod
    dot_prod_inst (
      .a0 (a[i][0])
      ,.a1 (a[i][1])
      ,.a2 (a[i][2])
      ,.a3 (a[i][3])
      ,.b0 (b[0][j])
      ,.b1 (b[1][j])
      ,.b2 (b[2][j])
      ,.b3 (b[3][j])
      ,.c  (c[i][j])
      ,.d  (d[i][j]));
  end
end
endgenerate
```

# Testbench Design

```
for (i=0; i<=3; i=i+1) begin : row_i
 for (j=0; j<=3; j=j+1) begin : col_i
  a[i][j] <= 1.0;
  b[i][j] <= 1.0;
  c[i][j] <= 1.0;
 end
end

$write("d=[");
for (i=0; i<=3; i=i+1) begin : row_d
 $write("\n");
 for (j=0; j<=3; j=j+1) begin : col_d
  $write("%f ",d[i][j]);
 end
end
```

- The tb **initial** contains loops that assign test inputs to a, b andc matrices.
- After a 1 ns wait, we print out the resulting matrix d.
- Notice the **for** loop constructs here are different from those used to **generate** the dot product units!
- This code will simulate, but not synthesize as **real** types are not synthesis-compatible as of now.

# Google Tensor Processing Unit

- Google getting into hardware design. Three spins of TPU already!
- TPU is a specialized chip for AI $\rightarrow$ matrix-matrix/tensor operations
- **Key Idea**: Systolic design
  - Lots of simple parallel computational units
  - Data movement strictly localized to immediate neighbours
- Tightly coupled with TensorFlow (programming APIs)
- \$1–3 USD per TPU per hour (as of today!)

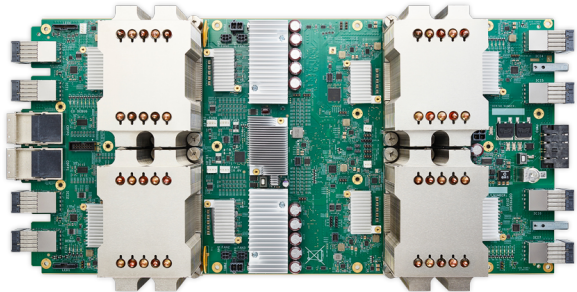


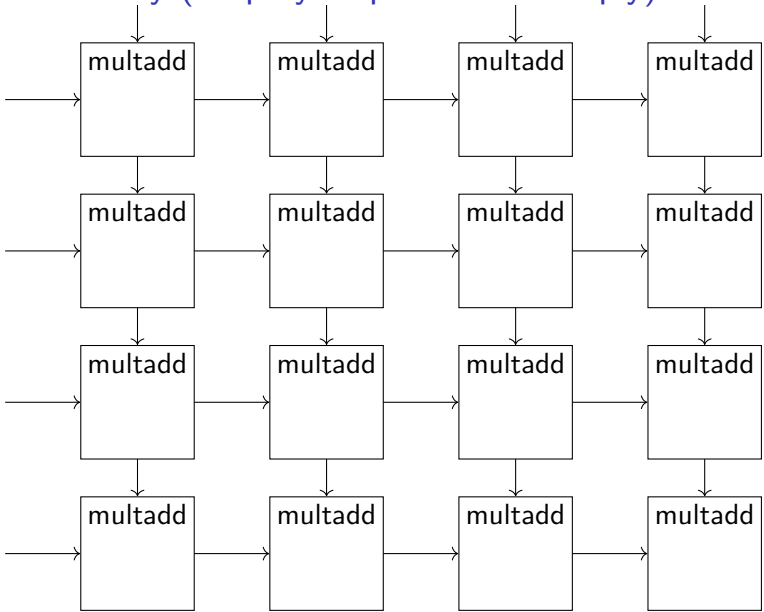
Figure: TPU v3 device → liquid cooled!

https://cloud.google.com/tpu/

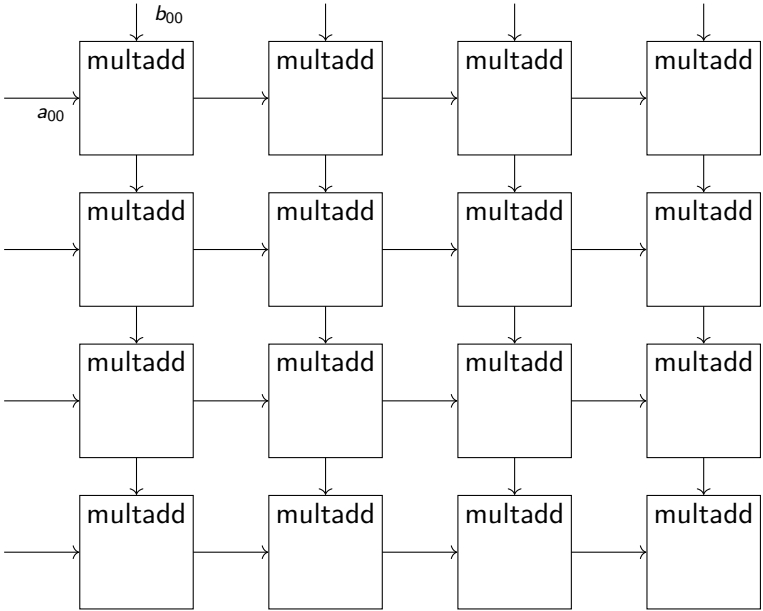
# TPU History

- First generation
  - 64K 8-bit multiply-add operations @700 MHz $\rightarrow$ 44 Tops/s
- Second generation
  - 32K 32-bit (and single-precision float) multiply-add operations @700 MHz $\rightarrow$ 45 TFLOPS (float)
- Third generation
  - Not a lot of details $\rightarrow 2\times$ better than TPU v2
- Fourth generation
  - $\rightarrow 2\times$ better than TPU v3 + Optical switching

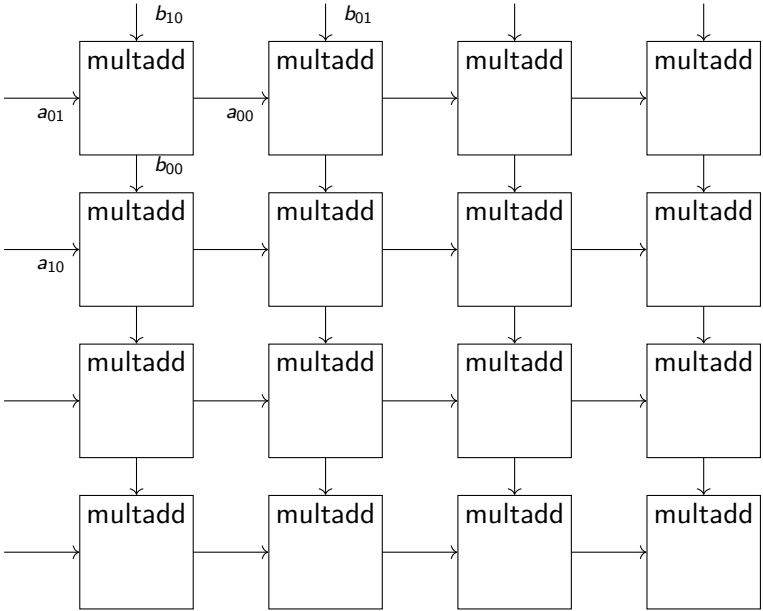
# Systolic Array (Step-by-step Matrix Multiply)



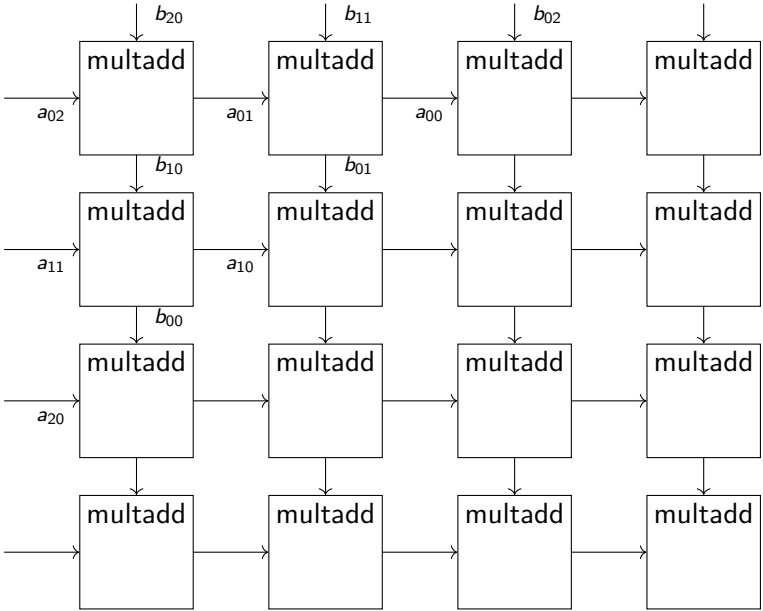
# Systolic Array (Step-by-step Matrix Multiply)



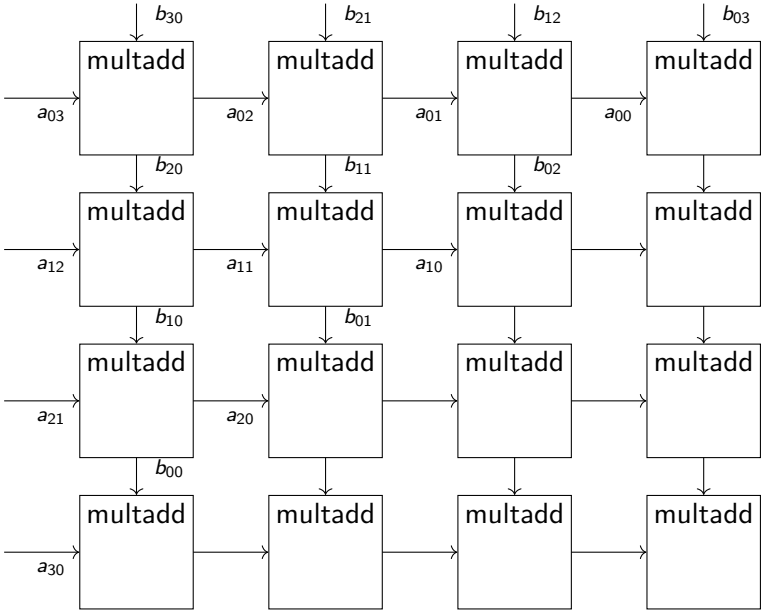
# Systolic Array (Step-by-step Matrix Multiply)



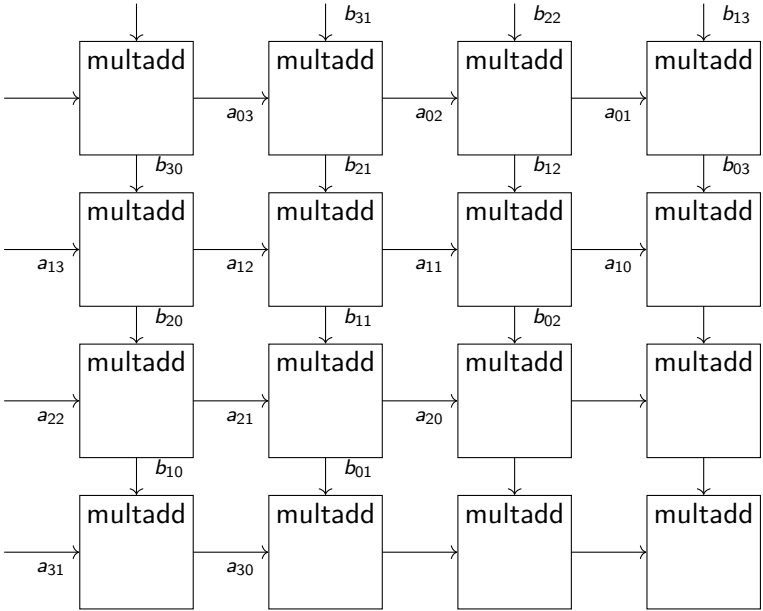
# Systolic Array (Step-by-step Matrix Multiply)



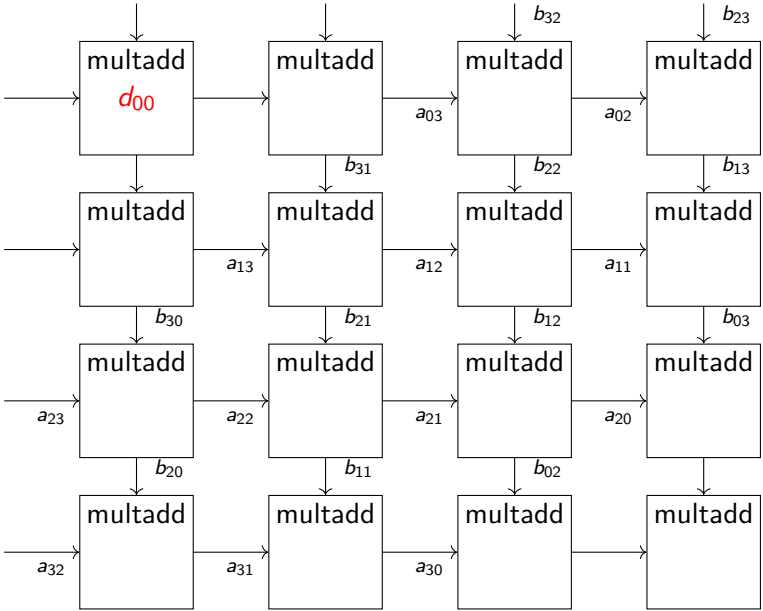
# Systolic Array (Step-by-step Matrix Multiply)



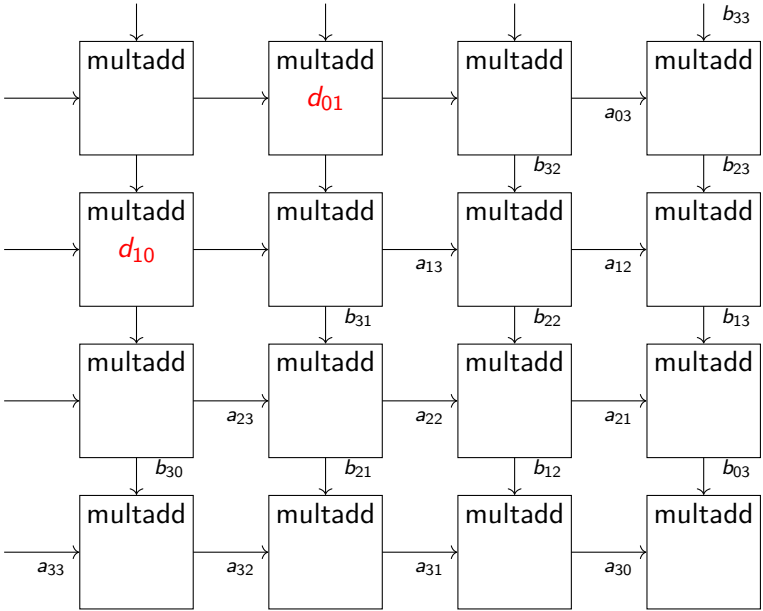
# Systolic Array (Step-by-step Matrix Multiply)



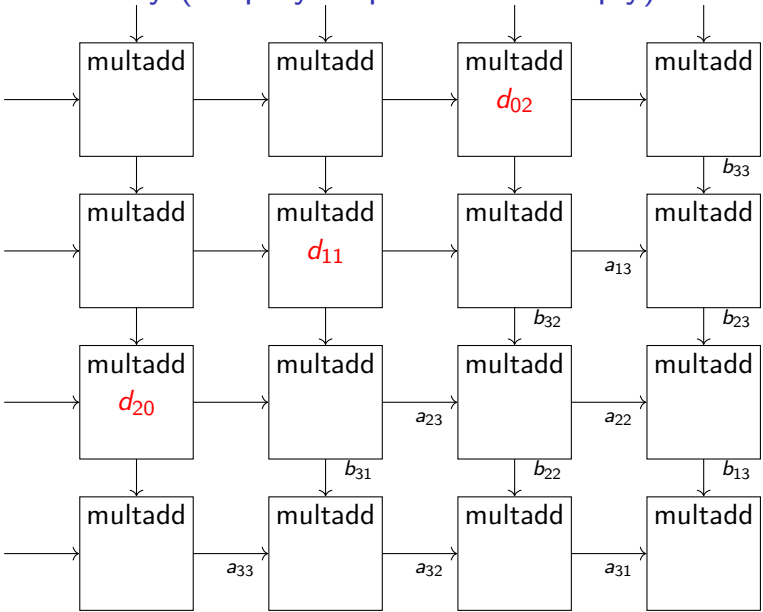
# Systolic Array (Step-by-step Matrix Multiply)



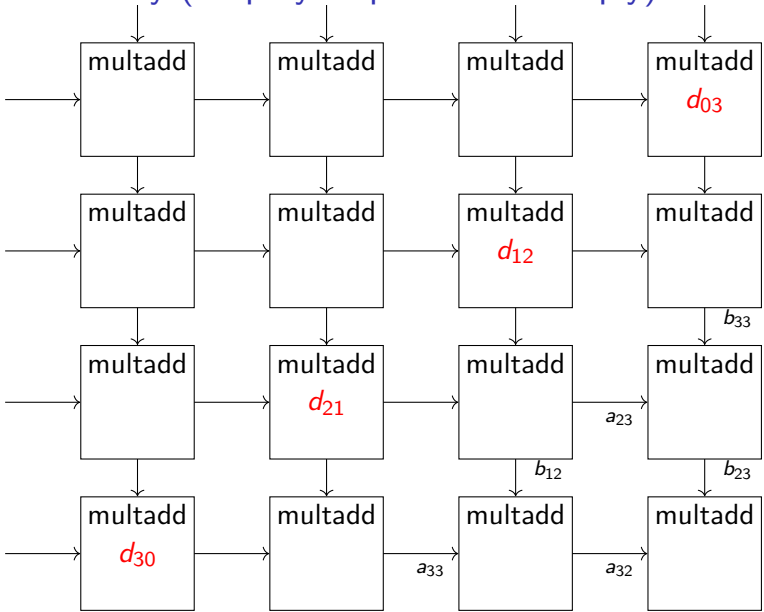
# Systolic Array (Step-by-step Matrix Multiply)



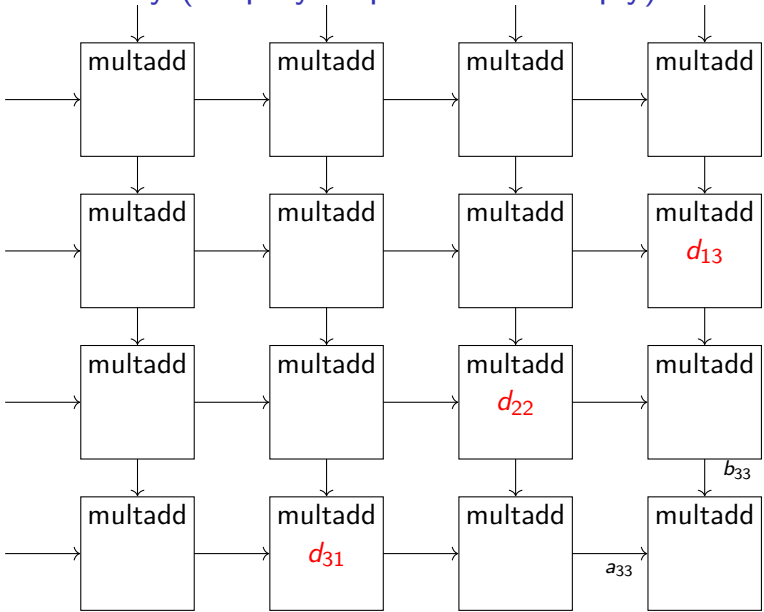
# Systolic Array (Step-by-step Matrix Multiply)



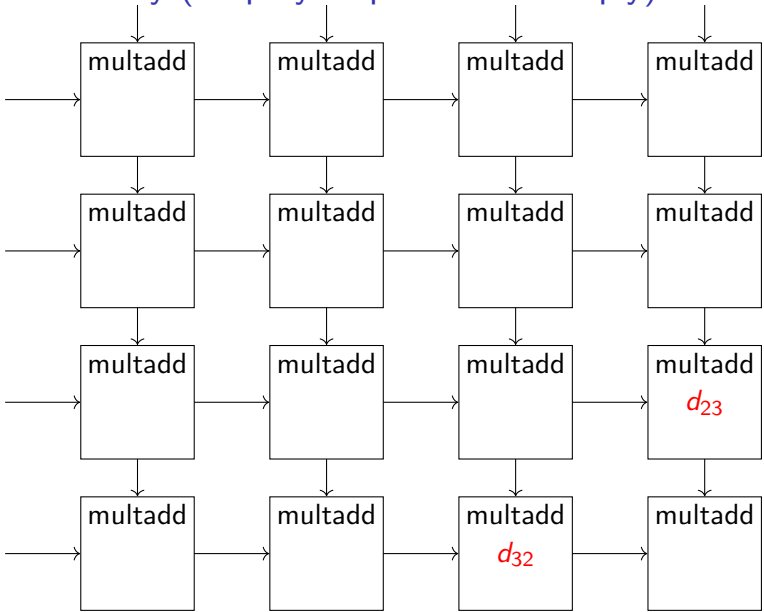
# Systolic Array (Step-by-step Matrix Multiply)



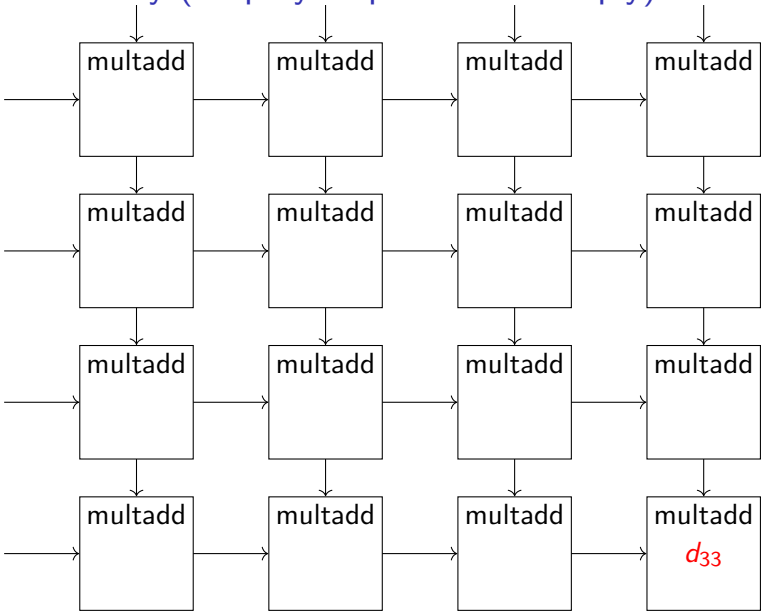
# Systolic Array (Step-by-step Matrix Multiply)



# Systolic Array (Step-by-step Matrix Multiply)



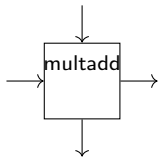
# Systolic Array (Step-by-step Matrix Multiply)

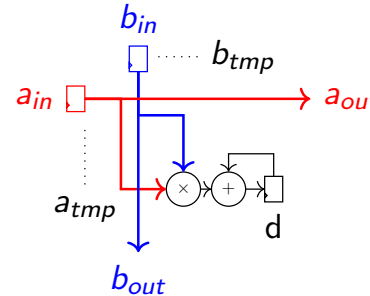


# Matrix-Matrix Multiplication on Systolic Array

- Send $B$ matrix along column, and $A$ matrix along rows of a 2D array
- Each step along the way, the compute block performs a small multiply-add operation
- Data movement is co-ordinated to ensure correct data arrives at correct compute block in the correct time!
- All data movement is local $\rightarrow$ hardware design is super easy!
- In the end, final product $D$ is available in every compute block $\rightarrow$ result must be read out (not shown)

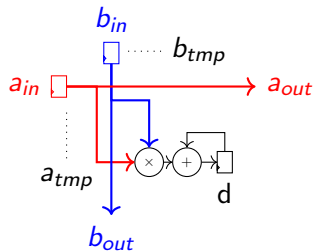
# The multadd Building Block




- The multadd block is a simple 8b×8b multiplier followed by a 32b accumulator
- All communication in the chip is strictly local → nearest neighbour → no long wires!
  - Matrix elements *streamed* into systolic core from top+left links
  - Data forwarded to links below+right for further use by neighouring elements
- The building block in the TPUv1 was just 8b multiplication of neuron weights, and input → later revisions added floating-point and custom formats
- Design is so easy, even Google engineers can build this :)

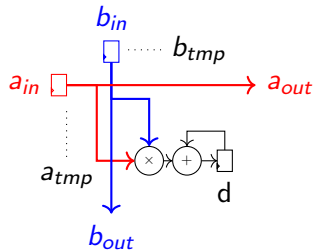
# RTL Description of the multadd block




*TPUv1 has 8b multipliers, 32b accumulators*

```
module systolic_leaf
    (input wire clk,
     input wire rst,
     input wire signed [7:0] a_in,
     input wire signed [7:0] b_in,
     output wire signed [7:0] a_out,
     output wire signed [7:0] b_out,
     output wire signed [31:0] d_out
     );
```

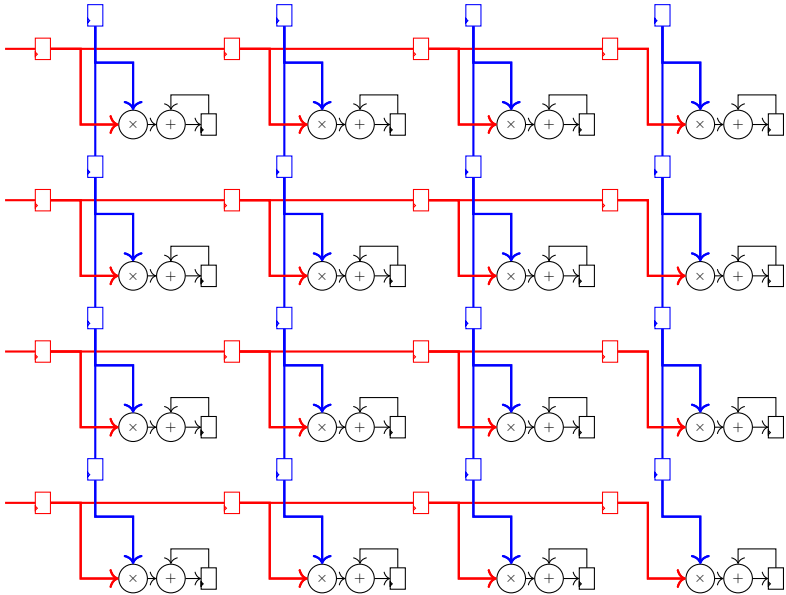
# RTL Description of the multadd block




*TPUv1 has 8b multipliers, 32b accumulators*

```
reg signed [31:0] d;
reg signed [7:0] a_tmp;
reg signed [7:0] b_tmp;

always@(posedge clk) begin
  if (rst) begin
    d     <= 0;
    a_tmp <= 0;
    b_tmp <= 0;
  end else begin
    d <= d + a_tmp*b_tmp;
    a_tmp <= a_in;
    b_tmp <= b_in;
  end
end

assign a_out = a_tmp;
assign b_out = b_tmp;
assign d_out = d;
```

# Systolic Array Design (4×4 design)



# RTL Description of the Systolic Array

```
module systolic_complete
  # (parameter SIZE = 4)
    (input wire clk,
     input wire rst,
     input wire signed [8*SIZE-1:0] a,
     input wire signed [8*SIZE-1:0] b,
     output wire signed [32*SIZE*SIZE-1:0] d
     );
```

# RTL Description of the Systolic Array

```
genvar i,j;

wire signed [8*SIZE*(SIZE+1)-1 : 0] horizontal_wires;
wire signed [8*SIZE*(SIZE+1)-1 : 0] vertical_wires;

generate
for (i=0; i<=SIZE-1; i=i+1) begin : row
  assign horizontal_wires[8*(i+1)-1 : 8*i] = a[8*(i+1)-1 : 8*i];
  assign vertical_wires[8*(i+1)-1 : 8*i] = b[8*(i+1)-1 : 8*i];
  for (j=0; j<=SIZE-1; j=j+1) begin : col
    systolic_leaf
      systolic_leaf_inst (
              .clk   (clk),
              .rst   (rst),
              .a_in  (horizontal_wires[8*(j*(SIZE+1) + (i+1))-1:8*(j*(SIZE+1) + i)]),
              .b_in  (vertical_wires[8*(j*SIZE + (i+1))-1:8*(j*SIZE + i)]),
              .a_out (horizontal_wires[8*(j*(SIZE+1) + (i+2))-1:8*(j*(SIZE+1) + (i+1))]),
              .b_out (vertical_wires[8*((j+1)*SIZE + (i+1))-1:8*((j+1)*SIZE + i)]),
              .d_out (d[32*(j*SIZE + (i+1))-1:32*(j*SIZE + i)])
      );
  end
end
endgenerate

endmodule
```

# Wrapup

- Top-down design of hardware design $\rightarrow$ NVIDIA Tensor Cores, Google TPU Systolic array case studies
- It is possible to approximate (to the first order) even complex hardware designs
- Hardware design complexity can be tamed with replication via `generate` statements