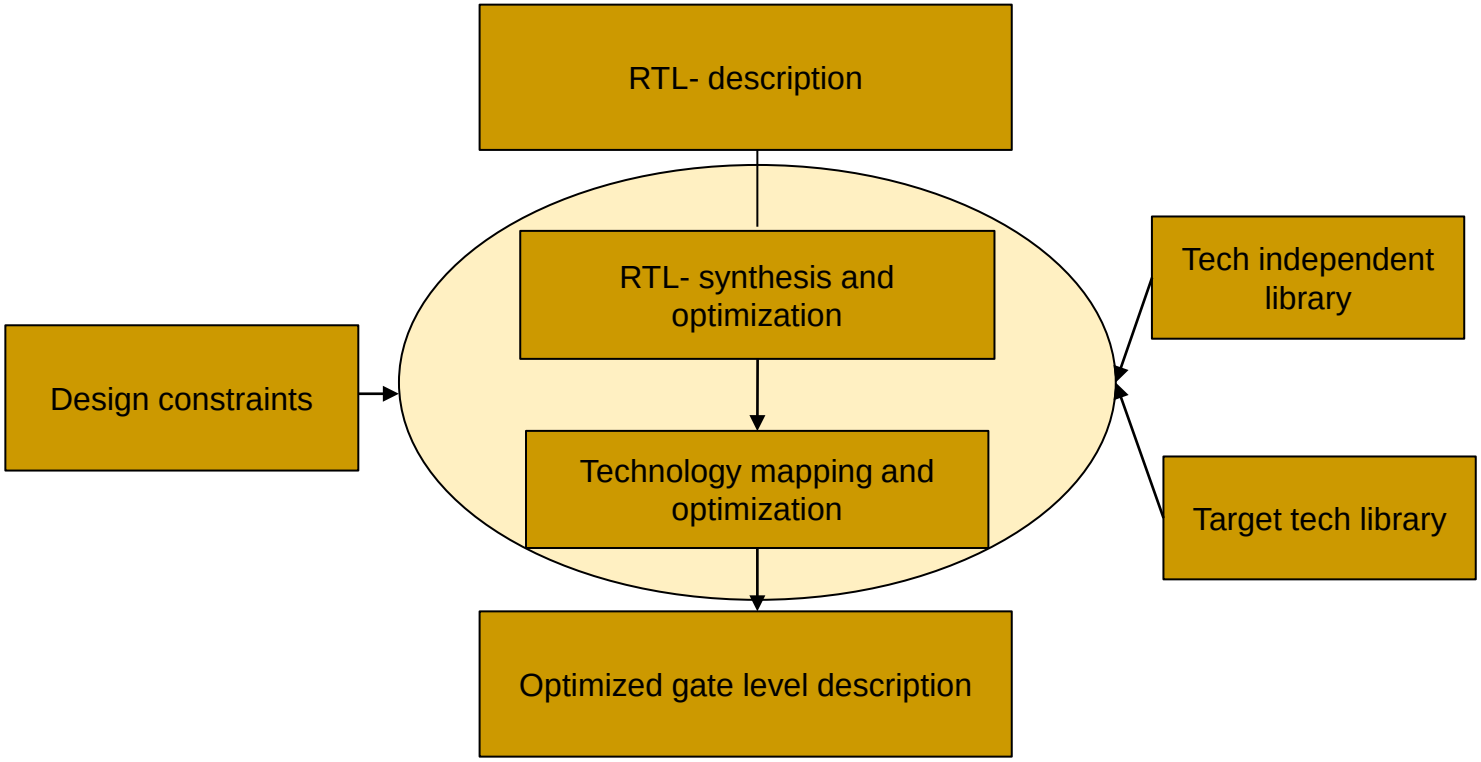
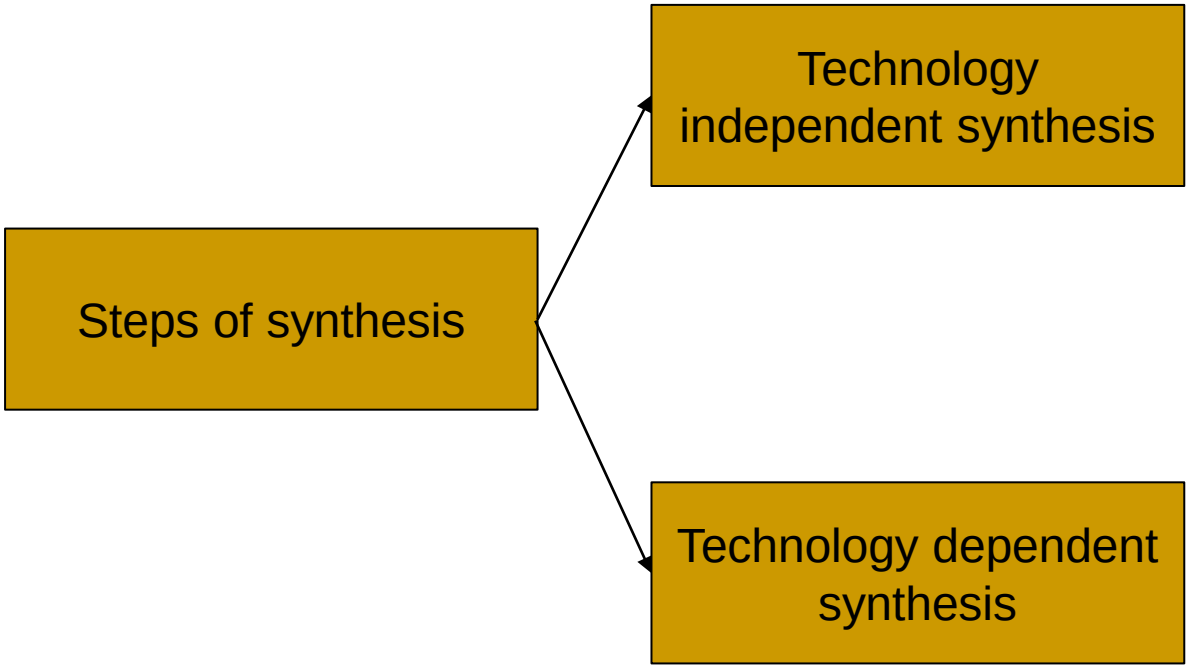
Process of Synthesis

Process of Synthesis



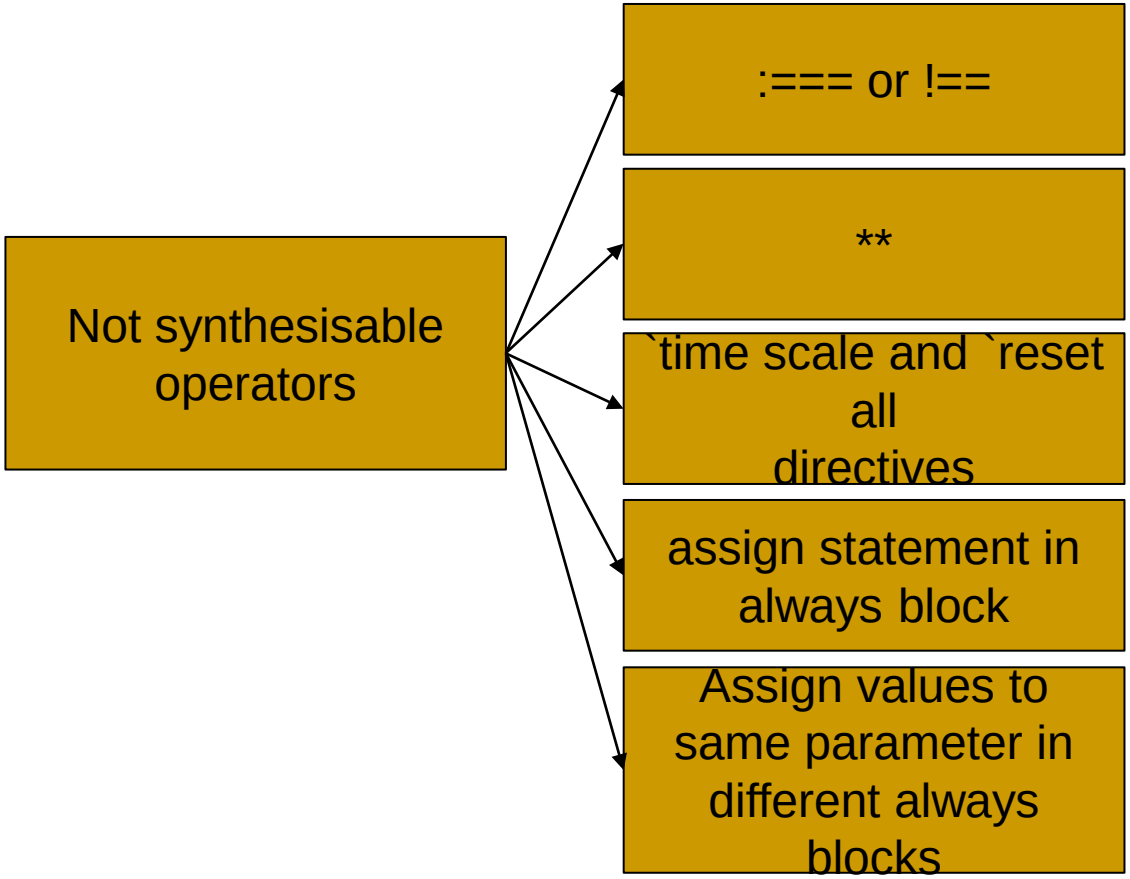
SYNOPSYS®

Steps of Synthesis



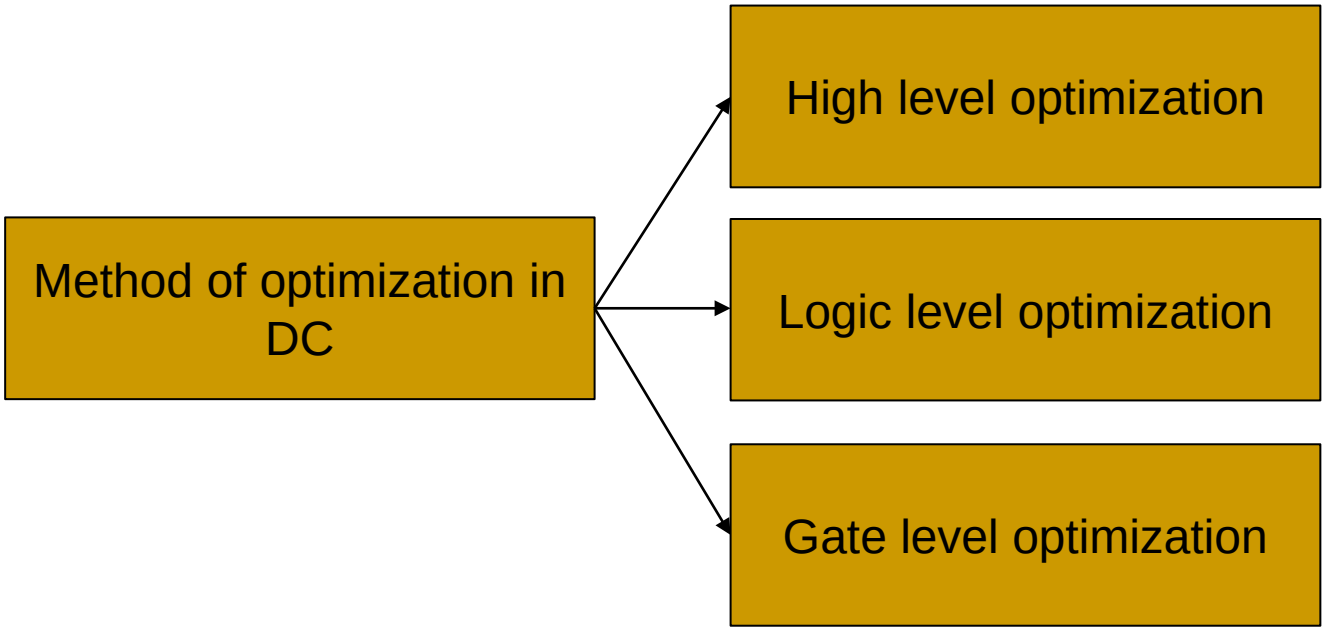
SYNOPSYS®

Not Synthesisable Operators

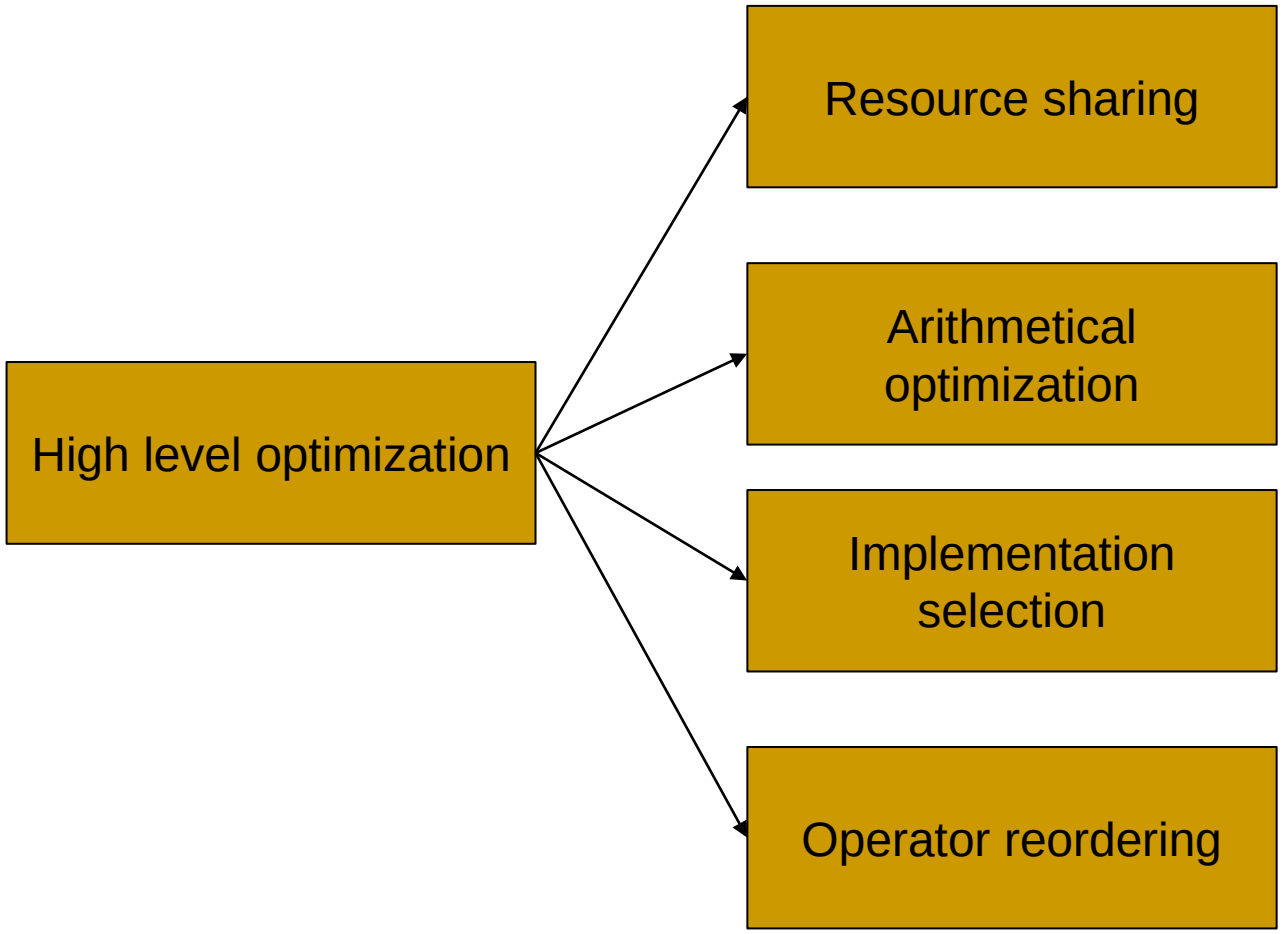


SYNOPSYS®

Optimization Methods in Design Compiler

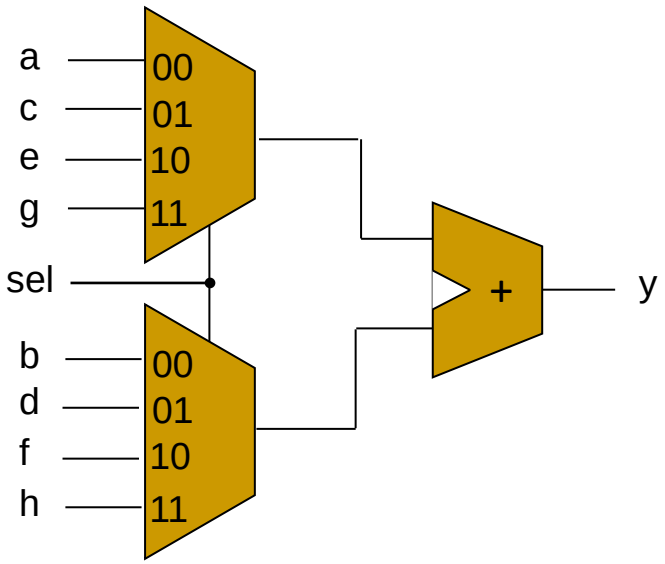


High Level Optimization



Resource Sharing

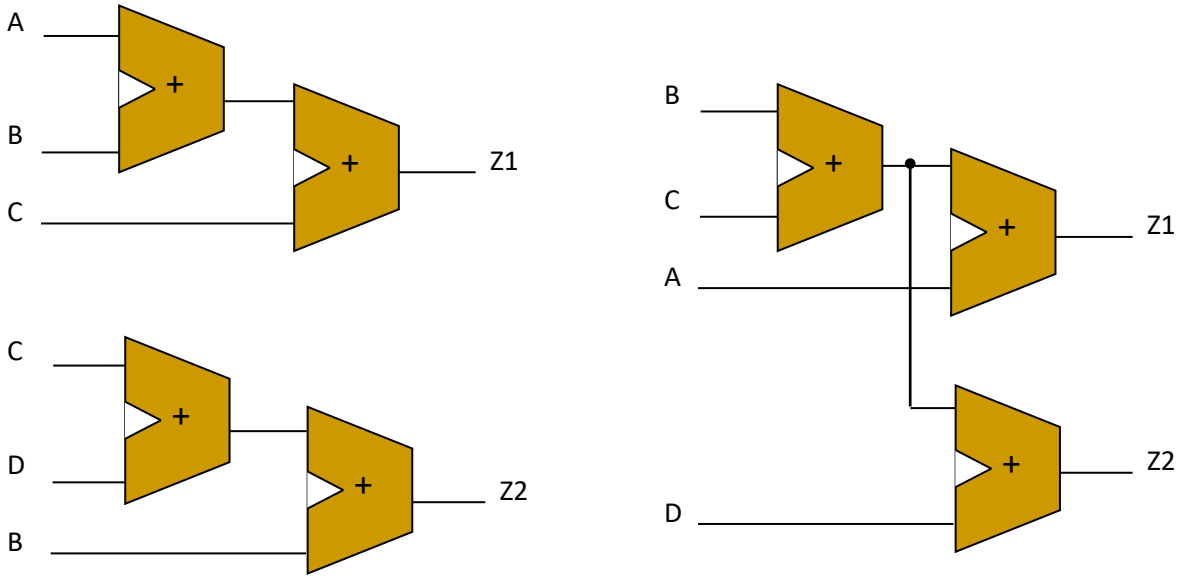
```
if(sel==2'b00) y=a+b;
else if(sel==2'b01)
   y=c+d;
else if(sel==2'b10)
   y=e+f;
else y=g+h;
....
```



SYNOPSYS

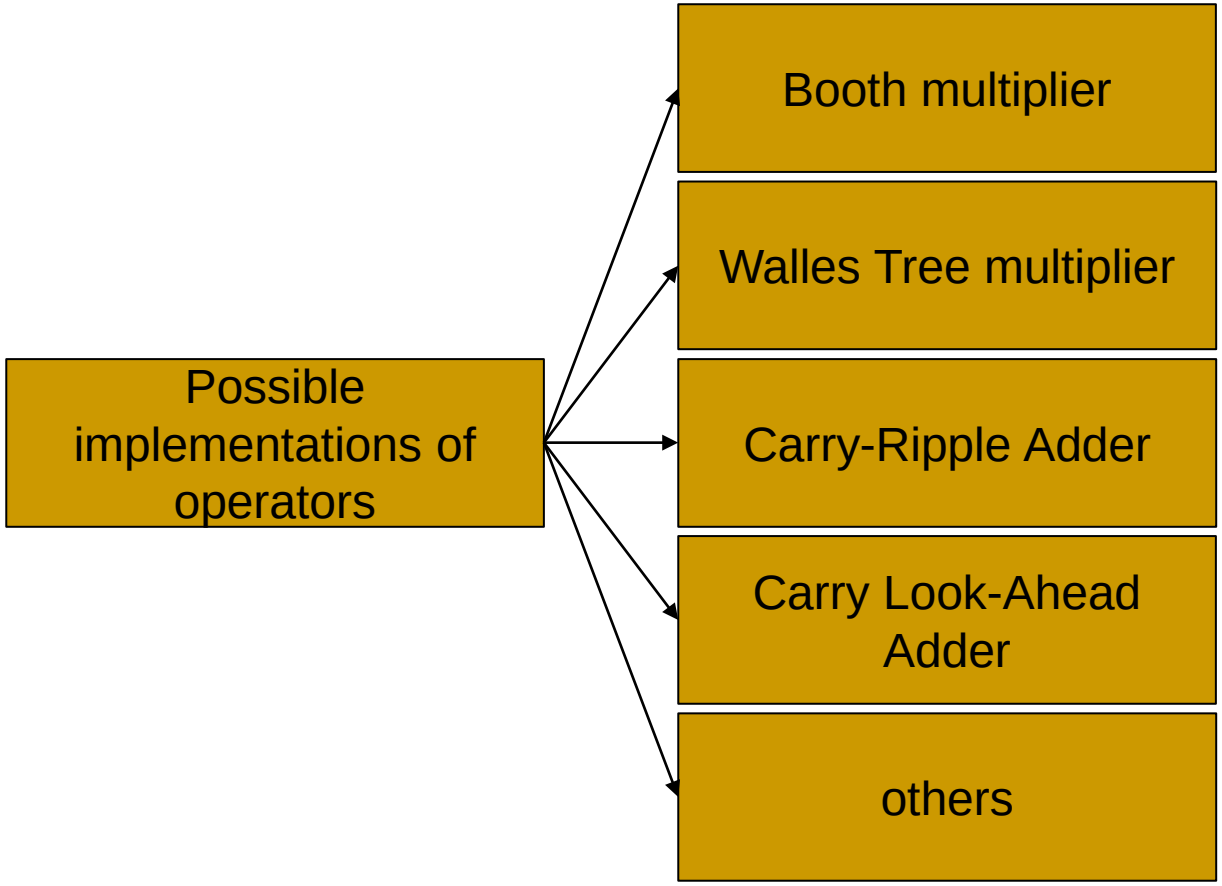
Arithmetical Optimization

- Z1= A+B+C Z2= C+D+B



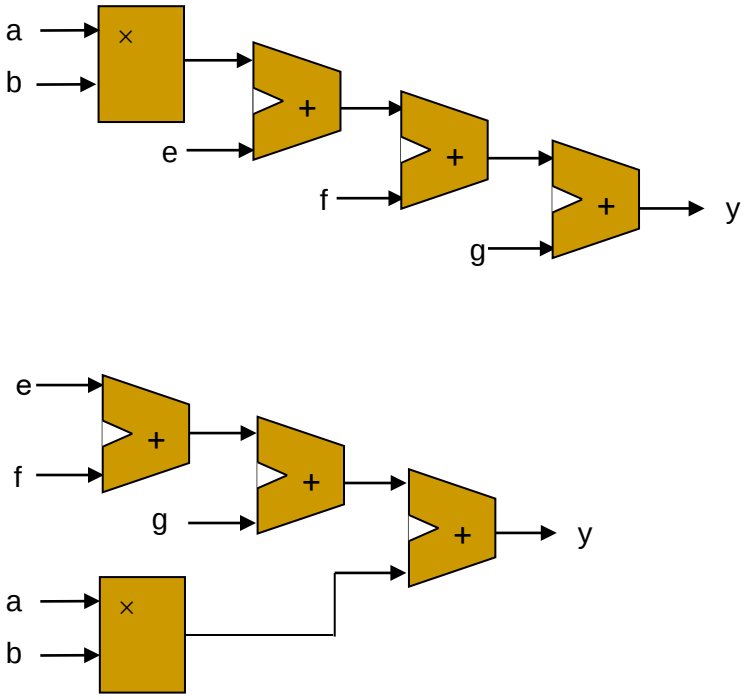
SYNOPSYS®

Implementation Selection



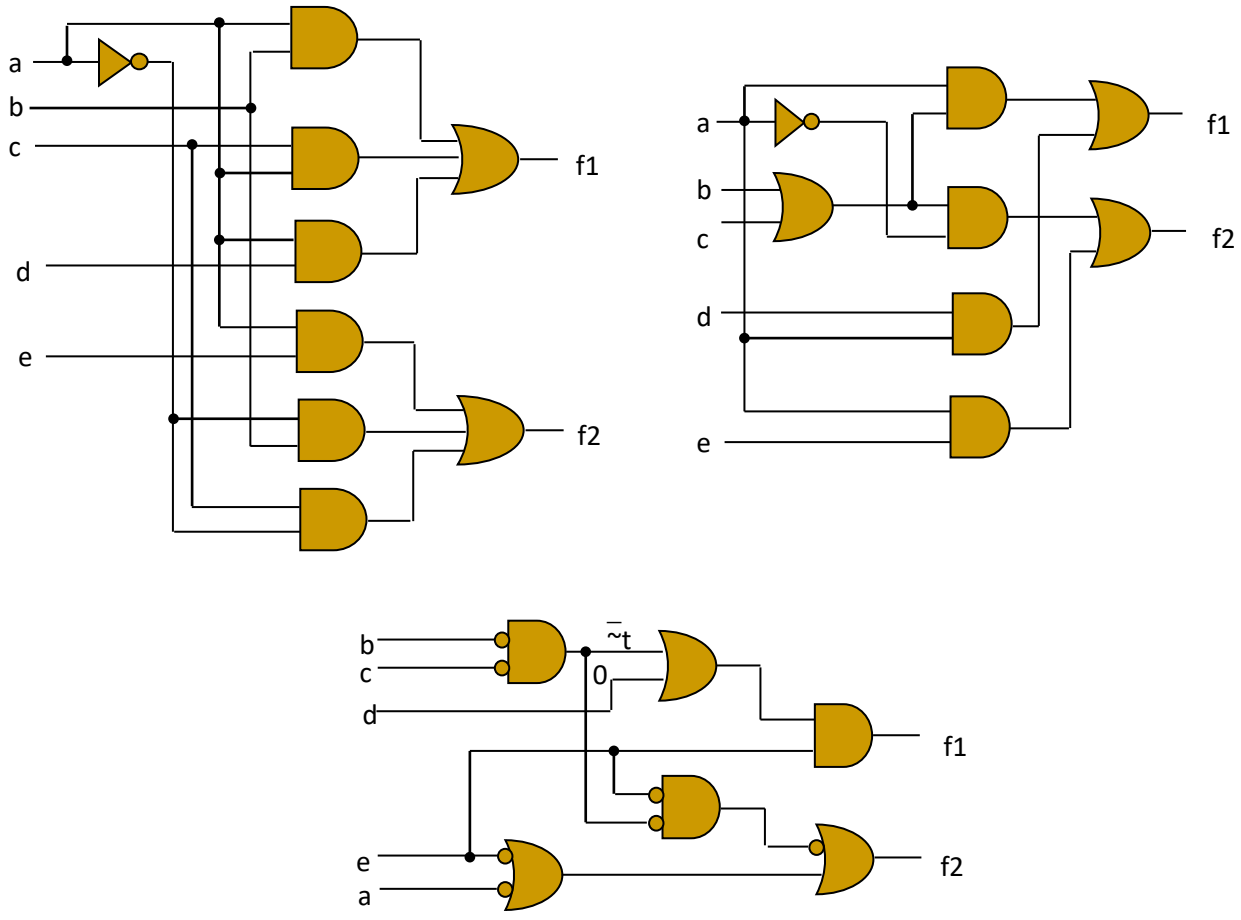
Operator Reordering

Y=a*b+e+f+g



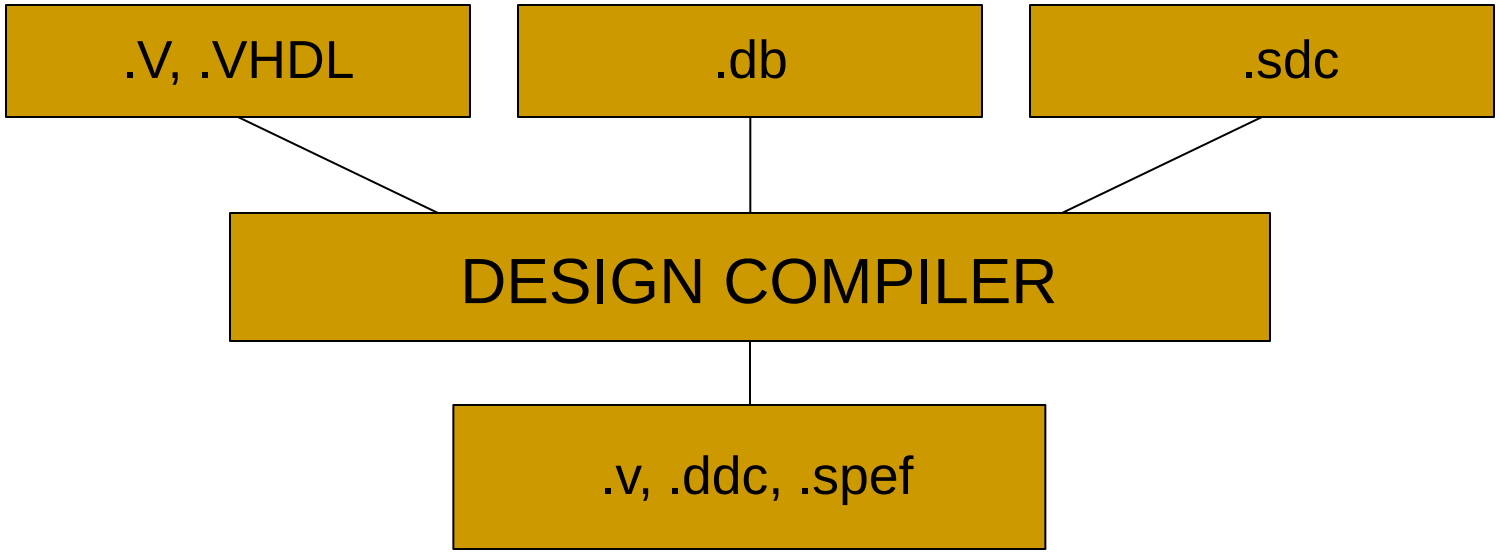
SYNOPSYS®

Gate Level Optimization



SYNOPSYS®

Inputs and Output of Logic Synthesis Tool (DC)



SYNOPSYS®

Systolic Arrays

Systolic Arrays: Motivation

- Goal: design an accelerator that has
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth

- Idea: Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - such that they collectively transform a piece of input data before outputting it to memory

- Benefit: Maximizes computation done on a single piece of data element brought from memory

src: O. Mutlu, ETH

Systolic Arrays

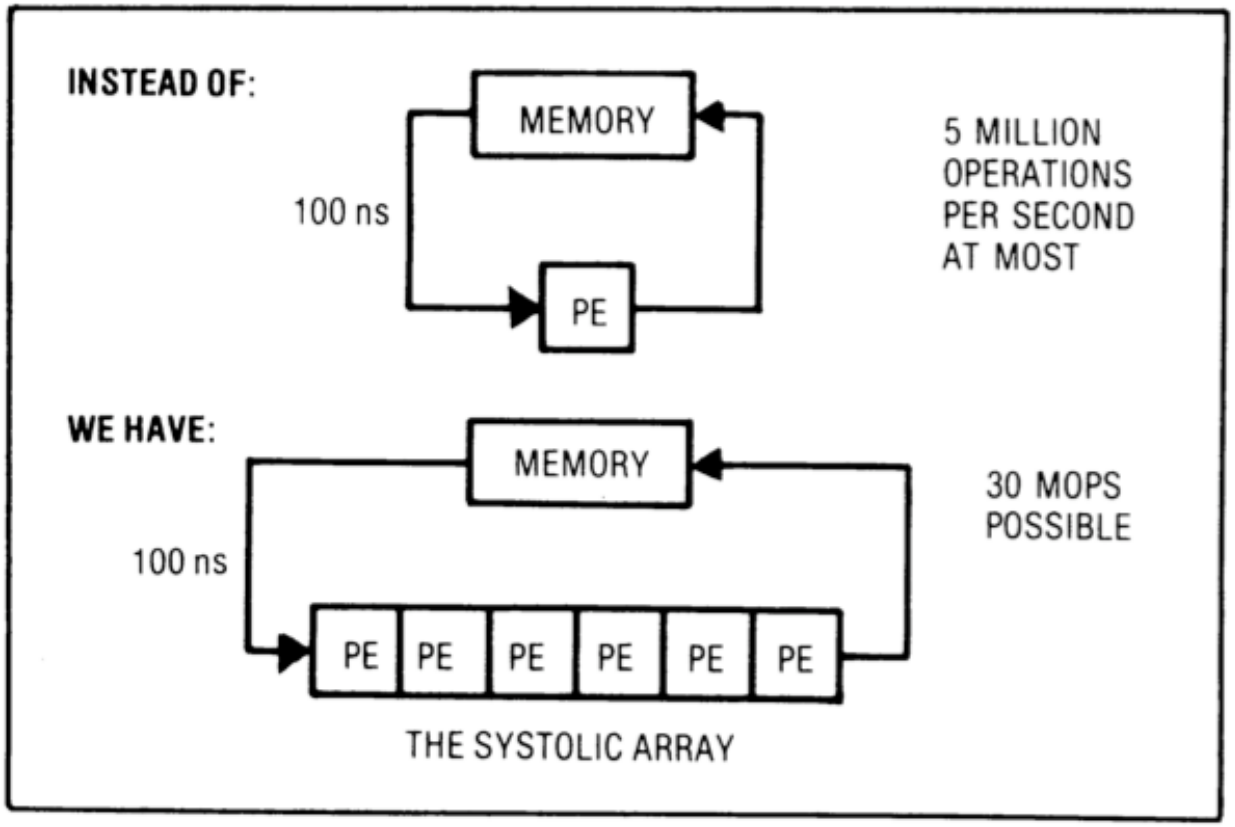


Figure 1. Basic principle of a systolic system.

- H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

src: O. Mutlu, ETH

Systolic Arrays

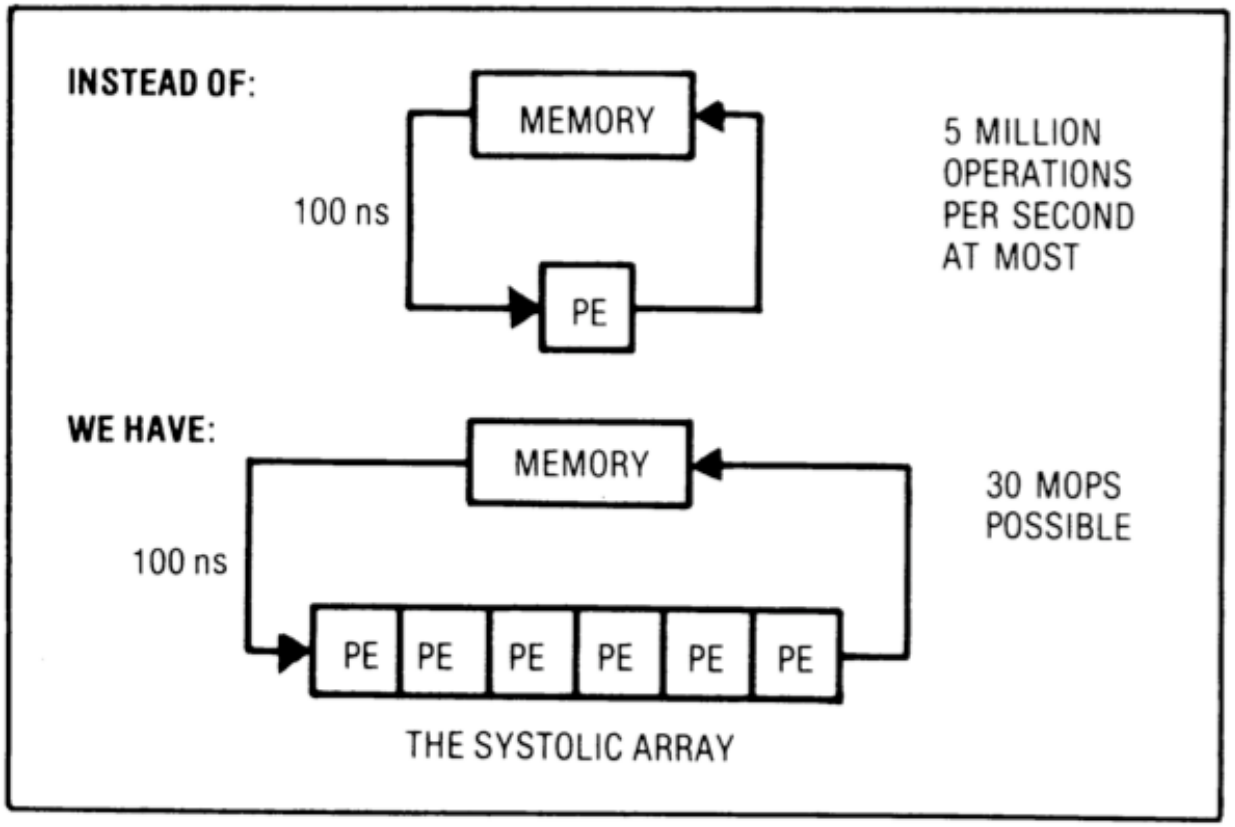


Figure 1. Basic principle of a systolic system.

Memory: heart
Data: blood
PEs: cells

Memory pulses data through PEs

- H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

src: O. Mutlu, ETH

Why Systolic Architectures?

src: O. Mutlu, ETH

Why Systolic Architectures?

- Idea: Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory

- Similar to blood flow: heart → many cells → heart
  - Different cells "process" the blood
  - Many veins operate simultaneously
  - Can be many-dimensional

- Why? Special purpose accelerators/architectures need
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth

src: O. Mutlu, ETH

Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - Balance computation and memory bandwidth

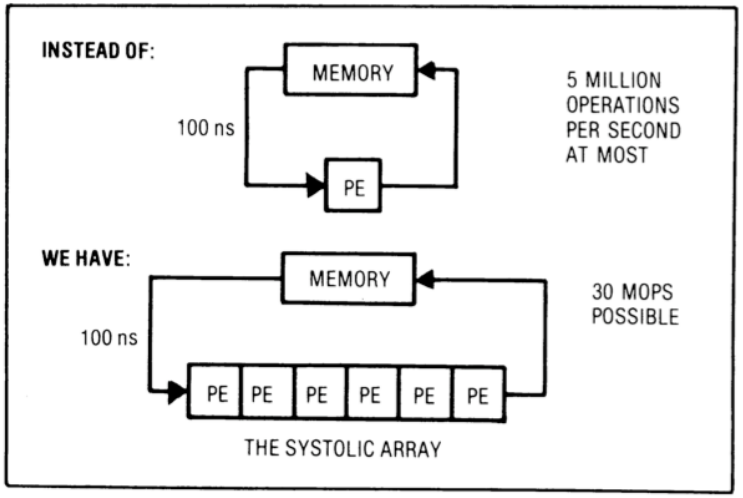


Figure 1. Basic principle of a systolic system.

- Differences from pipelining:
  - These are individual PEs
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)

src: O. Mutlu, ETH

Systolic Architectures

- Groq Microarchitecture
  - Neural Networks

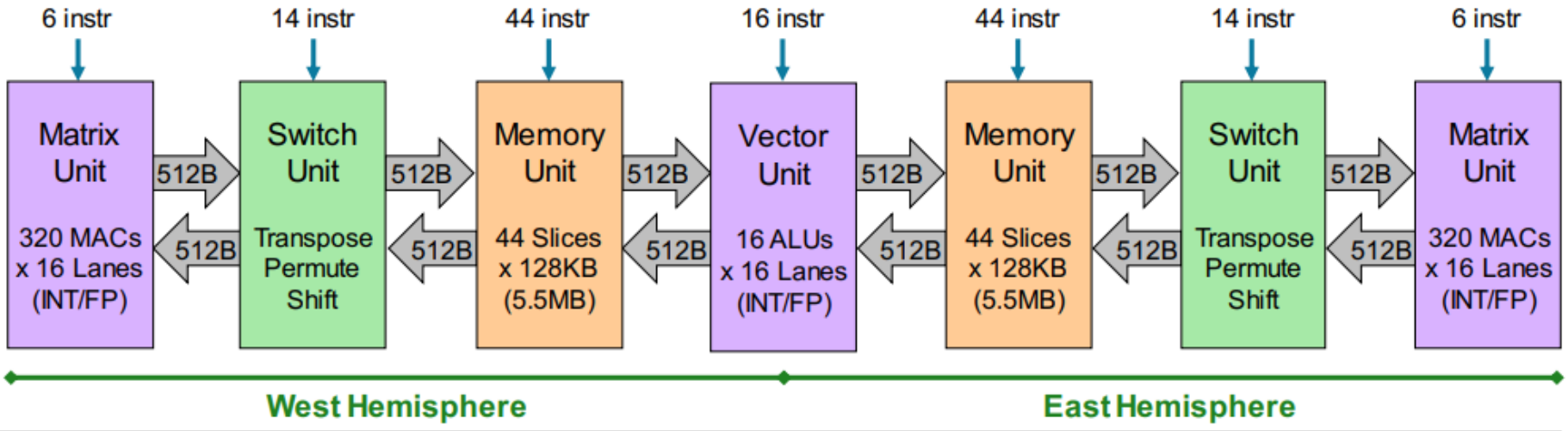


Figure 2. TSP superlane block diagram. Every superlane is bilaterally symmetric with an east side and a west side. It contains 16 lanes, each of which is 8 bits wide. Data flows from east to west and from west to east.

src: https://groq.com/wp-content/uploads/2020/04/Groq-Rocks-NNs-Linley-Group-MPR-2020Jan06.pdf

Systolic Architectures

- Groq Microarchitecture
  - Neural Networks

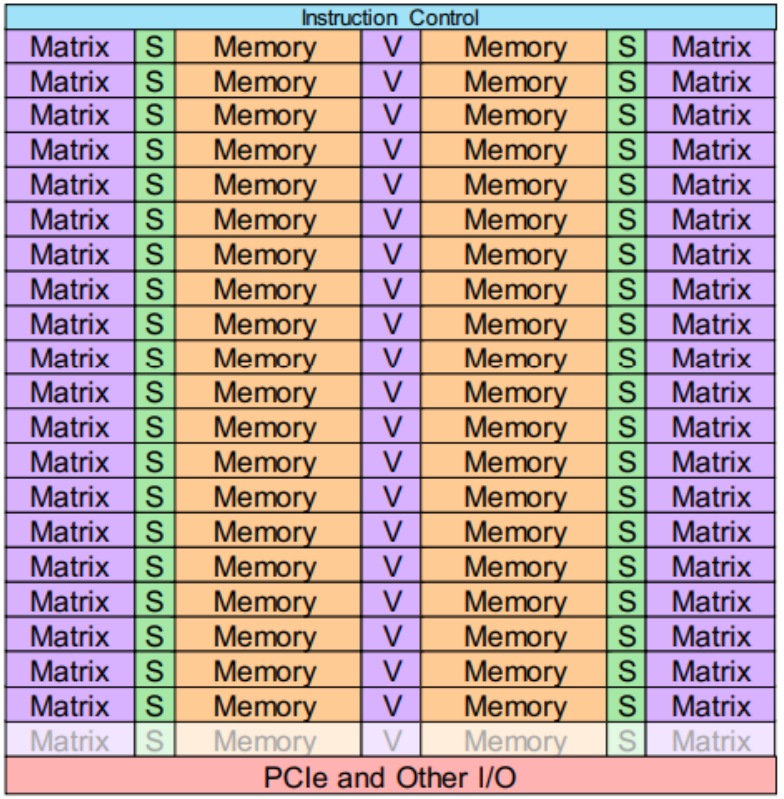


Figure 3. Groq TSP die layout. Built in 14nm CMOS, the 725mm^2 chip has 20 superlanes plus an extra superlane for redundancy. Instruction control requires only 3% of the die area. Diagram not to scale.

src: https://groq.com/wp-content/uploads/2020/04/Groq-Rocks-NNs-Linley-Group-MPR-2020Jan06.pdf

Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc …
  - Many image processing tasks
  - Machine learning: up to hundreds of convolutional layers in Convolutional Neural Networks (CNN)

Given the sequence of weights $\{w_1, w_2, \ldots, w_k\}$
and the input sequence $\{x_1, x_2, \ldots, x_n\}$,

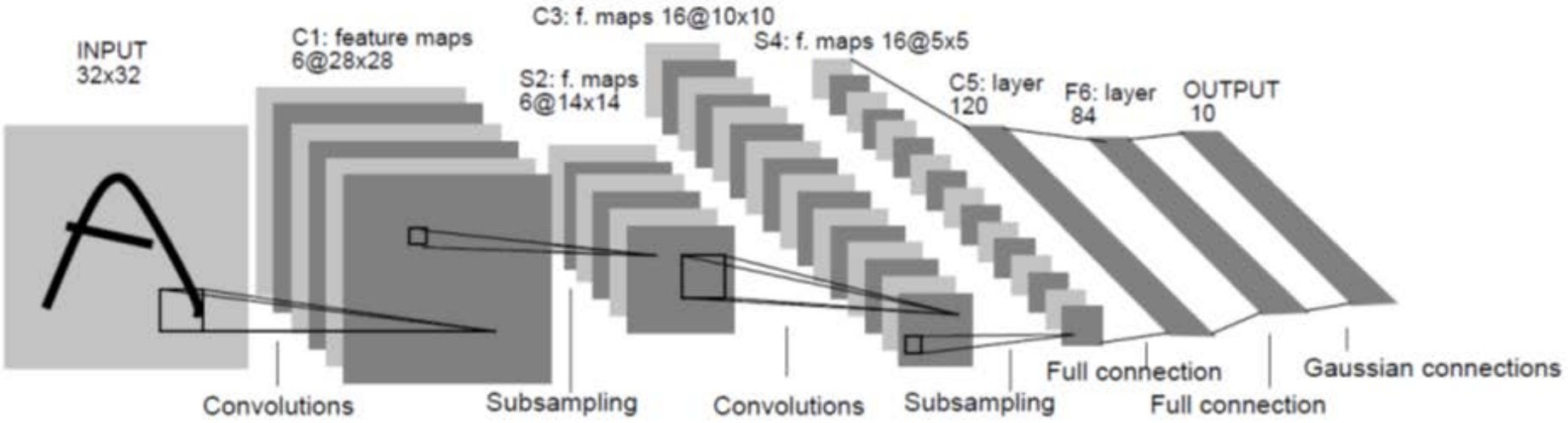
compute the result sequence $\{y_1, y_2, \ldots, y_{n+1-k}\}$
defined by

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}$$

src: O. Mutlu, ETH

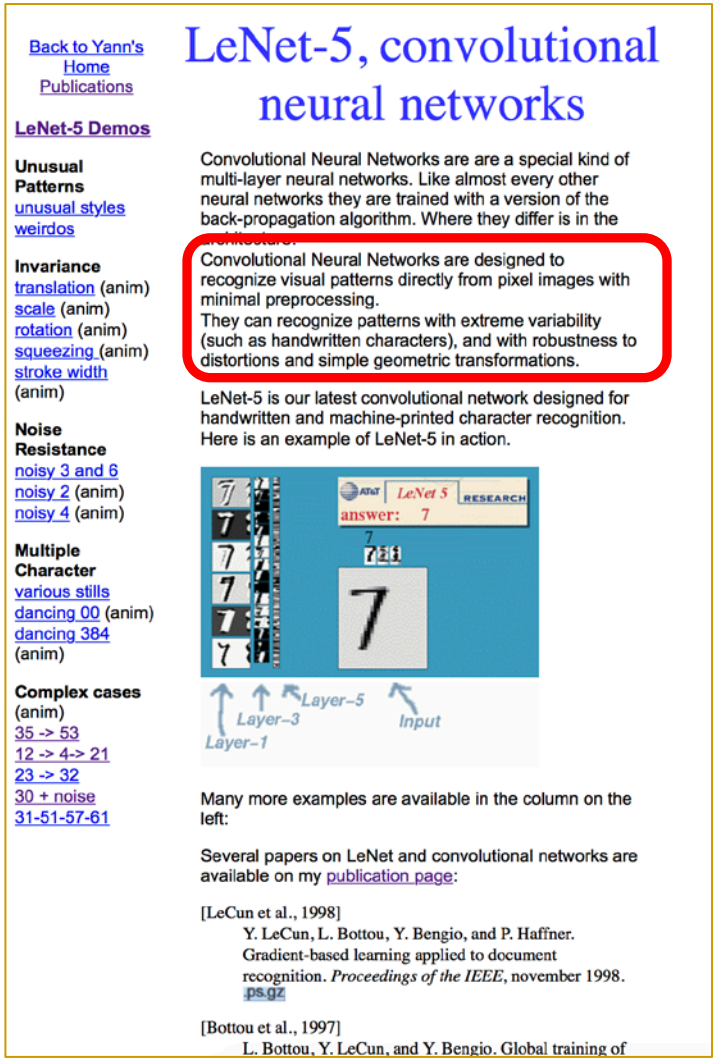
LeNet-5, a Convolutional Neural Network for Hand-Written Digit Recognition

This is a 1024*8 bit input, which will have a truth table of 2^{8196} entries



src: O. Mutlu, ETH

Convolutional Neural Networks: Demo



Back to Yann's Home Publications

LeNet-5 Demos

Unusual Patterns
unusual styles
weirdos

Invariance
translation (anim)
scale (anim)
rotation (anim)
squeezing (anim)
stroke width (anim)

Noise Resistance
noisy 3 and 6
noisy 2 (anim)
noisy 4 (anim)

Multiple Character
various stills
dancing 00 (anim)
dancing 384 (anim)

Complex cases (anim)
35 -> 53
12 -> 4-> 21
23 -> 32
30 + noise
31-51-57-61

LeNet-5, convolutional neural networks

Convolutional Neural Networks are are a special kind of multi-layer neural networks. Like almost every other neural networks they are trained with a version of the back-propagation algorithm. Where they differ is in the

Convolutional Neural Networks are designed to recognize visual patterns directly from pixel images with minimal preprocessing.
They can recognize patterns with extreme variability (such as handwritten characters), and with robustness to distortions and simple geometric transformations.

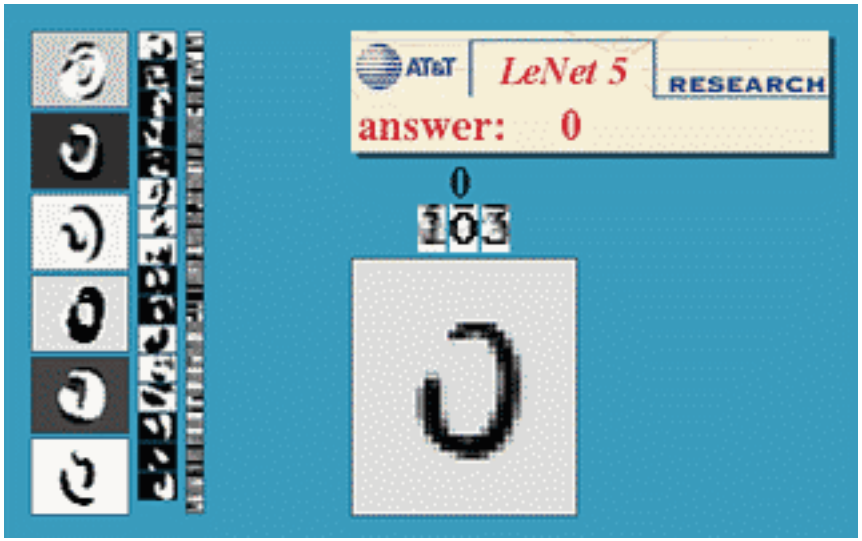
LeNet-5 is our latest convolutional network designed for handwritten and machine-printed character recognition. Here is an example of LeNet-5 in action.

Many more examples are available in the column on the left:

Several papers on LeNet and convolutional networks are available on my publication page:

[LeCun et al., 1998]
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998. .ps.gz

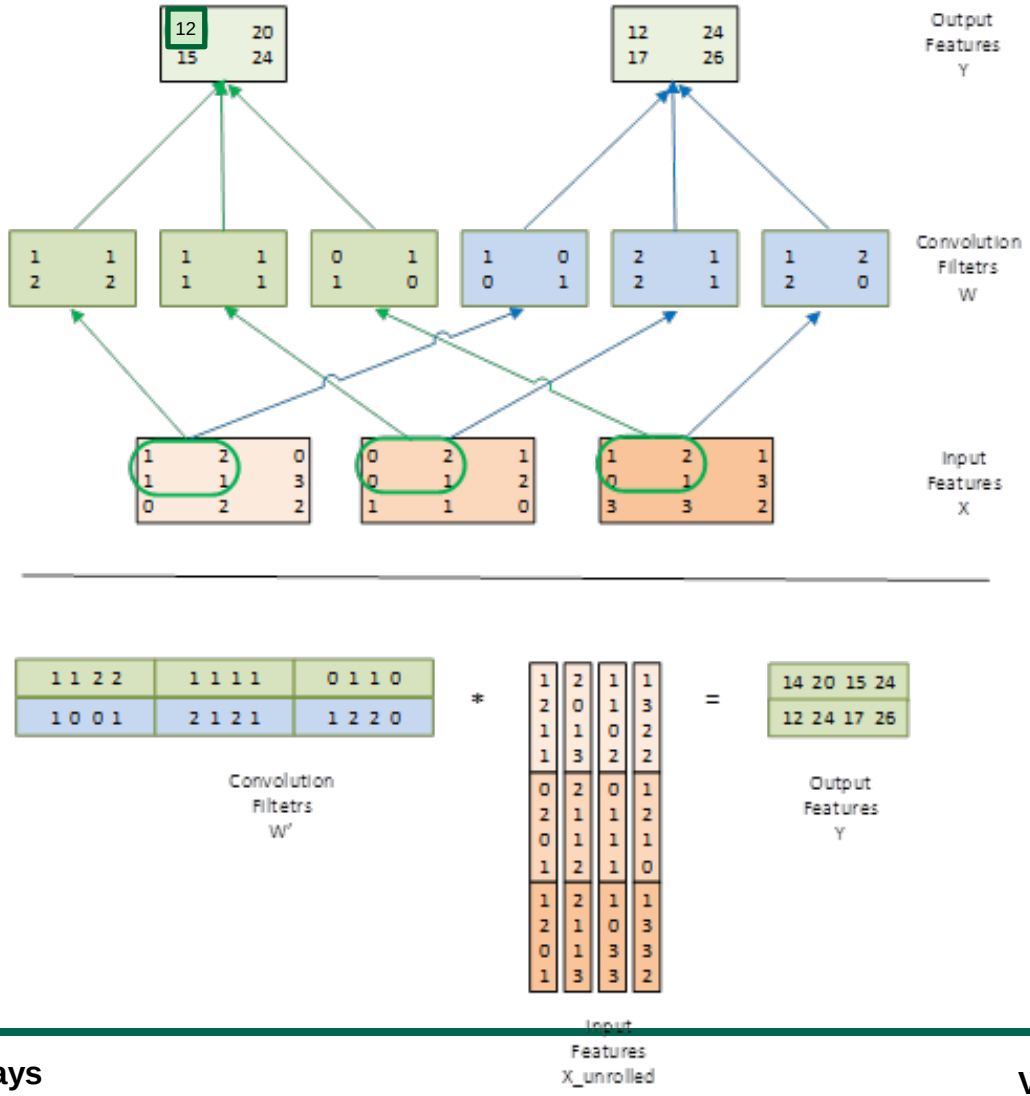
[Bottou et al., 1997]
L. Bottou, Y. LeCun, and Y. Bengio. Global training of



http://yann.lecun.com/exdb/lenet/index.html

src: O. Mutlu, ETH

Implementing a Convolutional Layer with Matrix Multiplication



Slide credit: Hwu & Kirk

src: O. Mutlu, ETH

Google TPU

- Google TPU v1
  - *N. P. Jouppi, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA 2017*

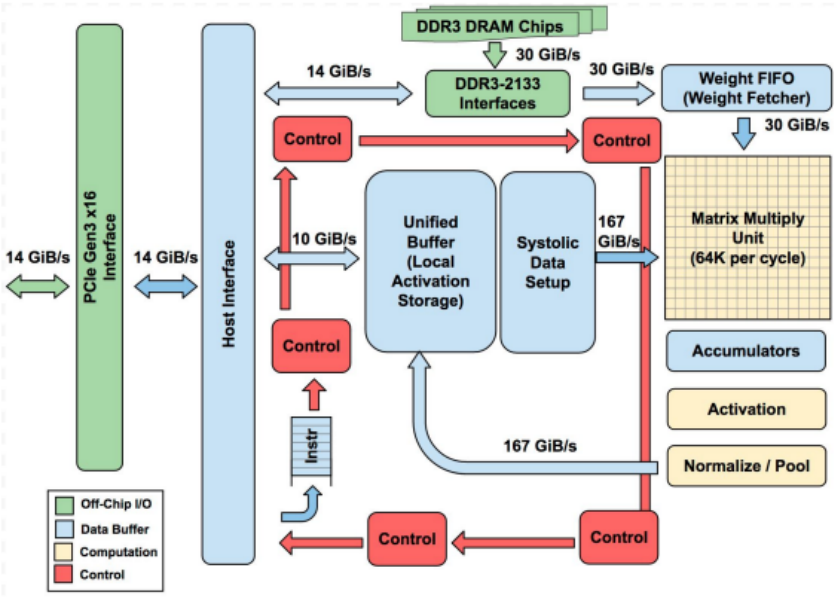


Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

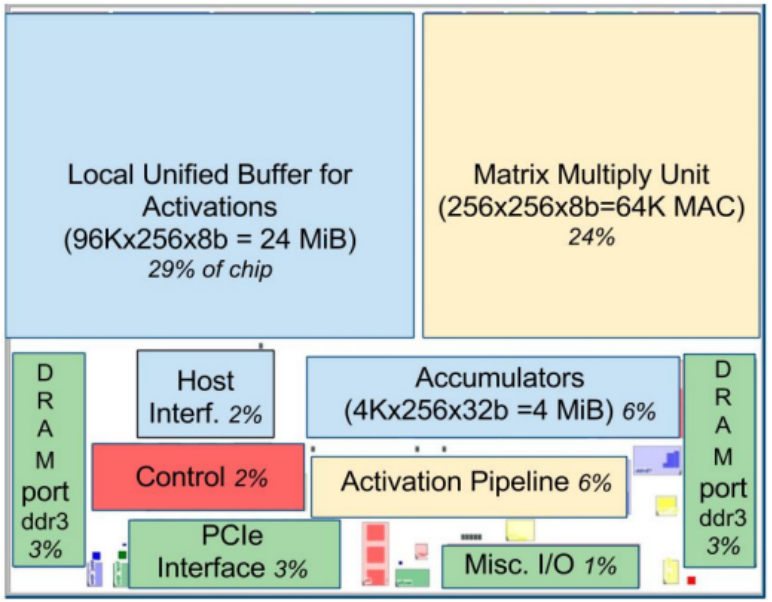


Figure 2. Floor Plan of TPU die. The shading follows Figure 1. The light (blue) data buffers are 37% of the die, the light (yellow) compute is 30%, the medium (green) I/O is 10%, and the dark (red) control is just 2%. Control is much larger (and much more difficult to design) in a CPU or GPU

Google TPU

- Google TPU v4
  - *N. P. Jouppi, et al. "TPUv4:An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings", ISCA 2023*

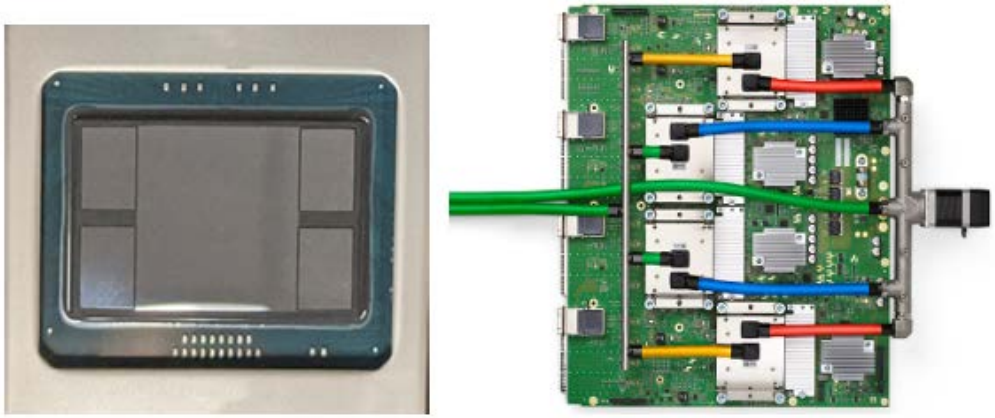


Figure 2: The TPU v4 package (ASIC in center plus 4 HBM stacks) and printed circuit board with 4 liquid-cooled packages. The board's front panel has 4 top-side PCIe connectors and 16 bottom-side OSFP connectors for inter-tray ICI links.

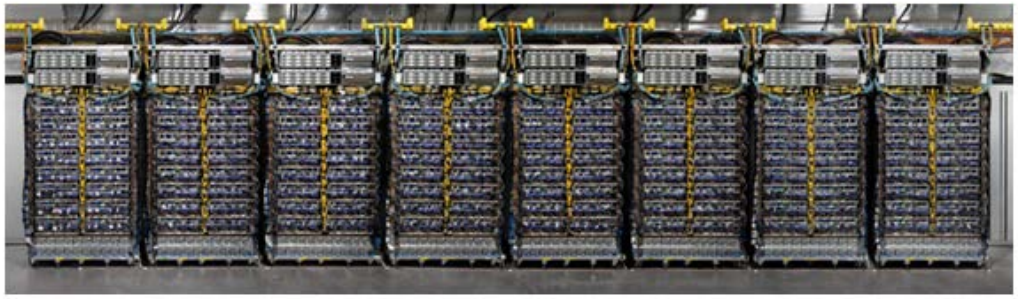
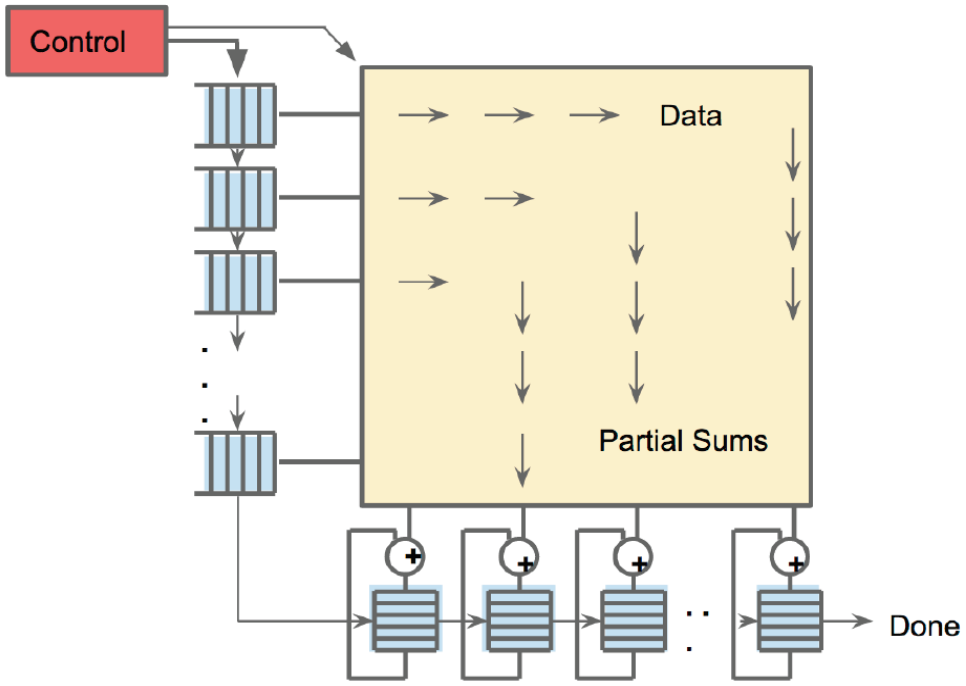


Figure 3: Eight of 64 racks for one 4096-chip supercomputer.

Google TPU

- Systolic Mac Array

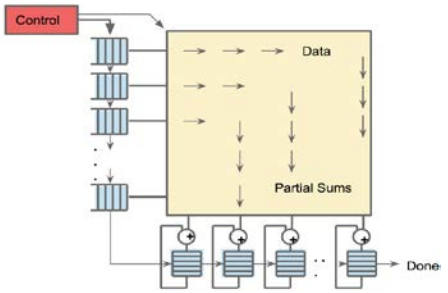


256×256 MAC array (64K MACs)

N. P. Jouppi, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA 2017

Systolic Mac Array

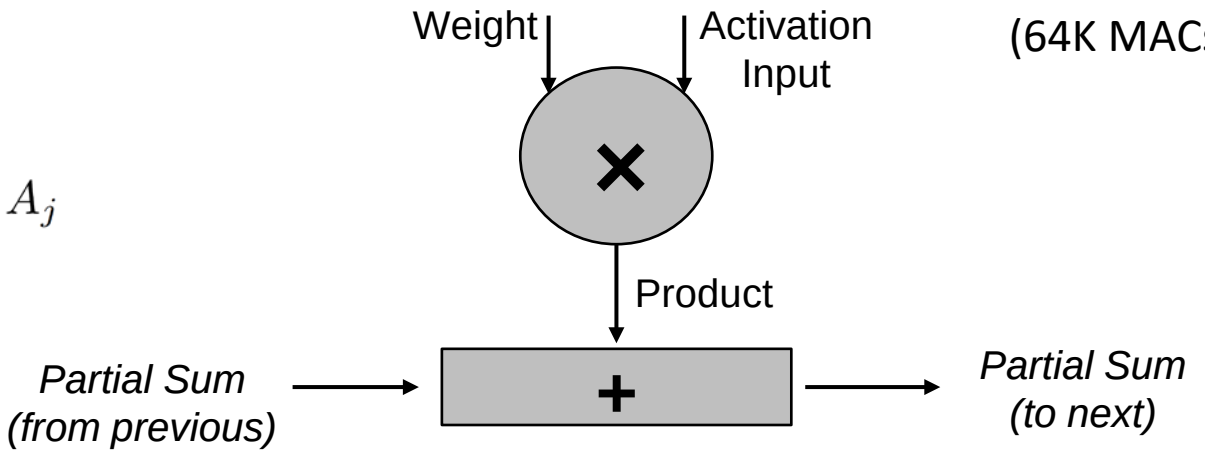
- DNN inference: >70% of time in GEMM Operations
- MAC Unit
  - Multiplier
  - Adder
  - 8bit weights, activations



256×256 MAC array
(64K MACs)

- Convolution:

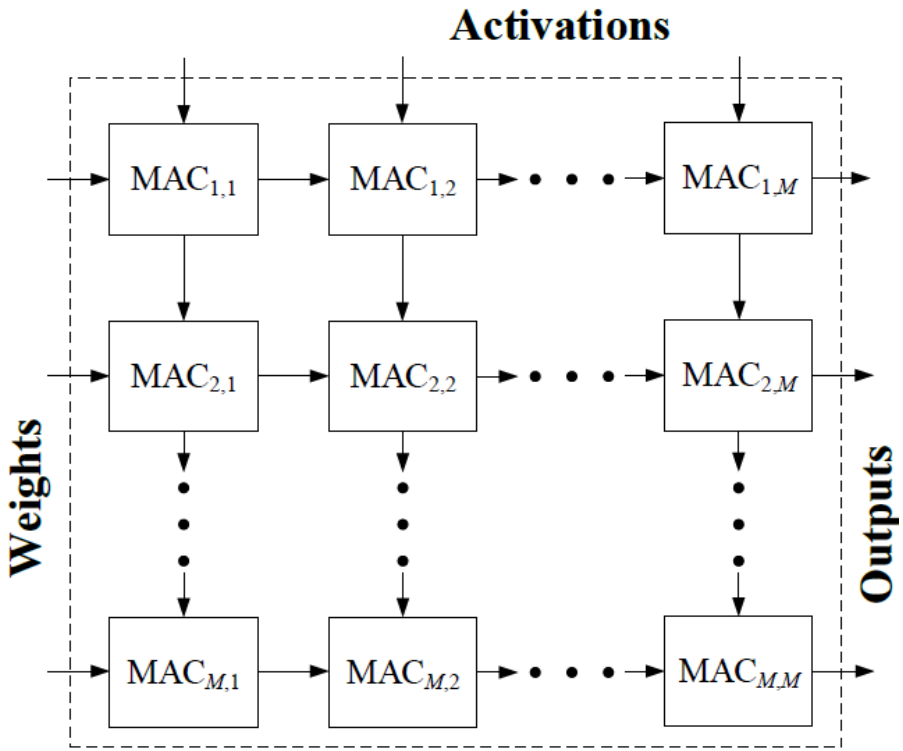
$$Y = B + \sum_{j=1}^{k} W_j \times A_j$$

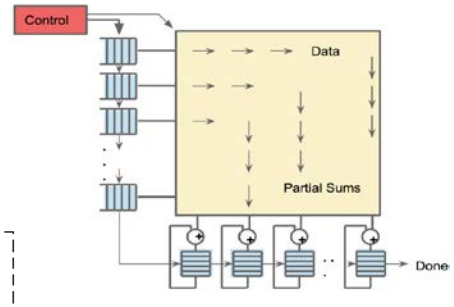


Systolic Mac Array

- DNN inference: >70% of time in GEMM Operations
- MAC Unit
  - Multiplier
  - Adder
  - 8bit weights, acti
- Convolution:

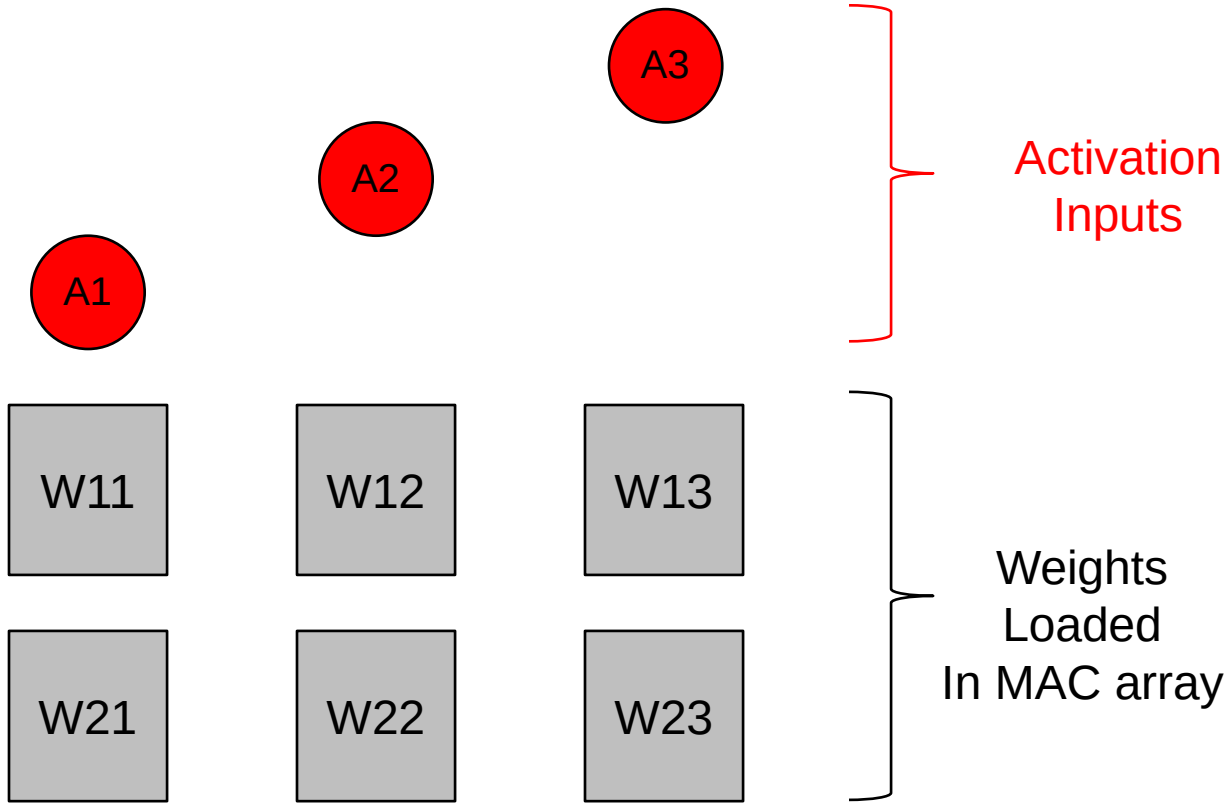
$$Y = B + \sum_{j=1}^{k} W_j \times .$$



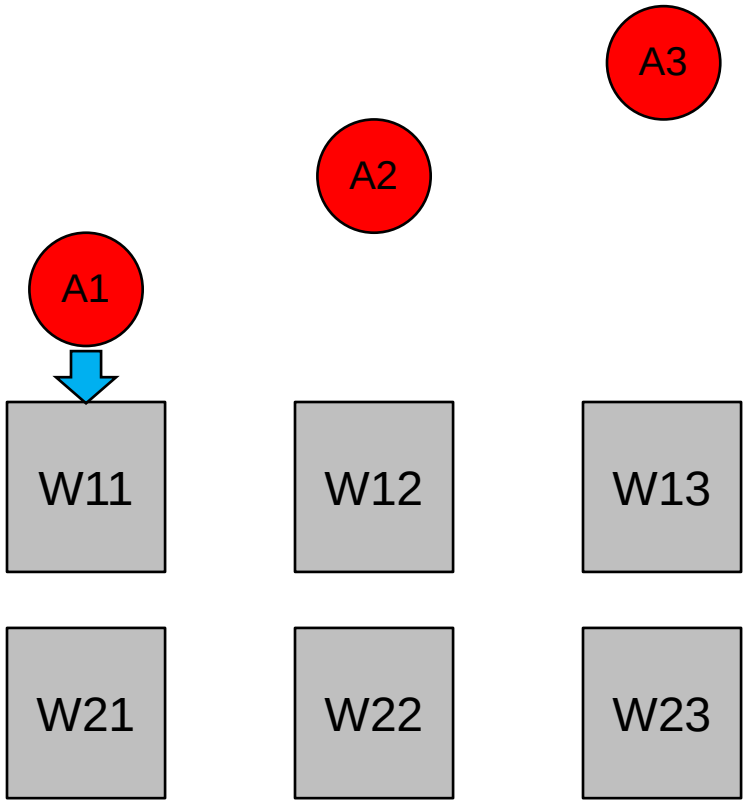


256×256 MAC array
(64K MACs)

Systolic Mac Array

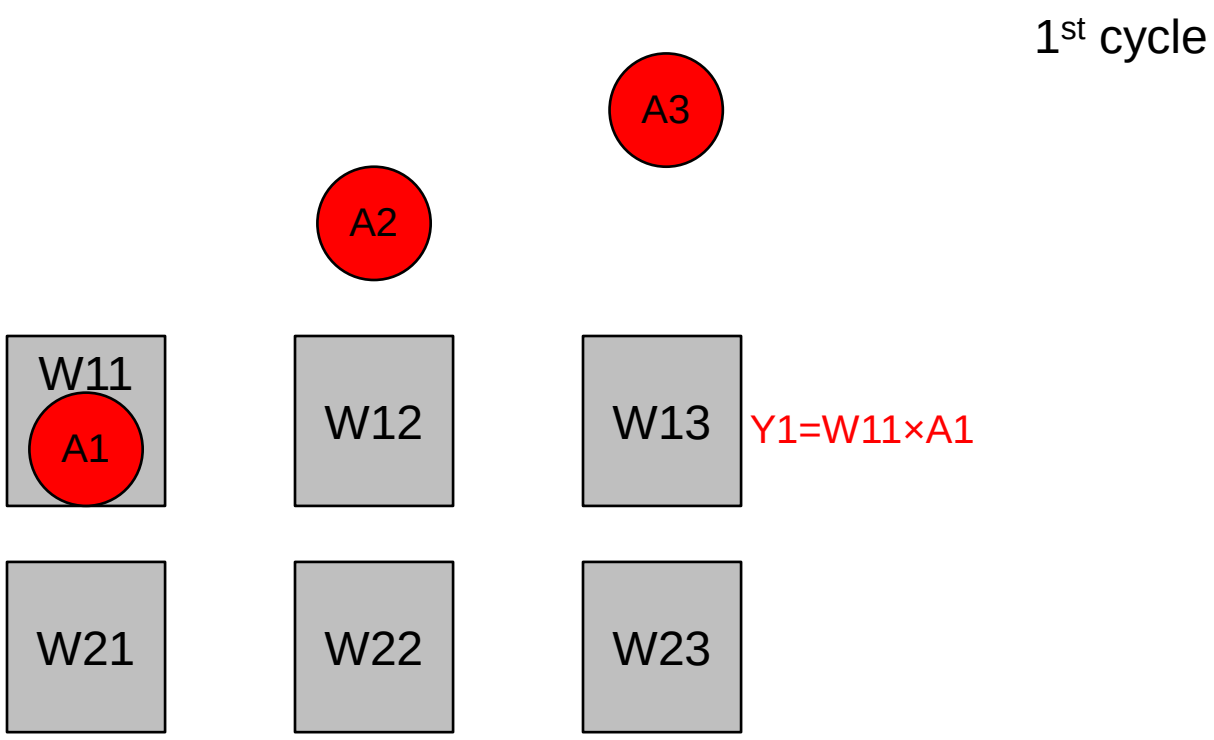


Systolic Mac Array

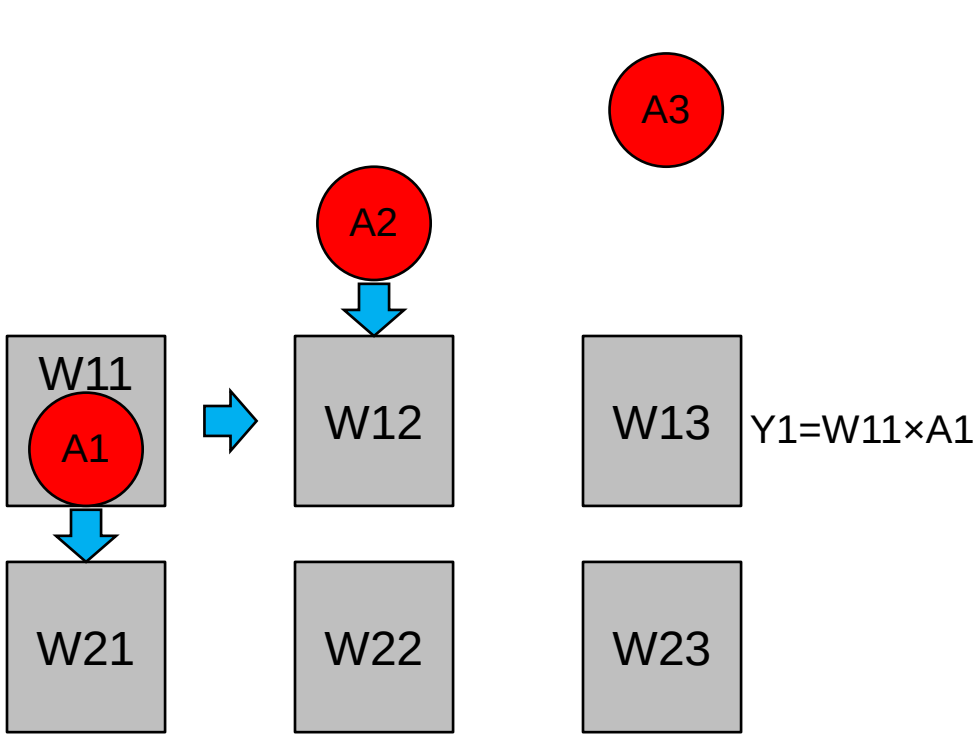


1st cycle

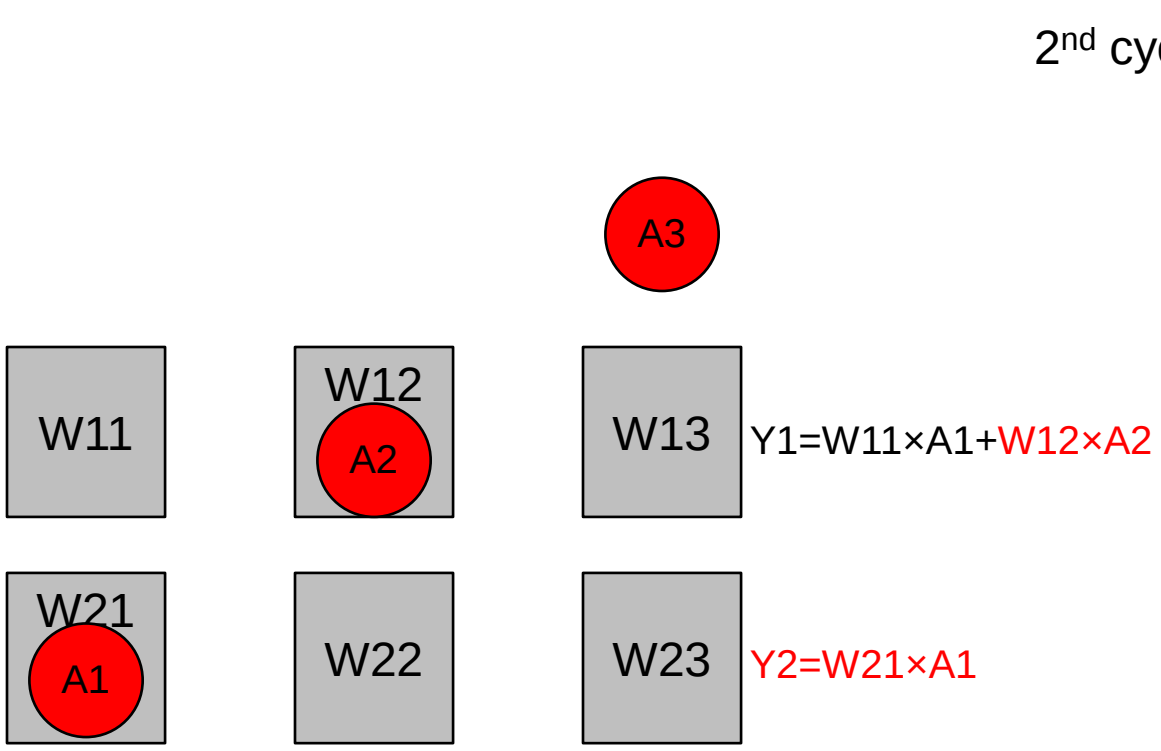
Systolic Mac Array



Systolic Mac Array

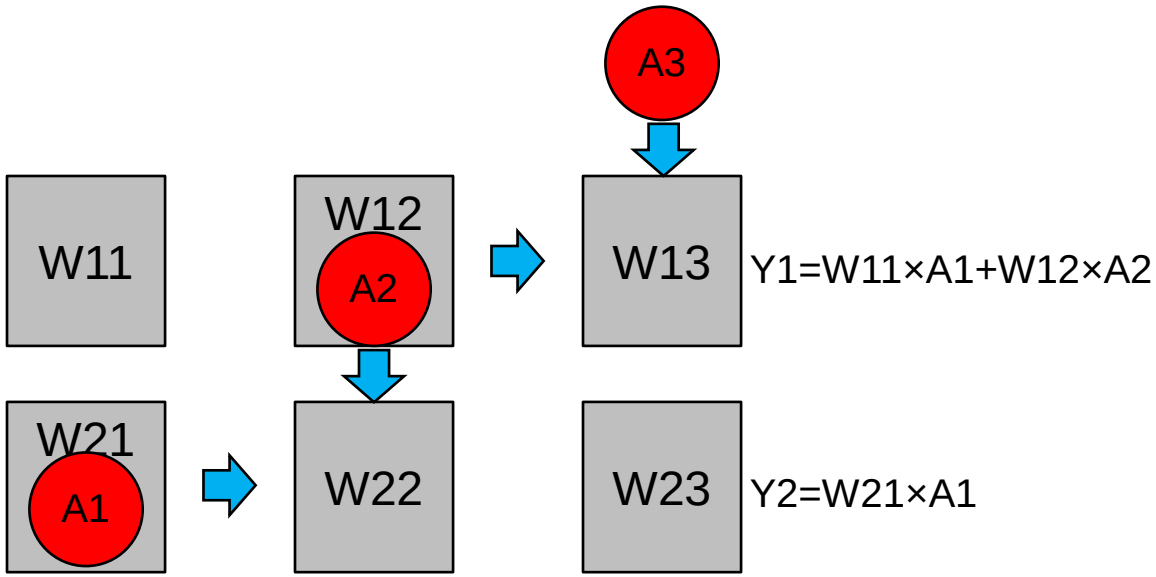


Systolic Mac Array



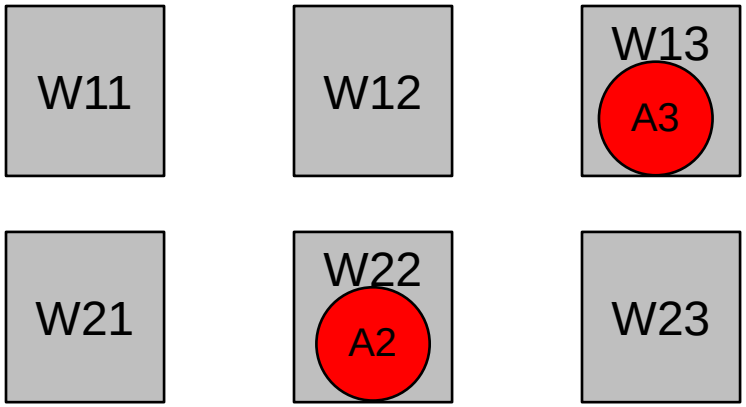
Systolic Mac Array

3rd cycle



Systolic Mac Array

3rd cycle

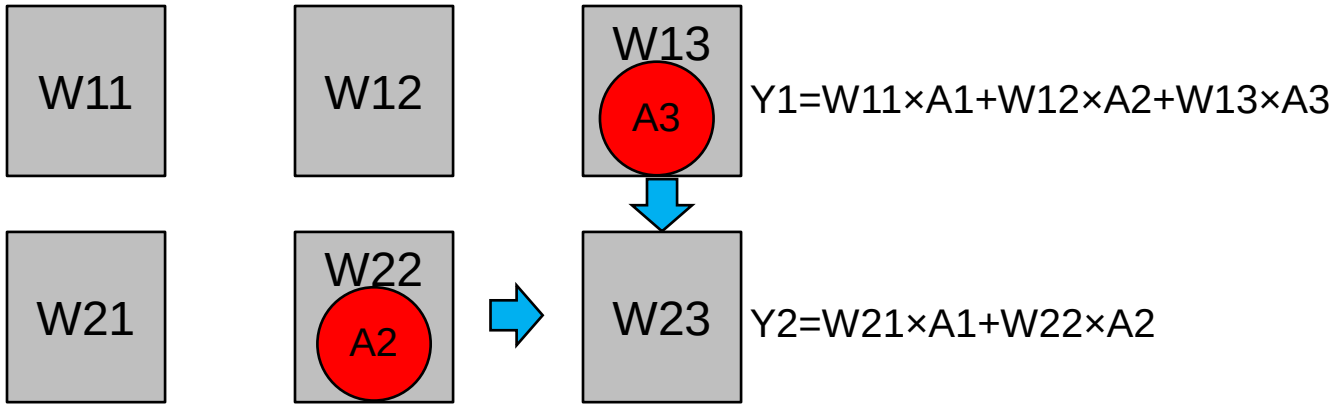


Y1=W11×A1+W12×A2+W13×A3

Y2=W21×A1+W22×A2

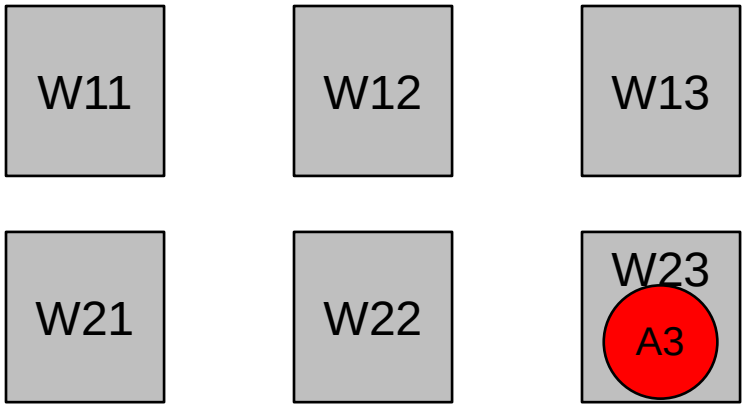
Systolic Mac Array

4th cycle



Systolic Mac Array

4th cycle



Y1=W11×A1+W12×A2+W13×A3

Y2=W21×A1+W22×A2+W23×A3

Systolic Mac Array

- Systolic mac array Verilog example

```
module macarray (clk, rst_n, newWeights_i, weight_i, input_i, OUT_o);
    parameter arrayLines = 16;
    parameter arrayColumns = 16;
    parameter inputWidth = 8;
    parameter weightWidth = 8;
    parameter accumulationWidth=32;

    input clk, rst_n, newWeights_i;
    input [weightWidth*arrayLines*arrayColumns-1:0] weight_i;
    input [inputWidth*arrayColumns-1:0] input_i;
    output [accumulationWidth*arrayLines-1:0] OUT_o;

    wire signed [inputWidth-1:0] activation_w [1:arrayLines+1] [1:arrayColumns+1];
    wire signed [accumulationWidth-1:0] accumulate_w [1:arrayLines+1][1:arrayColumns+1];
    wire signed [weightWidth-1:0] weight_w [1:arrayLines][1:arrayColumns+1];
```

Systolic Mac Array

- Systolic mac array Verilog example

```
    genvar gi, gj;
    generate
    //first column zero in
    for (gi = 1; gi <= arrayLines; gi=gi+1) begin: firstcolumn
        assign accumulate_w[gi][1] = {accumulationWidth{1'b0}};
    end
    //activations
    for (gj = 1; gj <= arrayColumns; gj=gj+1) begin: firstinpline
        assign activation_w[1][gj] = $signed(input_i[inputWidth*gj-1:inputWidth*(gj-1)]);
    end
    //weights
    for (gj = 1; gj <= arrayColumns; gj=gj+1) begin: firstinpline
        assign weight_w[1][gj] = $signed(weight_i[weightWidth*gj-1:weightWidth*(gj-1)]);
    end
    // generate macs
    for (gi = 1; gi <= arrayLines; gi=gi+1) begin: lines
        for (gj = 1; gj <= arrayColumns; gj=gj+1) begin: columns
            mac #(inputWidth, weightWidth, accumulationWidth) PE (    .clk(clk), .rst_n(rst_n),
                                                                      .newWeights_i(newWeights_i)
                                                                      .w_i(weight_w[gi][gj]),
                                                                      .in_i(activation_w[gi][gj]),
                                                                      .sumIN_i(accumulate_w[gi][gj]),
                                                                      .w_o(weight_w[gi+1][gj])
                                                                      .in_o(activation_w[gi+1][gj]),
                                                                      .sumOUT_o(accumulate_w[gi][gj+1]));
        end
    end
    //output
    for (gi = 1; gi <= arrayLines; gi=gi+1) begin: out
        assign OUT_o[accumulationWidth*gi-1:accumulationWidth*(gi-1)] = accumulate_w[gi][arrayColumns+1];
    end

    endgenerate

endmodule
```

Systolic Mac Array

- MAC unit

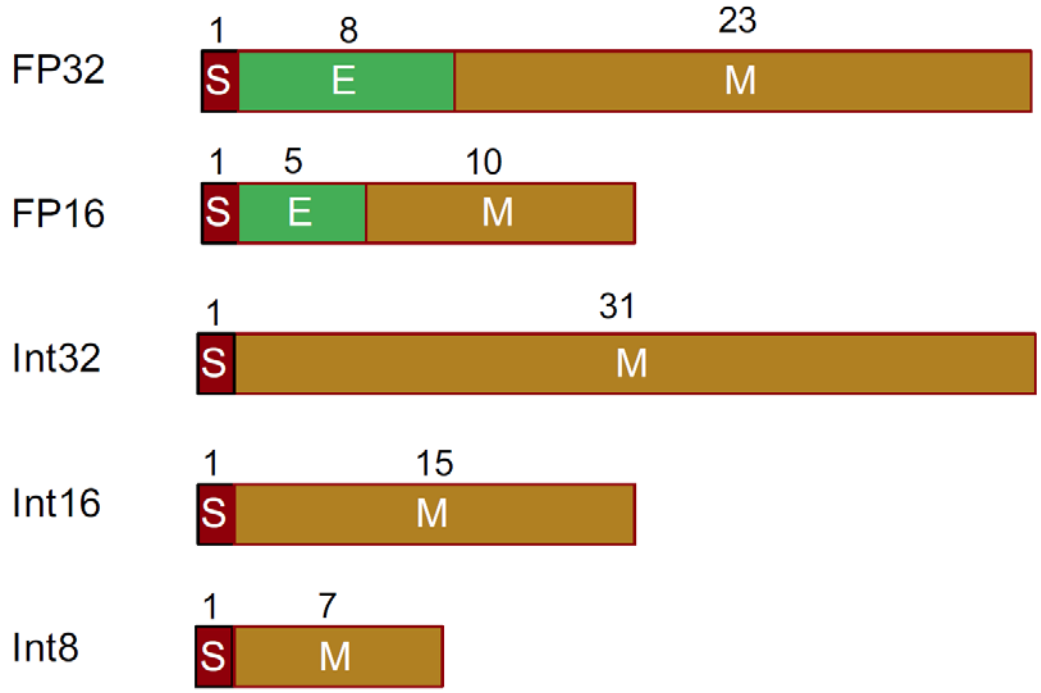
```
module mac (clk, newWeights_i, rst_n, w_i, in_i, sumIN_i, w_o, in_o, sumOUT_o);
    parameter inputWidth = 8;
    parameter weightWidth = 8;
    parameter adderWidth=32;
    localparam prodWidth = 2*inputWidth;

    input clk;
    input rst_n;
    input signed [weightWidth-1:0] w_i;
    input signed [inputWidth-1:0] in_i;
    input signed [adderWidth-1:0] sumIN_i;
    output signed [weightWidth-1:0] w_o;
    output signed [inputWidth-1:0] in_o;
    output signed [adderWidth-1:0] sumOUT_o;

    reg signed [weightWidth-1:0] w_reg;
    reg signed [inputWidth-1:0] in_reg;
    reg signed [adderWidth-1:0] sumOUT_reg;

    wire signed [prodWidth-1:0] prod_w;
    wire signed [adderWidth-1:0] sumOUT_w;
```

Systolic Mac Array

- MAC unit

```
    assign sumOUT_w = w_reg * in_i + sumIN_i;

    always @ (posedge clk or negedge rst_n) begin
        if (!rst_n) begin
          in_reg <= {multWidth{1'b0}};
          sumOUT_reg <= {adderWidth{1'b0}};
        end
        else begin
          in_reg <= in_i;
          sumOUT_reg <= sumOUT_w;
        end
    end

    always @ (posedge clk or negedge rst_n) begin
        if (!rst_n) begin
          w_reg <= {weightWidth{1'b0}};
        end
        else begin
          if (newWeights_i) w_reg <= w_i;
        end
    end

    assign in_o = in_reg;
    assign sumOUT_o = sumOUT_reg;
    assign w_o = w_reg;

endmodule
```

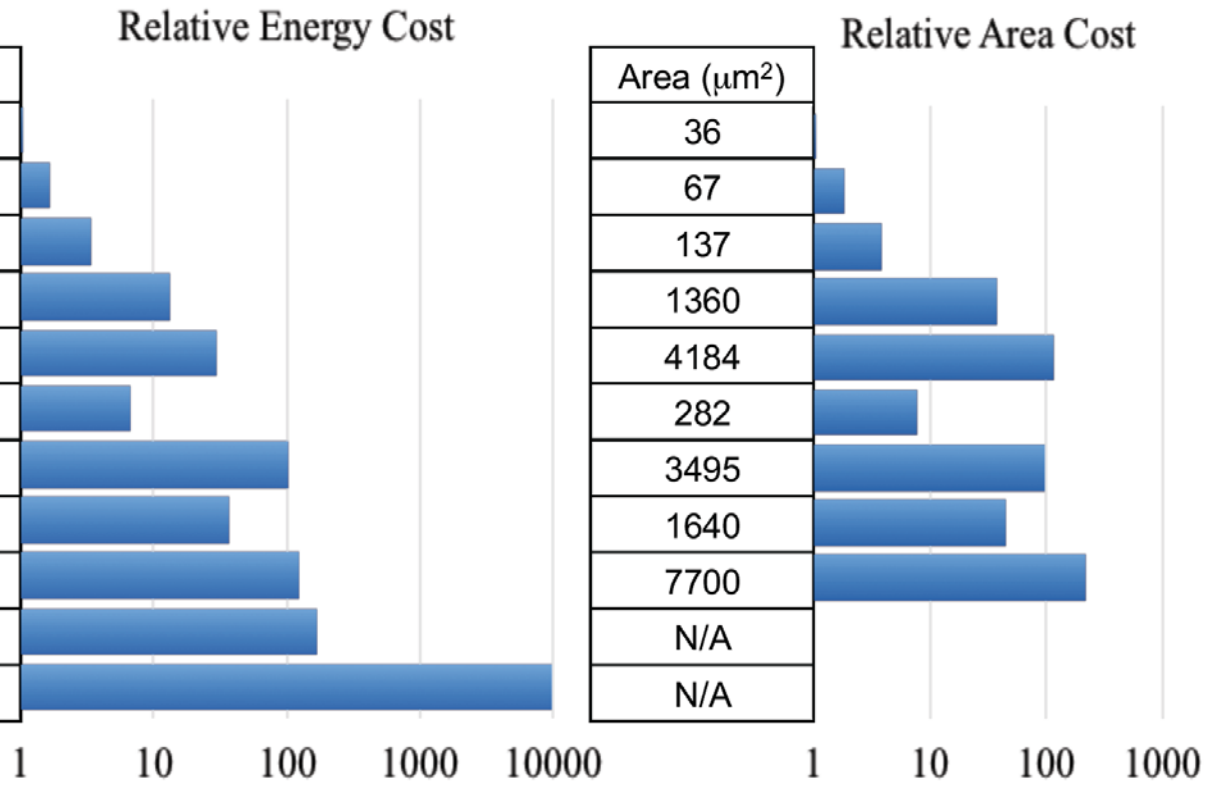
Number Representation



src: K. Pekmestzi, NTUA

Cost of Operations

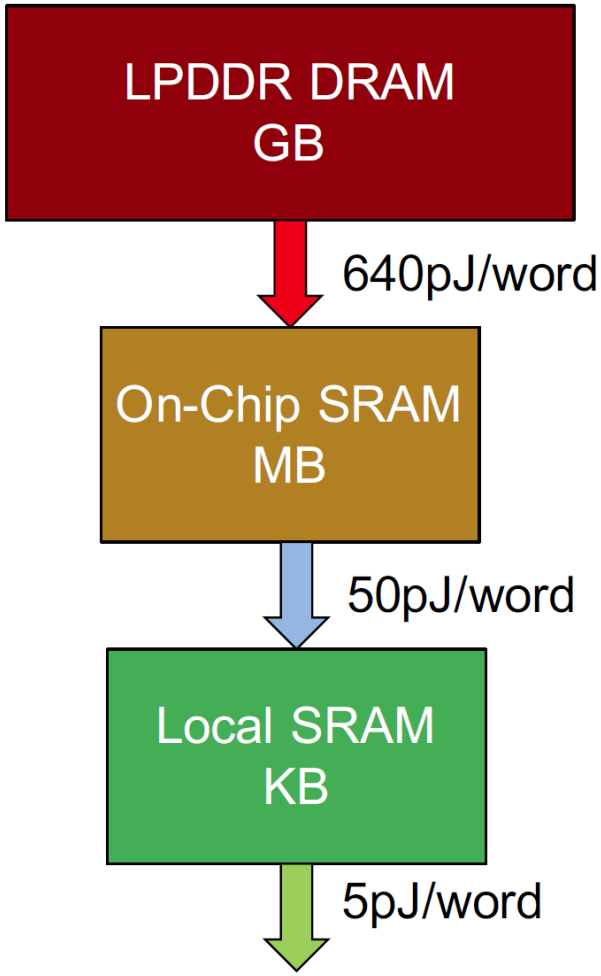
Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A



Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

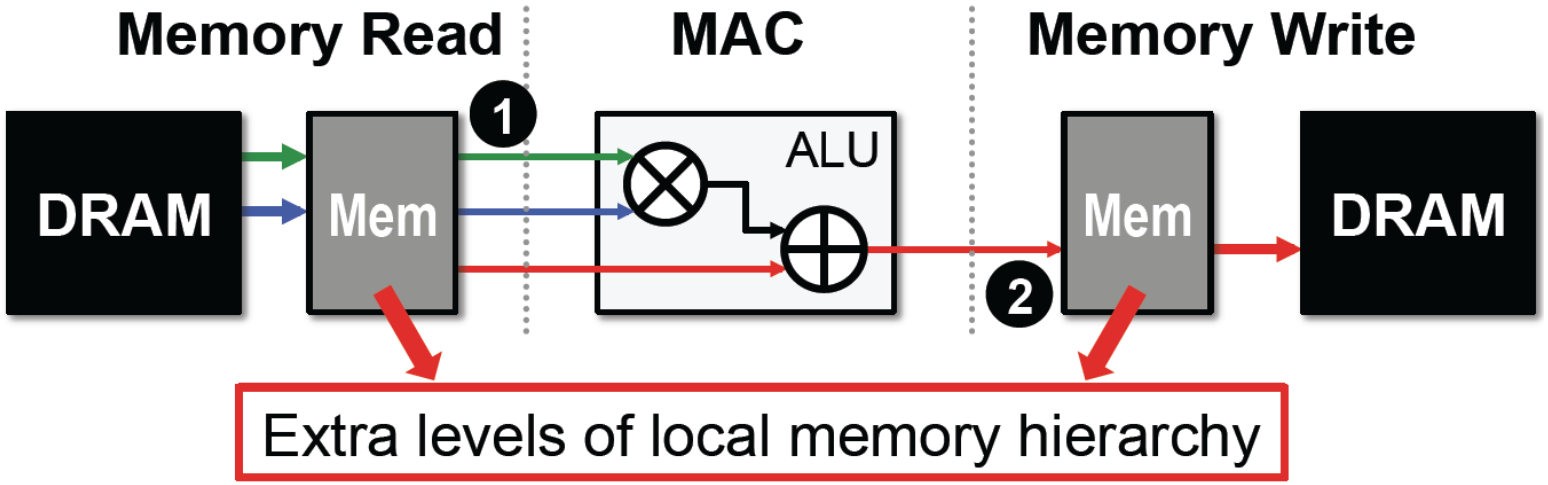
src: K. Pekmestzi, NTUA

The Importance of Staying Local



src: K. Pekmestzi, NTUA

Memory Access is the Bottleneck



Opportunities: ❶ **data reuse** ❷ **local accumulation**

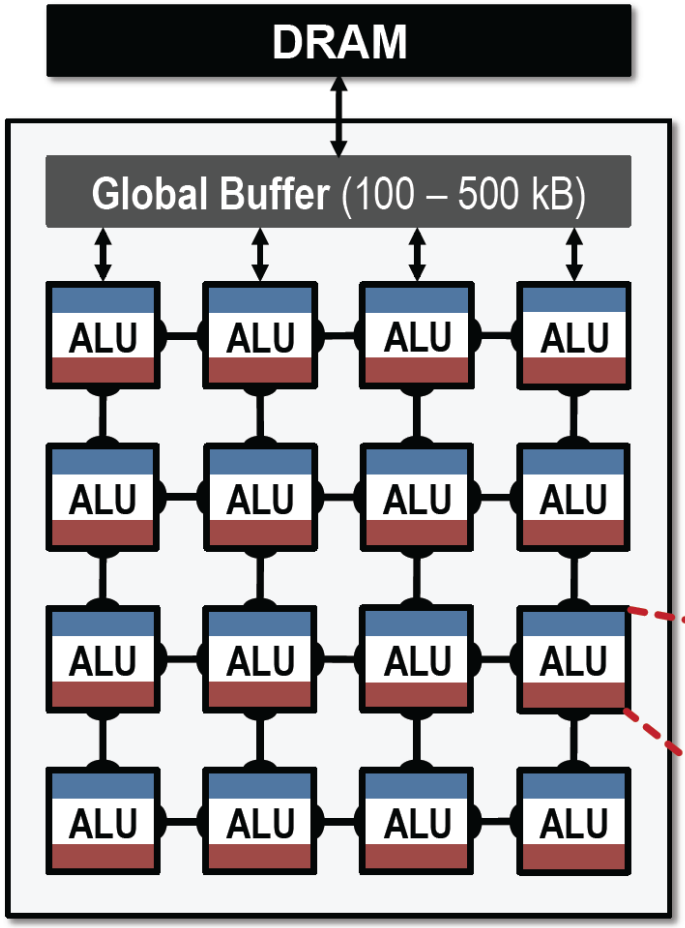
❶ Can reduce DRAM reads of filter/fmap by up to **500×**

❷ Partial sum accumulation does **NOT** have to access DRAM

Example: AlexNet has **724M** MACs => **2896M** with DRAM accesses which can be reduced to **61M** (best case).

src: K. Pekmestzi, NTUA

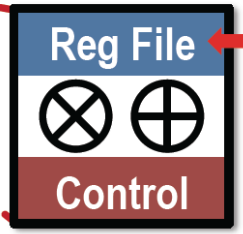
Spatial Architecture for DNN



Local Memory Hierarchy

- Global Buffer
- Direct inter-PE network
- PE-local memory (RF)

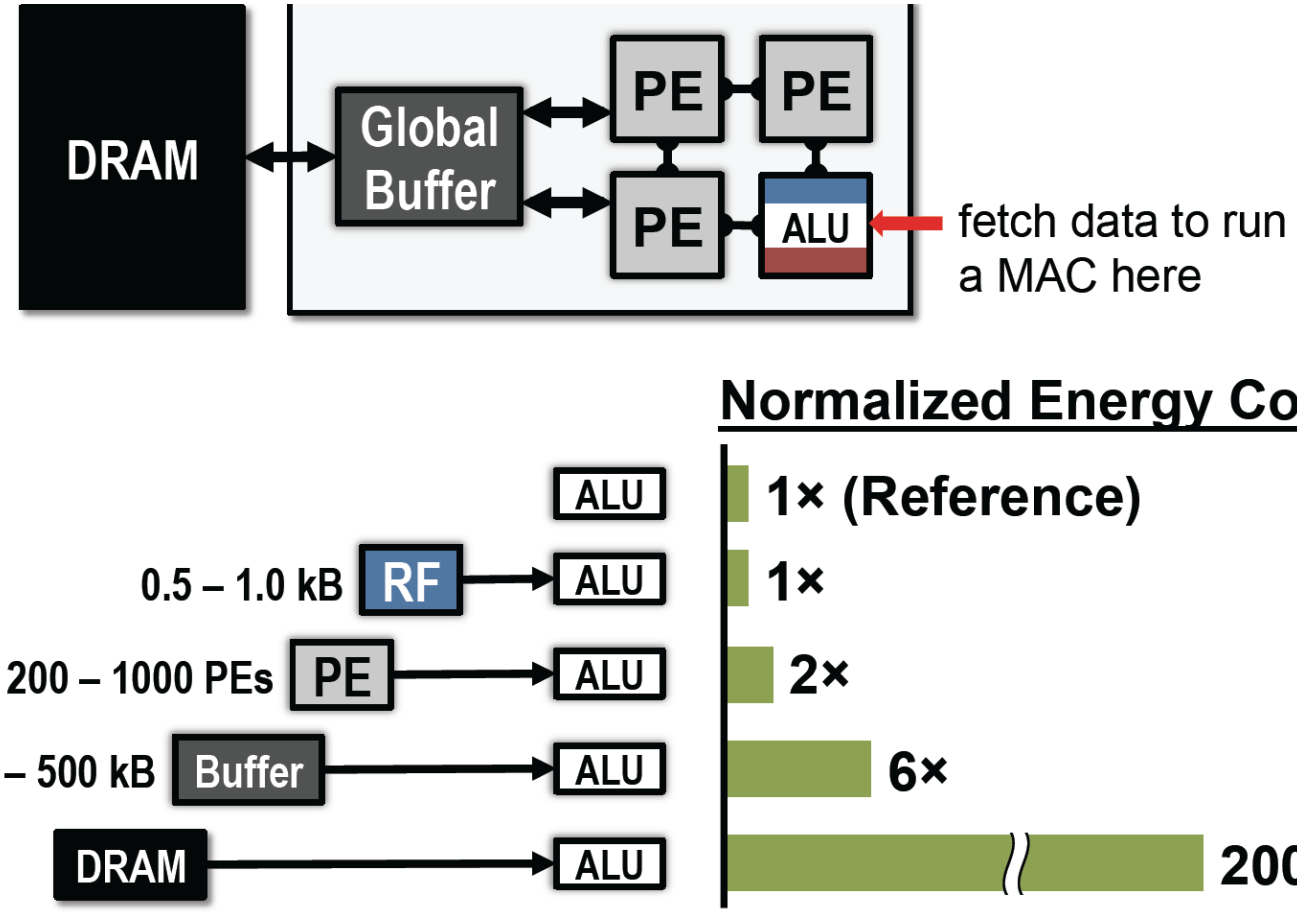
Processing Element (PE)



0.5 – 1.0 kB

src: K. Pekmestzi, NTUA

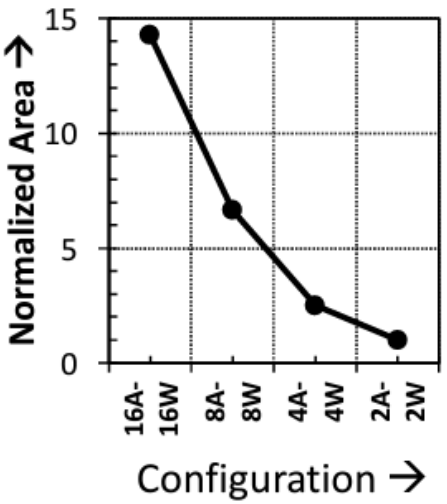
Low-Cost Local Data Access

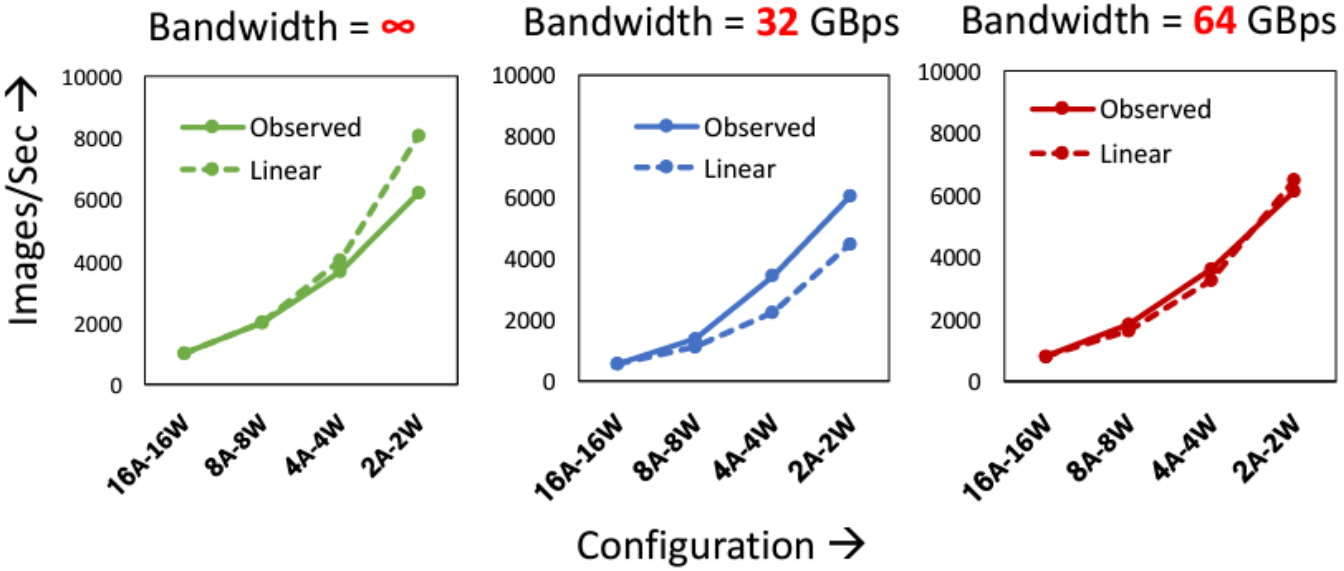


* measured from a commercial 65nm process

src: K. Pekmestzi, NTUA

Reduced Precision for Inference





J. Choi, et al., “PACT: Parameterized Clipping Activation for Quantized Neural Networks” 2018

Reduced Precision for Inference

Network	FullPrec	DoReFa				PACT			
		2b	3b	4b	5b	2b	3b	4b	5b
CIFAR10	0.916	0.882	0.899	0.905	0.904	0.897	0.911	0.913	0.917
SVHN	0.978	0.976	0.976	0.975	0.975	0.977	0.978	0.978	0.979
AlexNet	0.551	0.536	0.550	0.549	0.549	0.550	0.556	0.557	0.557
ResNet18	0.702	0.626	0.675	0.681	0.684	0.644	0.681	0.692	0.698
ResNet50	0.769	0.671	0.699	0.714	0.714	0.722	0.753	0.765	0.767

J. Choi, et al., “PACT: Parameterized Clipping Activation for Quantized Neural Networks” 2018

Arithmetic Circuits

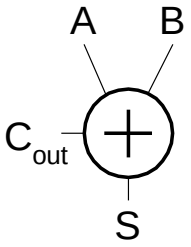
Adders

Single-Bit Addition

Half Adder

$S = A \oplus B$

$C_{out} = A \cdot B$

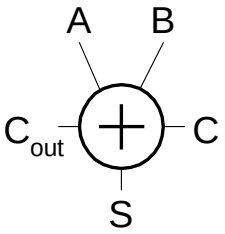


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

Full Adder

$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A, B, C)$$



A	B	C	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

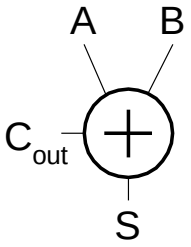
src: https://cmosvlsi.com/

Single-Bit Addition

Half Adder

$S = A \oplus B$

$C_{out} = A \cdot B$

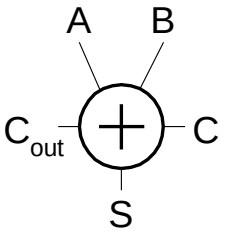


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder

$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A, B, C)$$



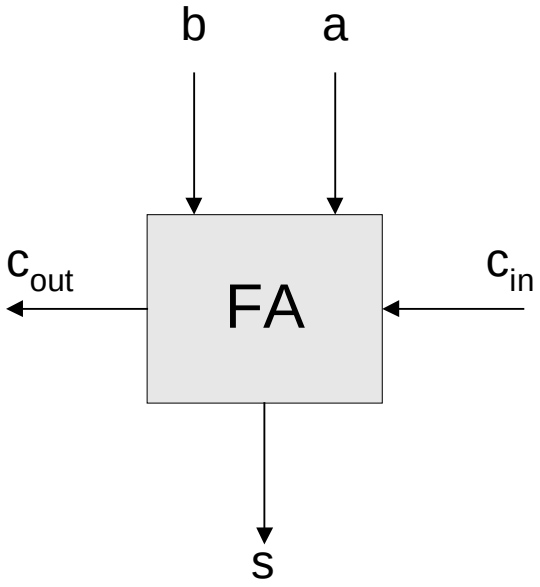
A	B	C	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

src: https://cmosvlsi.com/

Single-Bit Addition

- Full adder

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



s:

c_{in} \ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

c_{out}:

c_{in} \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = ab + ac_{in} + bc_{in}$$

$$s = c_{in}\bar{a}\bar{b} + \bar{a}\bar{c}_{in}b + bac_{in} + \bar{c}_{in}a\bar{b}$$

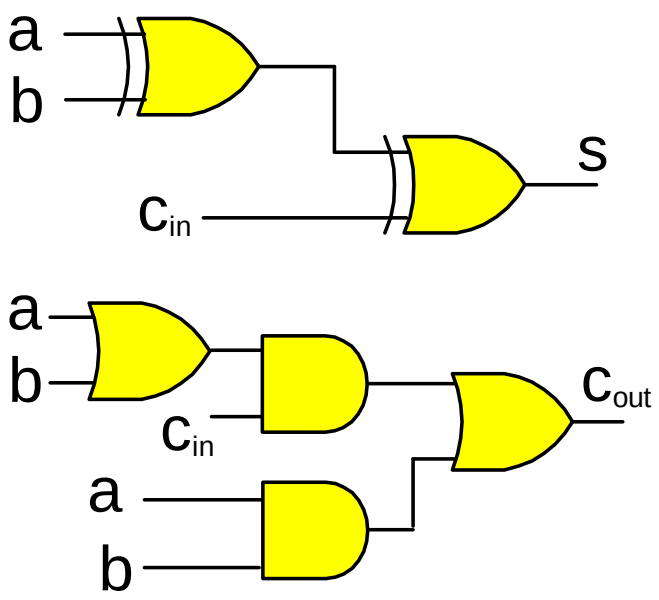
$$c_{out} = c_{in}b + ab + ac_{in}$$

src: K. Pekmestzi, NTUA

Full Adder

- 1 bit adder

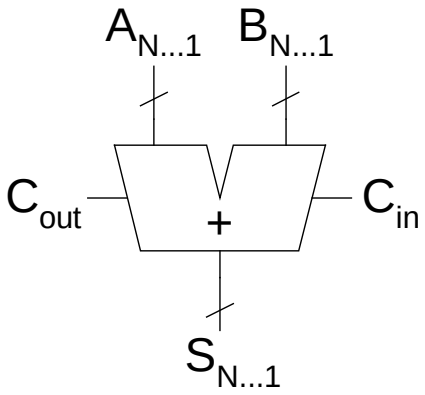
$$s = a \oplus b \oplus c_{in}$$
$$c_{out} = ab + c_{in}(a + b)$$

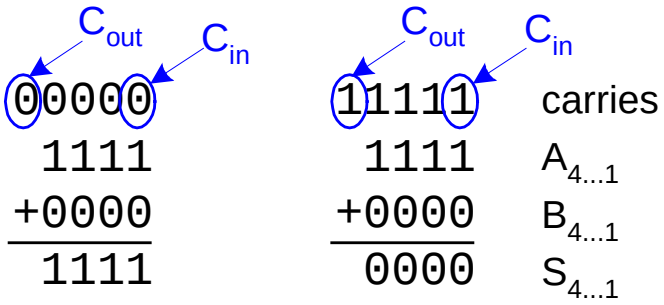


src: K. Pekmestzi, NTUA

Carry Propagate Adders

- N-bit adder called CPA
  - Each sum bit depends on all previous carries
  - How do we compute all these carries quickly?

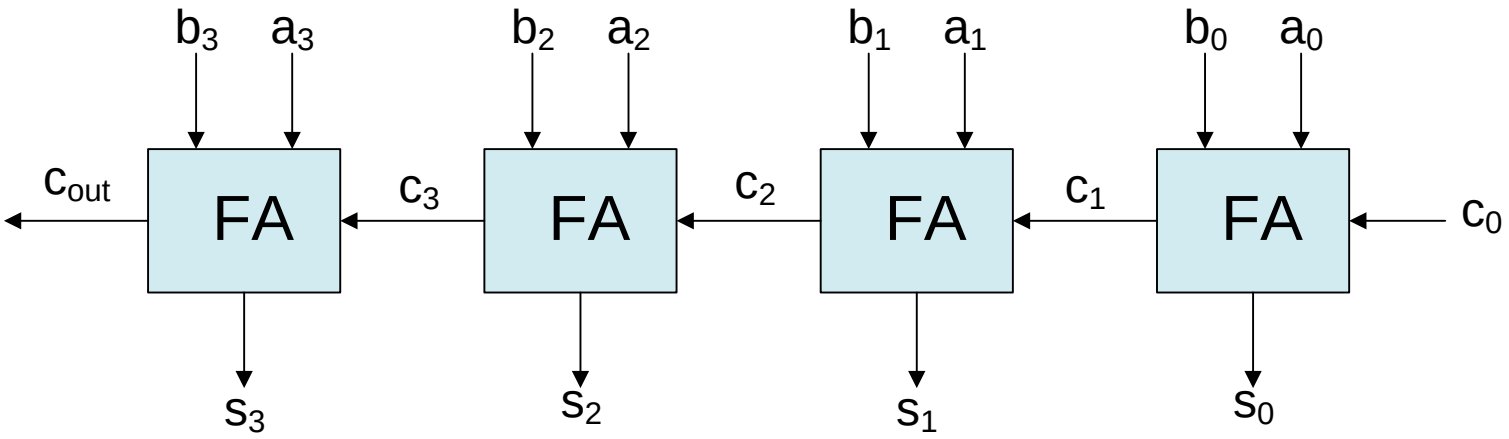




src: https://cmosvlsi.com/

RCA 4-bit

- 4-bit Ripple Carry Adder (RCA)

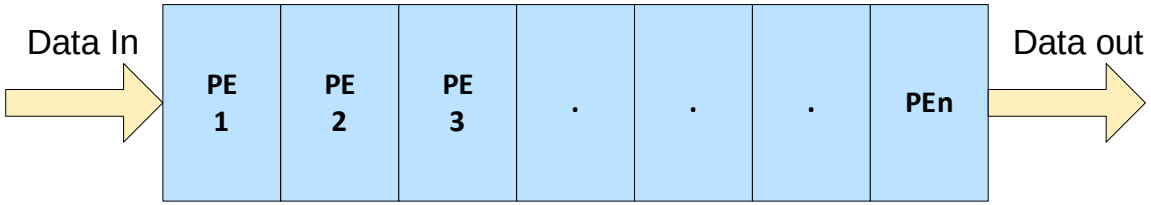


- Delay=$n \cdot T_{FA}$
  - On average the delay is practically Delay=$(\log_2 n) \cdot T_{FA}$
  - If the delays of S and Cout are different ($T_S > T_{Cout}$) then Delay=$T_S + (n-1) \cdot T_{Cout}$

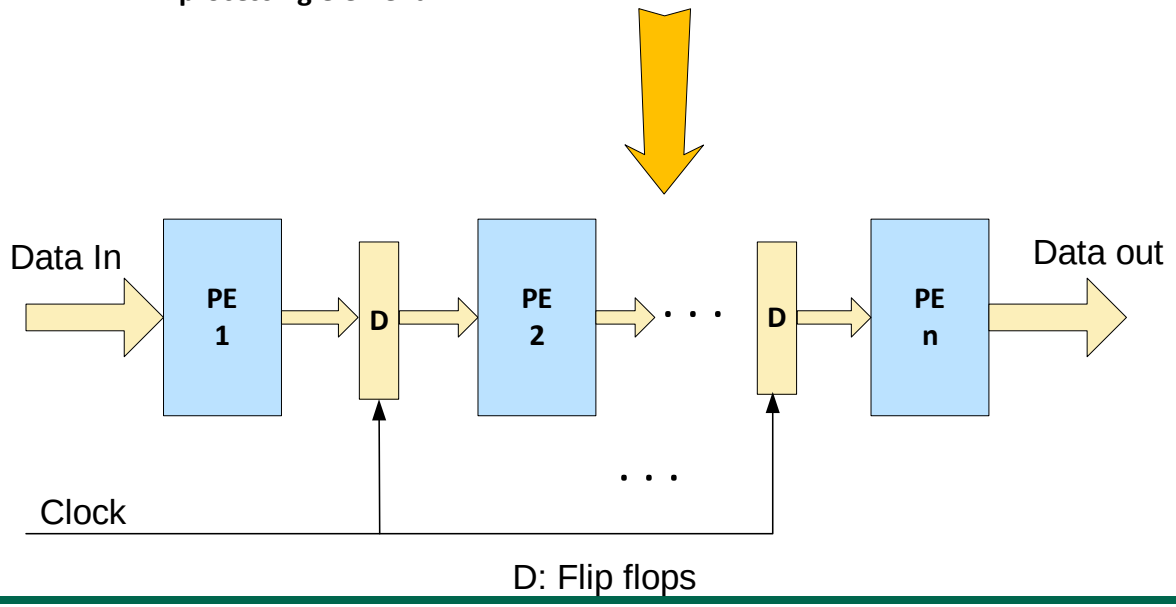
src: K. Pekmestzi, NTUA

Parallel to Systolic

- Converting a parallel circuit to a systolic one
  - Connections between PEs must pass through registers



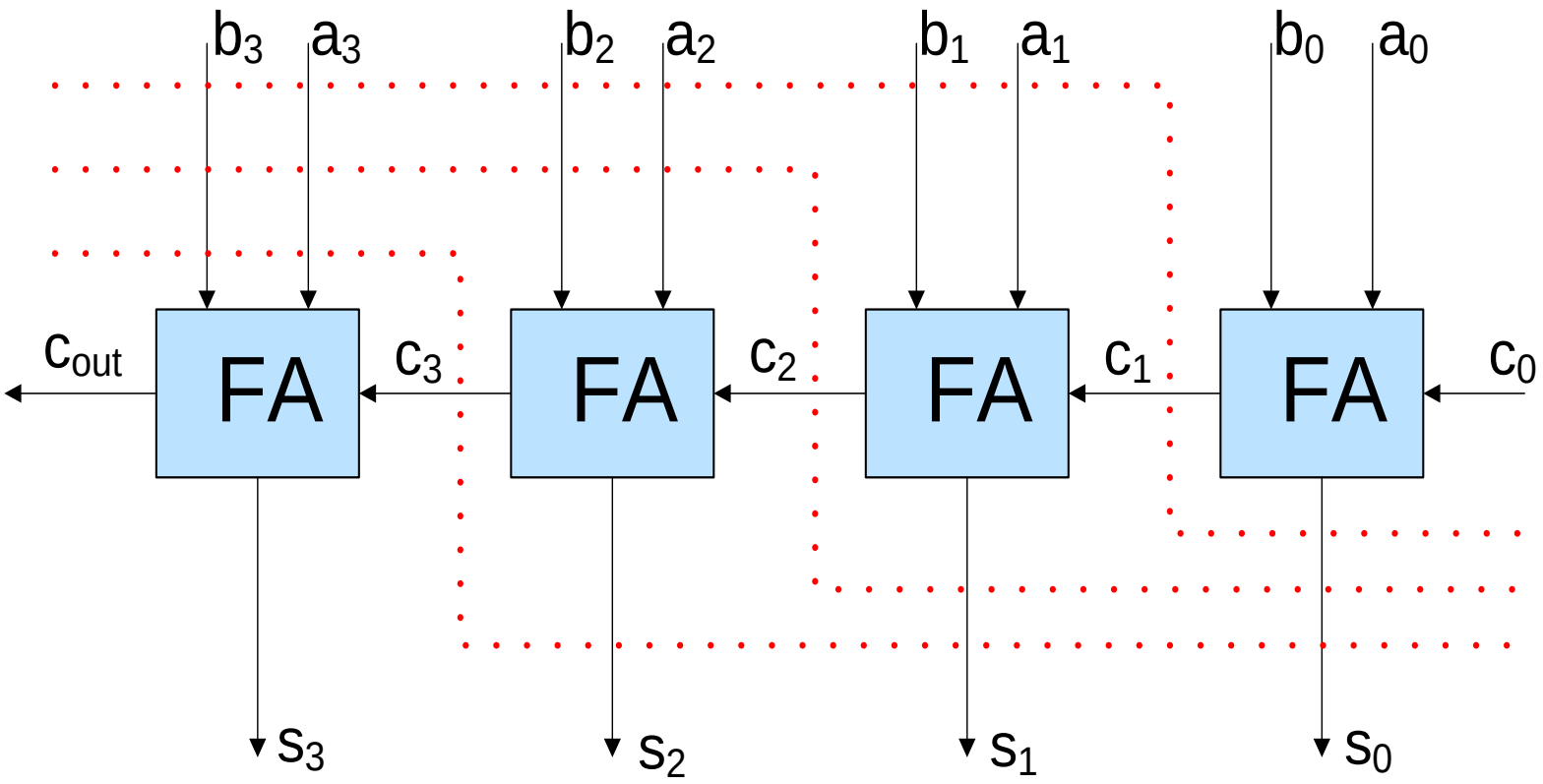
PE: processing element



src: K. Pekmestzi, NTUA

Parallel to Systolic

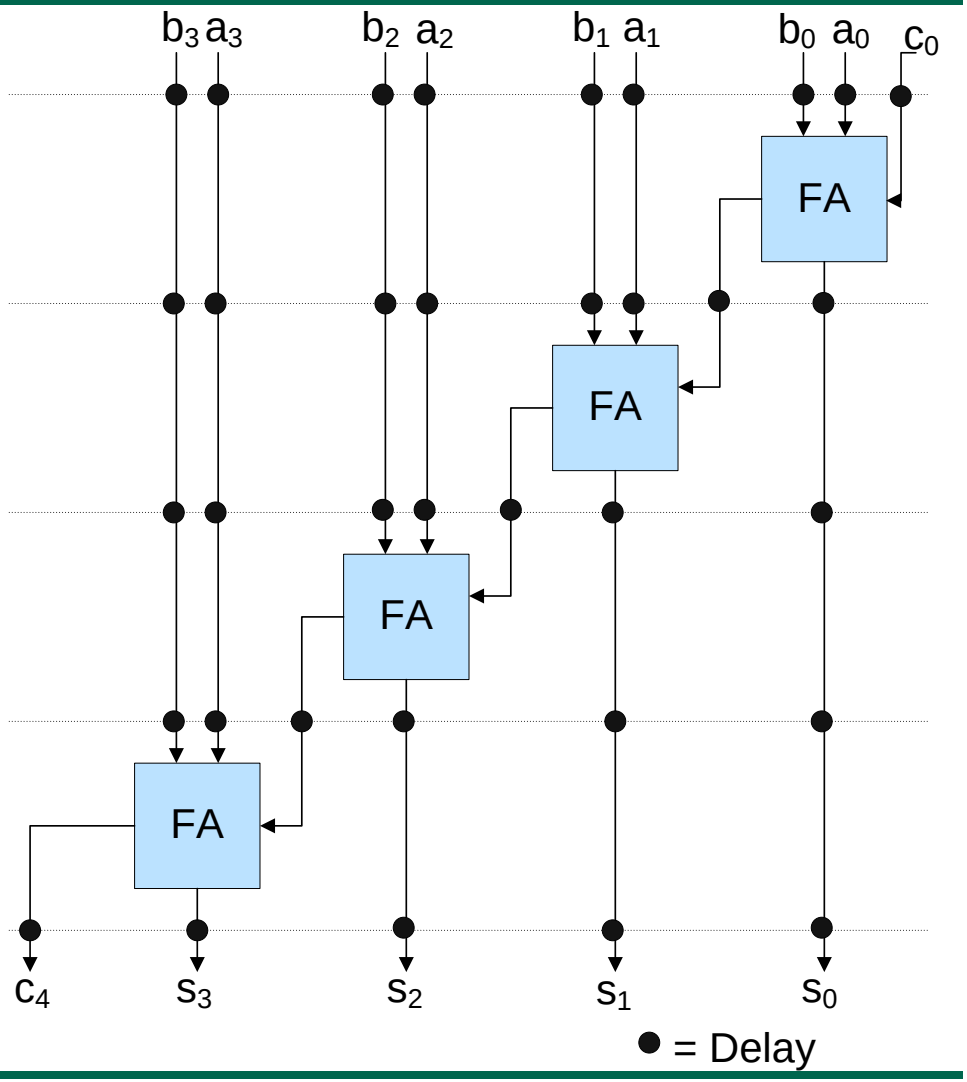
- Selecting delay points



src: K. Pekmestzi, NTUA

Systolic RCA

- 4 bit Systolic RCA

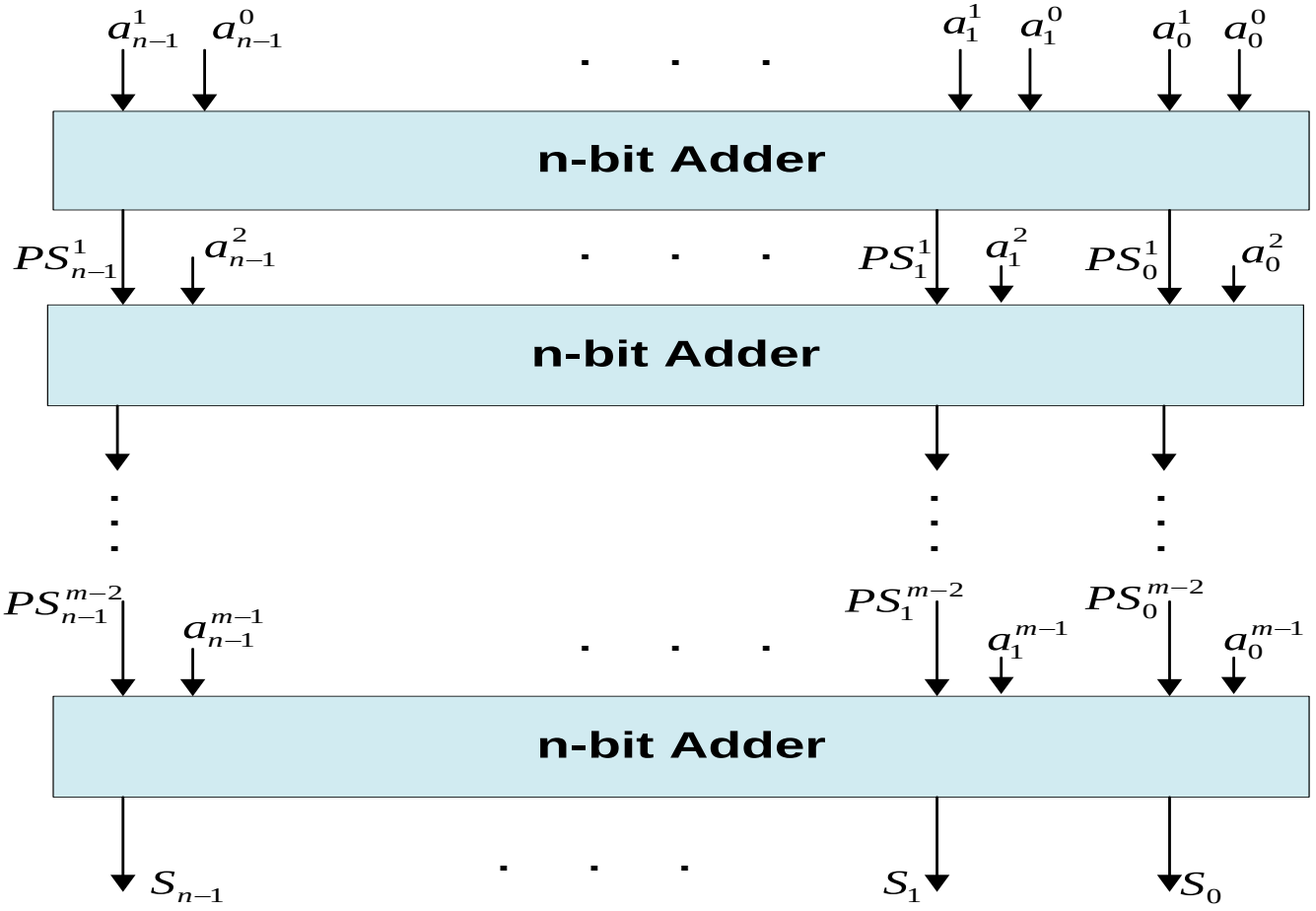


src: K. Pekmestzi, NTUA

Multi-operand Addition

- Add m n-bit numbers

$$S = \sum_{j=0}^{m-1} A^j$$

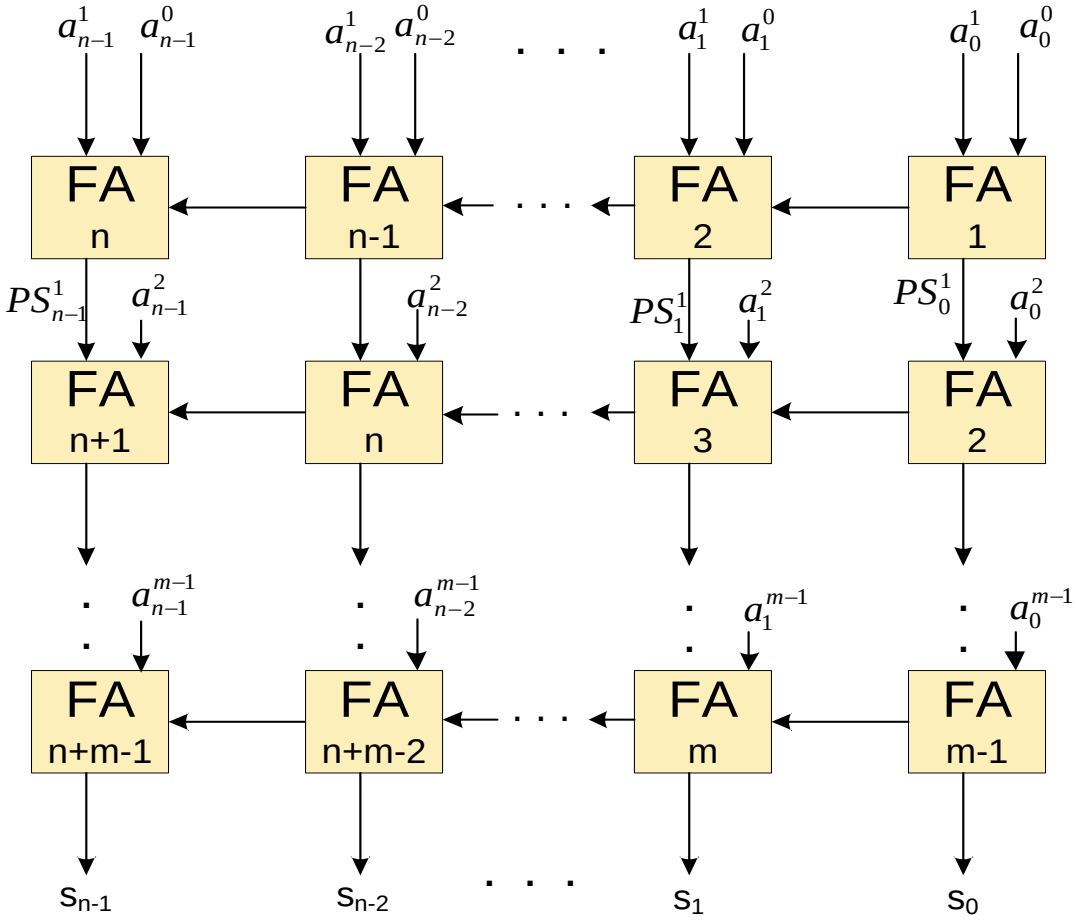


src: K. Pekmestzi, NTUA

Multi-operand Addition

- Delay T= (n+m-2)T_{FA}
- T= (m-2)T_S + nT_c

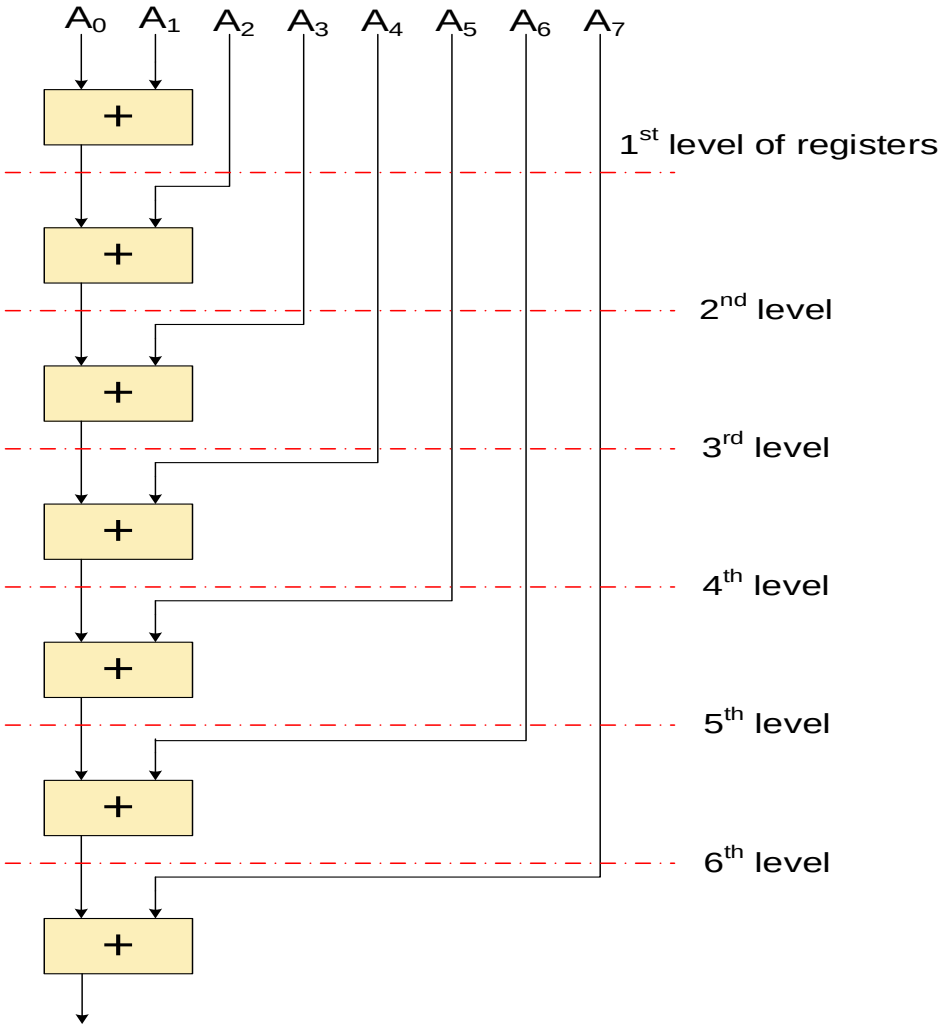
$$S = \sum_{j=0}^{m-1} A^j$$



src: K. Pekmestzi, NTUA

Systolic Multi-operand Addition

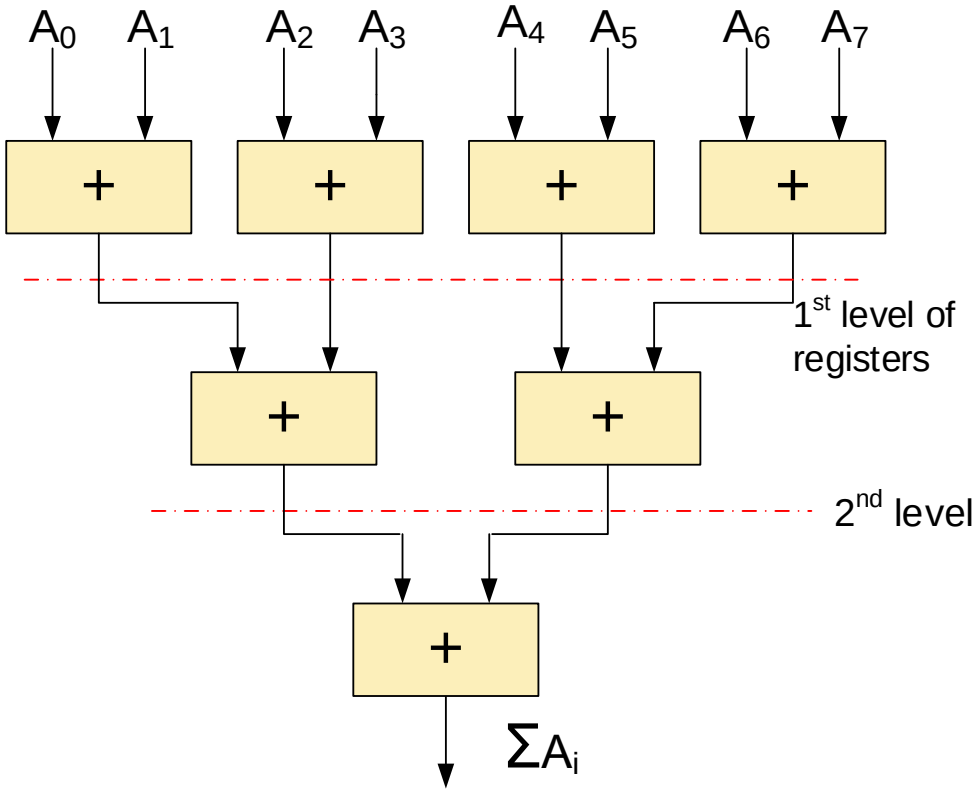
- m=8 summands
- Number of registers =m(m-1)/2=28
- Latency: $T = (m-1) \cdot T_{Clock} = 7 \cdot T_{Clock}$



src: K. Pekmestzi, NTUA

Systolic Multi-operand Addition

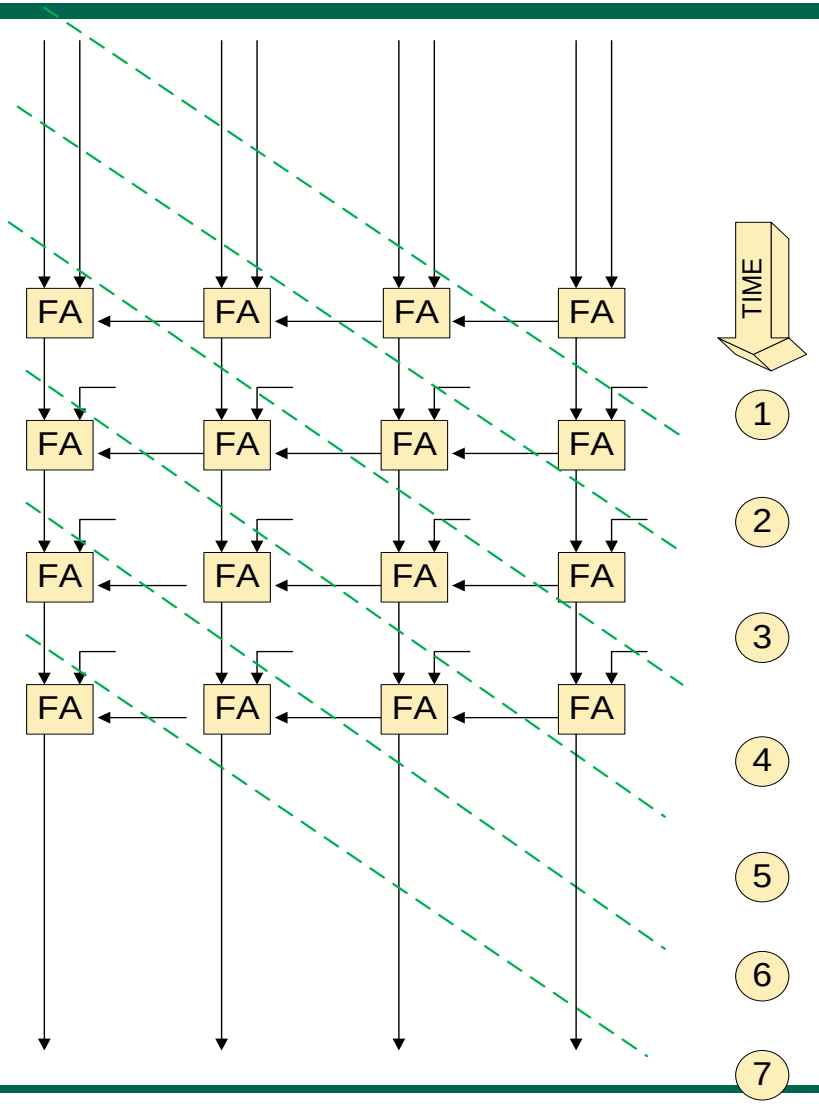
- Adder Tree
- m=8 summands
- Number of registers =(m-1)=7
- Latency: $T = \log_2 m \cdot T_{clock} = 3 \cdot T_{clock}$



src: K. Pekmestzi, NTUA

5x4 Systolic RCA

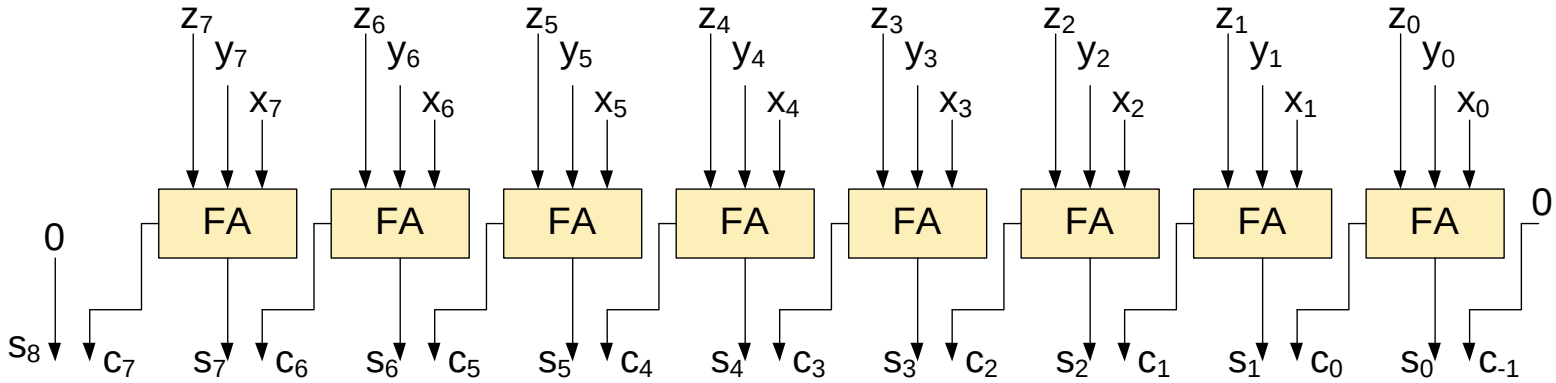
- Latency: $T = (n+m-2) \cdot T_{clock}$
- Frequency:
  $F = 1/(T_{FA}+T_D) = 1/T_{clock}$
  - T_D flip-flop induced delay

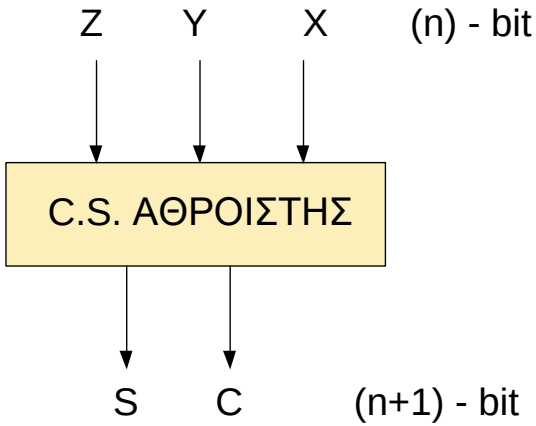


src: K. Pekmestzi, NTUA

Carry-save

- Carry-save operation



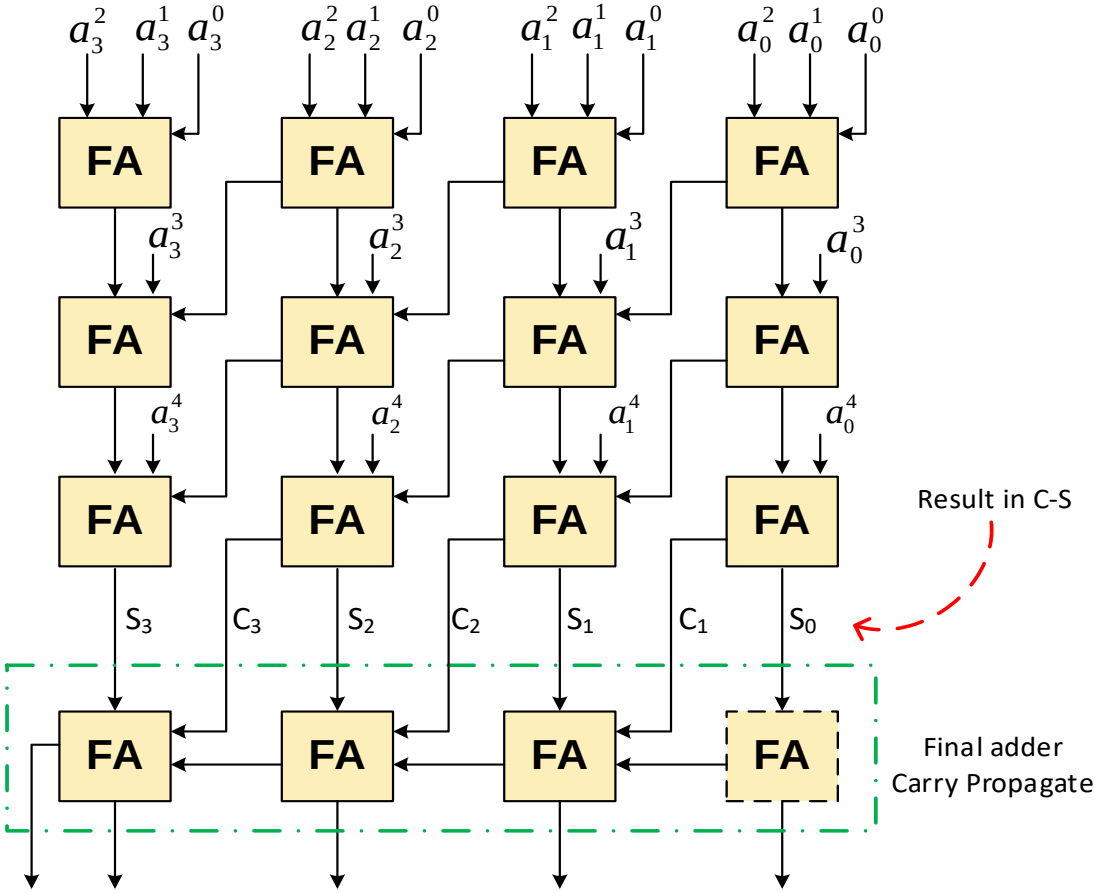


src: K. Pekmestzi, NTUA

Carry-save Addition

- 5 operand carry-save addition

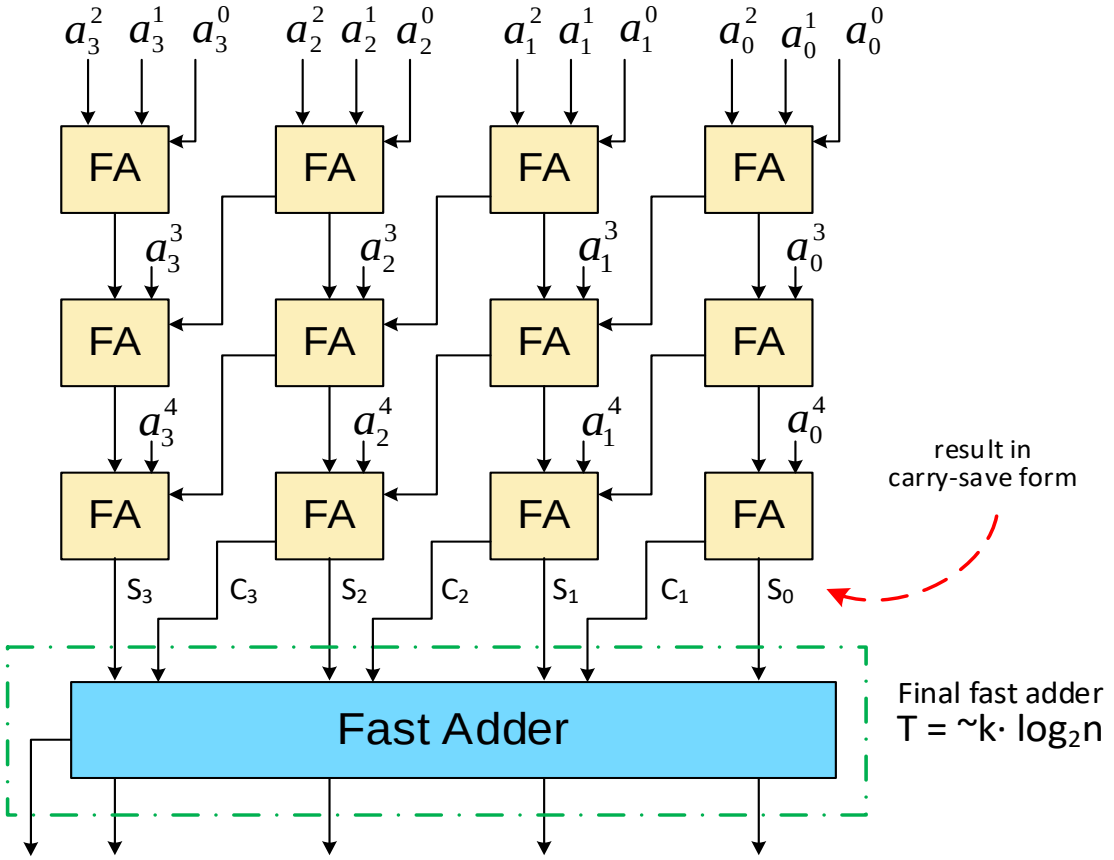
- Delay:
  T= (m-2)TFA+(n)TFA
  =(m+n-2)TFA
  - m=number of summands, n=bit size



src: K. Pekmestzi, NTUA

Carry-save Addition

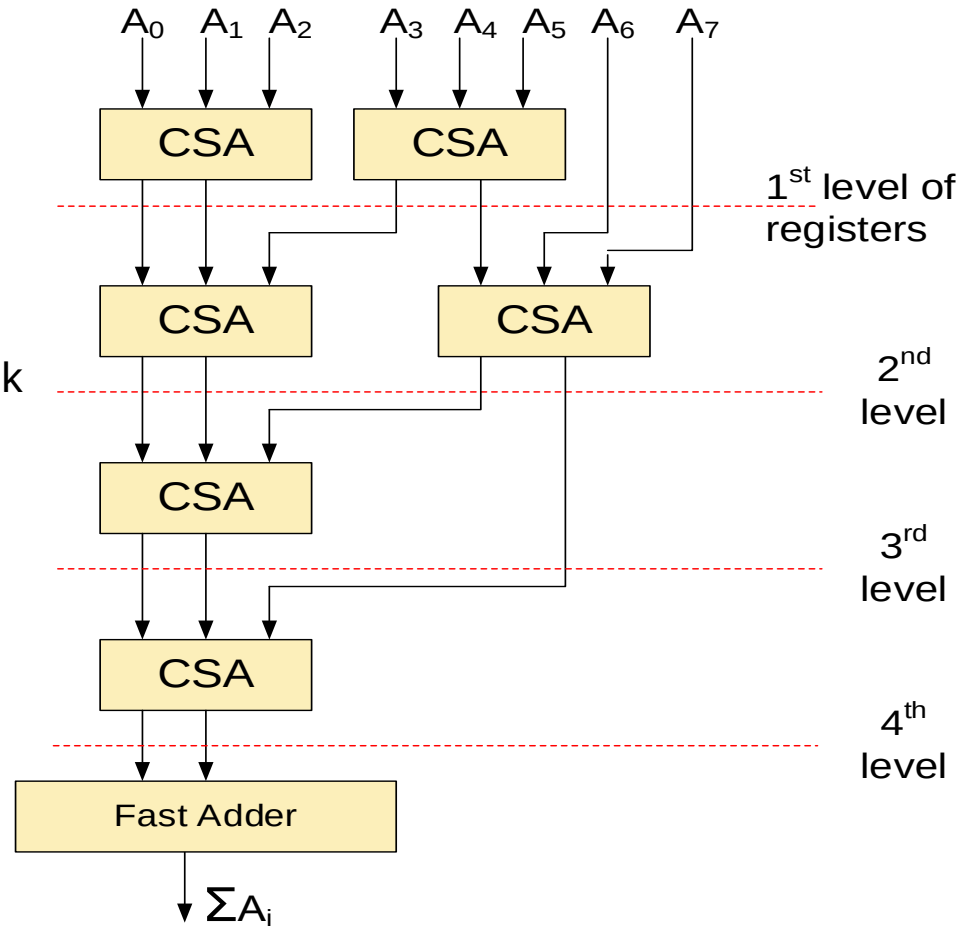
- 5 operand addition with carry-save
- Delay: $T=(m-2)T_{FA}+k\cdot \log_2 n T_{FA}$
  - m=number of summands, n=bit size



src: K. Pekmestzi, NTUA

Systolic Carry-save Addition

- 8 operand systolic carry-save addition
- Number of registers = 2m-3=15
- Latency = $\log_{3/2}m \cdot T_{Clock}$ = $1.7\log_2 m \cdot T_{Clock} = 5 \cdot T_{Clock}$



src: K. Pekmestzi, NTUA

Latency

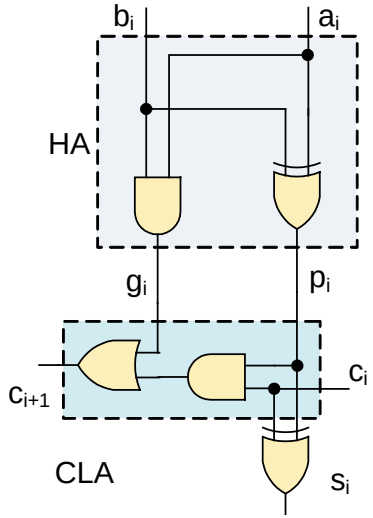
- Multi-operand addition latency

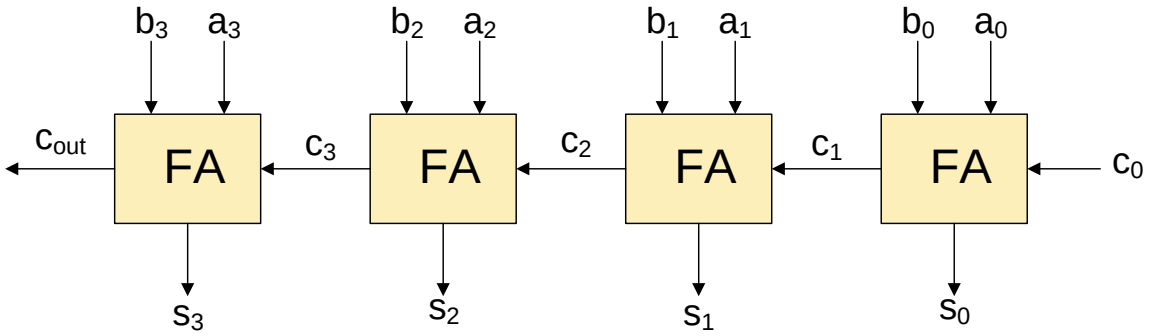
size m	Serial m-1	Tree $\log_2 m$	CS Tree $1.7\log_2 m$
8	7	3	5
12	11	4	5
16	15	4	7
32	31	5	9
64	63	6	10

src: K. Pekmestzi, NTUA

Carry Propagation

- Full adder: $s_i=(a_i \oplus b_i) \oplus c_i$ and $c_{i+1}=a_i \cdot b_i + (a_i+b_i) \cdot c_i$
- Denoting $p_i = a_i \oplus b_i$ and $g_i = a_i \cdot b_i$
  → $s_i=p_i \oplus c_i$ and $c_{i+1} = g_i + p_i \cdot c_i$
- Carry out:
  - If gi=1 → a carry is generated
  - If pi=1 → carry in is propagated





src: K. Pekmestzi, NTUA

PGK

- For a full adder, define what happens to carries
  - Generate: $C_{out} = 1$ independent of C

    $G = A \bullet B$
  - Propagate: $C_{out} = C$

    $P = A \oplus B$
  - Kill: $C_{out} = 0$ independent of C

    $K = \sim A \bullet \sim B$

src: K. Pekmestzi, NTUA

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} =$$

$$P_{i:j} =$$

- Base case

$$G_{i:i} \equiv$$

$$P_{i:i} \equiv$$

$$G_{0:0} \equiv$$

$$P_{0:0} \equiv$$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} =$$

- Base case

$$G_{i:i} \equiv$$

$$P_{i:i} \equiv$$

$$G_{0:0} \equiv$$

$$P_{0:0} \equiv$$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$G_{i:i} \equiv$ $\quad$ $G_{0:0} \equiv$

$P_{i:i} \equiv$ $\quad$ $P_{0:0} \equiv$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i$$

$$P_{i:i} \equiv$$

$$G_{0:0} \equiv$$

$$P_{0:0} \equiv$$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i \qquad G_{0:0} \equiv$$

$$P_{i:i} \equiv P_i = A_i \oplus B_i \qquad P_{0:0} \equiv$$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i$$

$$P_{i:i} \equiv P_i = A_i \oplus B_i$$

$$G_{0:0} \equiv G_0 = C_{in}$$

$$P_{0:0} \equiv$$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i \qquad G_{0:0} \equiv G_0 = C_{in}$$

$$P_{i:i} \equiv P_i = A_i \oplus B_i \qquad P_{0:0} \equiv P_0 = 0$$

- Sum:

$$S_i =$$

src: https://cmosvlsi.com/

Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i$$

$$P_{i:i} \equiv P_i = A_i \oplus B_i$$

$$G_{0:0} \equiv G_0 = C_{in}$$

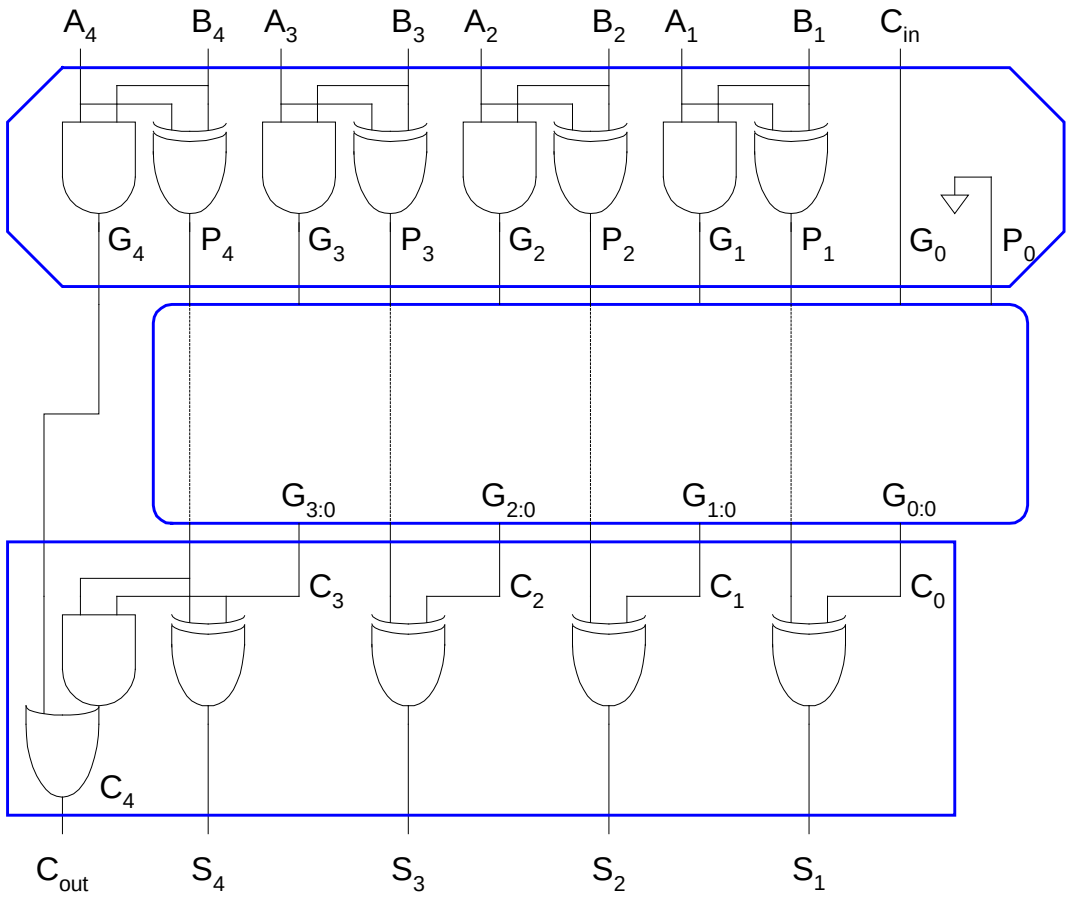
$$P_{0:0} \equiv P_0 = 0$$

- Sum:

$$S_i = P_i \oplus G_{i-1:0}$$

src: https://cmosvlsi.com/

PG Logic



1: Bitwise PG logic

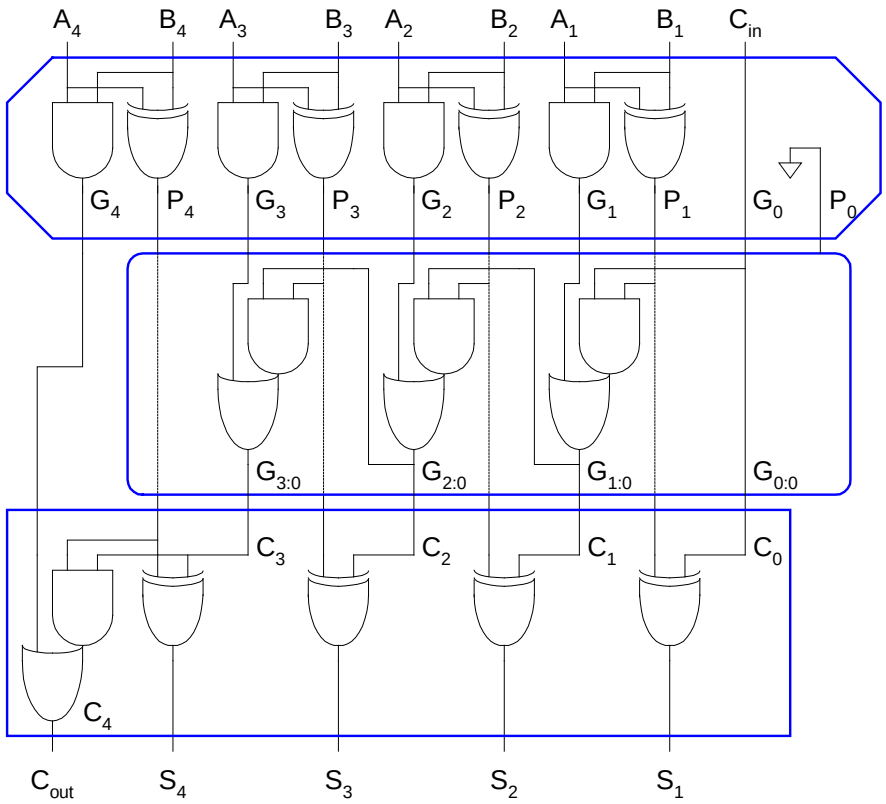
2: Group PG logic

3: Sum logic

src: https://cmosvlsi.com/

Carry-Ripple Revisited

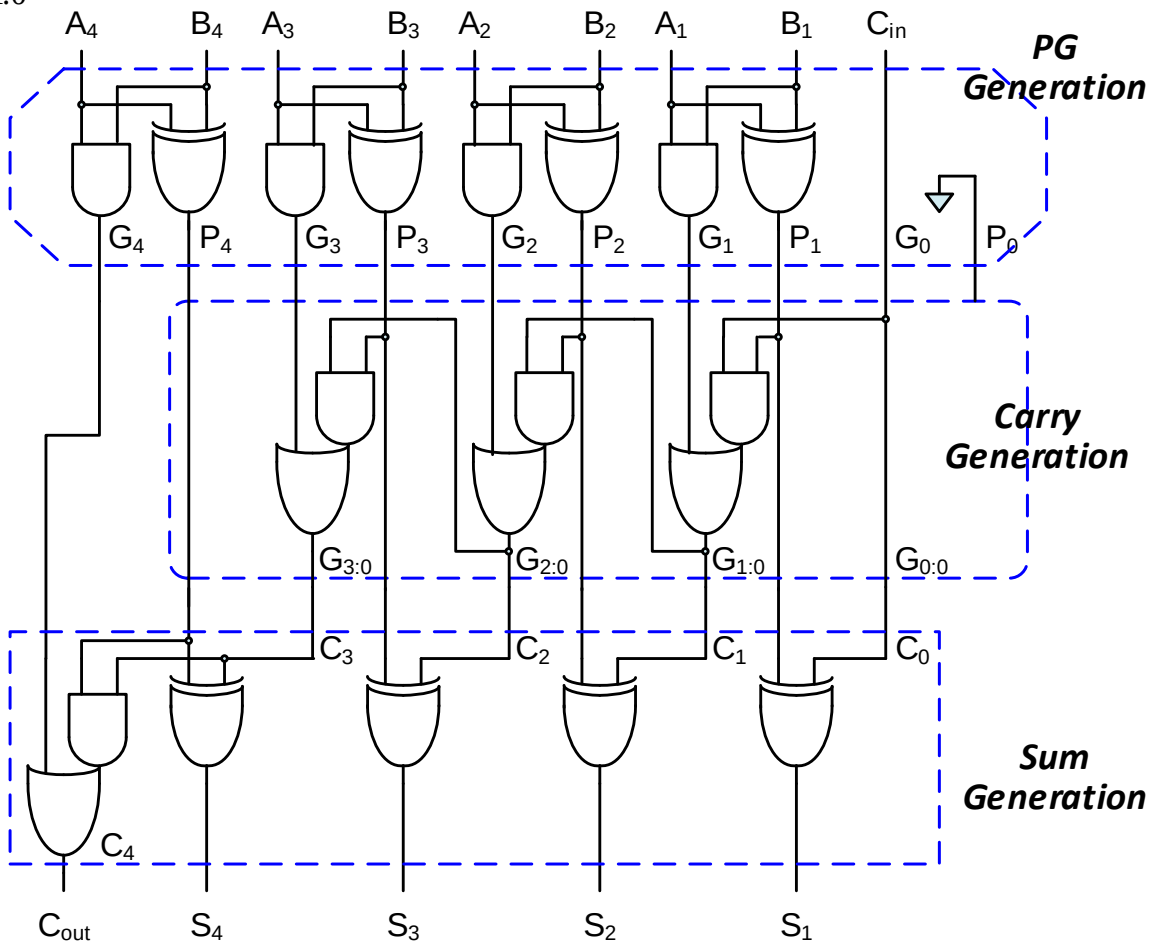
- $G_{i:0} = G_i + P_i \cdot G_{i-1:0}$



src: https://cmosvlsi.com/

RCA

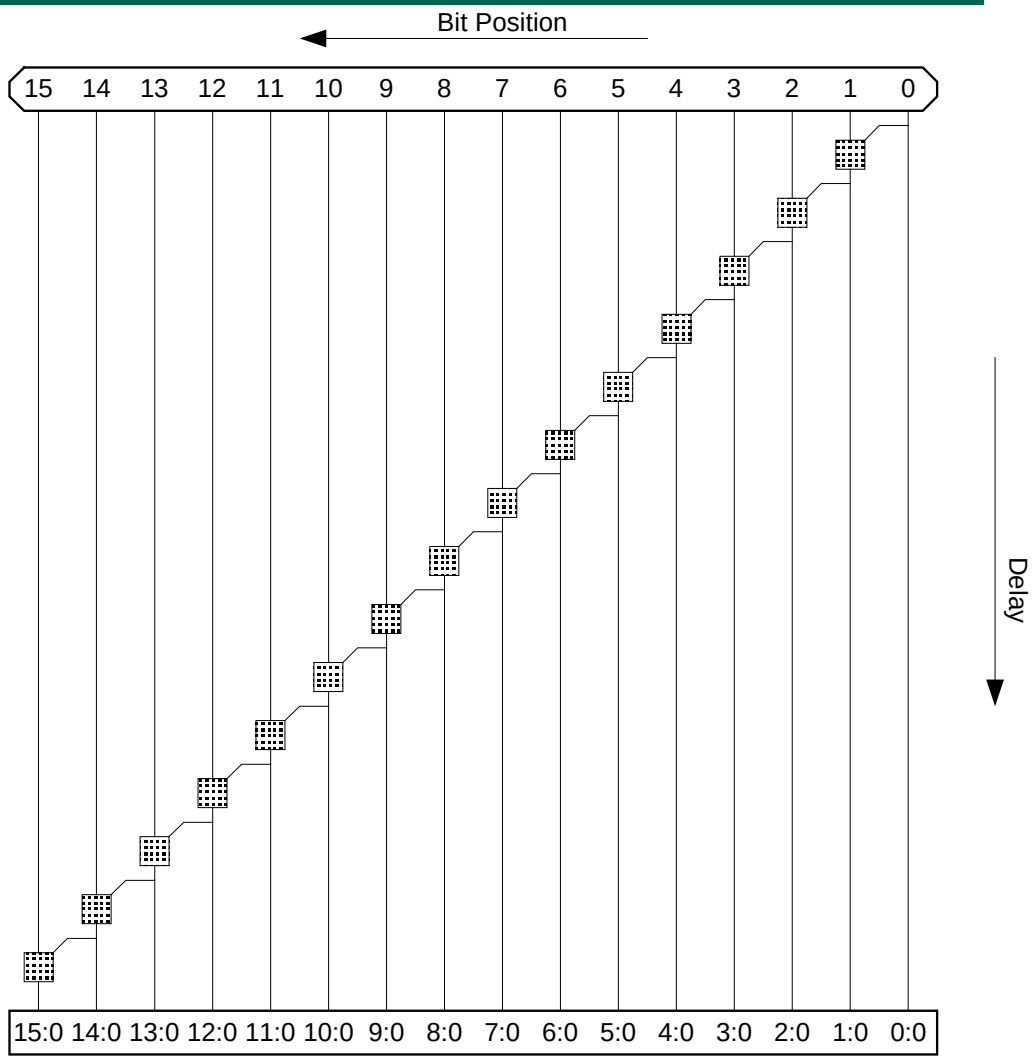
- $G_{i:0} = G_i + P_i \cdot G_{i-1:0}$



src: https://cmosvlsi.com/

Carry-Ripple PG Diagram

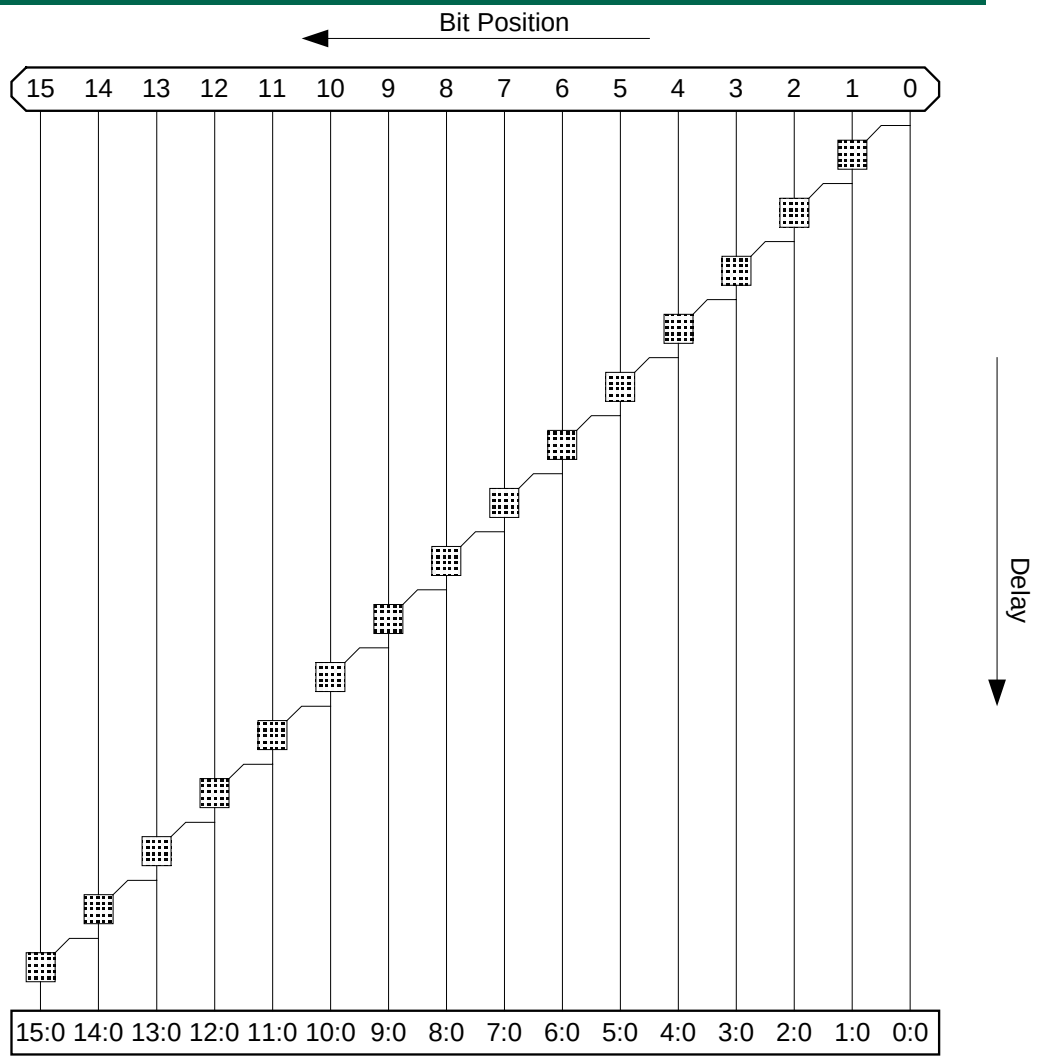
$t_{\text{ripple}} =$



src: https://cmosvlsi.com/

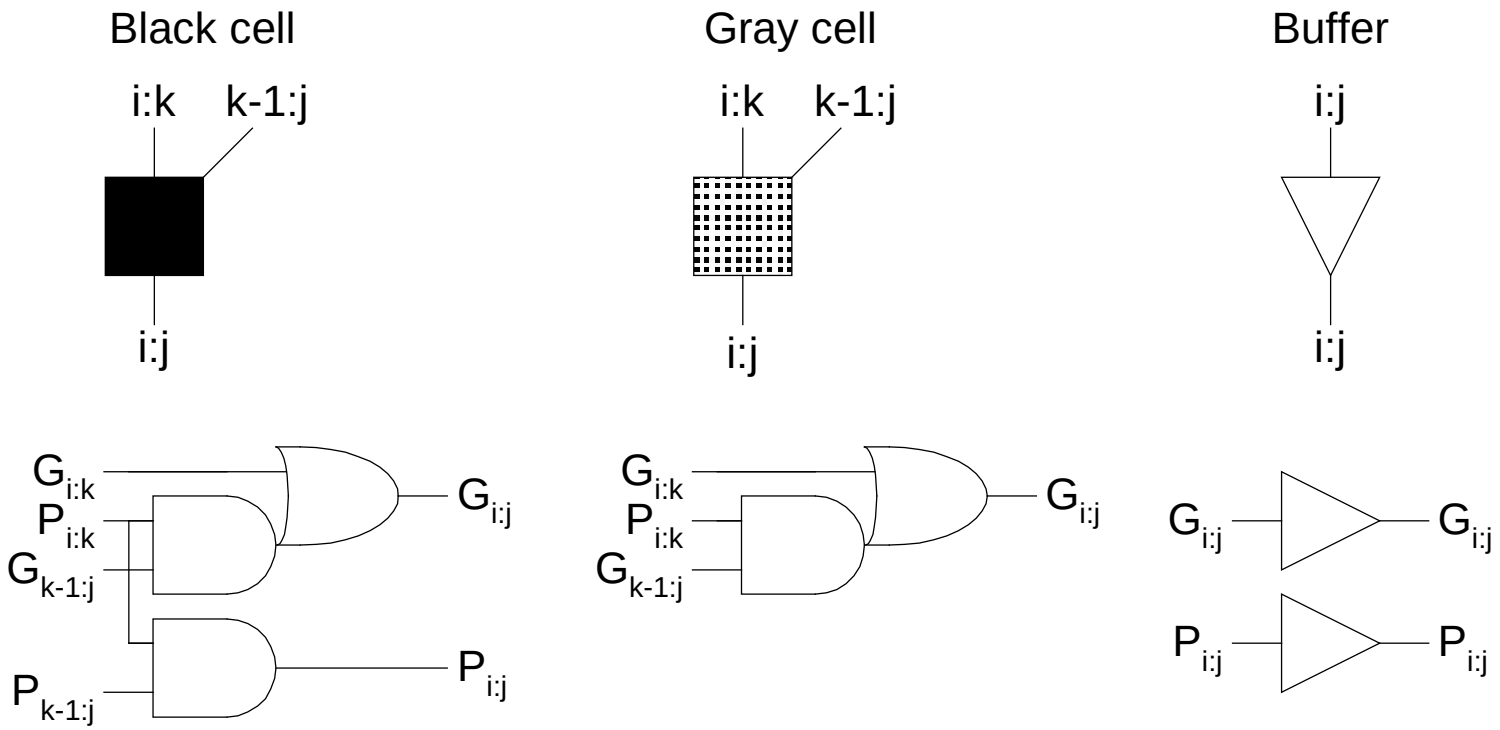
Carry-Ripple PG Diagram

$$t_{\text{ripple}} = t_{pg} + (N-1)t_{AO} + t_{\text{xor}}$$



src: https://cmosvlsi.com/

PG Diagram Notation



src: https://cmosvlsi.com/

Carry Prediction Adders

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$

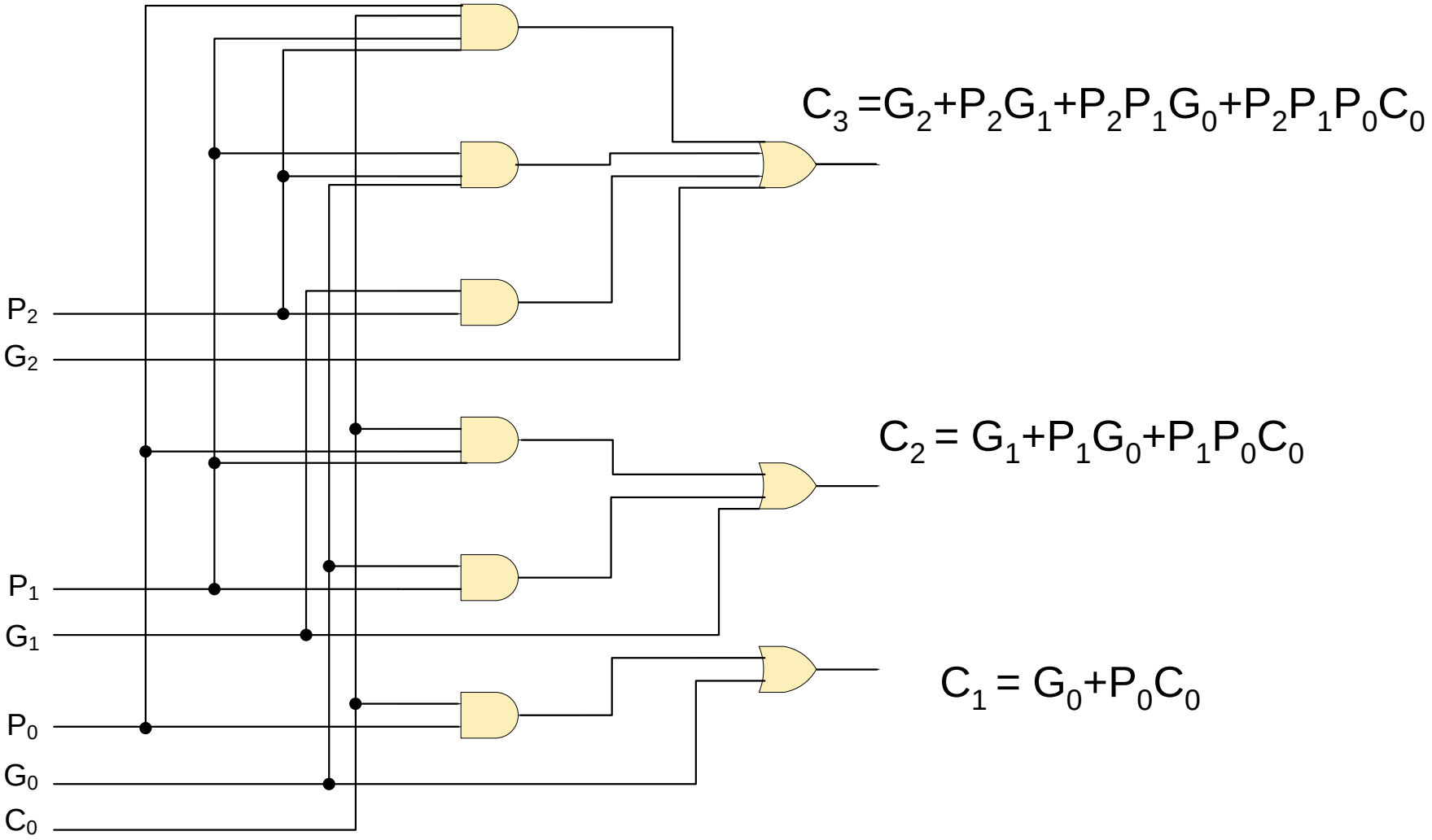
$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$

For n:

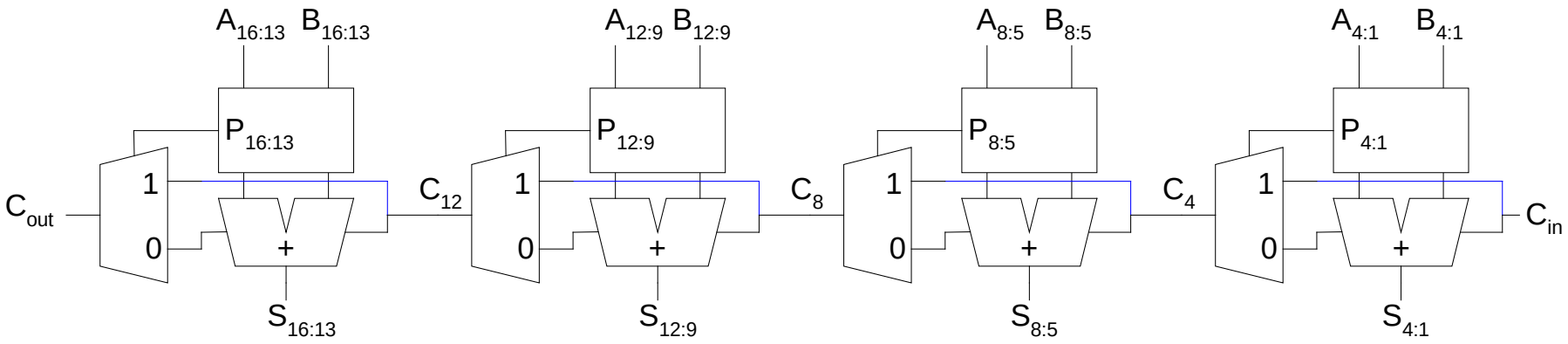
$$C_{n+1} = G_n + P_n(G_{n-1} + P_{n-1}(G_{n-2} + P_{n-2}(G_{n-3} + P_{n-3}(\ldots + P_2(G_1 + P_1 C_1))))) =$$
$$= G_n + P_n G_{n-1} + P_n P_{n-1} G_{n-2} + P_n P_{n-1} P_{n-2} G_{n-3} + \ldots + P_n P_{n-1} P_{n-2} \ldots P_2 G_1 + P_n P_{n-1} P_{n-2} \ldots P_2 P_1 C_1$$

Carry Prediction Adders



Carry-Skip Adder

- Carry-ripple is slow through all N stages
- Carry-skip allows carry to skip over groups of n bits
  - Decision based on n-bit propagate signal



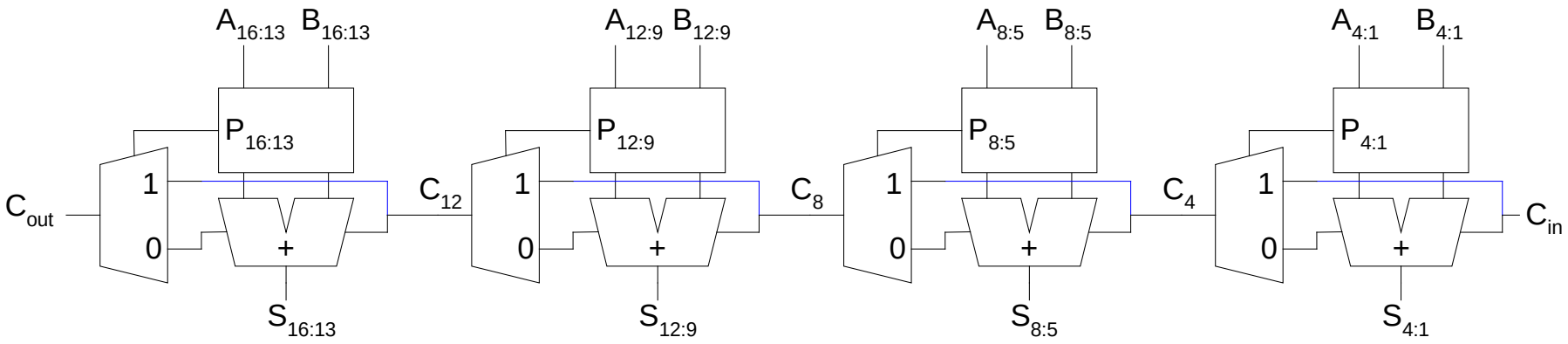
For k n-bit groups (N = nk)

$$t_{\text{skip}} =$$

src: https://cmosvlsi.com/

Carry-Skip Adder

- Carry-ripple is slow through all N stages
- Carry-skip allows carry to skip over groups of n bits
  - Decision based on n-bit propagate signal

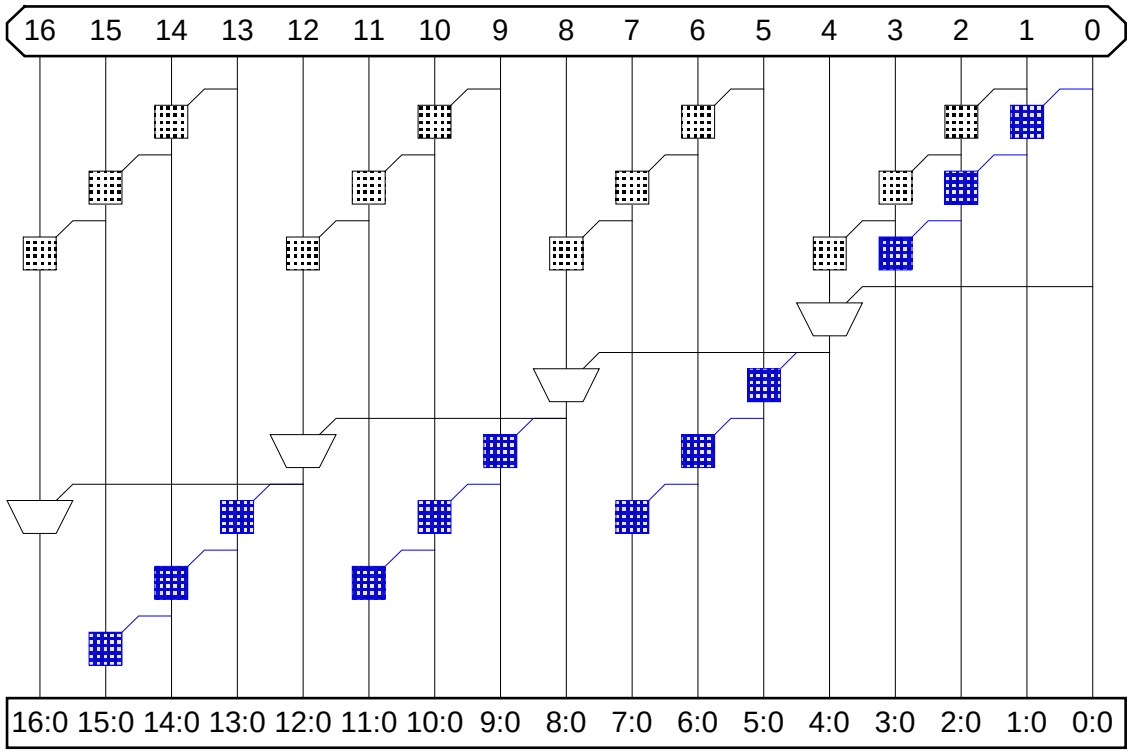


For k n-bit groups (N = nk)

$$t_{\text{skip}} = t_{pg} + \left[2(n-1) + (k-1)\right] t_{AO} + t_{\text{xor}}$$

src: https://cmosvlsi.com/

Carry-Skip PG Diagram

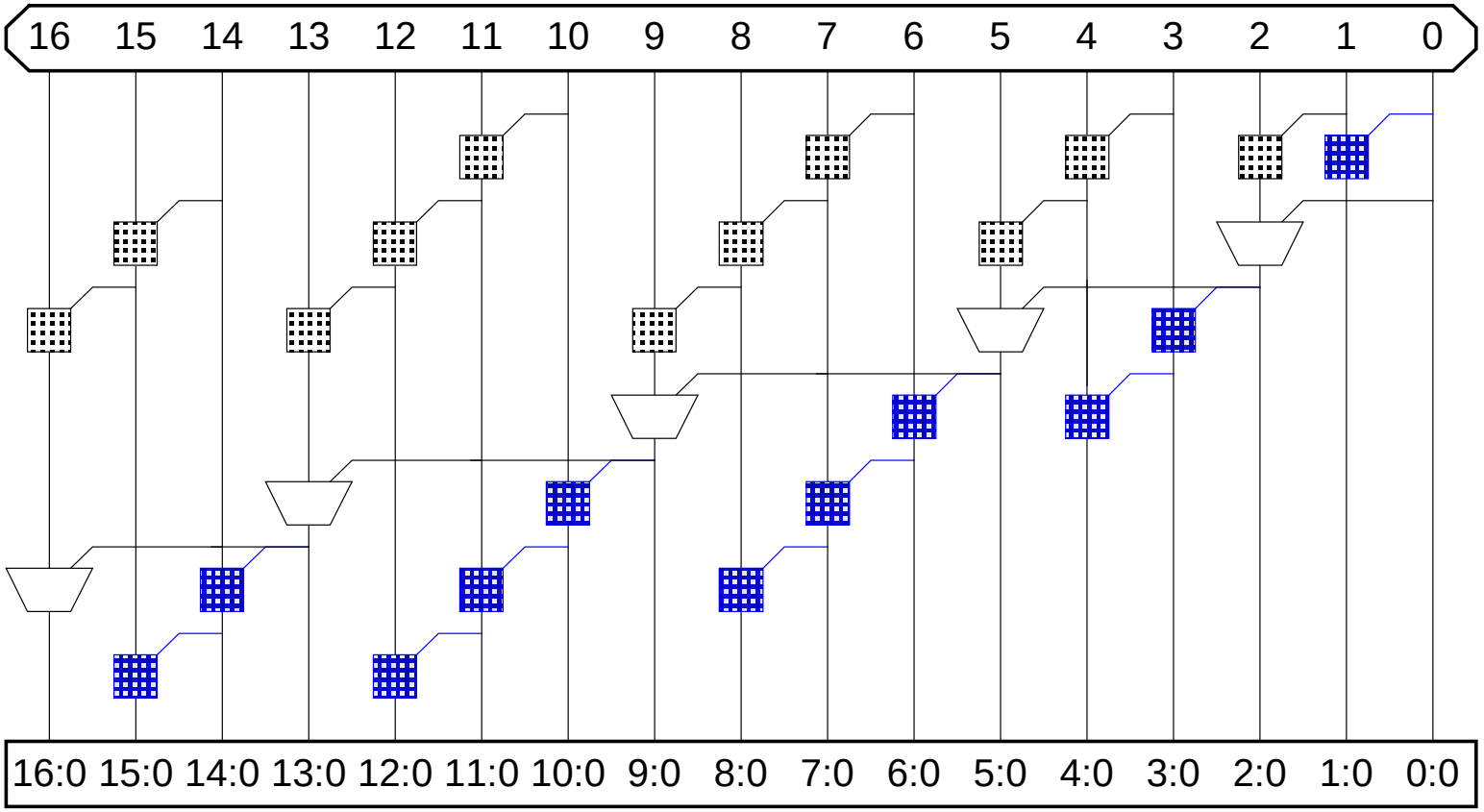


For k n-bit groups (N = nk)

$$t_{\text{skip}} = t_{pg} + \left[2(n-1) + (k-1)\right] t_{AO} + t_{\text{xor}}$$

src: https://cmosvlsi.com/

Variable Group Size

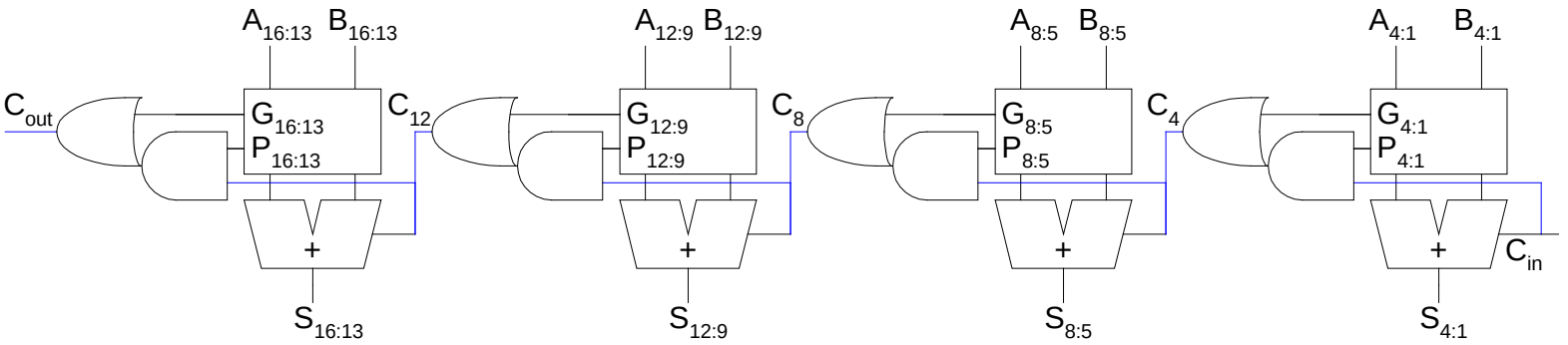


Delay grows as O(sqrt(N))

src: https://cmosvlsi.com/

Carry-Lookahead Adder

- Carry-lookahead adder computes $G_{i:0}$ for many bits in parallel.
- Uses higher-valency cells with more than two inputs.



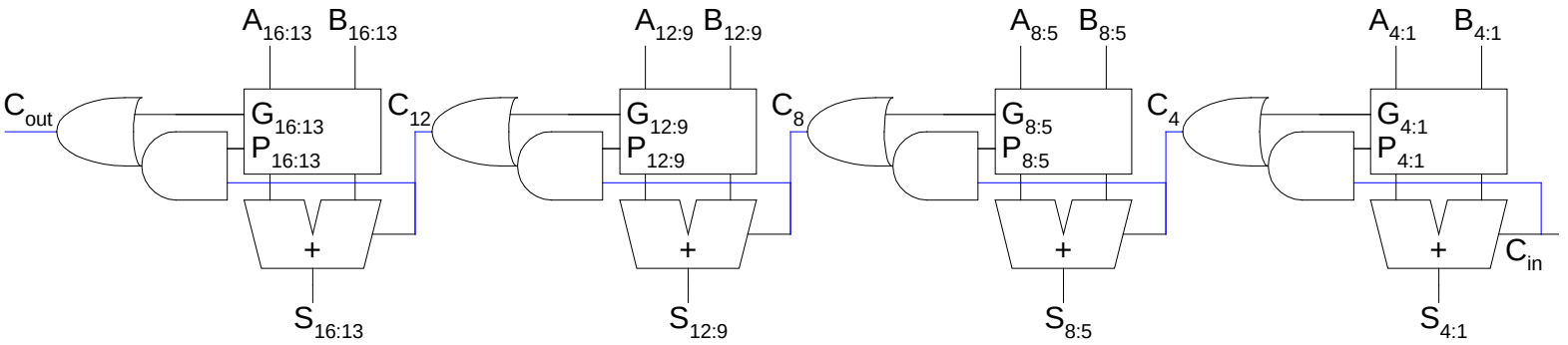
N=k·b, b=block size, k=number of blocks

t_{add} =

src: https://cmosvlsi.com/

Carry-Lookahead Adder

- Carry-lookahead adder computes $G_{i:0}$ for many bits in parallel.
- Uses higher-valency cells with more than two inputs.

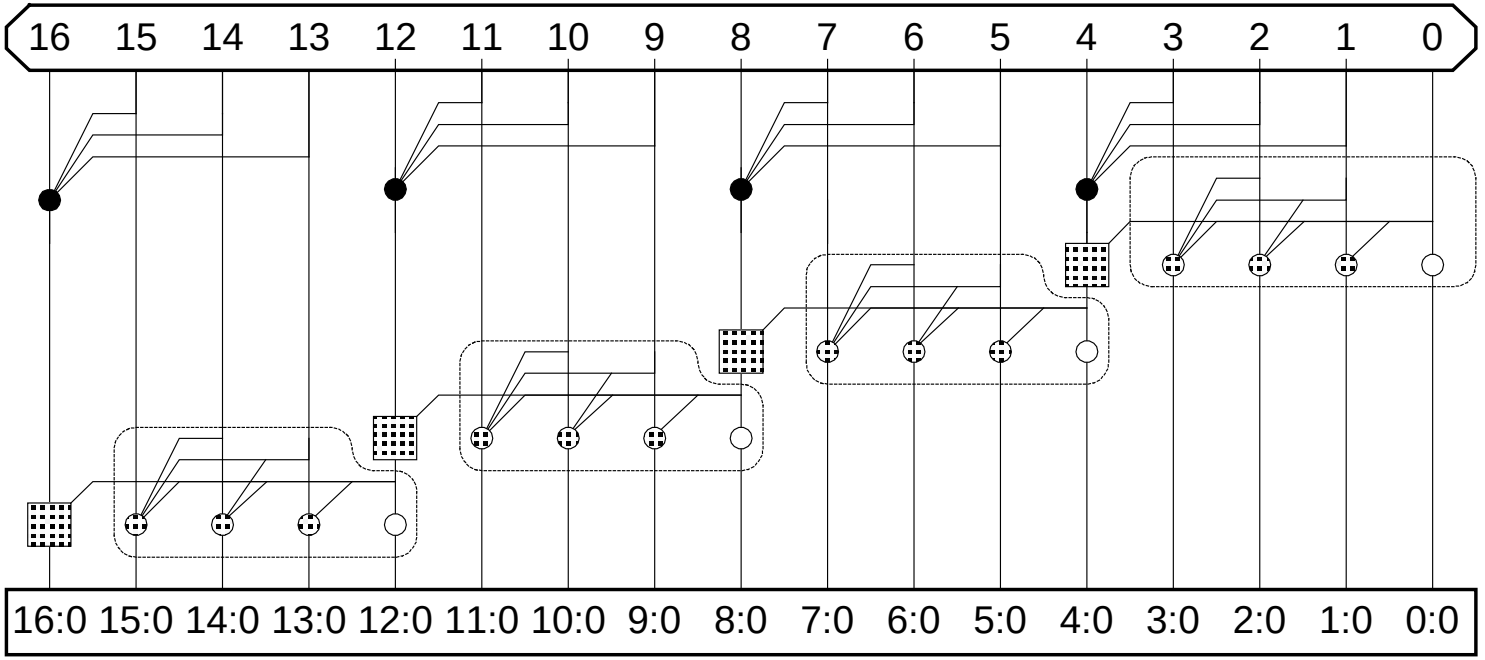


N=k·b, b=block size, k=number of blocks

$$t_{add} = t_{pg} + t_{pg(b)} + [(b-1)+(k-1)]\cdot t_{AO} + t_{XOR}$$

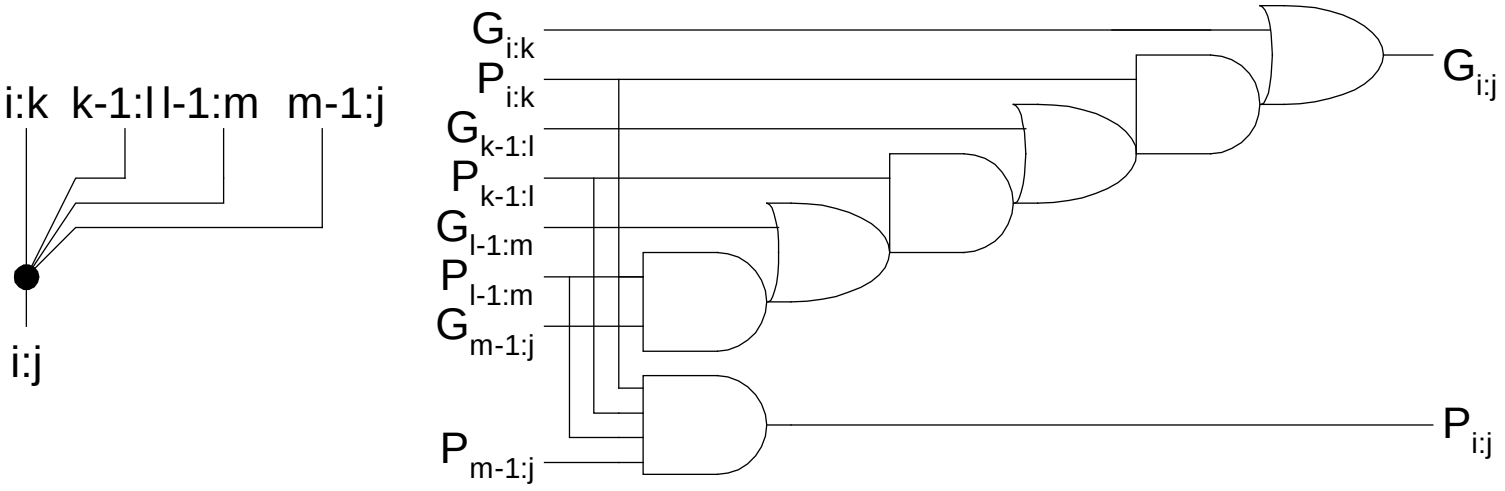
src: https://cmosvlsi.com/

CLA PG Diagram



src: https://cmosvlsi.com/

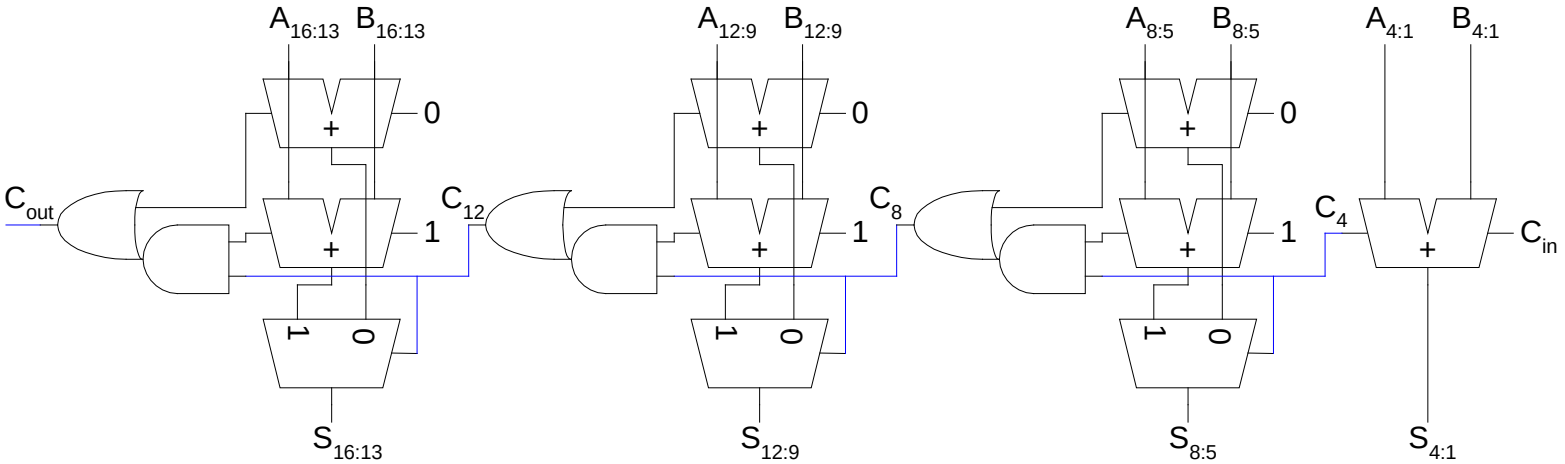
Higher-Valency Cells



src: https://cmosvlsi.com/

Carry-Select Adder

- Trick for critical paths dependent on late input X
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes n-bit sums
  - For both possible carries into n-bit group



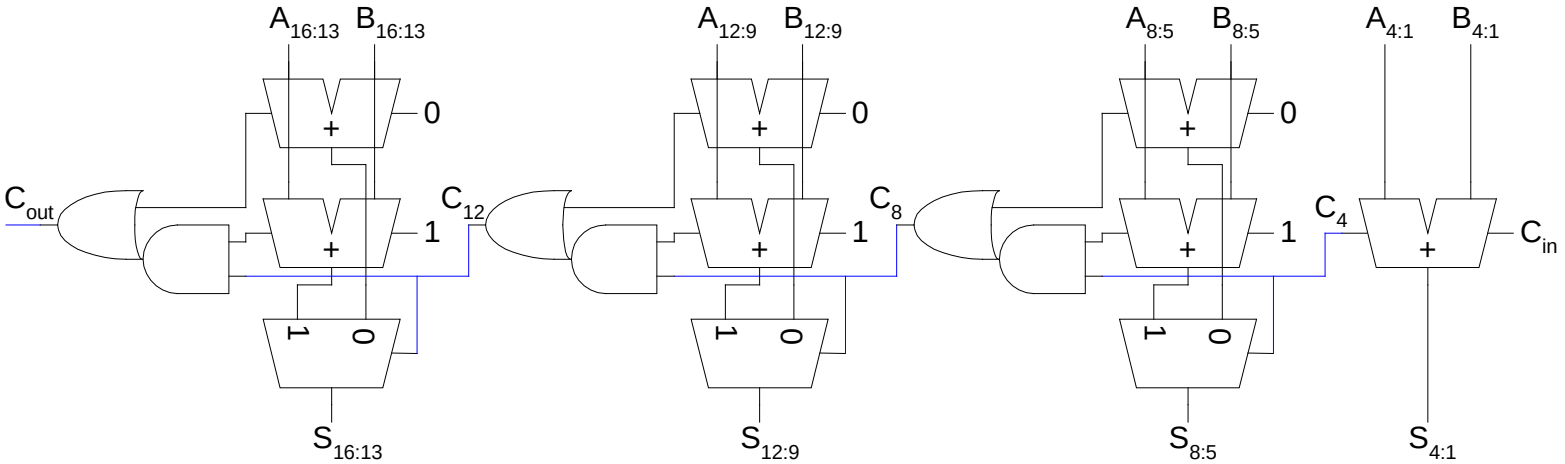
N=k·b, b=block size, k=number of blocks

t_{add} =

src: https://cmosvlsi.com/

Carry-Select Adder

- Trick for critical paths dependent on late input X
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes n-bit sums
  - For both possible carries into n-bit group



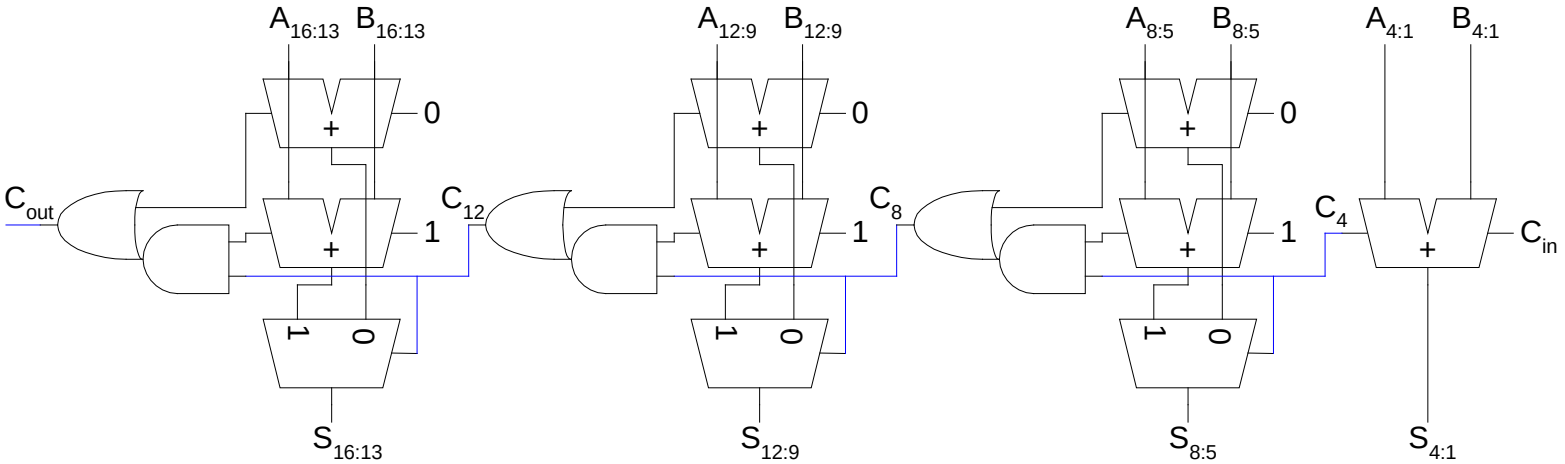
N=k·b, b=block size, k=number of blocks

$t_{add} = t_{pg}$

src: https://cmosvlsi.com/

Carry-Select Adder

- Trick for critical paths dependent on late input X
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes n-bit sums
  - For both possible carries into n-bit group



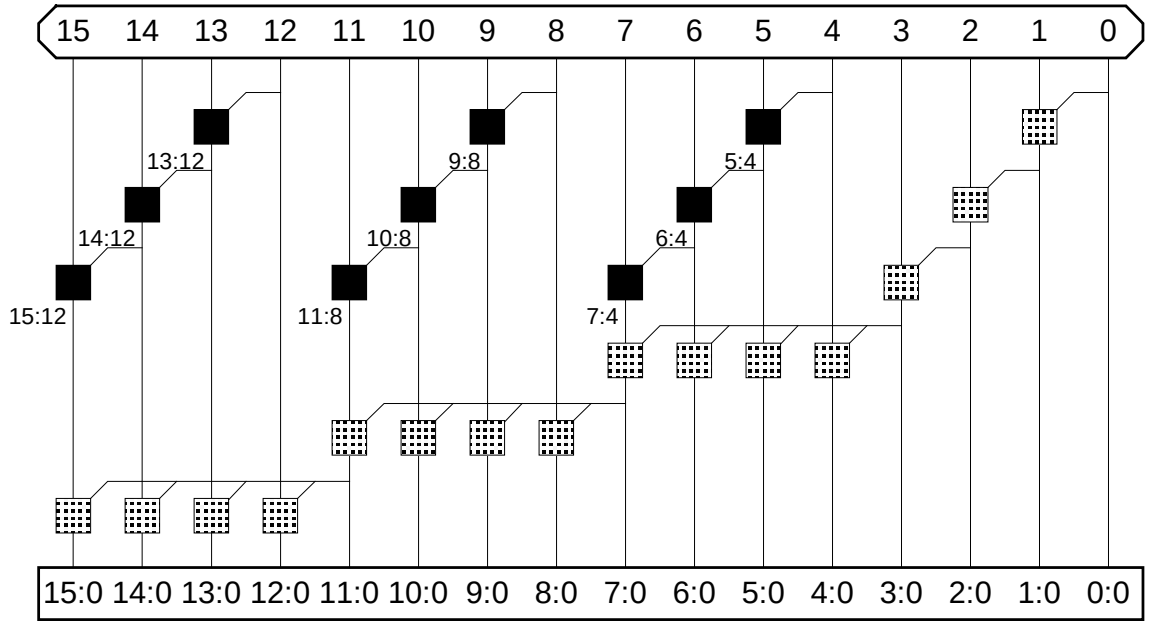
N=k·b, b=block size, k=number of blocks

$t_{add} = t_{pg} + [b+(k-2)] \cdot t_{AO} + t_{mux}$

src: https://cmosvlsi.com/

Carry-Increment Adder

- Factor initial PG and final XOR out of carry-select

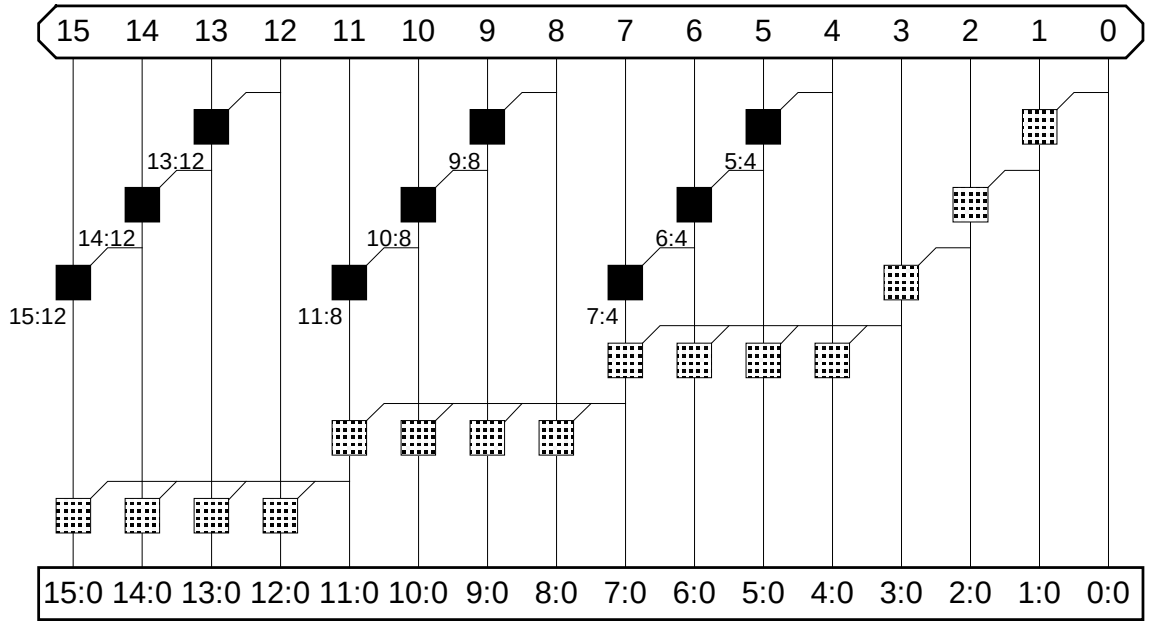


$$t_{\text{increment}} =$$

src: https://cmosvlsi.com/

Carry-Increment Adder

- Factor initial PG and final XOR out of carry-select

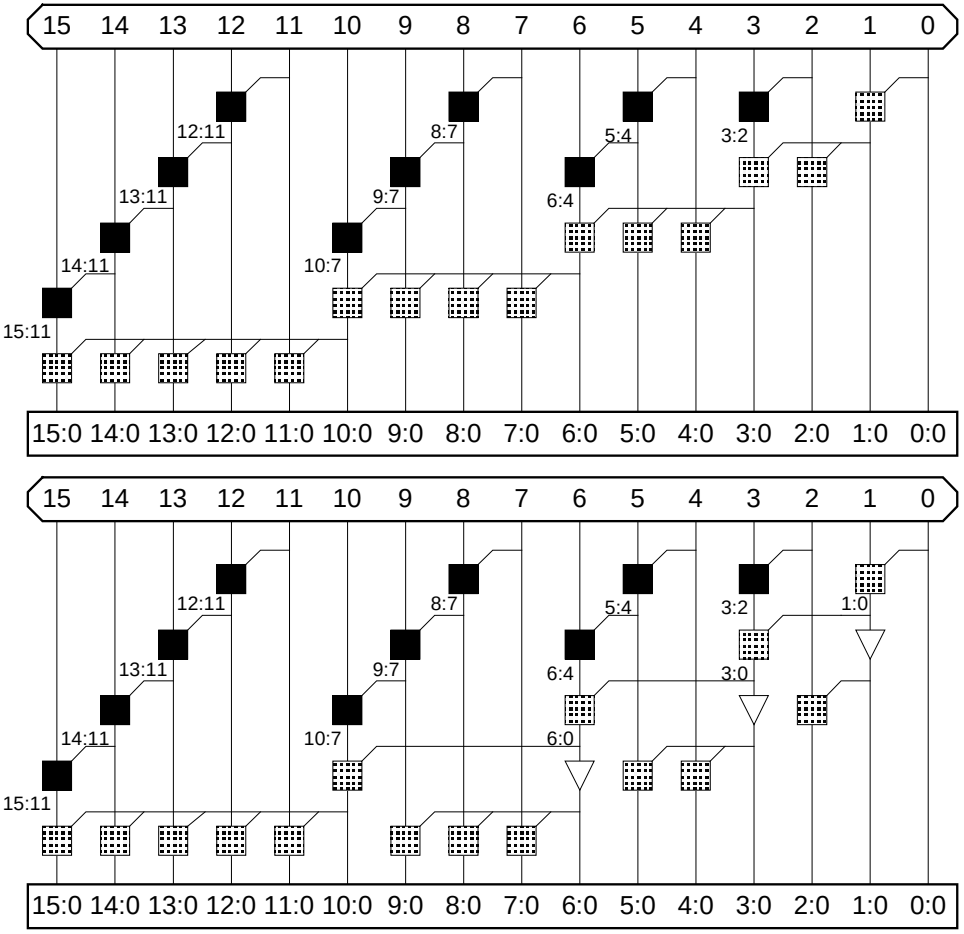


$$t_{\text{increment}} = t_{pg} + \left[(n-1) + (k-1) \right] t_{AO} + t_{\text{xor}}$$

src: https://cmosvlsi.com/

Variable Group Size

- Also buffer noncritical signals



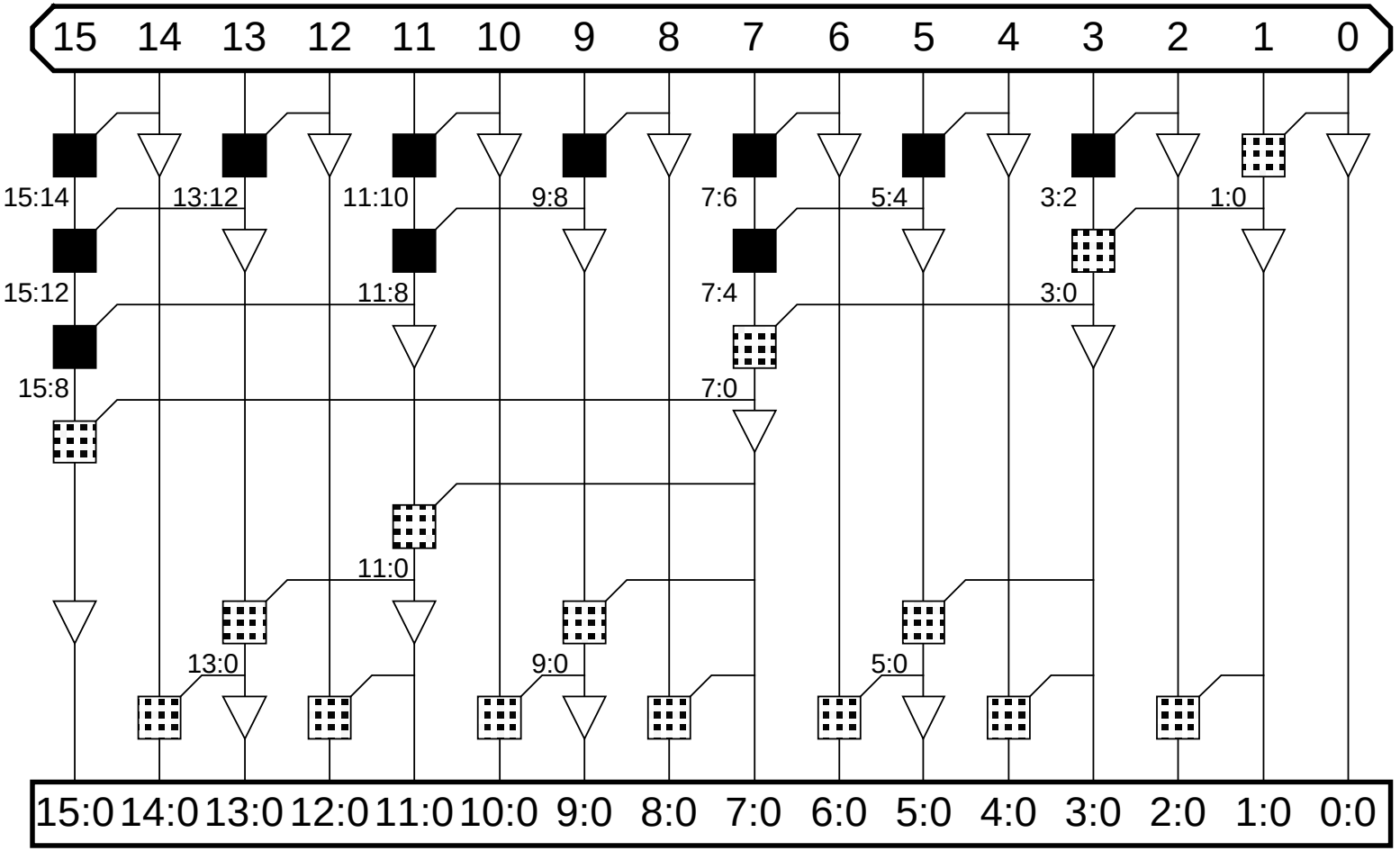
src: https://cmosvlsi.com/

Tree Adder

- If lookahead is good, lookahead across lookahead!
  - Recursive lookahead gives O(log N) delay
- Many variations on tree adders

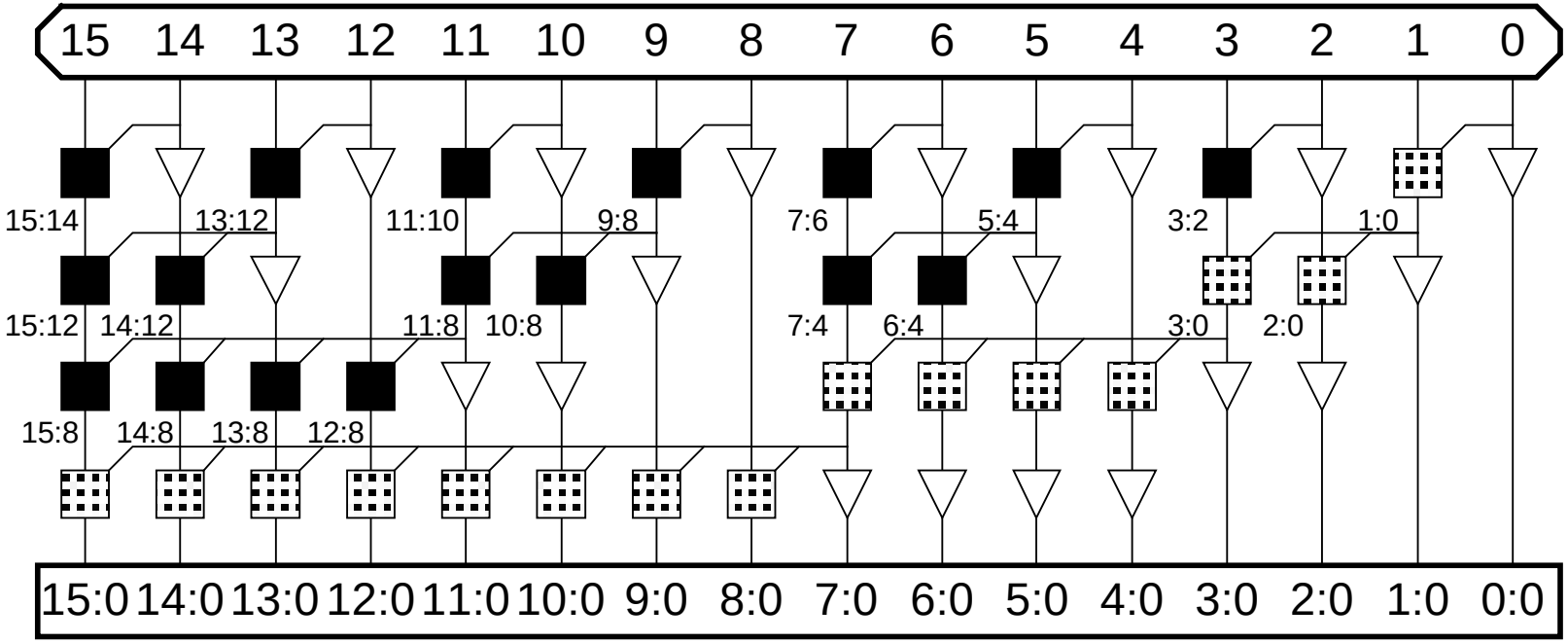
src: https://cmosvlsi.com/

Brent-Kung



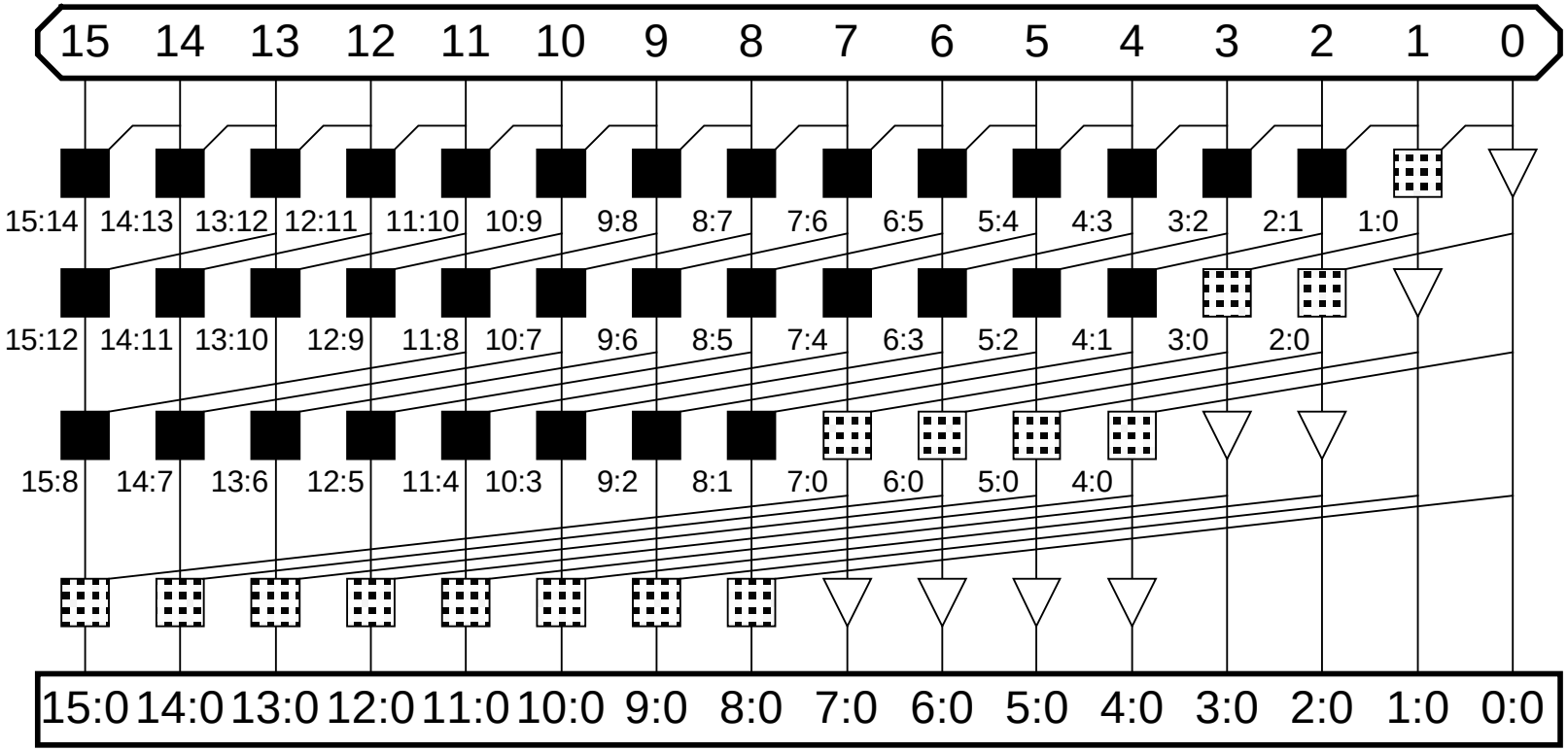
src: https://cmosvlsi.com/

Sklansky



src: https://cmosvlsi.com/

Kogge-Stone



src: https://cmosvlsi.com/

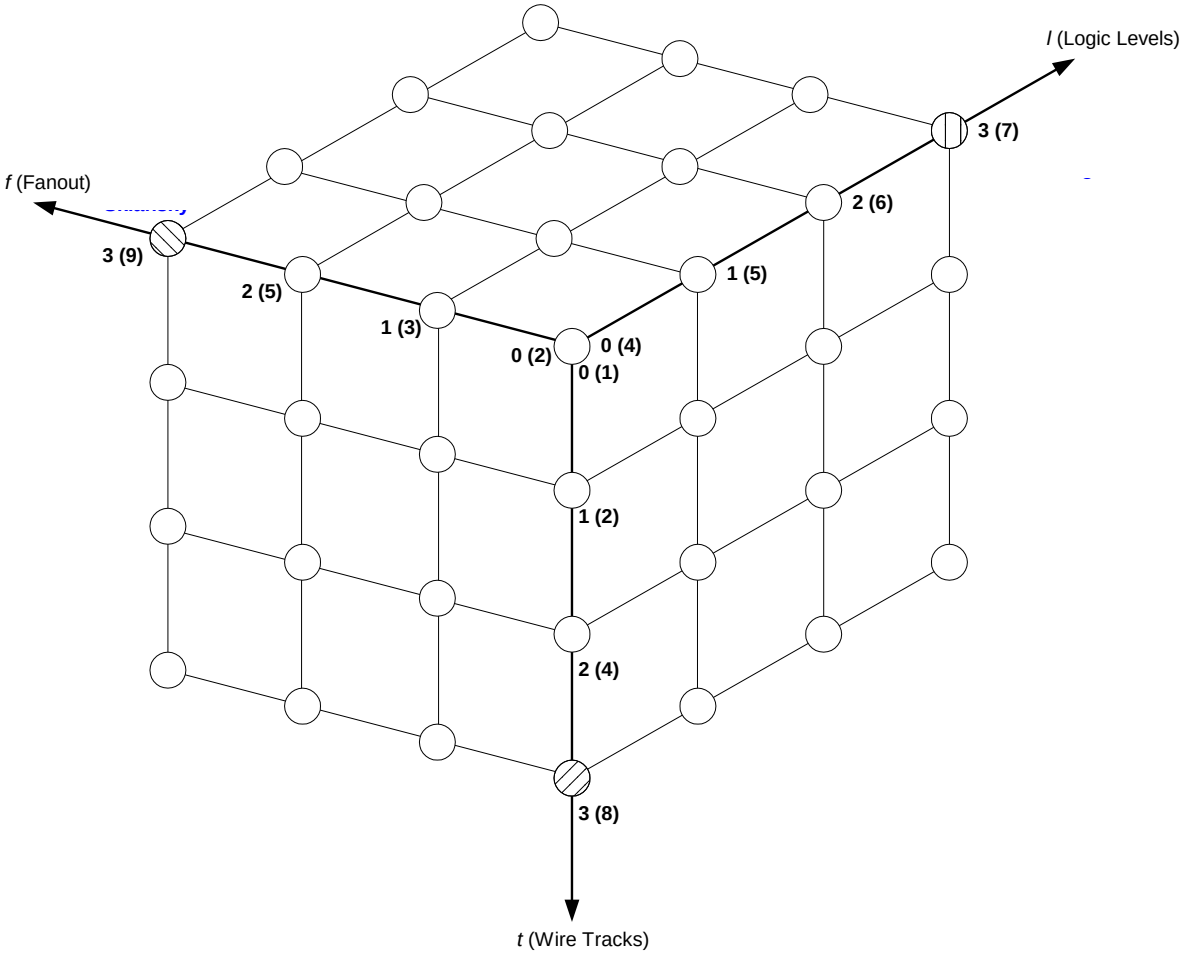
Tree Adder Taxonomy

- Ideal N-bit tree adder would have
  - $L = \log N$ logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels
- Describe adder with 3-D taxonomy (l, f, t)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: 2^t
- Known tree adders sit on plane defined by

$$l + f + t = L\text{-}1$$

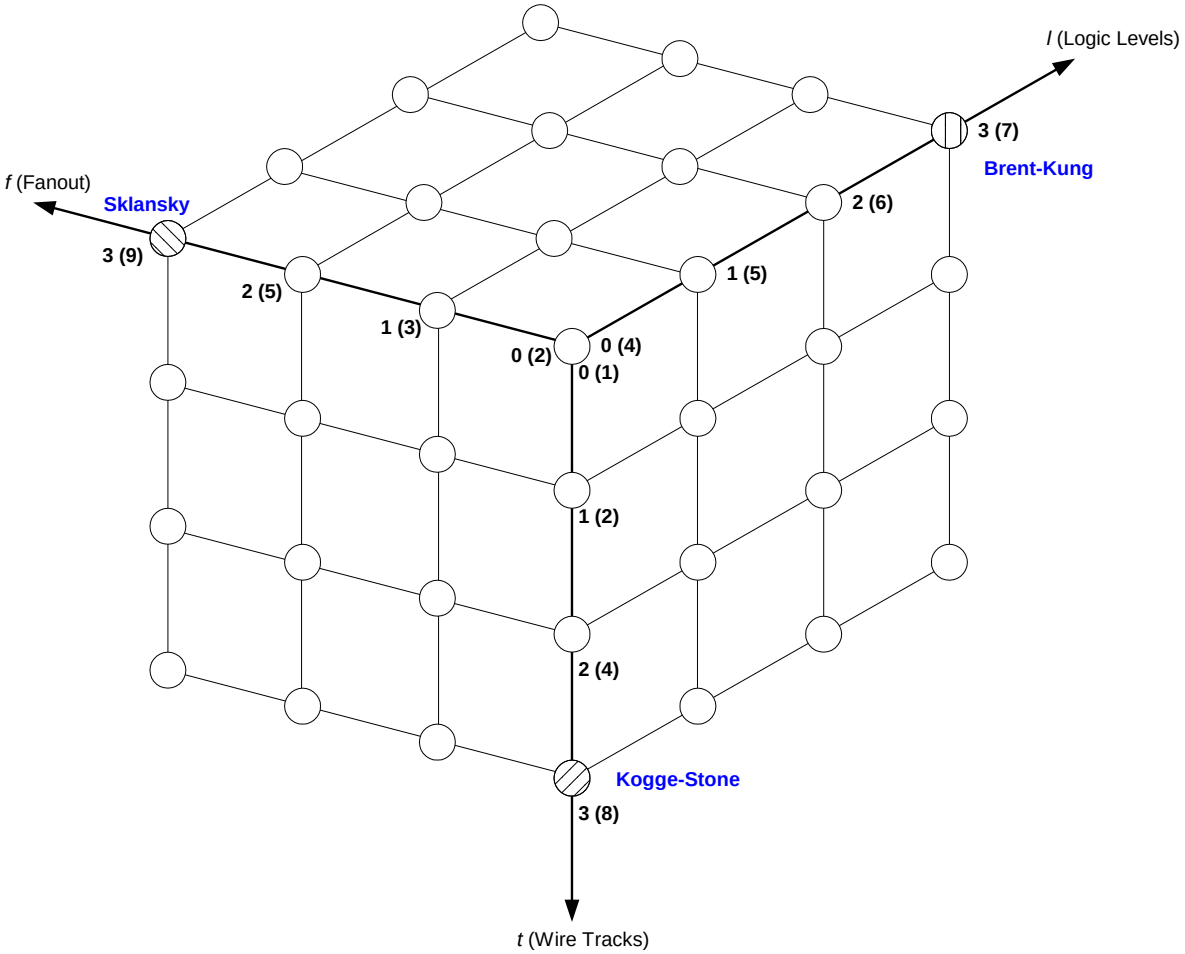
src: https://cmosvlsi.com/

Tree Adder Taxonomy



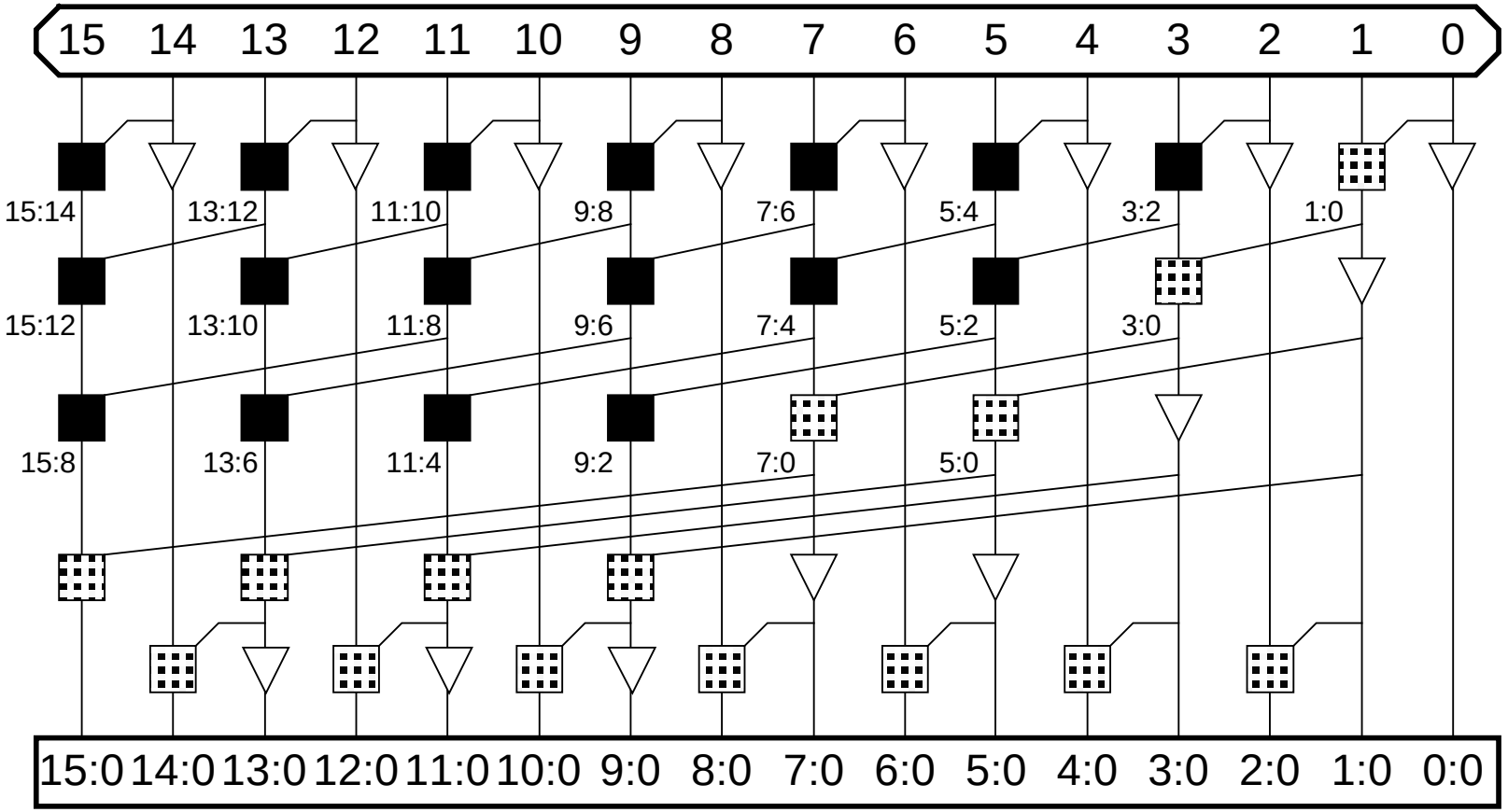
src: https://cmosvlsi.com/

Tree Adder Taxonomy



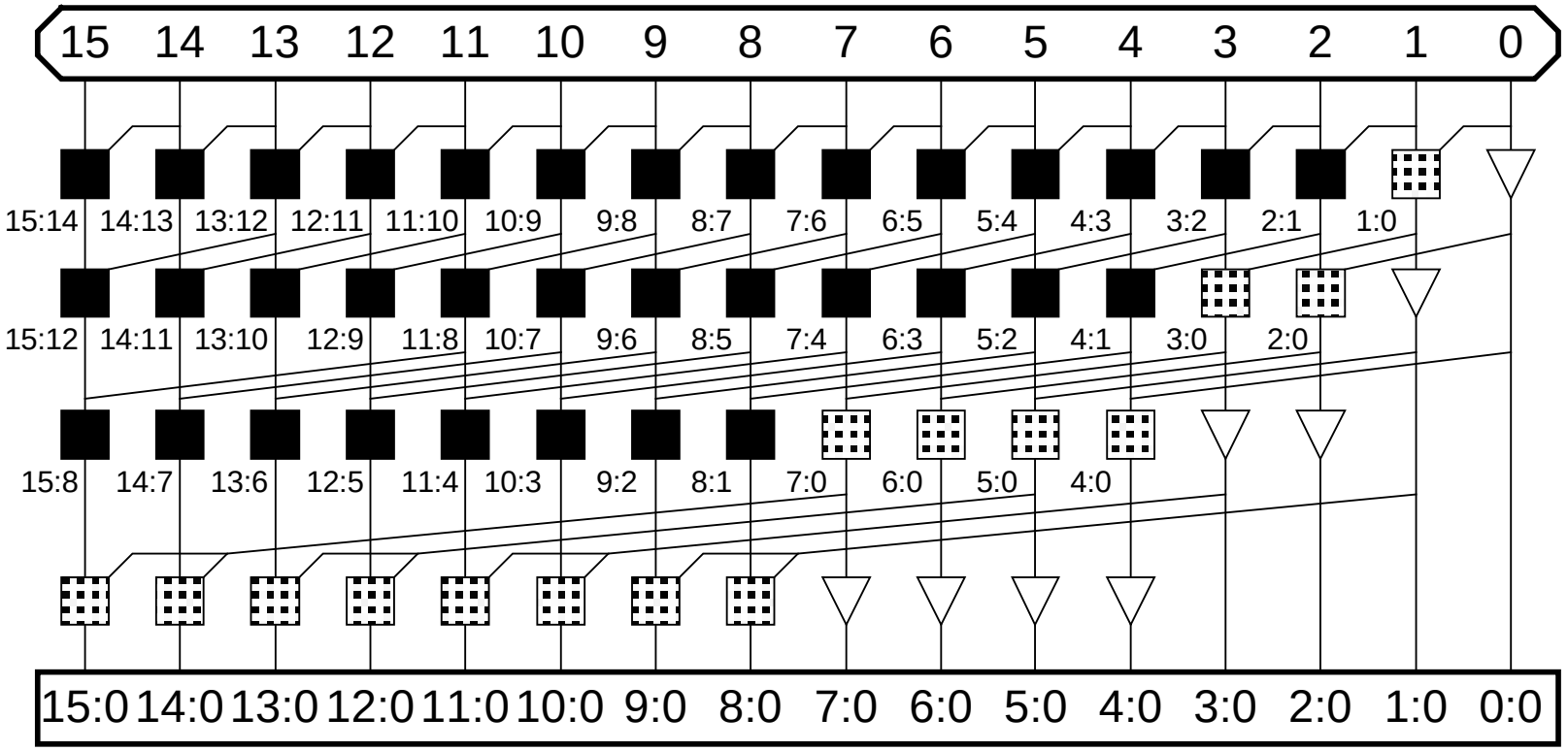
src: https://cmosvlsi.com/

Han-Carlson



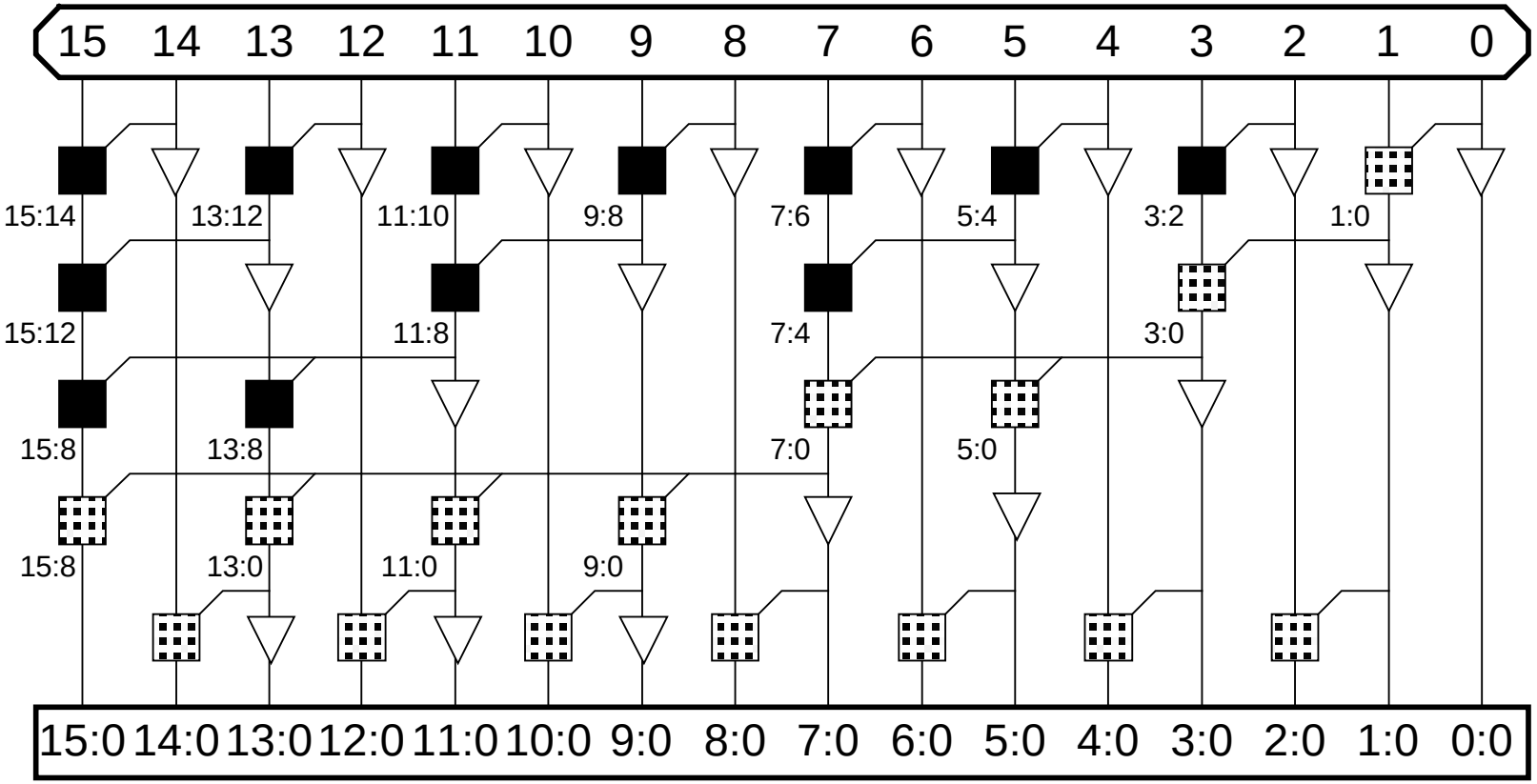
src: https://cmosvlsi.com/

Knowles [2, 1, 1, 1]



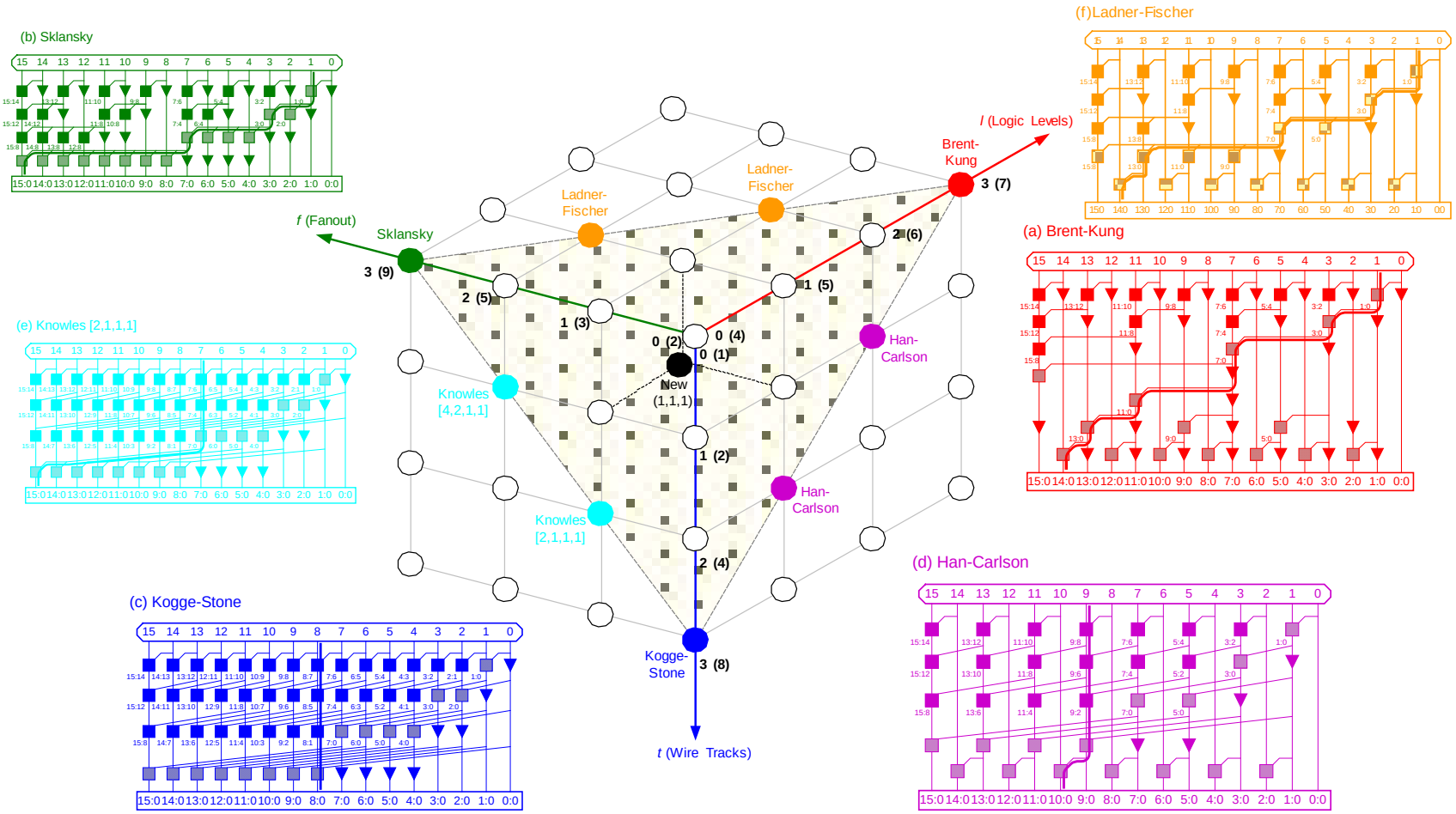
src: https://cmosvlsi.com/

Ladner-Fischer



src: https://cmosvlsi.com/

Taxonomy Revisited



src: https://cmosvlsi.com/

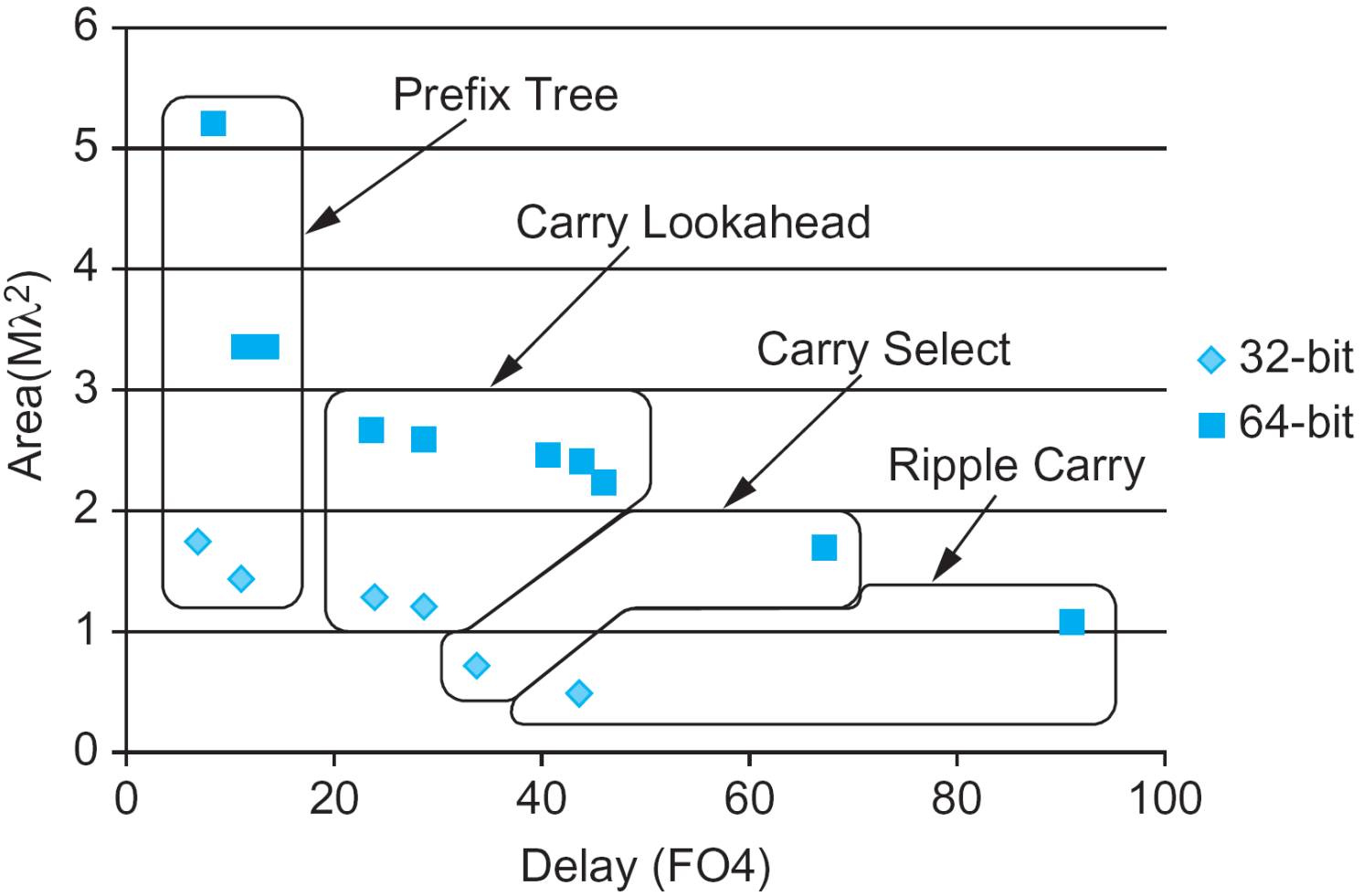
Summary

- Adder architectures offer area / power / delay tradeoffs.
- Choose the best one for your application.

Architecture	Classification	Logic Levels	Max Fanout	Tracks	Cells
Carry-Ripple		N-1	1	1	N
Carry-Skip n=4		N/4 + 5	2	1	1.25N
Carry-Inc. n=4		N/4 + 2	4	1	2N
Brent-Kung	(L-1, 0, 0)	$2\log_2 N - 1$	2	1	2N
Sklansky	(0, L-1, 0)	$\log_2 N$	N/2 + 1	1	$0.5\ N\log_2 N$
Kogge-Stone	(0, 0, L-1)	$\log_2 N$	2	N/2	$N\log_2 N$

src: https://cmosvlsi.com/

Summary

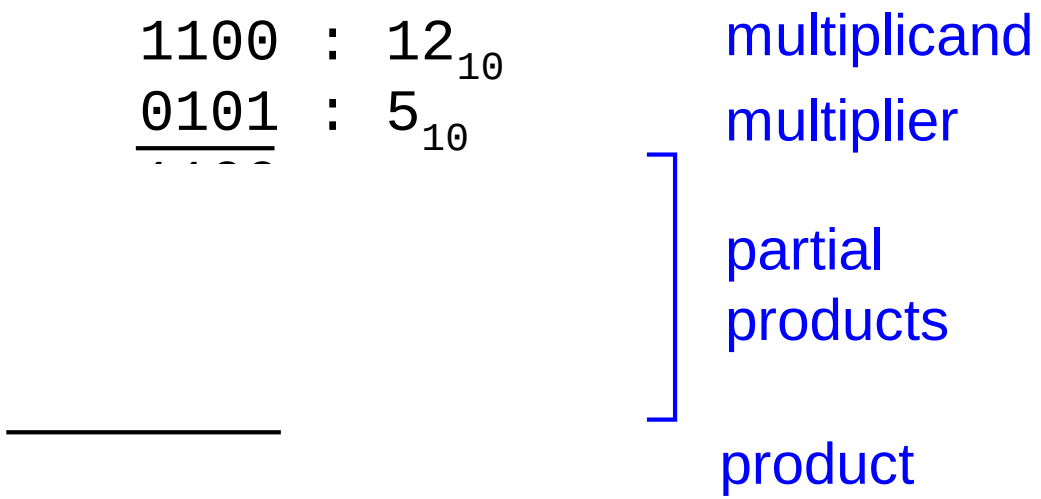


src: https://cmosvlsi.com/

Multipliers

Multiplication

- Example:

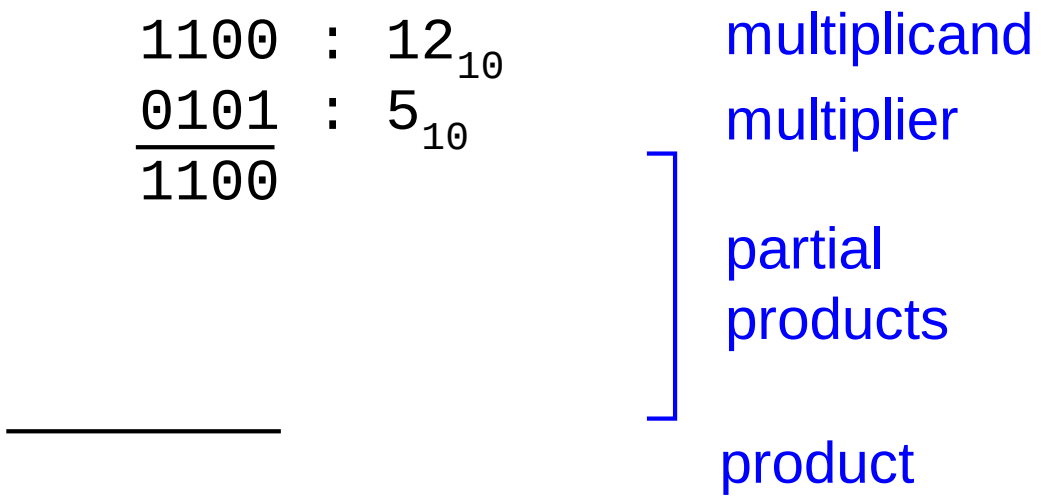


- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

src: https://cmosvlsi.com/

Multiplication

- Example:

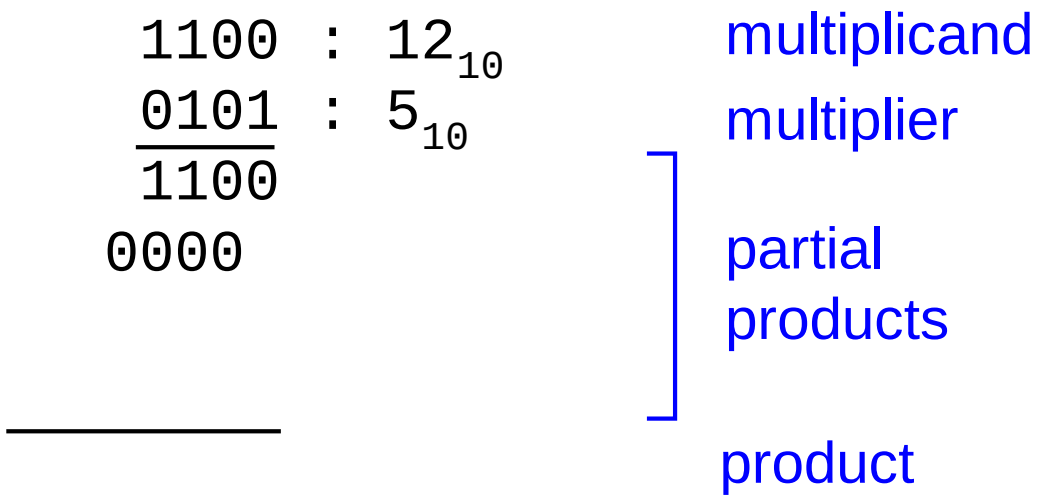


- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

src: https://cmosvlsi.com/

Multiplication

- Example:

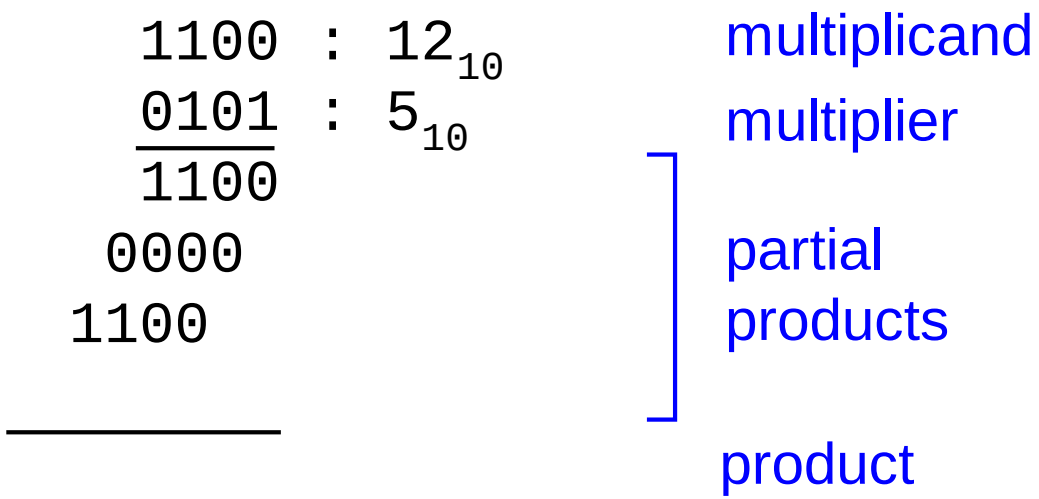


- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

src: https://cmosvlsi.com/

Multiplication

- Example:

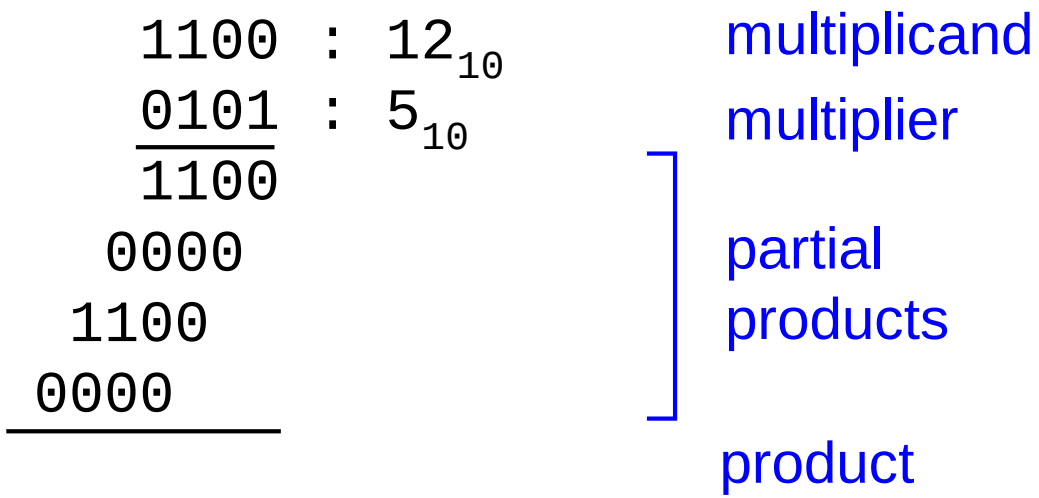


- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

src: https://cmosvlsi.com/

Multiplication

- Example:

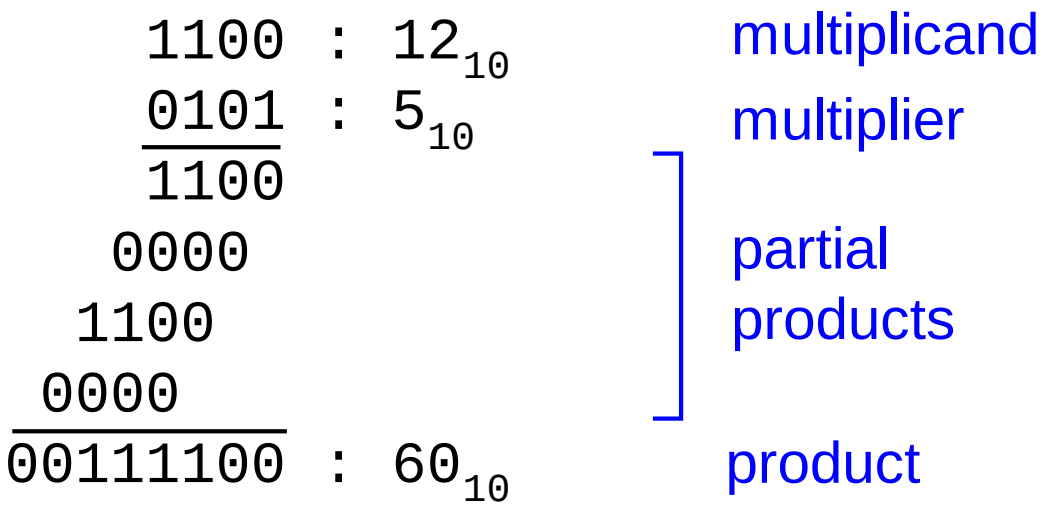


- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

src: https://cmosvlsi.com/

Multiplication

- Example:



- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

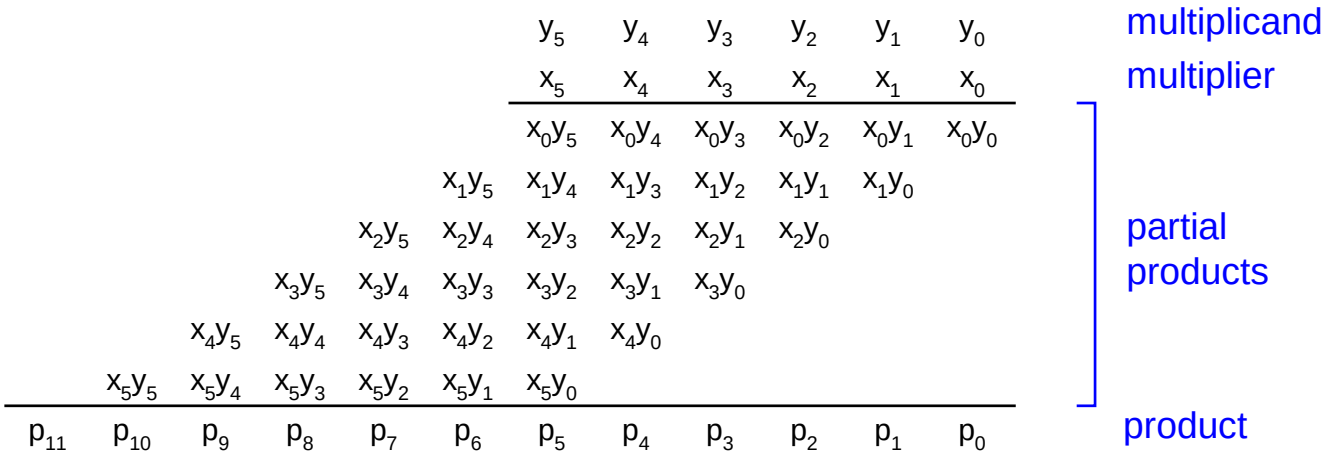
src: https://cmosvlsi.com/

General Form

- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$

- Product:

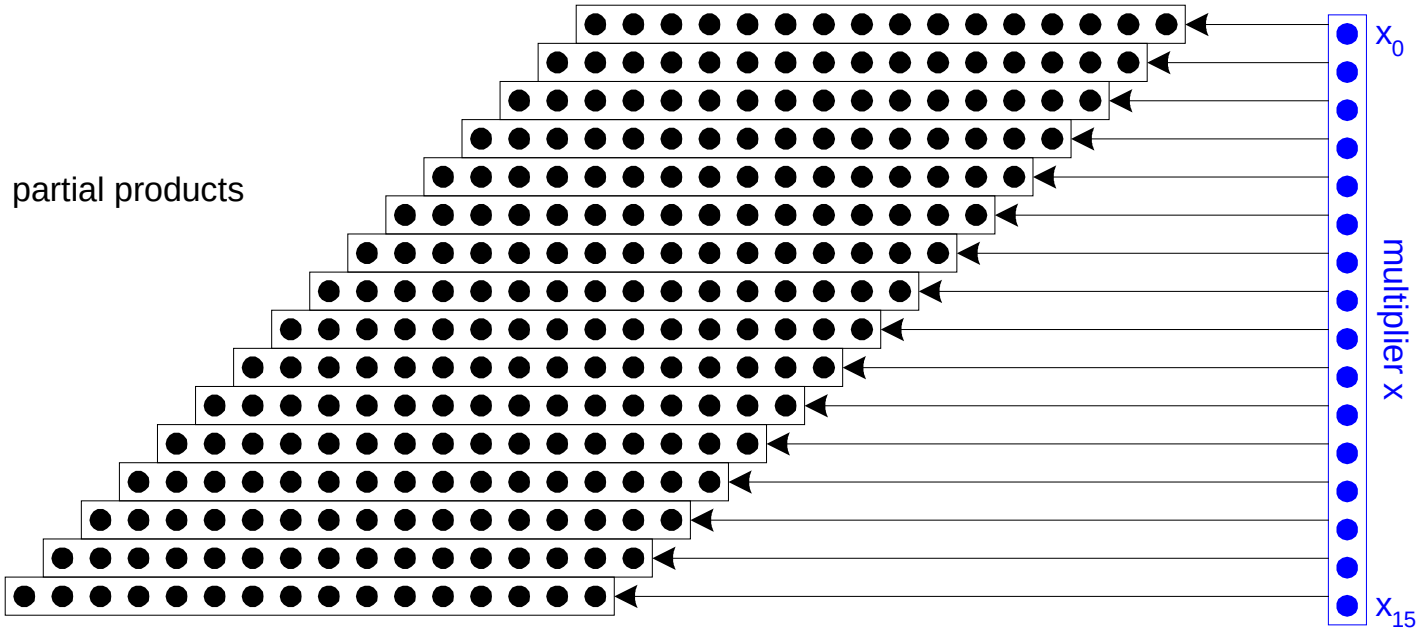
$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$



src: https://cmosvlsi.com/

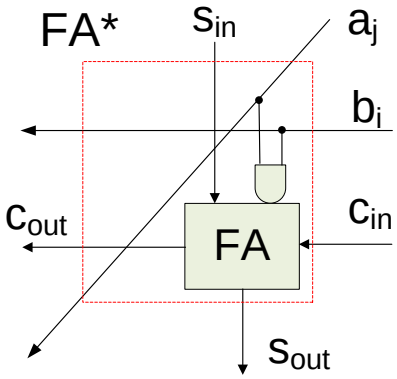
Dot Diagram

- Each dot represents a bit



src: https://cmosvlsi.com/

RCA Array Multiplier



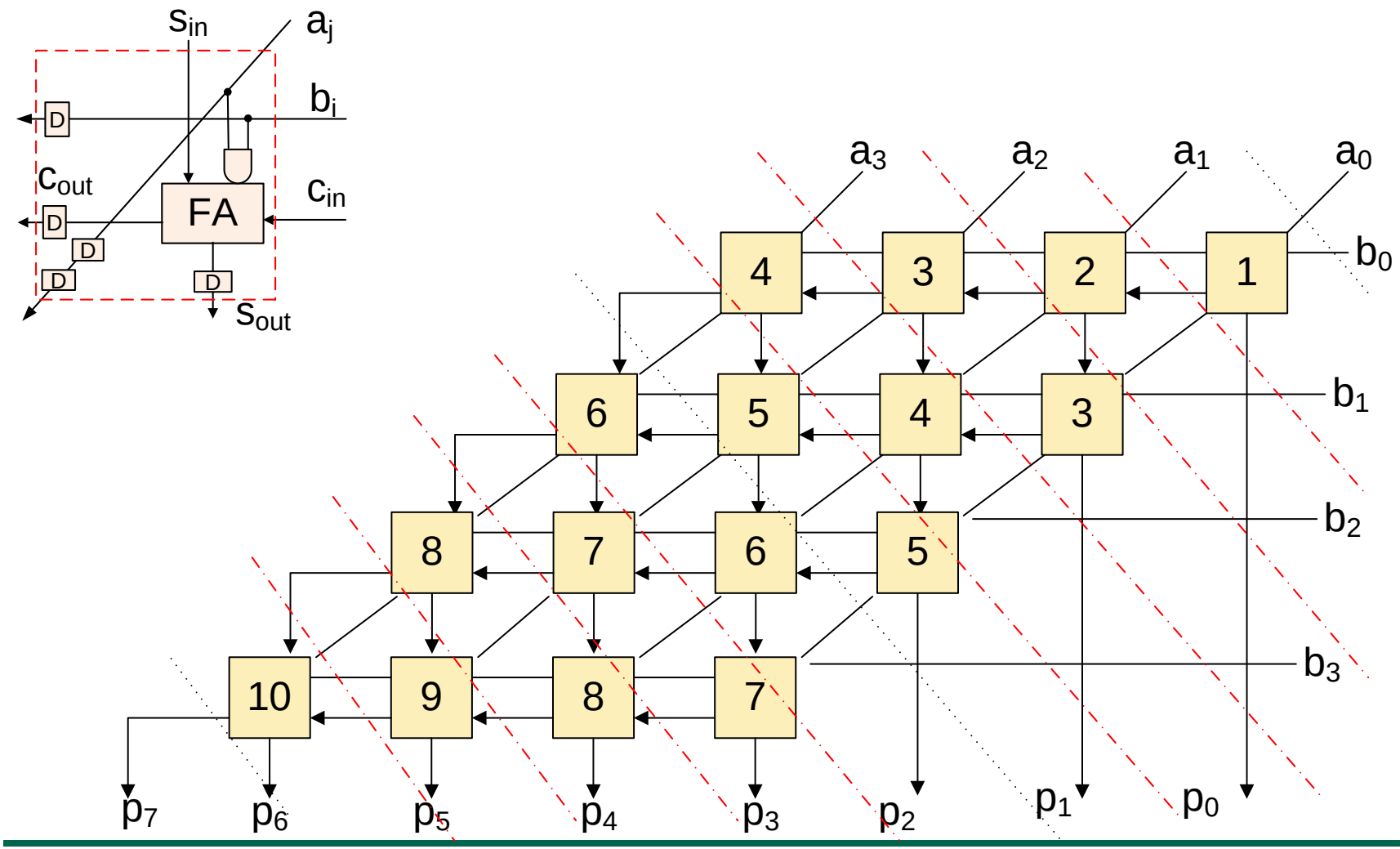
$P(n+m)=a(n)\times b(m)$

Delay:

$T= (2n+m-2)T_{FA}+T_{AND}$

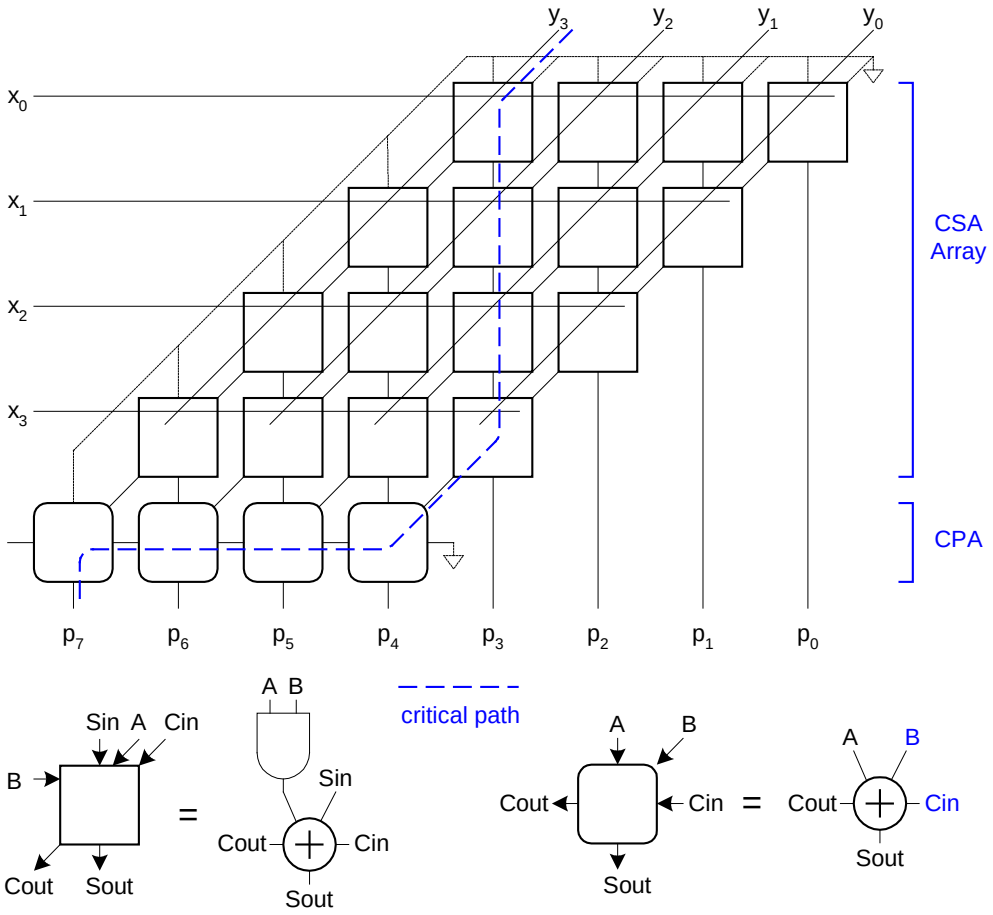
src: K. Pekmestzi, NTUA

Systolic RCA Array Multiplier



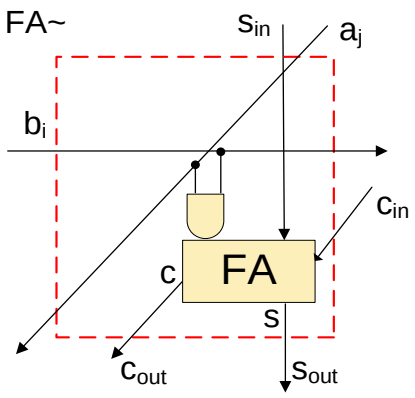
src: K. Pekmestzi, NTUA

CSA Array Multiplier



src: https://cmosvlsi.com/

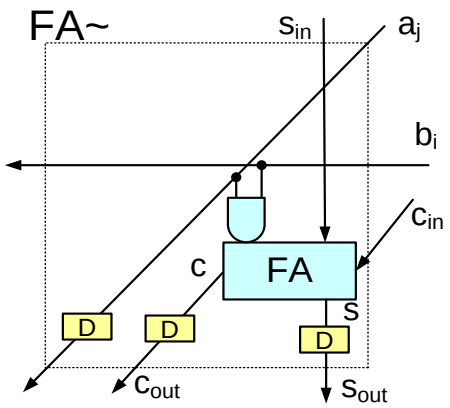
4x4 CSA Array Multiplier

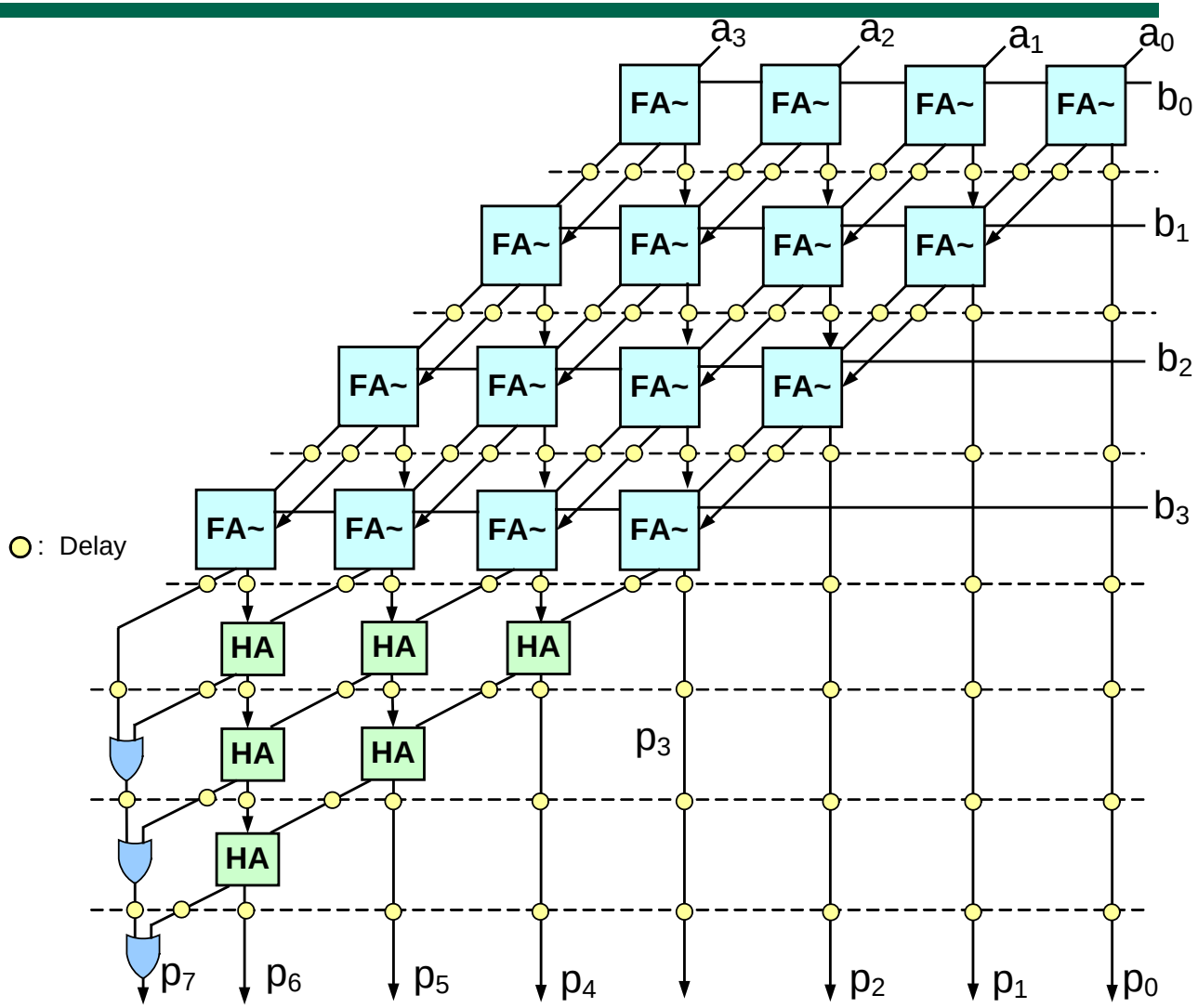


P(n+m)=a(n)b(m)
Delay
$T= (n+m)T_{FA}+T_{AND}$

src: K. Pekmestzi, NTUA

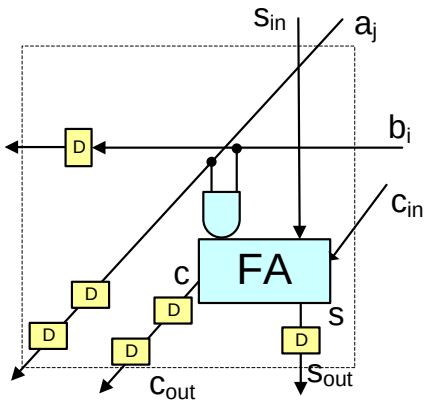
Pipeline 4x4 CSA Array Multiplier

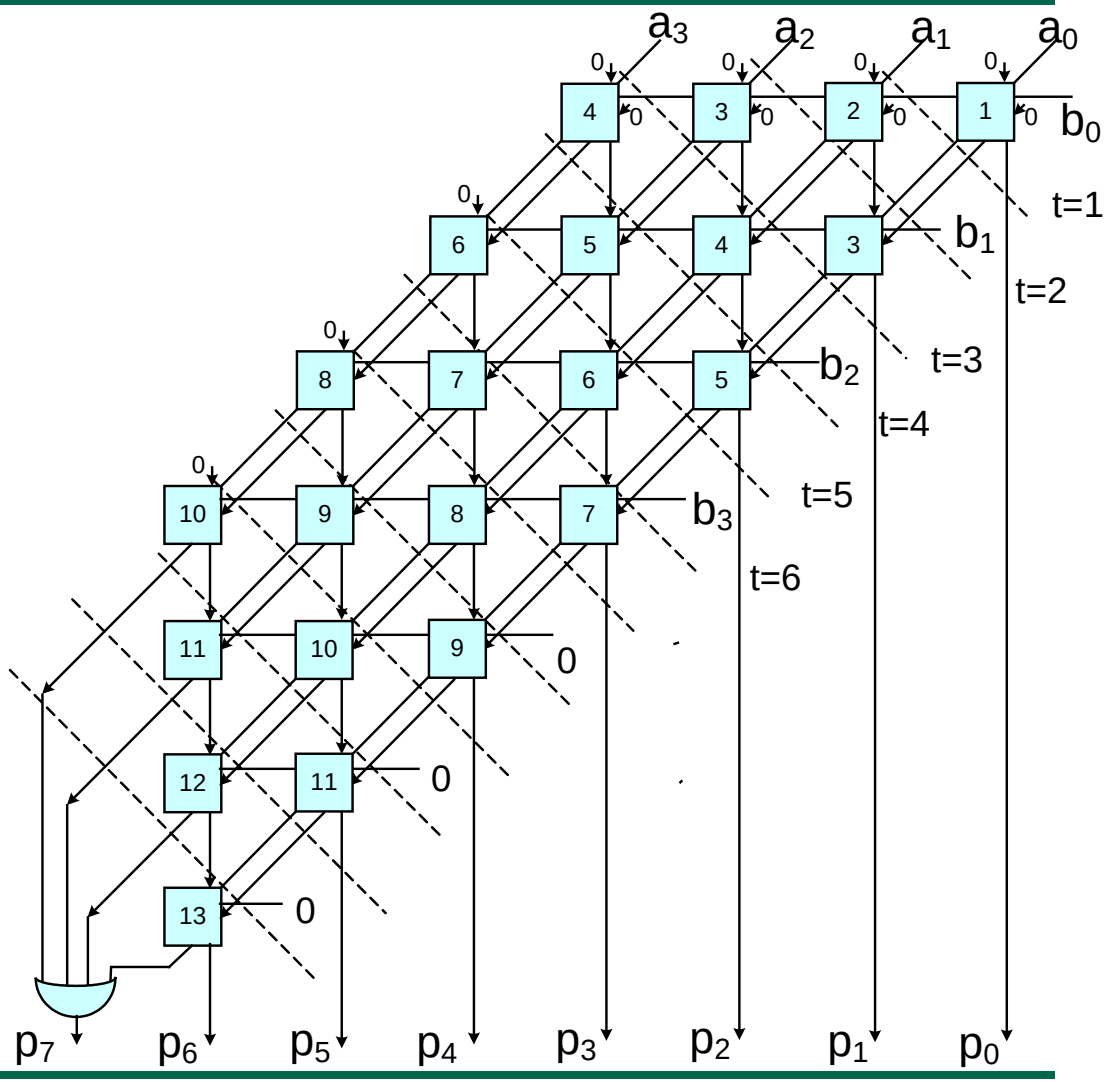




src: K. Pekmestzi, NTUA

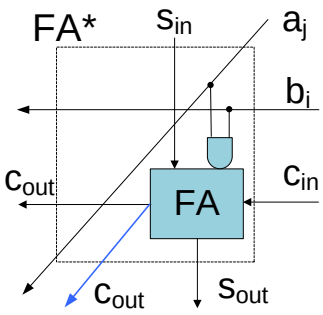
Systolic 4x4 CSA Array Multiplier



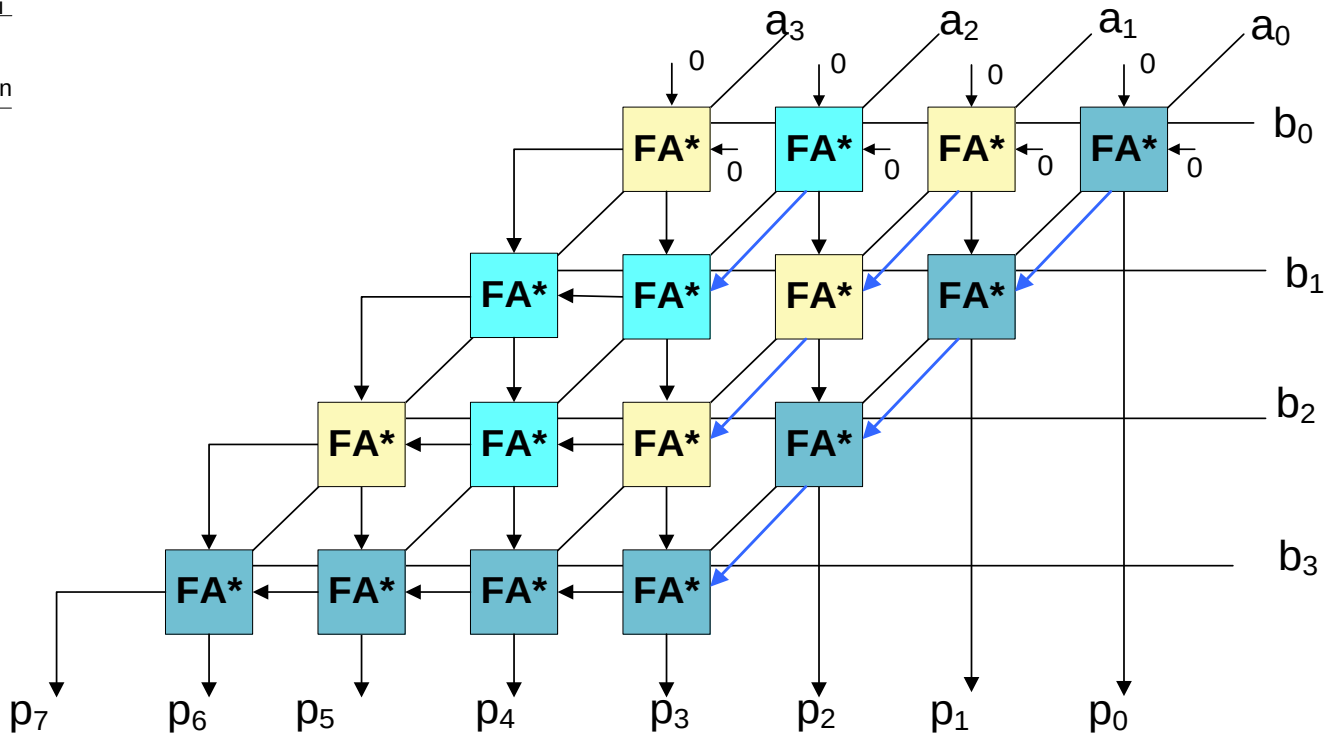


src: K. Pekmestzi, NTUA

Array Multiplier w/ RCA & Carry-Save



P(n+m)=a(n)b(m)
Delay:
$T = (n+m-1)T_{FA}+T_{AND}$

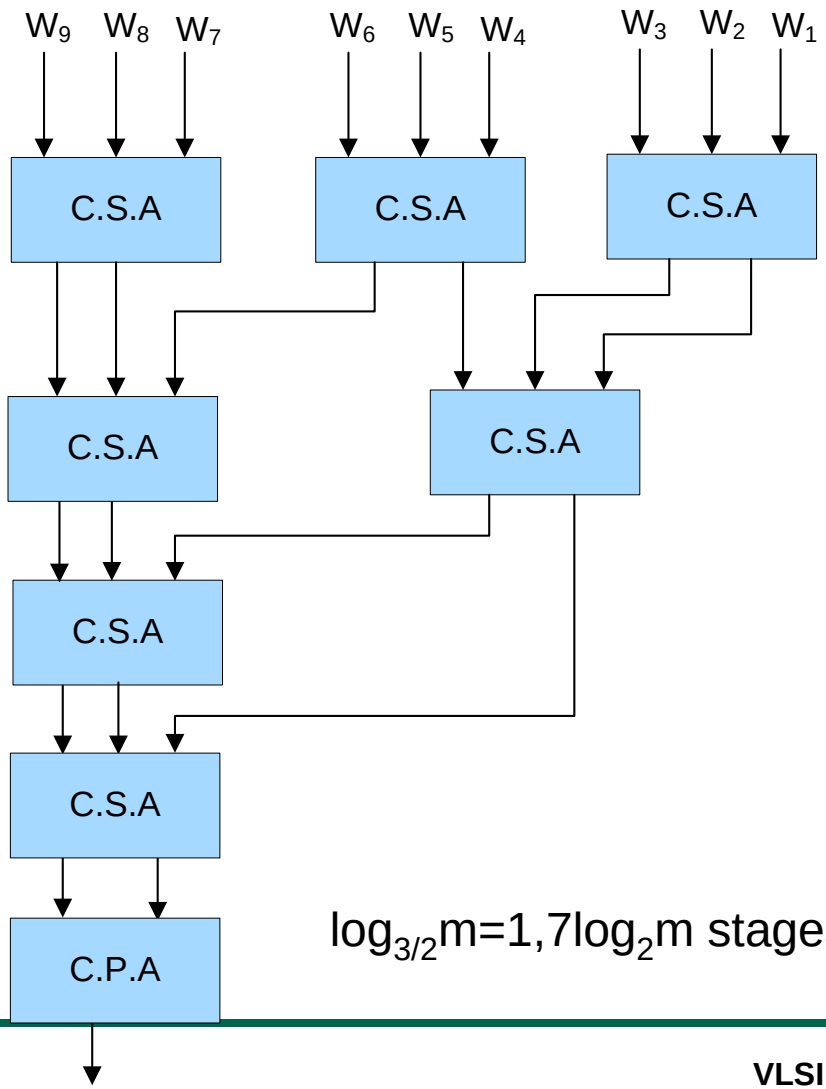


src: K. Pekmestzi, NTUA

Wallace Tree

$P(n+m)=a(n)\cdot b(m)$

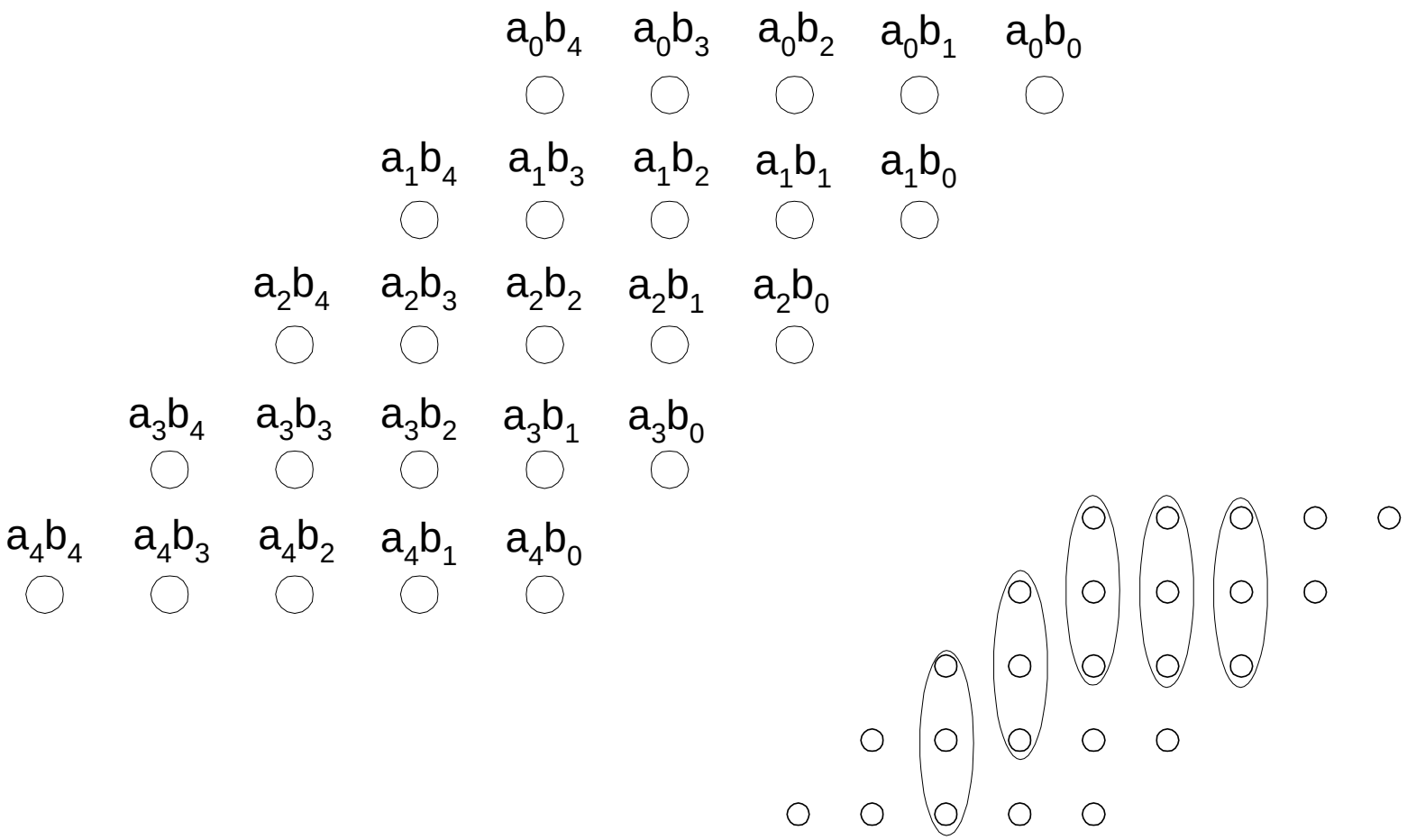
DelaY:
$T=(1,7\log_2 m)\cdot T_{FA}+T_{AND}+\log_2 n\cdot T_{pg}$



$\log_{3/2} m=1,7\log_2 m$ stages

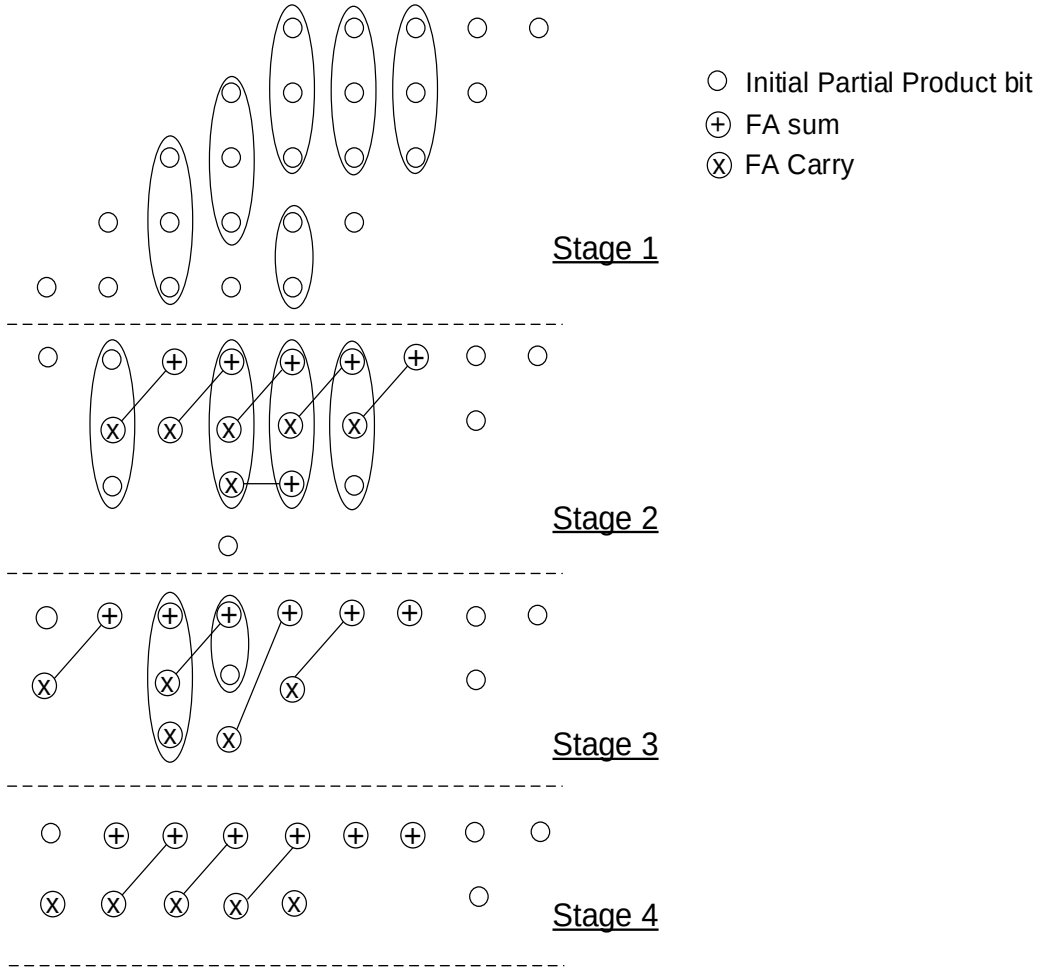
src: K. Pekmestzi, NTUA

Partial Product Grouping



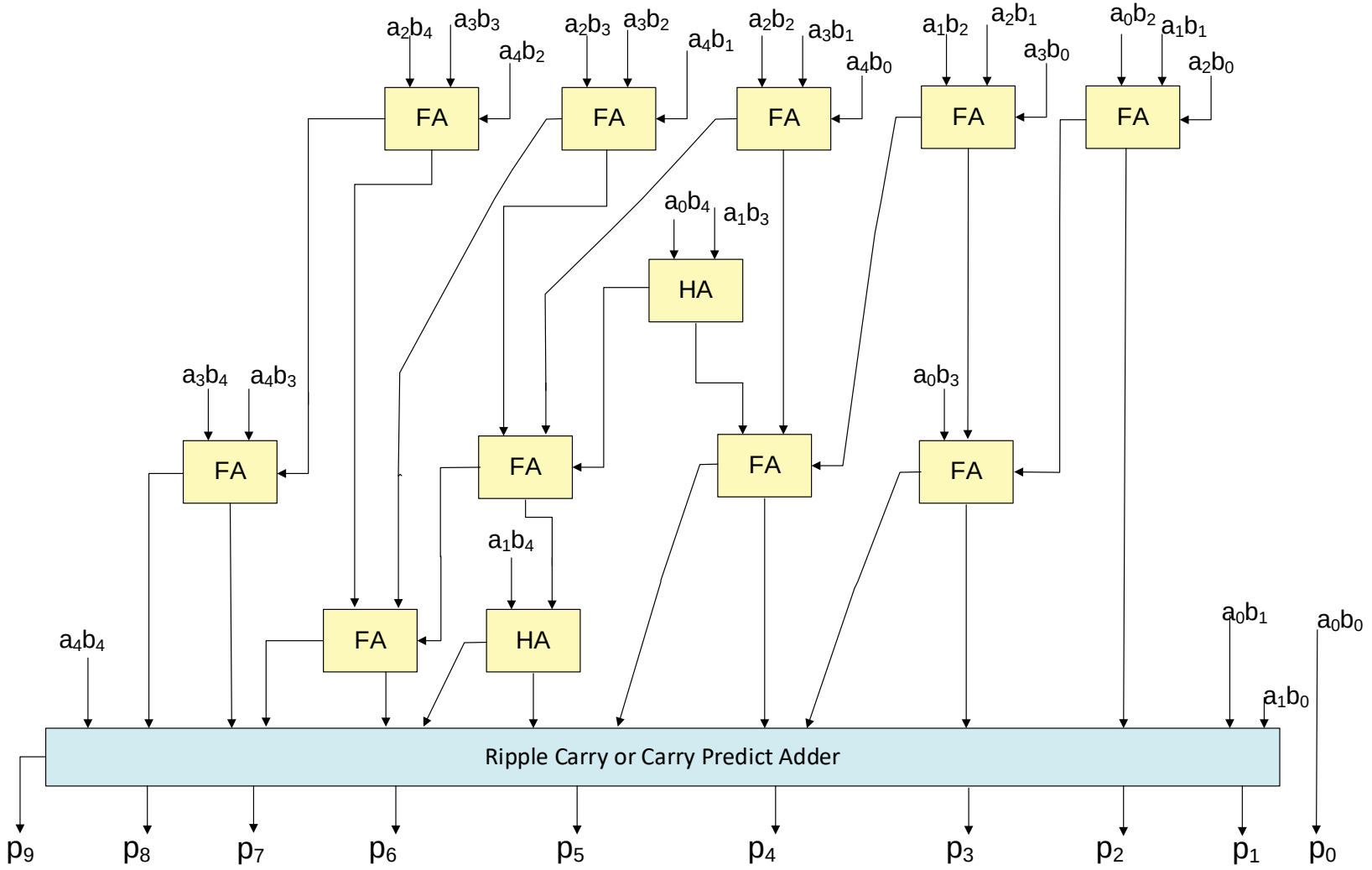
src: K. Pekmestzi, NTUA

Wallace Reduction



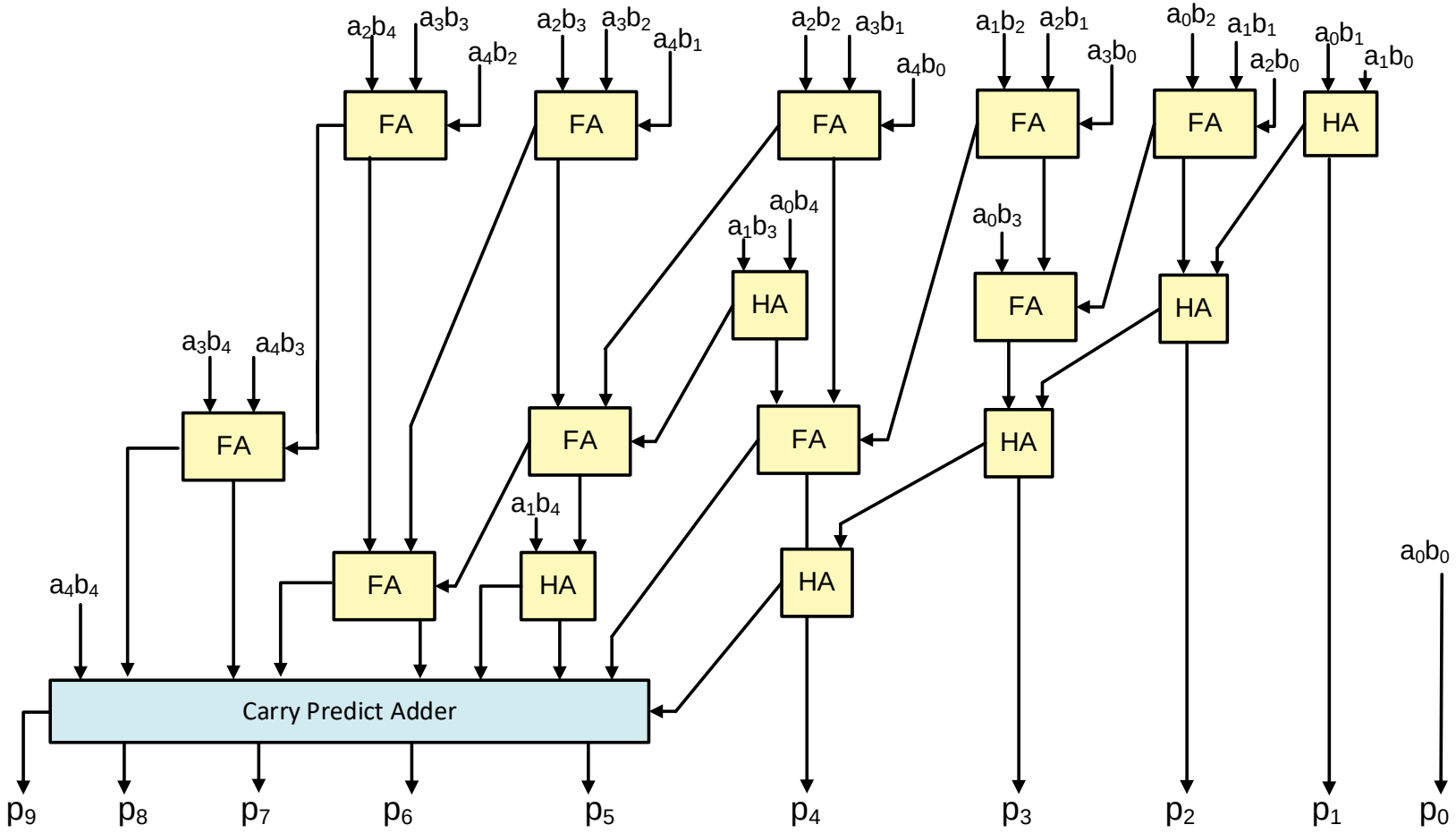
src: K. Pekmestzi, NTUA

5x5 Wallace Multiplier



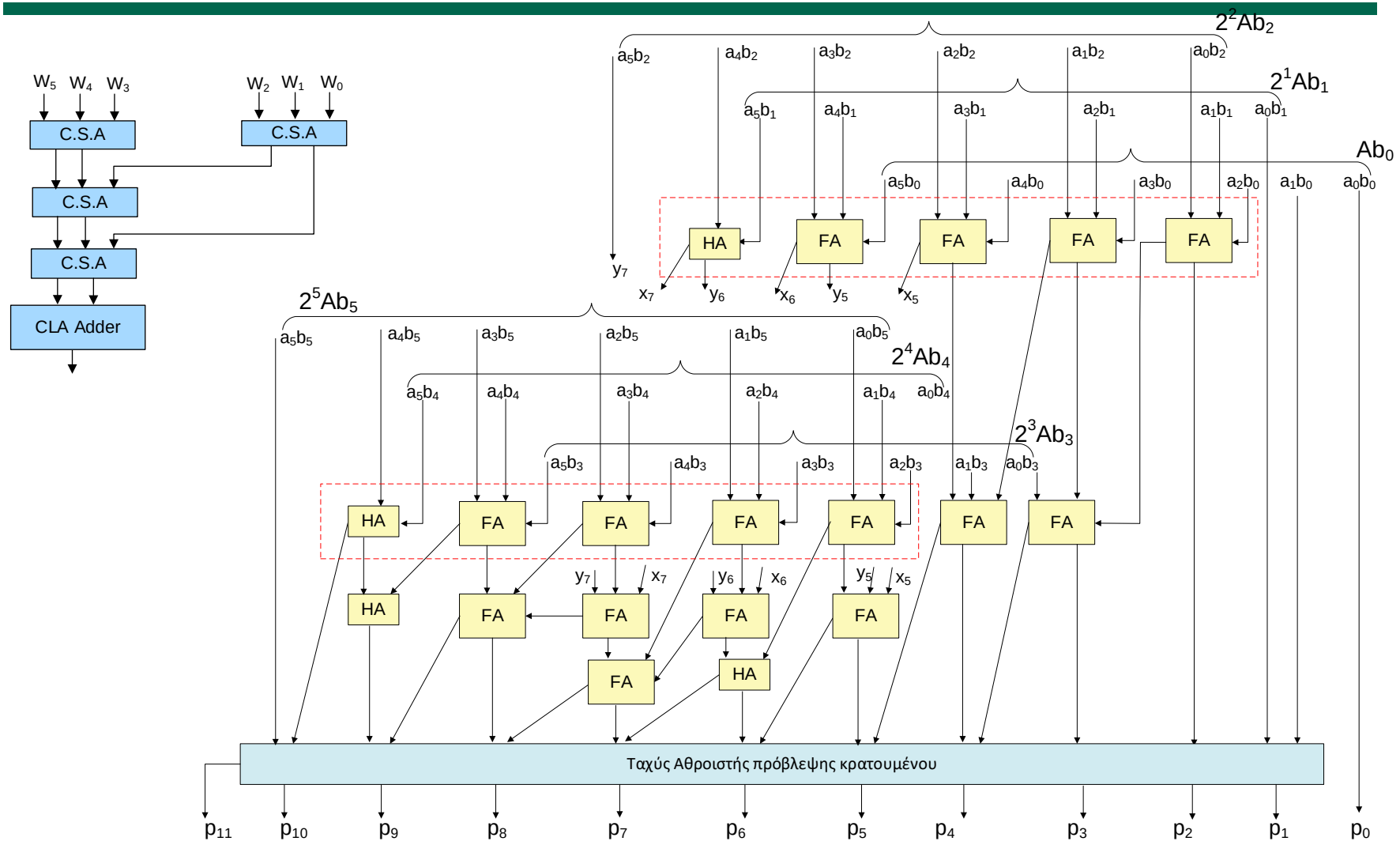
src: K. Pekmestzi, NTUA

5x5 Wallace Multiplier



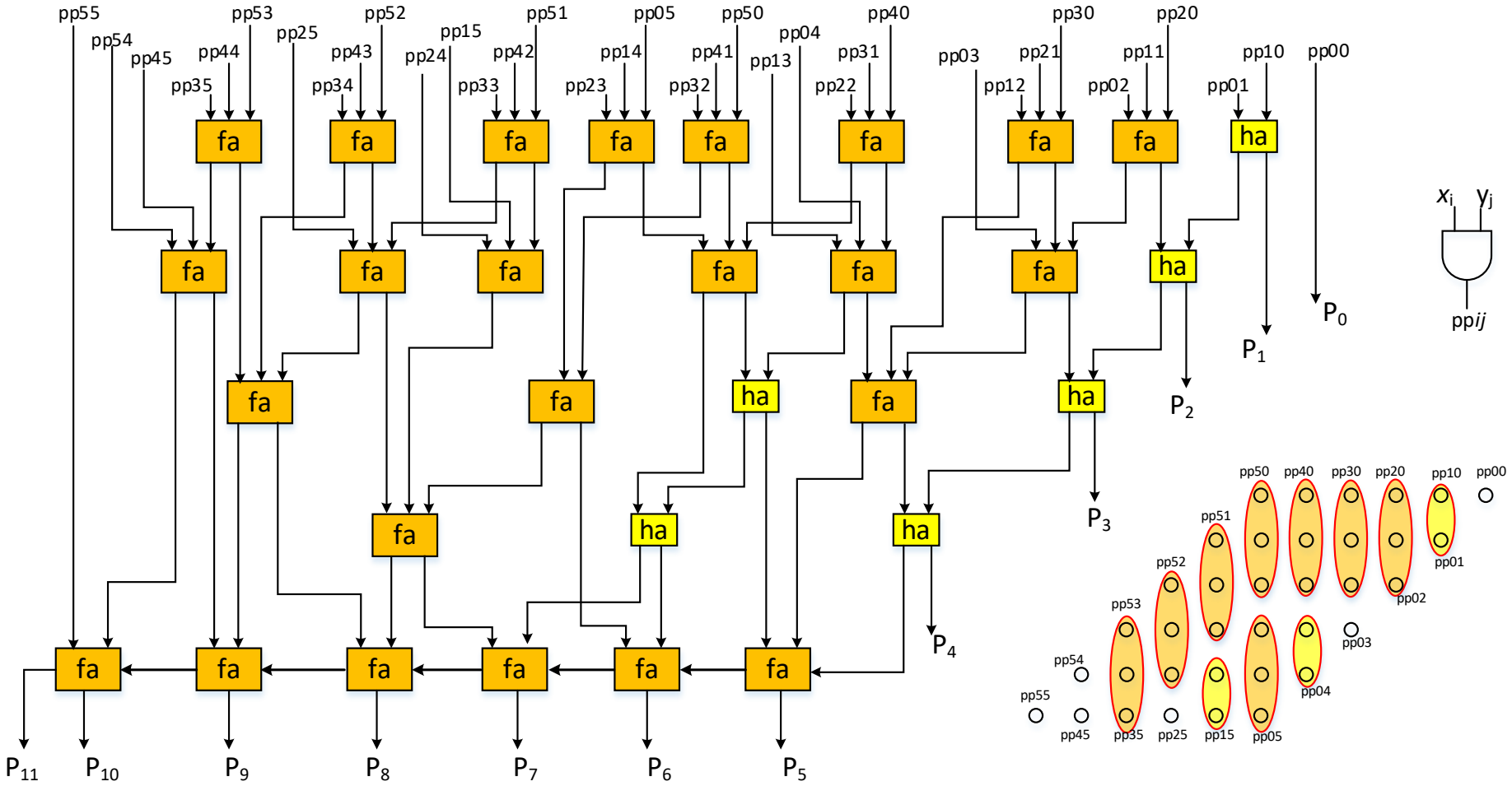
src: K. Pekmestzi, NTUA

6x6 Wallace Multiplier



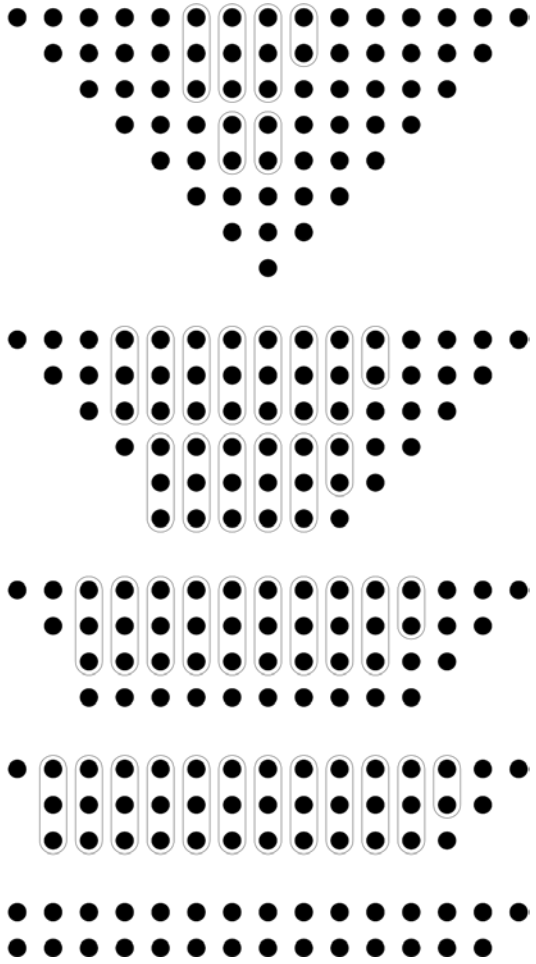
src: K. Pekmestzi, NTUA

6x6 Wallace Multiplier

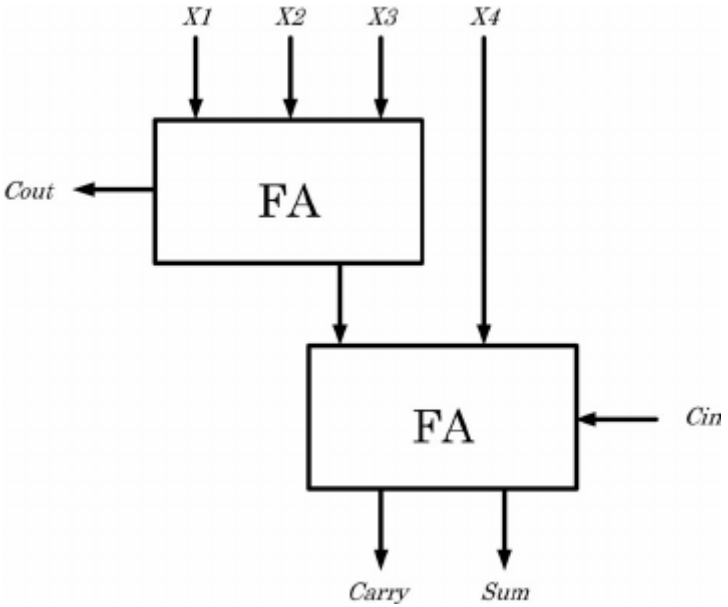


src: K. Pekmestzi, NTUA

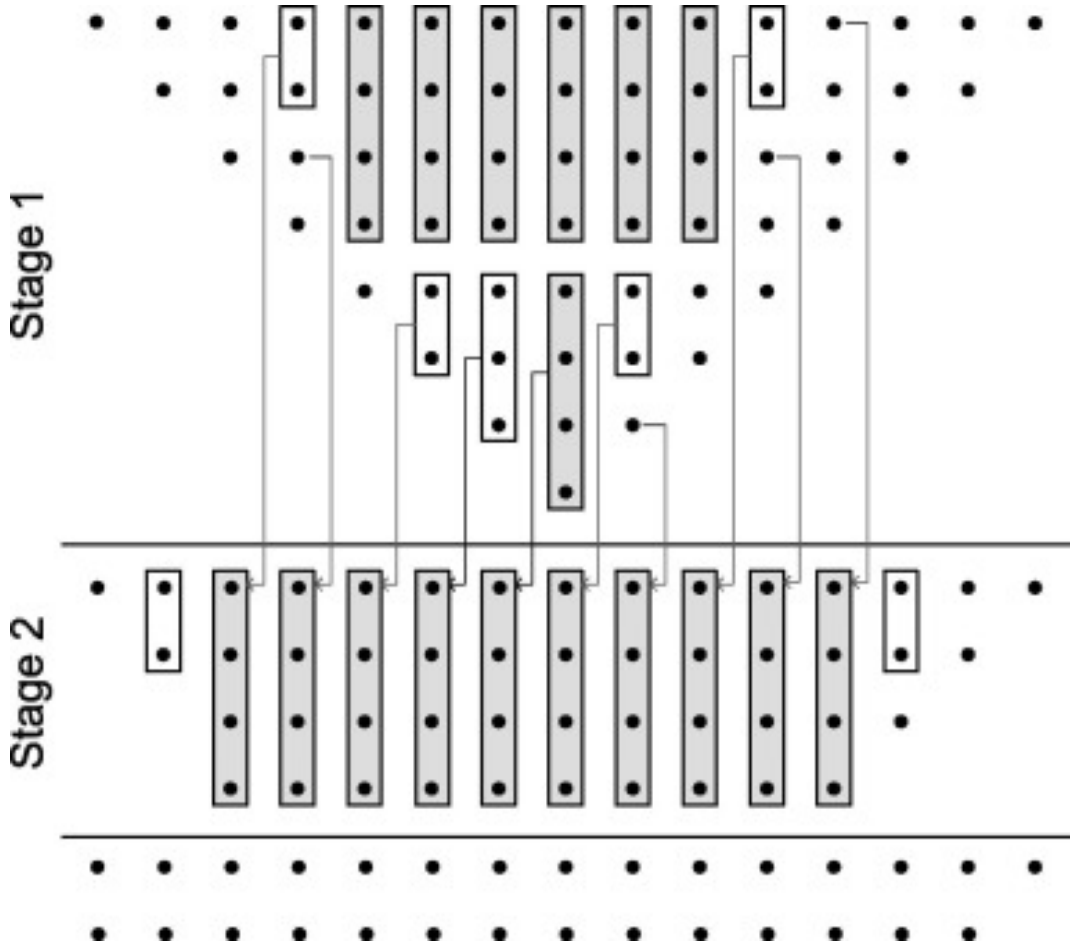
8x8 Dadda Multiplier



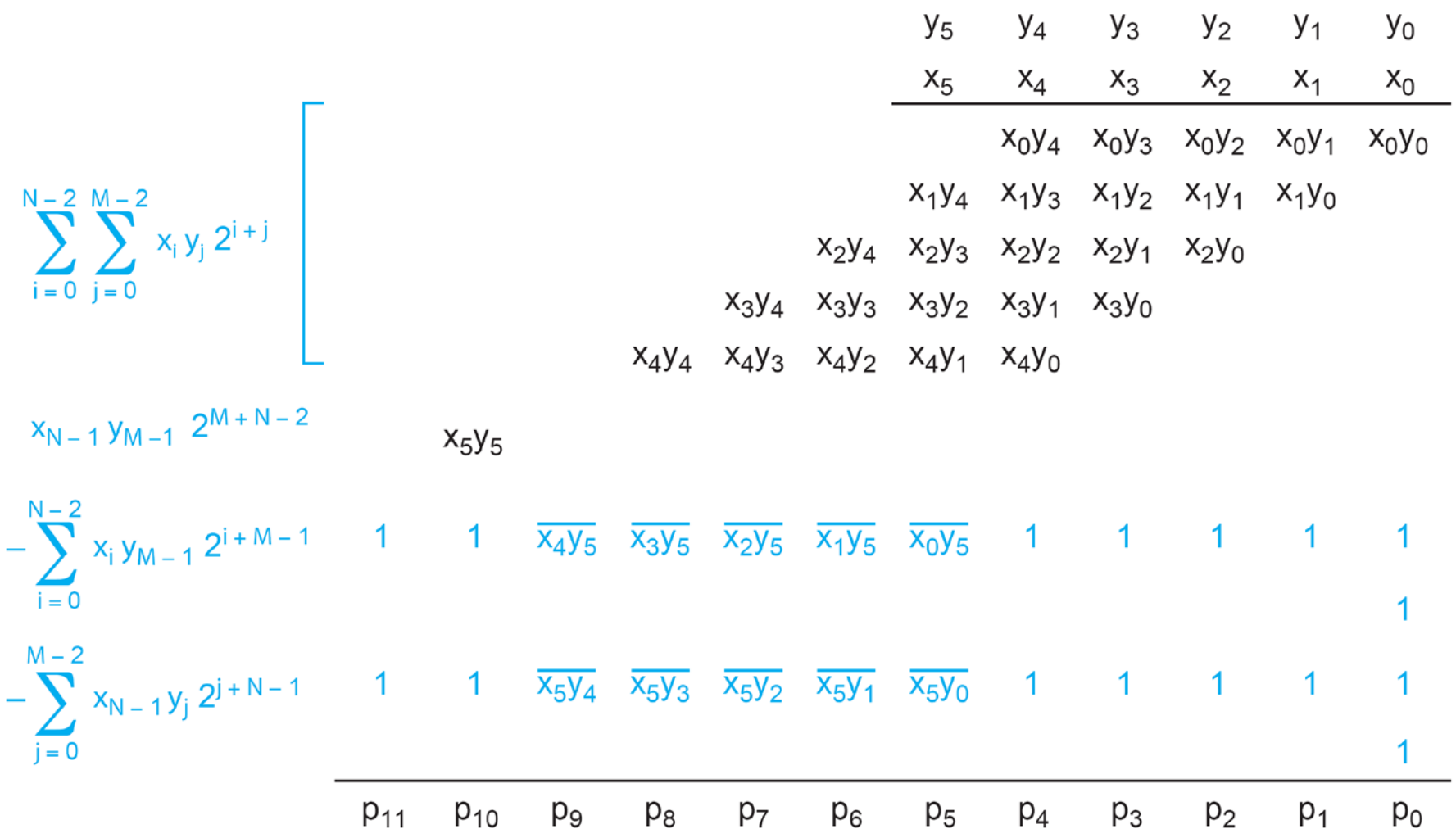
4:2 Compressor



8x8 Dadda Multiplier

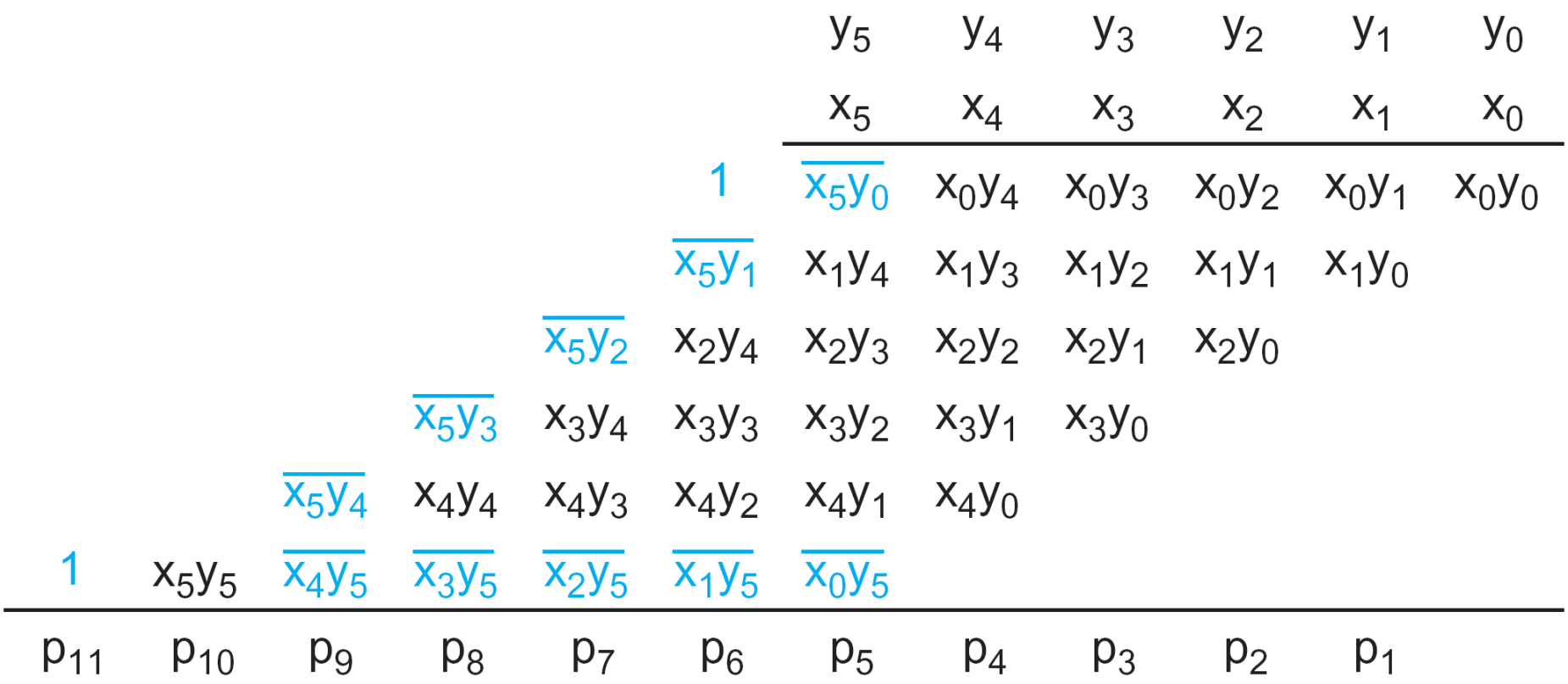


Signed Multiplier



src: https://cmosvlsi.com/

Signed Multiplier



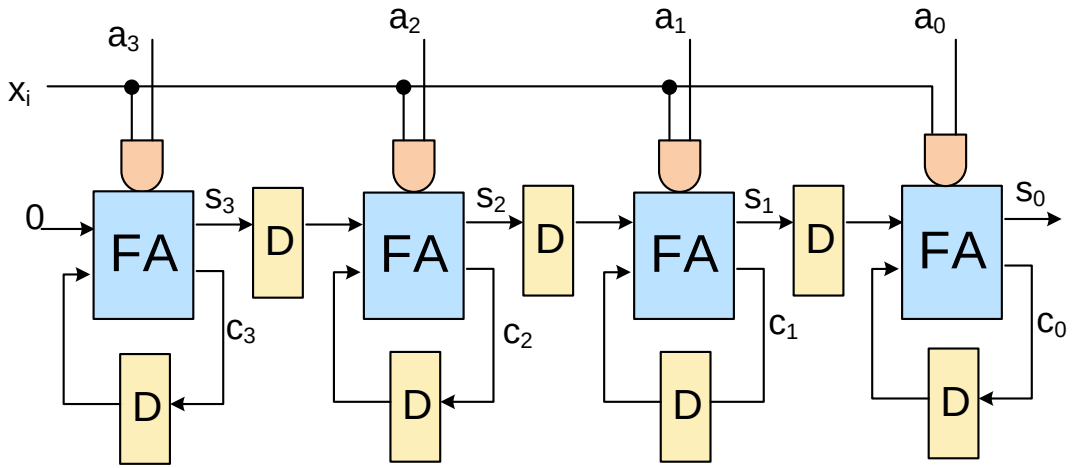
Simplified partial products for two's complement multiplier

src: https://cmosvlsi.com/

Serial Array Multiplier

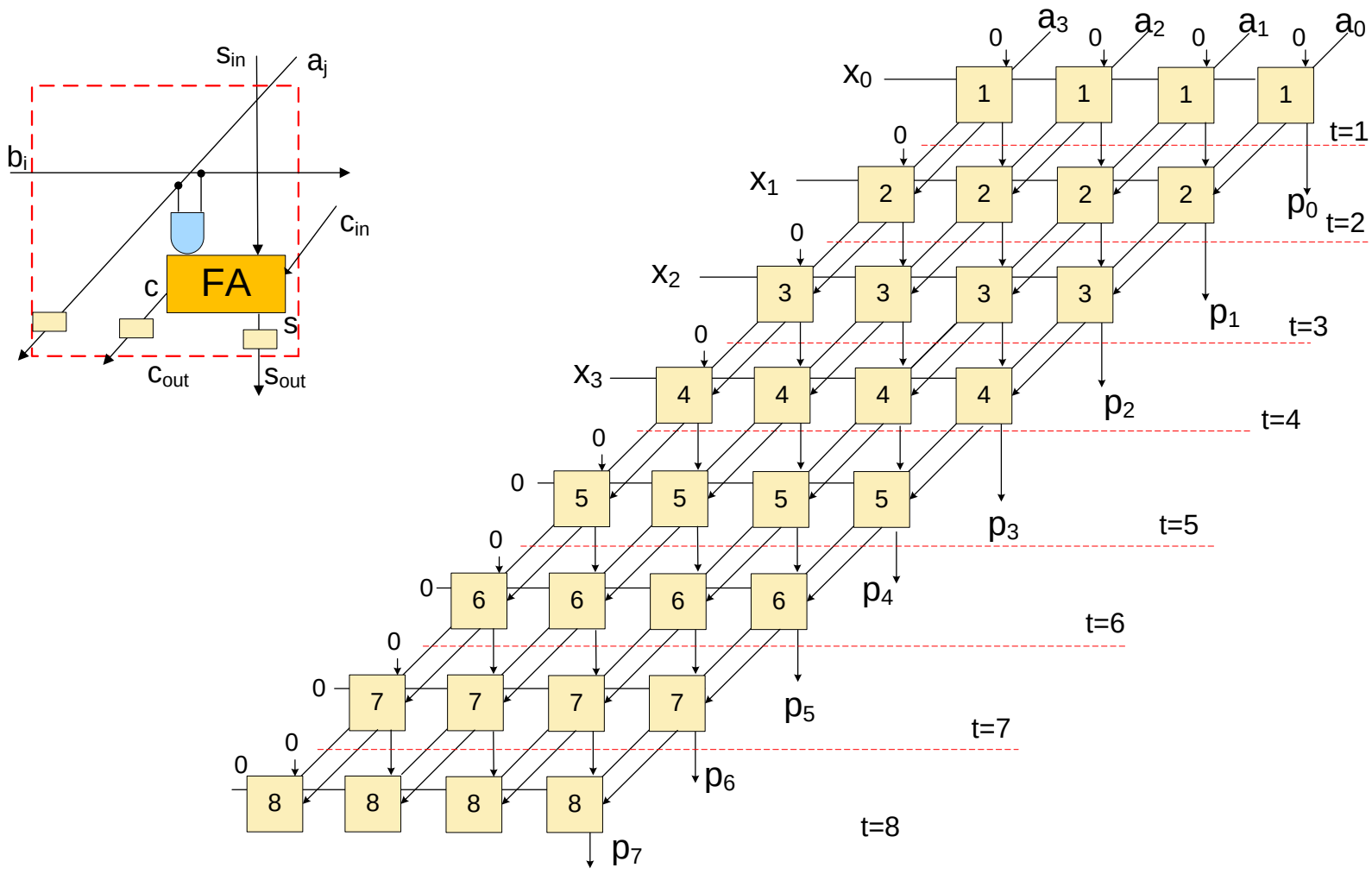
$$AX = Ax_0 2^0 + Ax_1 2^1 + Ax_2 2^2 + Ax_3 2^3$$

				$\alpha_3 x_0$	$\alpha_2 x_0$	$\alpha_1 x_0$	$\alpha_0 x_0$	
			$\alpha_3 x_1$	$\alpha_2 x_1$	$\alpha_1 x_1$	$\alpha_0 x_1$		
		$\alpha_3 x_2$	$\alpha_2 x_2$	$\alpha_1 x_2$	$\alpha_0 x_2$			
	$\alpha_3 x_3$	$\alpha_2 x_3$	$\alpha_1 b_3$	$\alpha_0 x_3$				
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	Product



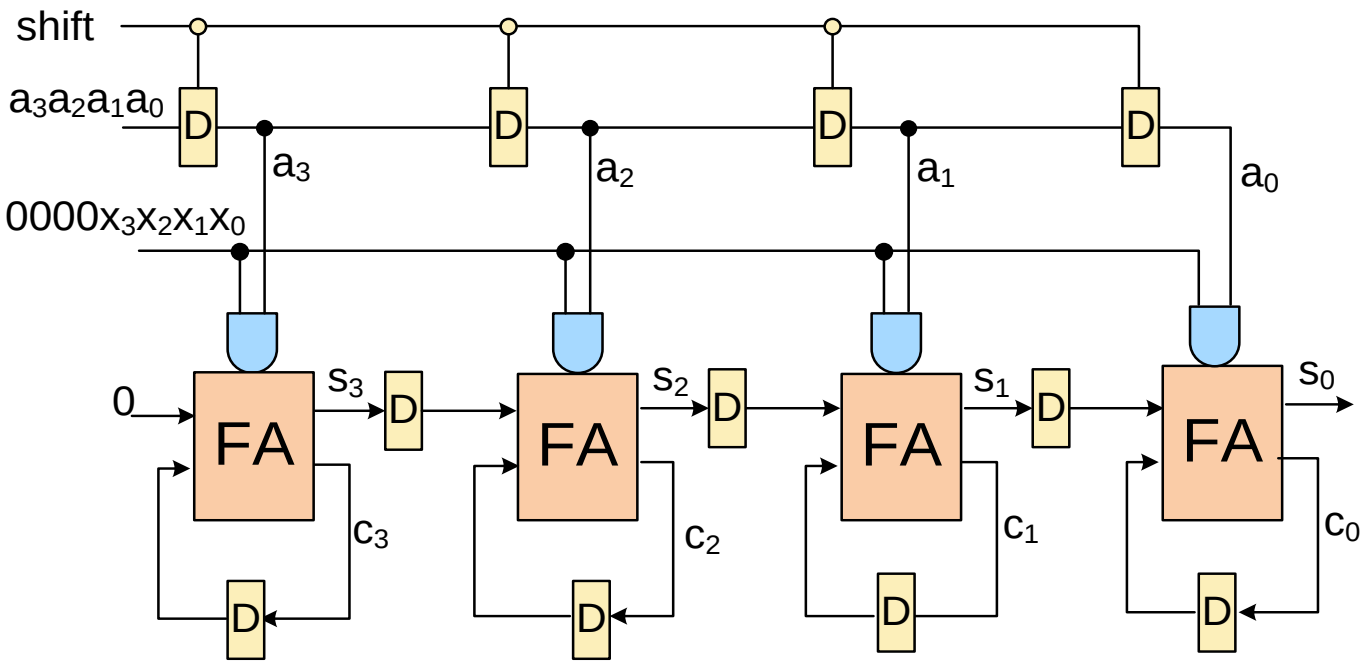
src: K. Pekmestzi, NTUA

Unroll in Time



src: K. Pekmestzi, NTUA

Save Operand



shift	1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
a	$a_0a_1a_2a_3$ (stable a) $a_0a_1a_2a_3$
x	0 0 0 0 $x_0x_1x_2x_3$ 0 0 0 0 (new x)
p=ax	$p_0p_1p_2p_3p_4p_5p_6p_7$ (new p)

src: K. Pekmestzi, NTUA

Fewer Partial Products

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-2^r encoding
- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder ☹

src: https://cmosvlsi.com/

Booth Encoding

- Instead of 3Y, try –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, try –2Y + 4Y in next partial product

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	SINGLE_i	DOUBLE_i	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

src: https://cmosvlsi.com/

Booth Encoding

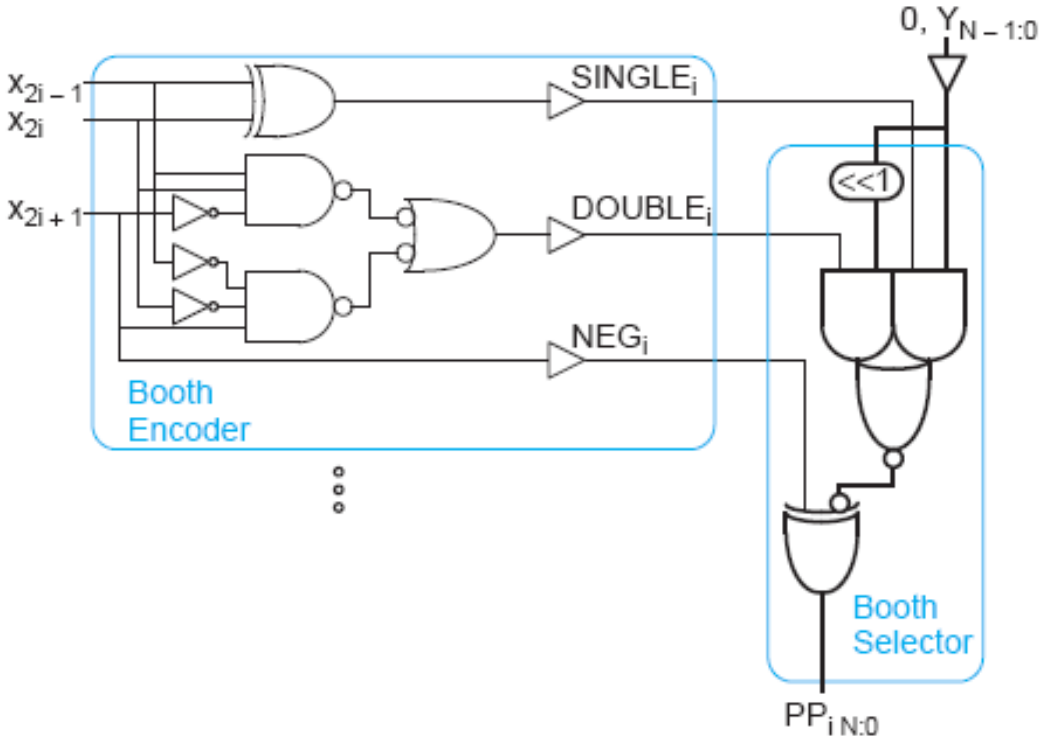
- Instead of 3Y, try –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, try –2Y + 4Y in next partial product

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	SINGLE_i	DOUBLE_i	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	$-0\ (=0)$	0	0	1

src: https://cmosvlsi.com/

Booth Hardware

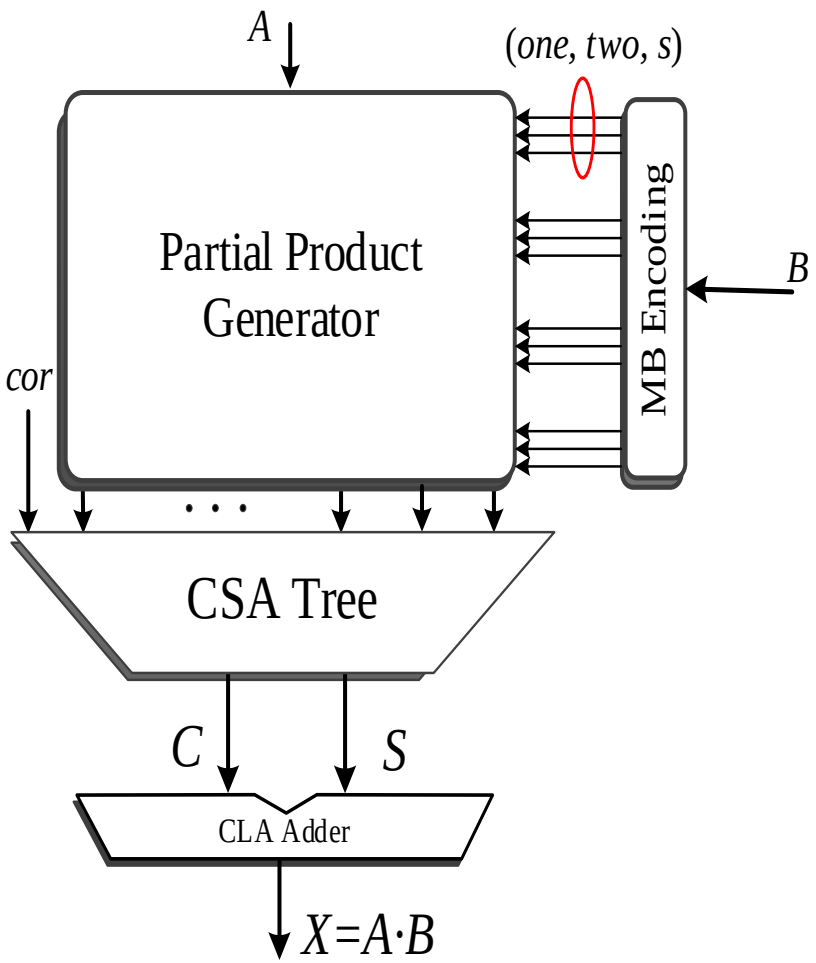
- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits



src: https://cmosvlsi.com/

Modified Booth Multiplier

Multiplicand A in 2's complement

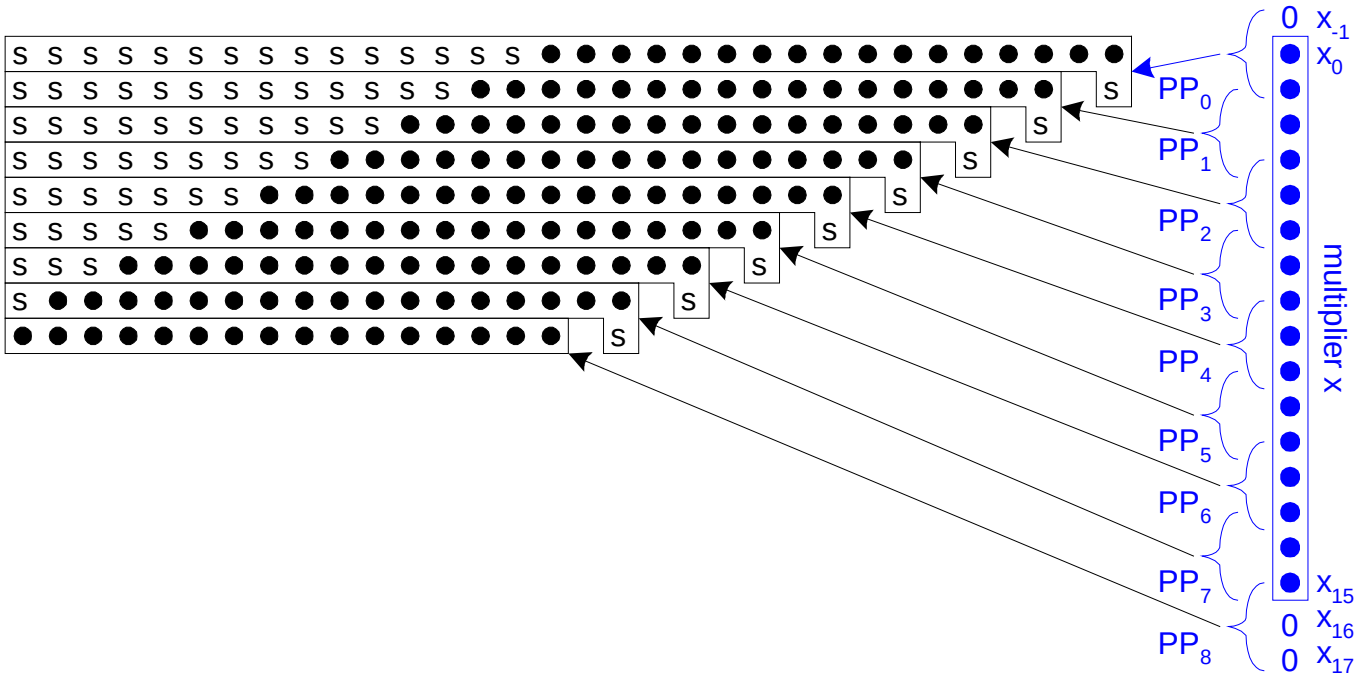


Multiplier B in 2's complement

src: K. Pekmestzi, NTUA

Sign Extension

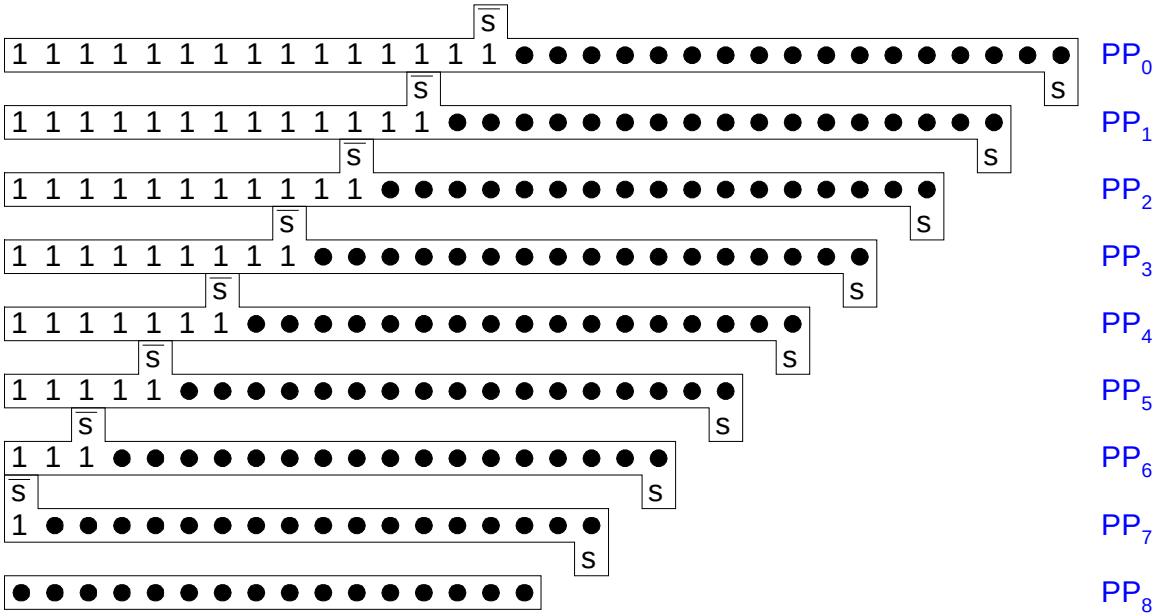
- Partial products can be negative
  - Require sign extension, which is cumbersome
  - High fanout on most significant bit



src: https://cmosvlsi.com/

Simplified Sign Ext.

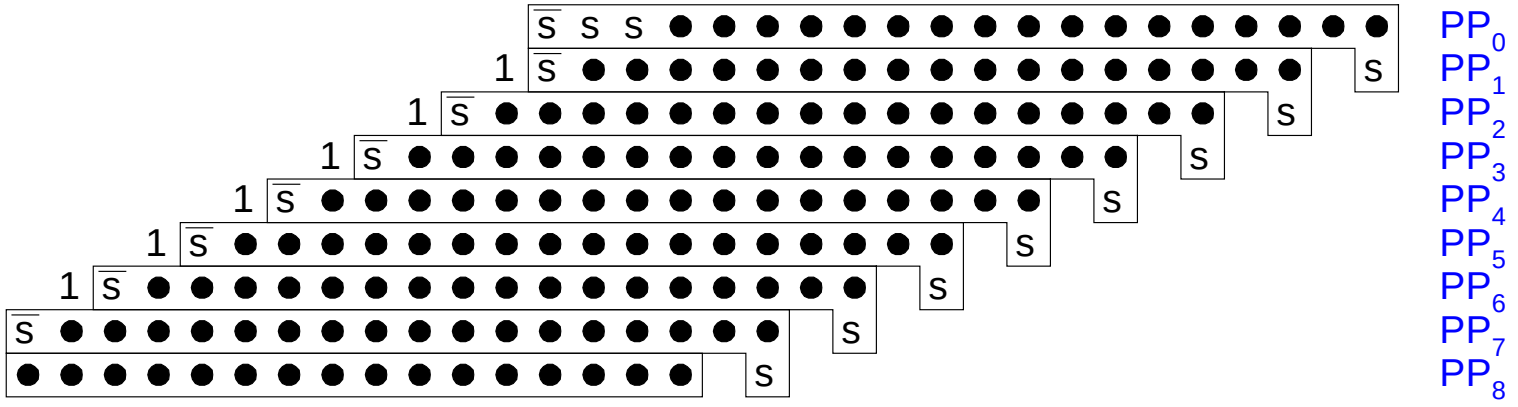
- Sign bits are either all 0's or all 1's
  - Note that all 0's is all 1's + 1 in proper column
  - Use this to reduce loading on MSB



src: https://cmosvlsi.com/

Even Simpler Sign Ext.

- No need to add all the 1's in hardware
  - Precompute the answer!



src: https://cmosvlsi.com/

Advanced Multiplication

- Signed vs. unsigned inputs
- Higher radix Booth encoding
- Array vs. tree CSA networks

src: https://cmosvlsi.com/