Basic Building Blocks

Lecture 5

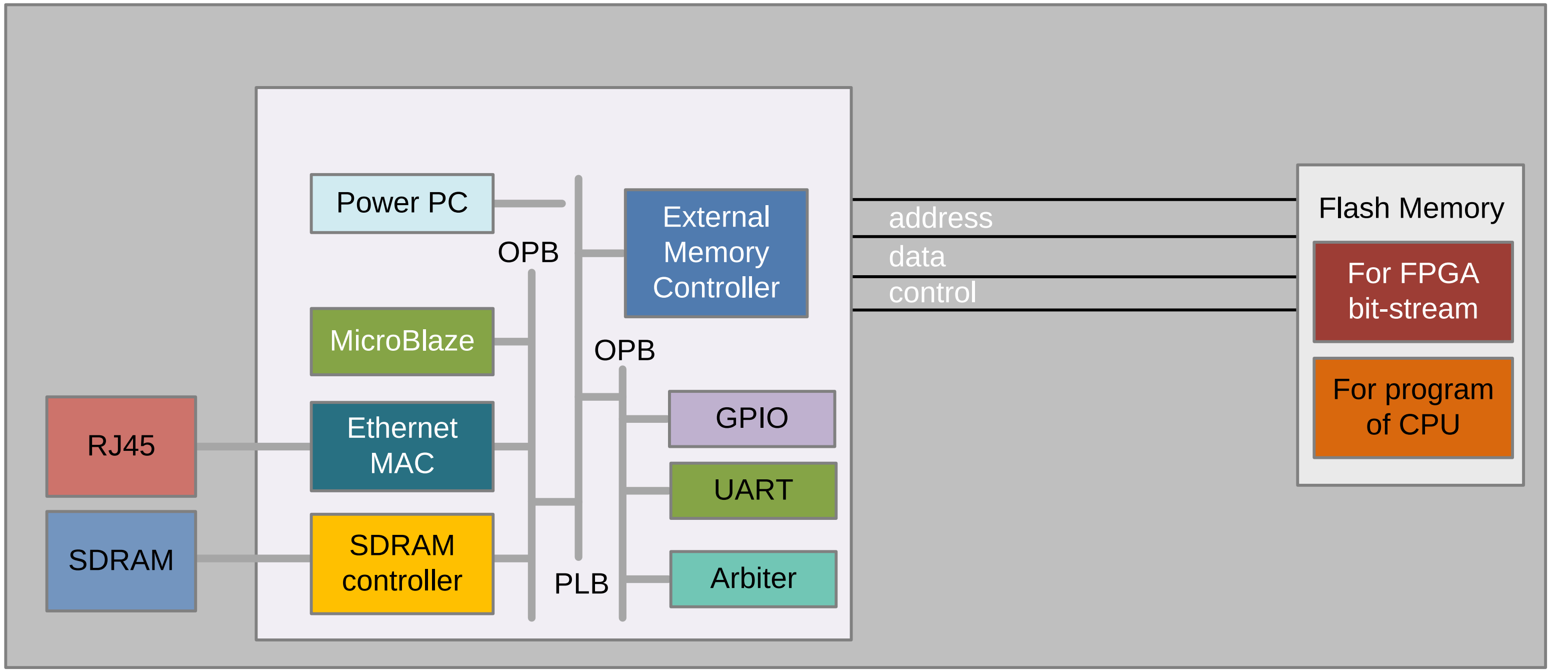
Dr. Shoab A. Khan

New Trends in FPGAs' Architecture

Embedded Units in FPGAs

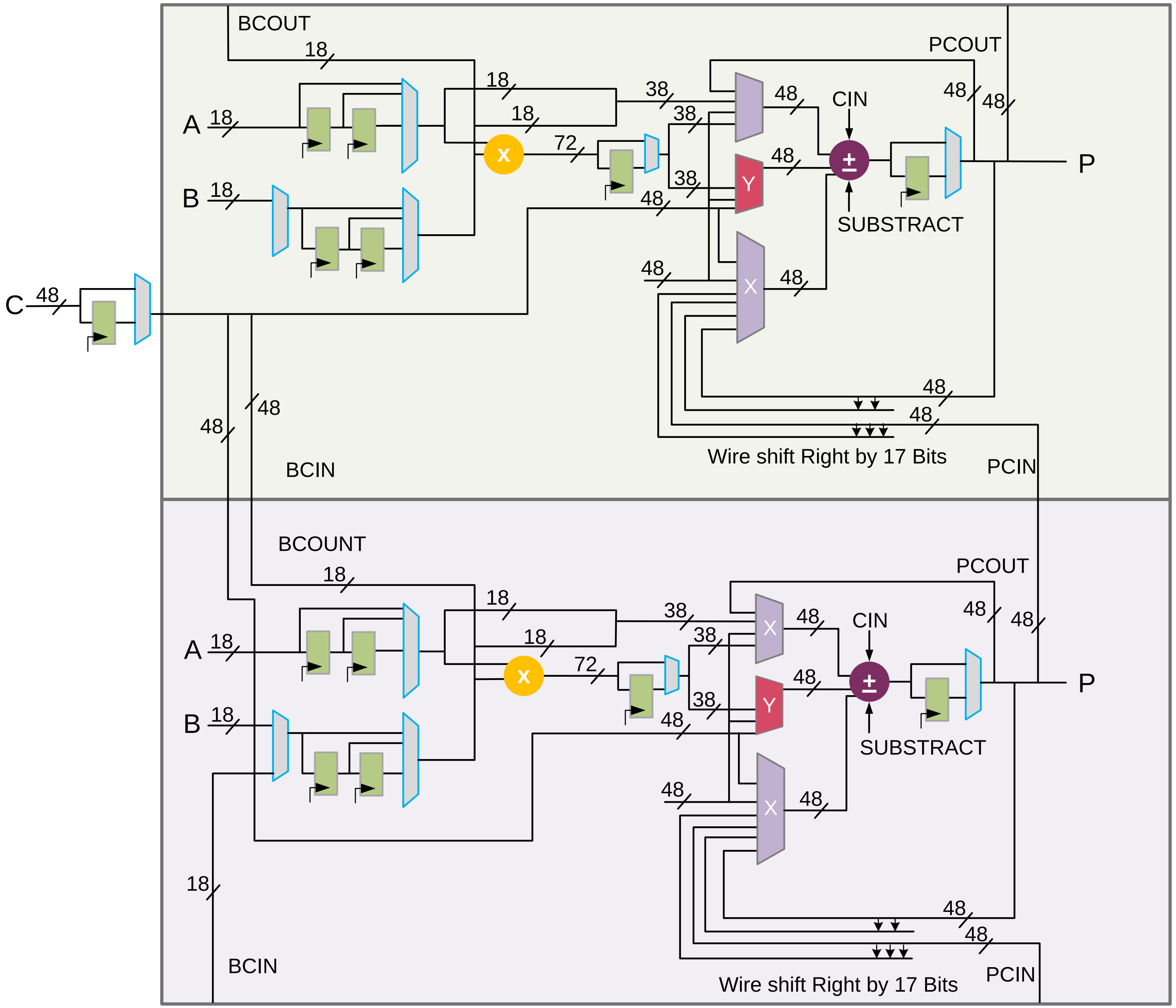
Embedding in FPGAs

- FPGA with PowerPC, MicroBlaze, Ethernet MAC and other embedded interfaces

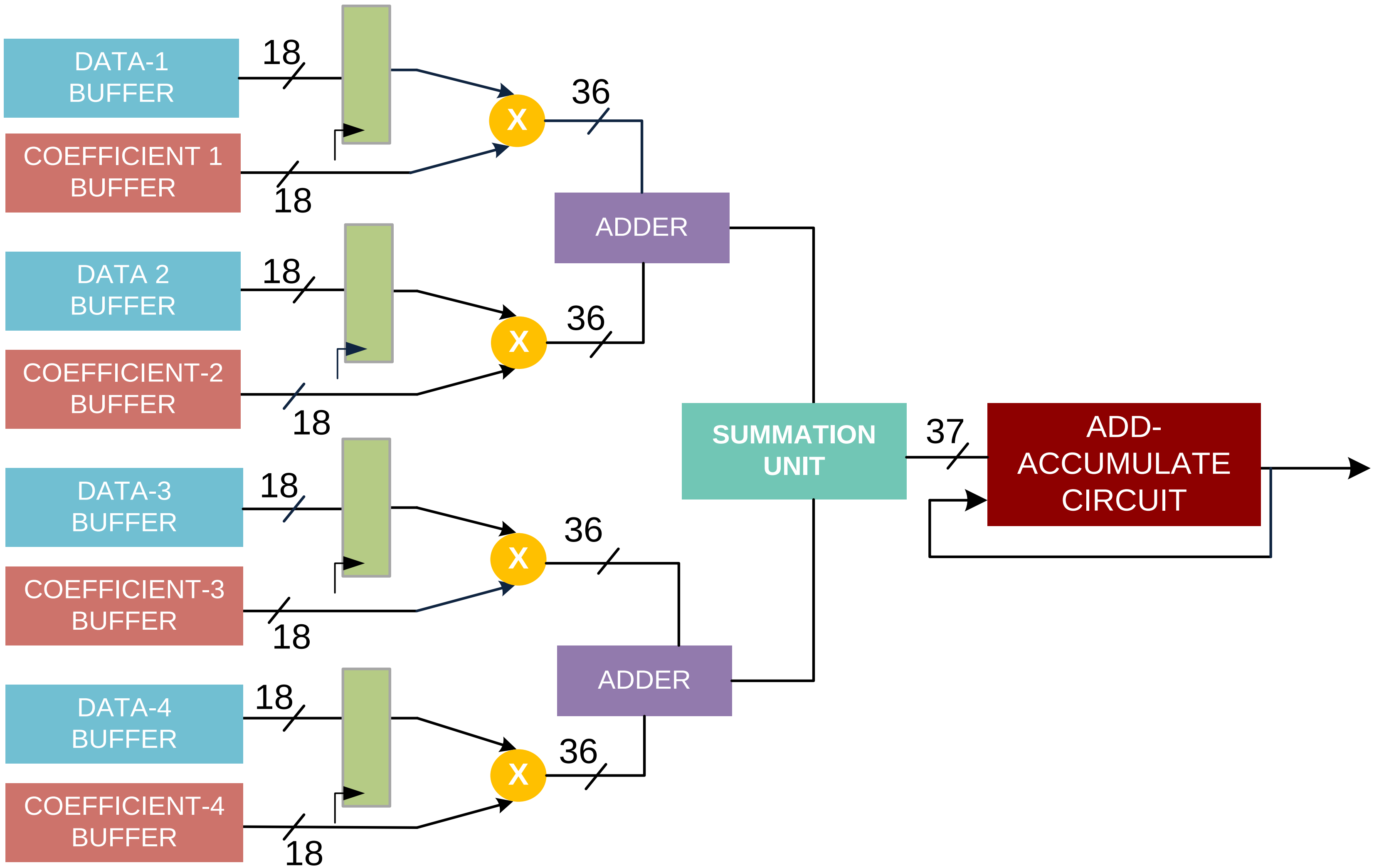


Embedded Arithmetic Units in FPGAs

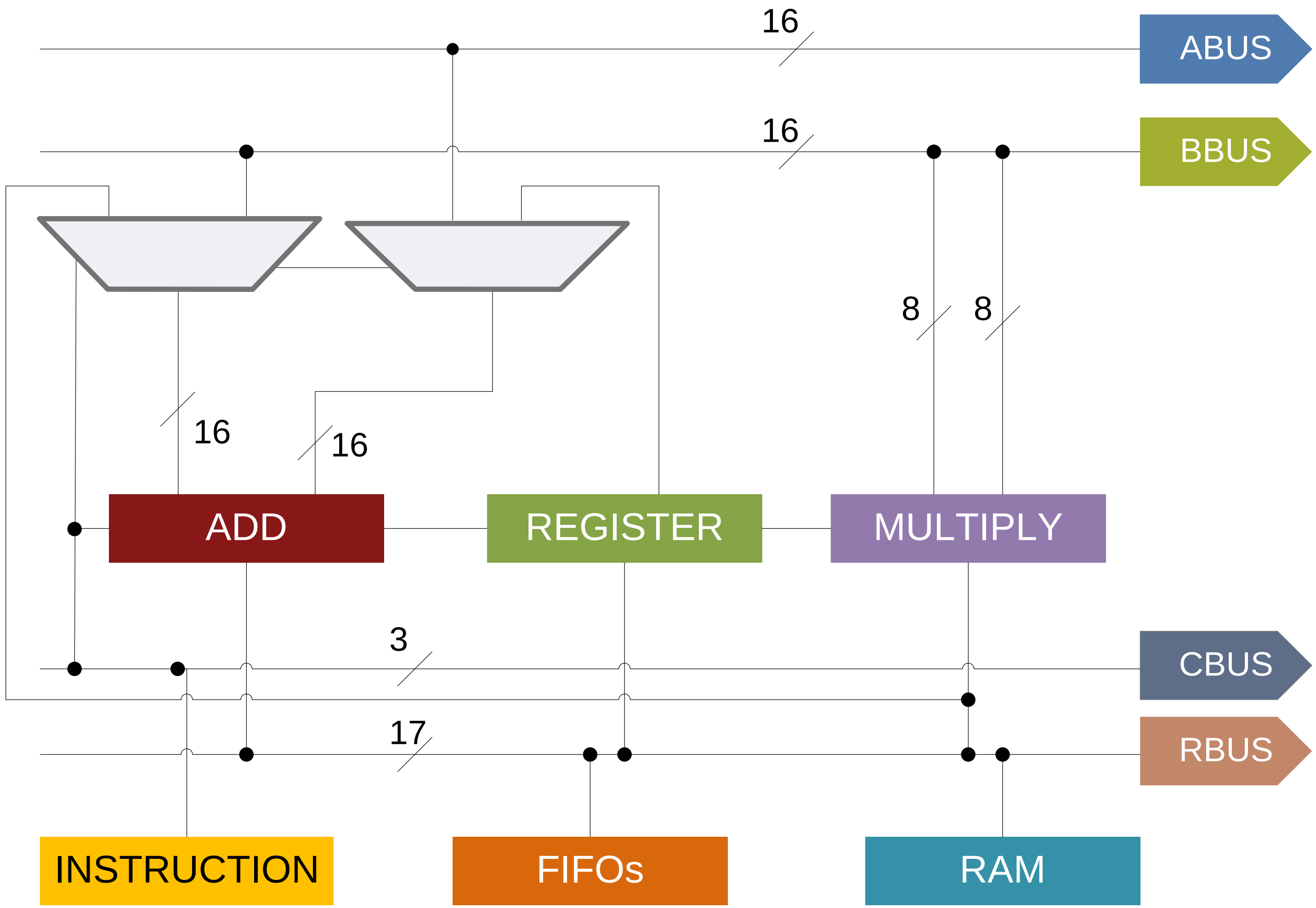
(a) DSP48 in Virtex™-4 FPGA (Derived from Xilinx documentation)



(b)18x18 multiplier and adder in Altera FPGA



8x8 multiplier and 16-bit adder in Quick Logic FPGA



18x18 multiplier in Virtex-II, Virtex-II pro and Spartan™-3 FPGA

Instantiation of Embedded Blocks

- ISE-provided template

```
// MULT18X18: 18 x 18 signed asynchronous multiplier
// Virtex-II/II-Pro, Spartan-3
// Xilinx HDL Language Template, version 9.1i
MULT18X18 MULT18X18_inst (
.P(P), // 36-bit multiplier output
.A(A), // 18-bit multiplier input
.B(B) // 18-bit multiplier input);
// End of MULT18X18_inst instantiation
```

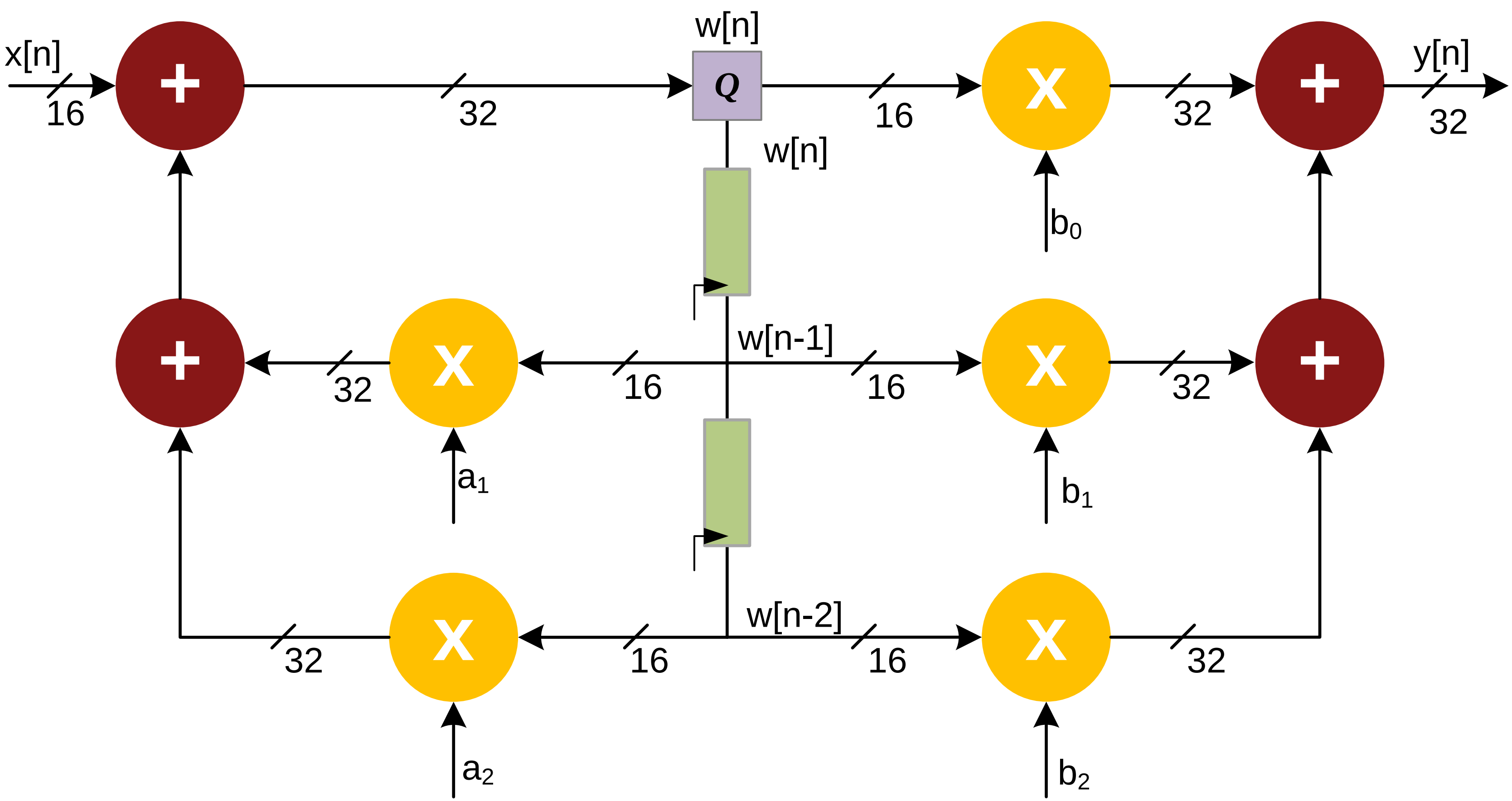
Embedded Multipliers

- Automatic instantiation

$$w[n] = a_1 w[n-1] + a_2 w[n-2] + x[n]$$

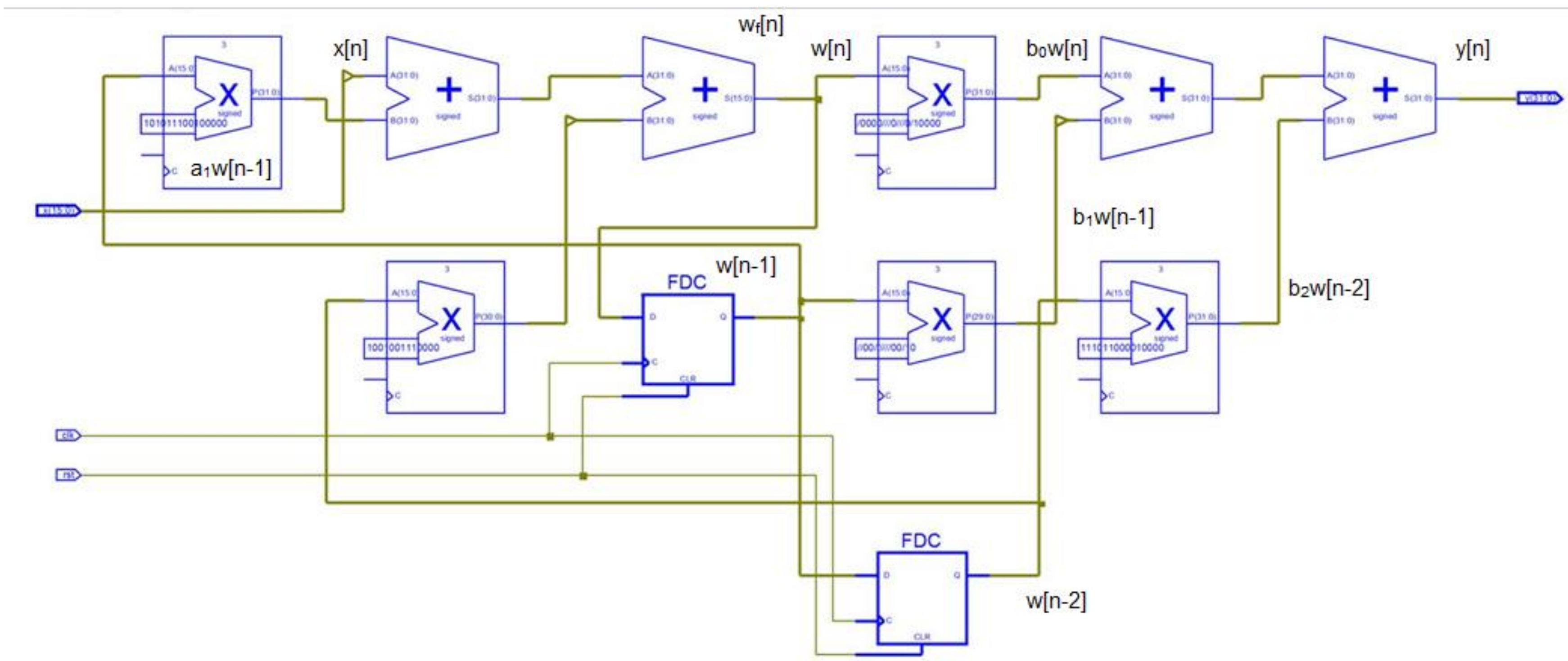
$$y[n] = b_0 w[n] + b_1 w[n-1] + b_2 w[n-2]$$

Block diagram of a 2nd Order IIR filter in Direct Form II Realization



(a)

RTL schematic generated by Xilinx's Integrated Software Environment



(b)

Synthesis of the design on Spartan™-3 FPGA, the multiplication and addition operations are mapped on DSP48 Multiply Accumulate (MAC) embedded blocks

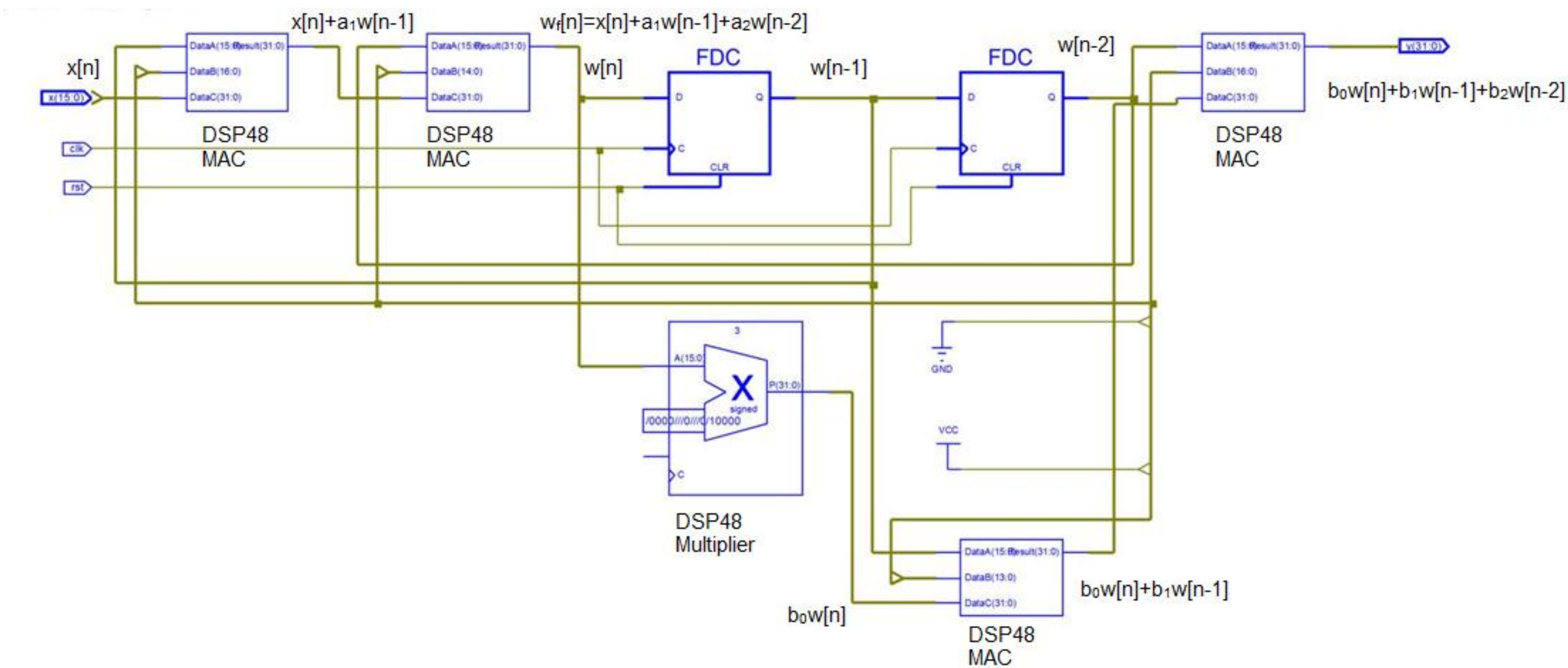
Selected Device : 3s400pq208-5

Minimum period: 10.917ns (Maximum Frequency: 91.597MHz)

Number of Slices:	58 out of 3584	1%
Number of Slice Flip Flops:	32 out of 7168	0%
Number of 4 input LUTs:	109 out of 7168	1%
Number of IOs:	50	
Number of bonded IOBs:	50 out of 141	35%
Number of MULT18X18s:	5 out of 16	31%
Number of GCLKs:	1 out of 8	12%

(c)

RTL schematic generated by Xilinx ISE for Virtex™ 4 target device . The multiplication and addition operations are mapped on DSP48 multiply accumulate (MAC) embeded blocks

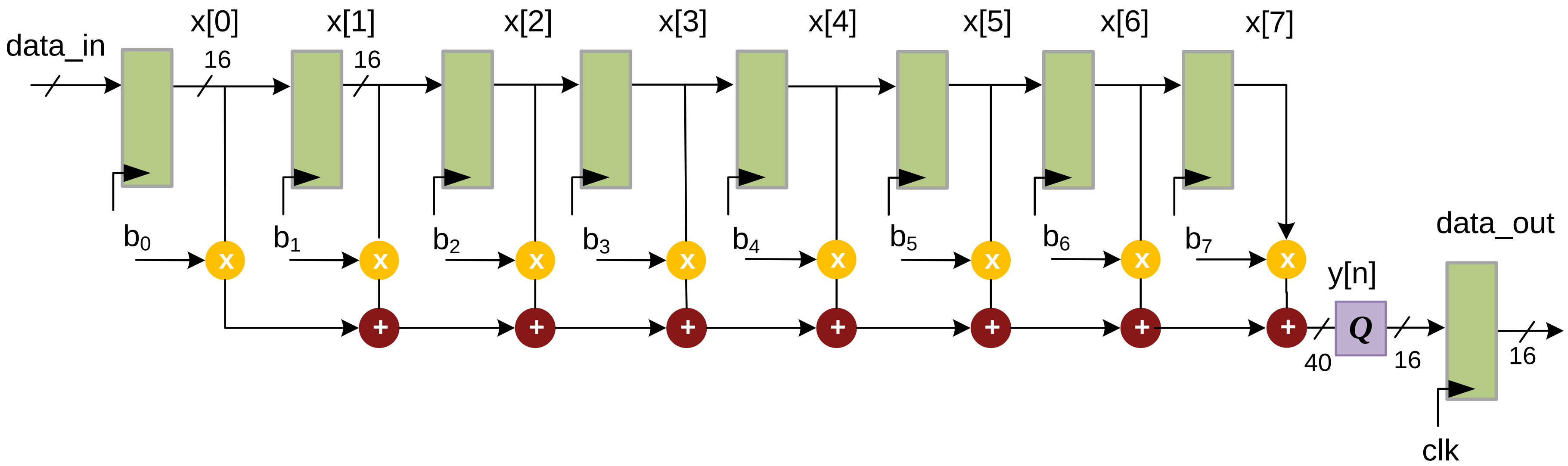


(d)

```
module iir(xn, clk, rst, yn);

// x[n] is in Q1.15 format
input signed [15:0] xn;
input clk, rst;

// y[n] is in Q2.30 format
output signed [31:0] yn;

// Full precision w[n] in Q2.30 format
wire signed [31:0] wfn;

// Quantized w[n] in Q1.15 format
wire signed [15:0] wn;

// w[n-1]and w[n-2] in Q1.15 format
reg signed [15:0] wn_1, wn_2;

// all the coefficients are in Q1.15 format
wire signed [15:0] b0 = 16'ha7b0;
wire signed [15:0] b1 = 16'hf2b2;
wire signed [15:0] b2 = 16'h7610;
wire signed [15:0] a1 = 16'h5720;
wire signed [15:0] a2 = 16'h1270;
```

```
// w[n] in Q2.30 format with one redundant sign bit
 assign wfn = wn_1*a1+wn_2*a2;

/* through away redundant sign bit and keeping
  16 MSB and adding x[n] to get w[n] in Q1.15 format */

 assign wn = wfn[30:15]+xn;

// computing y[n] in Q2.30 format with one redundant sign bit
 assign yn = b0*wn + b1*wn_1 + b2*wn_2;

always @(posedge clk or posedge rst)
begin
		if(rst)
		begin
			wn_1 <= 0;
			wn_2 <=0;
		end
		else
		begin
			wn_1 <= wn;
			wn_2 <= wn_1;
		end
End

Endmodule
```

An 8-tap Direct Form (DF)-I FIR filter



Example of Optimized Mapping

```
// ----------------------------------------------------------------------
// Module: fir_filter
// ----------------------------------------------------------------------
// ----------------------------------------------------------------------
// Discrete-Time FIR Filter
// ------------------------------
// Filter Structure  : Direct-Form FIR
// Filter Order      : 7
//
//        Input Format:  Q1.15
//        Output Format: Q1.15
// ----------------------------------------------------------------------

module fir_filter (
 input   clk,
 input   signed [15:0] data_in, //Q1.15
 output  reg signed [15:0] data_out //Q1.15
 );
```

Contd…

```
// Constants, filter is designed using Matlab FDATool, all coeffs are in Q1.15 format
parameter signed [15:0] b0 = 16'b1101110110111011;
parameter signed [15:0] b1 = 16'b1110101010001110;
parameter signed [15:0] b2 = 16'b0011001111011011;
parameter signed [15:0] b3 = 16'b0110100000001000;
parameter signed [15:0] b4 = 16'b0110100000001000;
parameter signed [15:0] b5 = 16'b0011001111011011;
parameter signed [15:0] b6 = 16'b1110101010001110;
parameter signed [15:0] b7 = 16'b1101110110111011;

reg  signed [15:0] xn [0:7]; // input sample delay line
wire signed [39:0] yn; // Q8.32

// Block Statements
always @(posedge clk)
Begin
	xn[0] <= data_in;
	xn[1] <= xn[0];
	xn[2] <= xn[1];
	xn[3] <= xn[2];
	xn[4] <= xn[3];
```

Contd…

```
        xn[5] <= xn[4];
        xn[6] <= xn[5];
        xn[7] <= xn[6];
        data_out <= yn[30:15]; // bring the output back in Q1.15 format
end

assign yn = xn[0] * b0 + xn[1] * b1 + xn[2] * b2 +
          xn[3] * b3 + xn[4] * b4 + xn[5] * b5 +
          xn[6] * b6 + xn[7] * b7;

endmodule  // fir_filter
```

Synthesis reports: (a)) Eight 18 $\times$ 18-bit embedded multipliers and seven adders from generic logic blocks are used on a Spartan™-3 family of FPGA

Selected device : 3s200pq208-5

Minimum period: 23.290 ns

(Maximum frequency: 42.936 MHz)

Number of slices:	185	out of	1920	9%
Number of Slice Flip Flops:	144	out of	3840	3%
Number of 4 input LUTs:	217	out of	3840	5%
Number of IOs:	33			
Number of bonded IOBs:	33	out of	141	23%
Number of MULT18X18s:	8	out of	12	66%
Number of GCLKs:	1	out of	8	12%

(a)

(b) Eight DSP48 embedded blocks are used once mapped on a Vertix-4 family of FPGA

Selected device : 4vlx15sf363-12

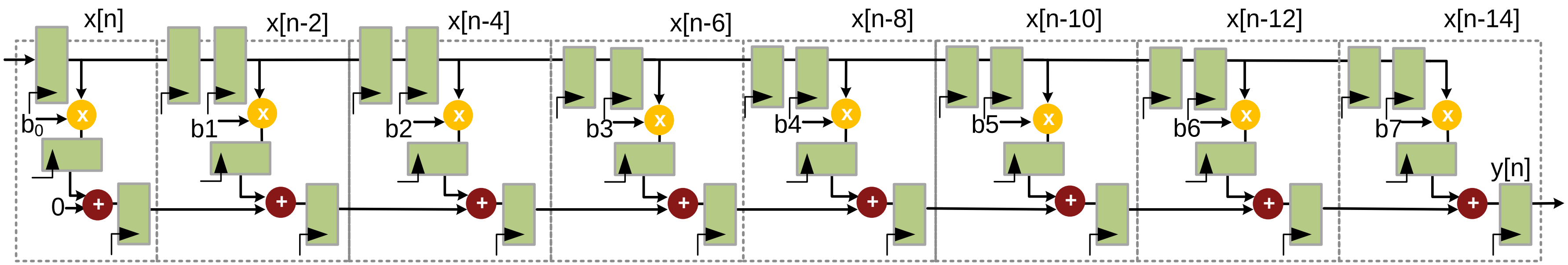
Minimum period: 16.958 ns

(Maximum frequency: 58.969 MHz)

Number of Slices:	9	out of	6144	0%
Number of Slice Flip Flops:	16	out of	12288	0%
Number of IOs:	33			
Number of bonded IOBs:	33	out of	240	13%
Number of GCLKs:	1	out of	32	3%
Number of DSP48s:	8	out of	32	25%

(b)

Optimized Mapping



Optimized Mapping

Selected Device : 4vlx15sf363-12
Minimum period: 1.891ns (Maximum Frequency: 528.821MHz)

Number of Slices:	9 out of 6144	0%
Number of Slice Flip Flops:	16 out of 12288	0%
Number of IOs:	33	
Number of bonded IOBs:	33 out of 240	13%
Number of GCLKs:	1 out of 32	3%
Number of DSP48s:	8 out of 32	25%

```
module fir_filter_pipeline (
  input   clk;
  input   signed [15:0] data_in;    //Q1.15
  output  signed [15:0] data_out;   //Q1.15

  // Constants, filter is designed using Matlab FDATool, all coeffs are in Q1.15 format
  parameter signed [15:0] b0 = 16'b1101110110111011;
  parameter signed [15:0] b1 = 16'b1110101010001110;
  parameter signed [15:0] b2 = 16'b0011001111011011;
  parameter signed [15:0] b3 = 16'b0110100000001000;
  parameter signed [15:0] b4 = 16'b0110100000001000;
  parameter signed [15:0] b5 = 16'b0011001111011011;
  parameter signed [15:0] b6 = 16'b1110101010001110;
  parameter signed [15:0] b7 = 16'b1101110110111011;
  reg  signed [15:0] xn   [0:13] ; // one stage pipelined input sample delay line
  reg  signed [32:0] prod [0:7];   // pipeline product registers in Q2.30 format
  wire signed [39:0] yn;           // Q10.30
  reg  signed [39:0] mac  [0:7];   // pipelined mac registers in Q10.30 format

  integer i;

  always @( posedge clk)
  begin
```

```
 xn[0] <= data_in;
      for (i=0; i<13; i=i+1)
            xn[i+1]=x[i];
        data_out <= yn[30:14];    // bring the output back in Q1.15 format
 end

 always @( posedge clk)
 begin
 prod[0] <= xn[0] * b0;
 prod[1] <= xn[2] * b1;
 prod[2] <= xn[4] * b2;
 prod[3] <= xn[6] * b3;
 prod[4] <= xn[8] * b4;
 prod[5] <= xn[10] * b5;
 prod[6] <= xn[12] * b6;
 prod[7] <= xn[14] * b7;
end
always @(posedge clk)
begin
            mac[0] <= prod[0];
            for (i=0; i<7; i=i+1)
                        mac[i+1] <= mac[i]+prod[i+1];
end

assign yn = mac[7];

endmodule                    // fir_filter
```

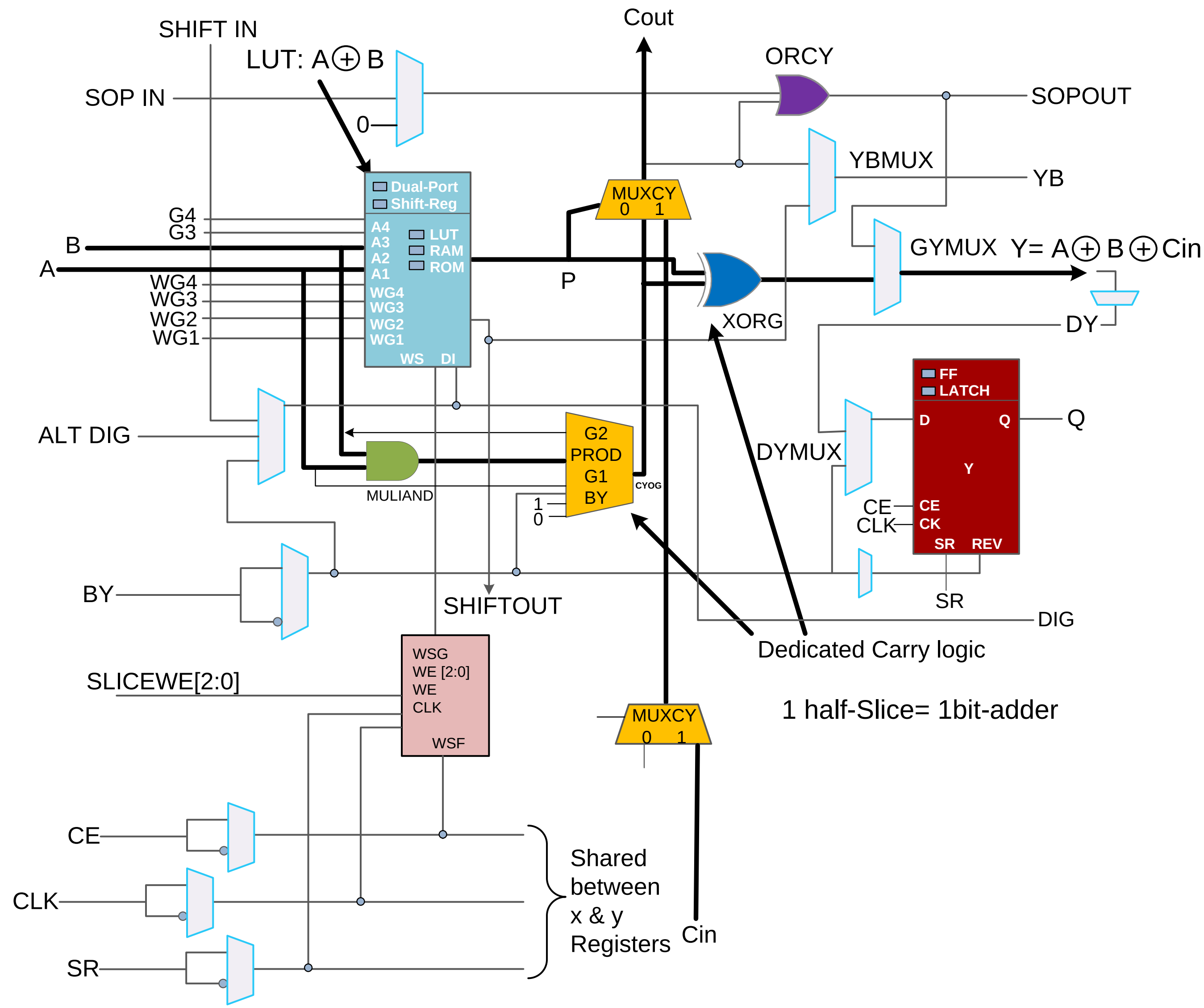
Carry Chain Logic in FPGAs

$$c_{i+1} = g_i + p_i c_i$$

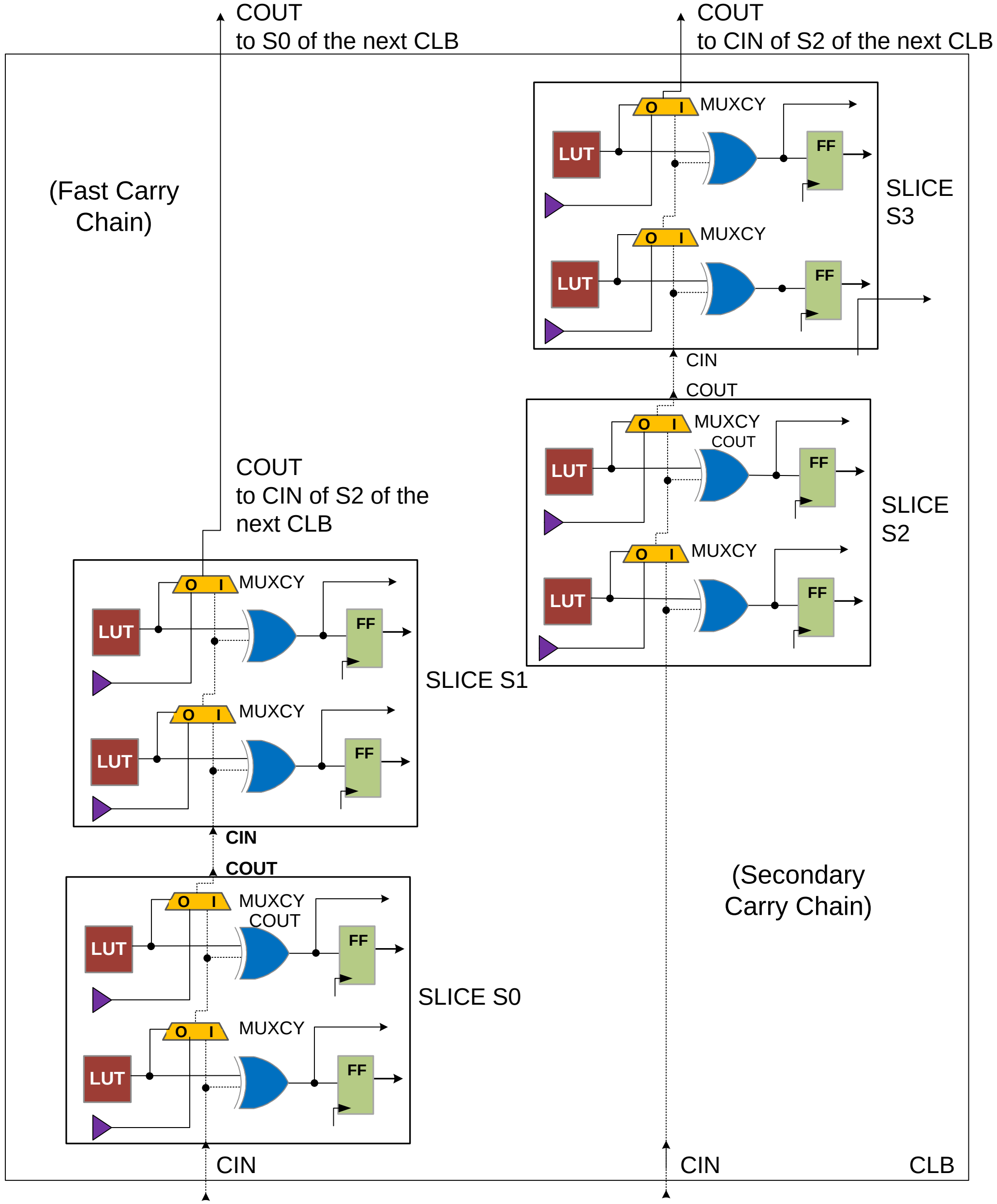
$$p_i = a_i + b_i$$

$$g_i = a_i b_i$$

Fast Carry Logic in Vertix™-II pro FPGA Slice

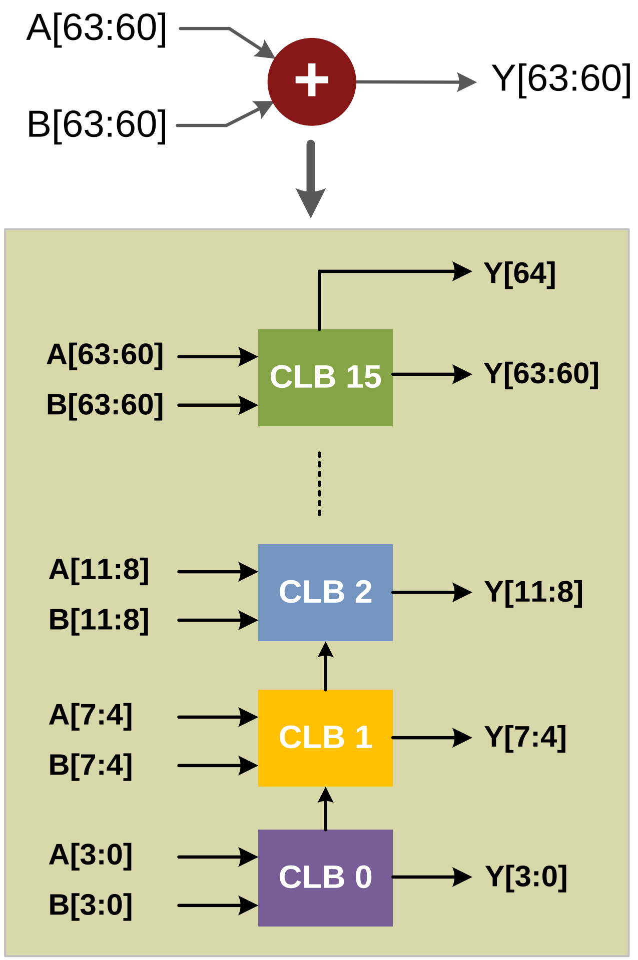


Fast Carry Logic



Fast Carry Chain

1 CLB = 4 Slices = 2,4bit adders



CLBs must be in same column

Parallel multiplier architecture

Designing Customized Multipliers

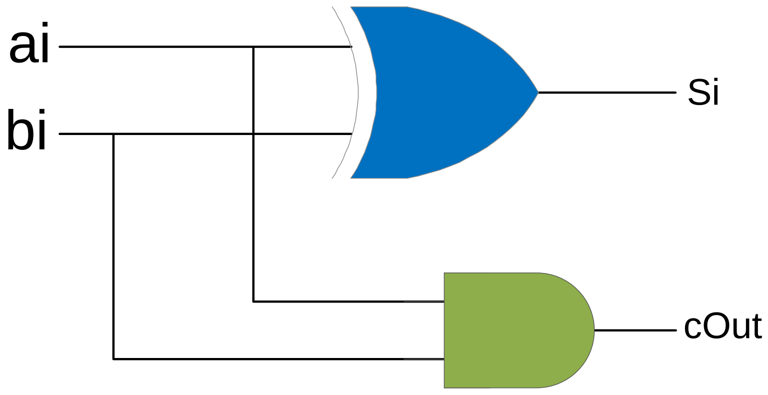
Adders

- Used in addition, subtraction, multiplication and division
- Speed of a signal processing or communication system ASIC depends heavily on these functional units

Half Adder using Data Flow modeling

```
module HALF_ADDER(ai, bi, si, cout);

    input  ai, bi;
    output  si, cout;


// data flow modeling

assign {cout,si} = ai + bi;

endmodule
```

Half Adder using Data Flow modeling



Half Adder

$$C_i = a_i b_i$$

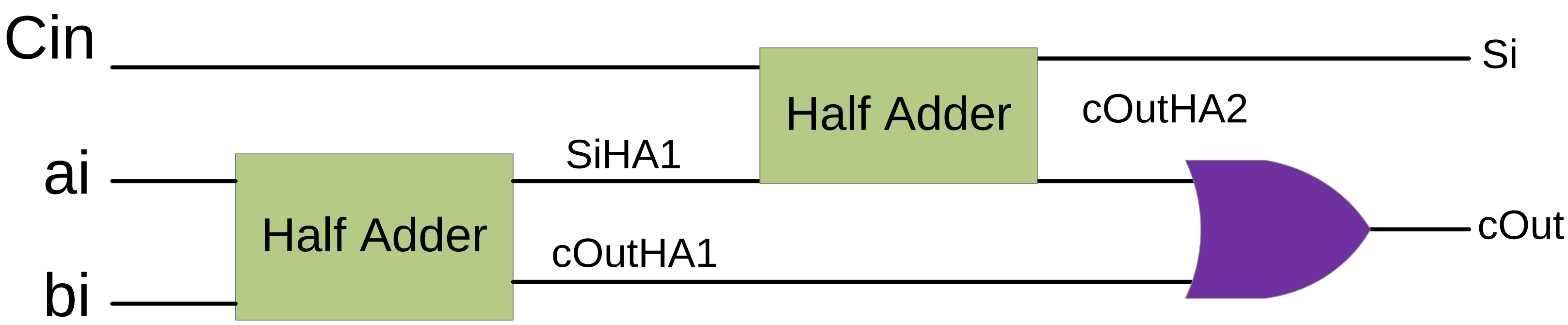
$$S_i = a_i \oplus b_i$$

Full Adder

Truth Table

x	y	z	C	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

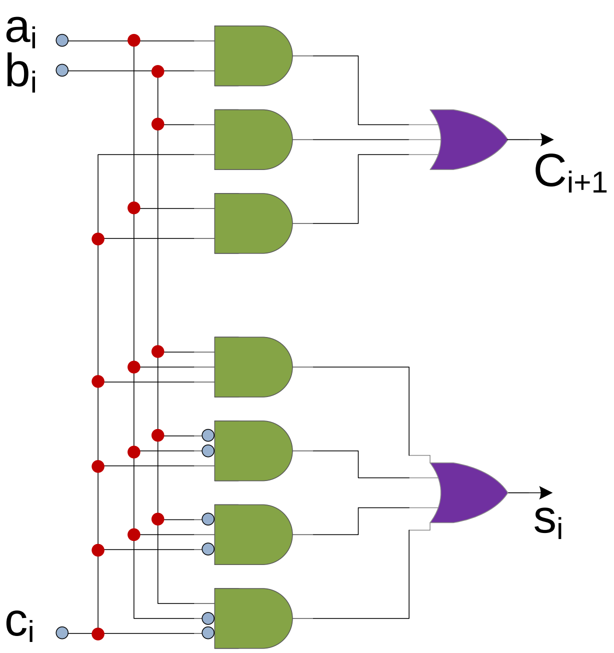
Full Adder



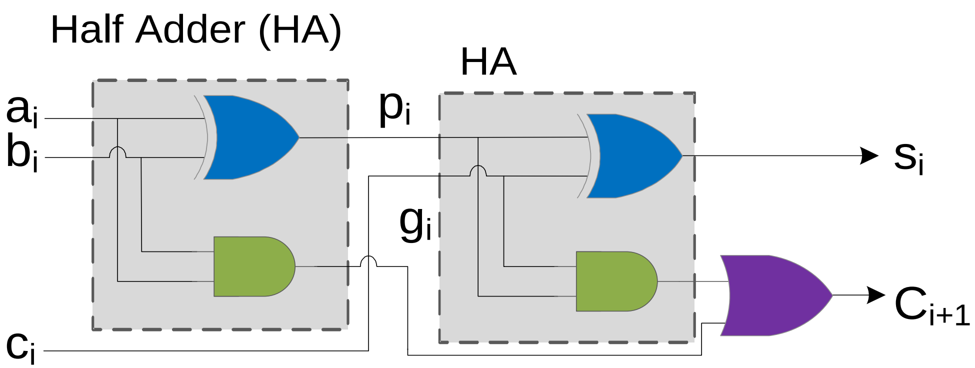
$$c_i = (a_i \oplus b_i)\, c_{i-1} + a_i b_i$$

$$s_i = a_i \oplus b_i \oplus c_i$$

Gate-level design options for a full adder

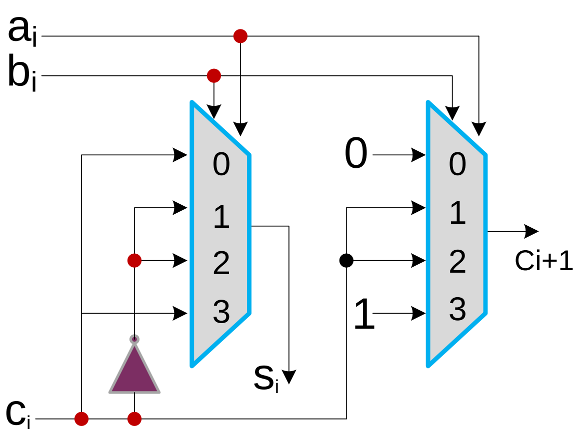


(a)

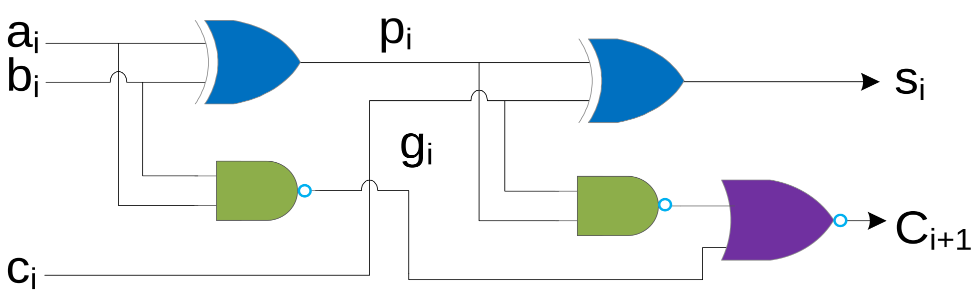


(b)

Contd ...



(c)



(d)

Full Adder: Implementation in Verilog

```
module FULL_ADDER(ai,bi,cin,si,cout);

    input  ai,bi;
    input  cin;
    output  si,cout;
    wire  SiHA1,CoutHA1,CoutHA2;

HALF_ADDER HA1(ai,bi,SiHA1,CoutHA1);  // instance HA1
HALF_ADDER HA2(SiHA1,cin,si,Cout);  //instance HA2
Or (cout,CoutHA1,CoutHA2);   // using or gate primitive

endmodule
```

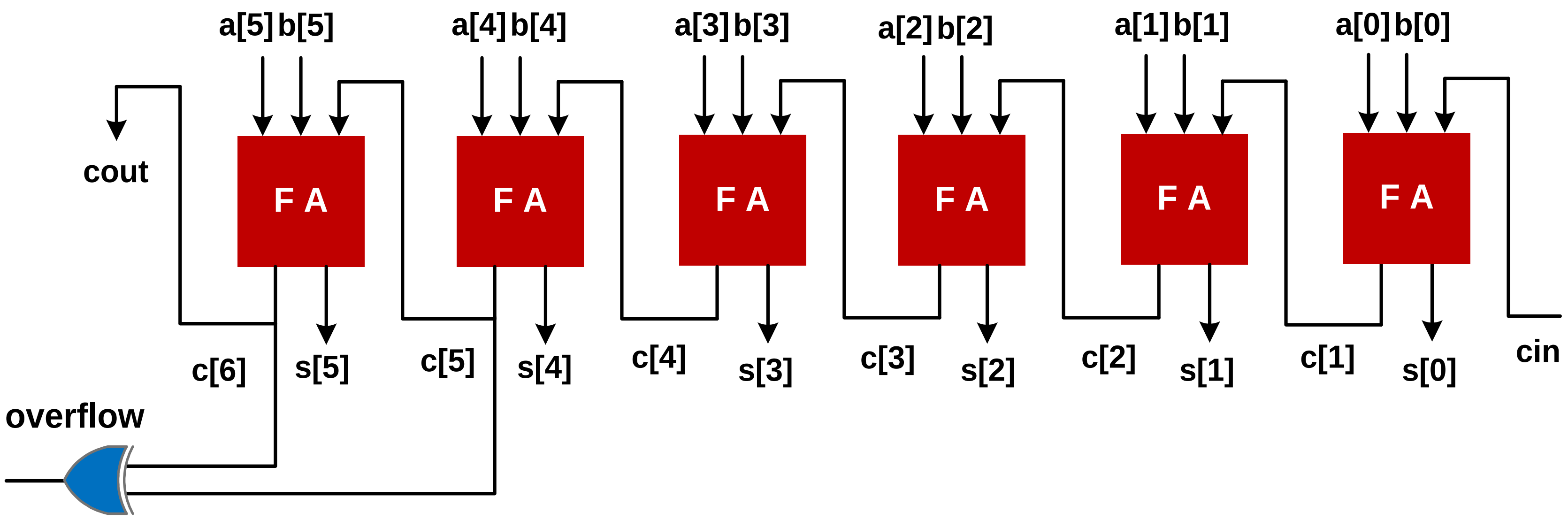
Full Adder Using Data Flow Modeling

```
module FULL_ADDER(ai,bi,cin,si,cout);

    input  ai,bi;
    input  cin;
    output  si,cout;


// through data flow level of abstraction

assign {cout,si} = ai + bi + cin;

endmodule
```

Full Adder Using Data Flow Modeling

```
module FULL_ADDER(ai,bi,cin,si,cout);

    input  ai,bi;
    input  cin;
    output  si,cout;


// through data flow level of abstraction

assign {cout,si} = ai + bi + cin;

endmodule
```

Ripple Carry Adder



```
module ripple_carry_adder #(parameter W=16)
        (input            clk,
         input      [W-1:0] a, b,
         input            cin,
         output reg [W-1:0] s_r,
         output reg         cout_r);


        wire [W-1:0] s;
        wire         cout;
        reg  [W-1:0] a_r, b_r;
        reg          cin_r;

        assign {cout,s} = a_r + b_r + cin_r;

        always@(posedge clk)
        begin
                a_r<=a;
                b_r<=b;
                cin_r<=cin;
                s_r<=s;
                cout_r<= cout;
        end
endmodule
```

RCA: Dataflow modeling

Six bit ripple carry adder through data flow modeling

```
// SIX BIT FULL ADDER ;
module fulladder_6bit(s,cout,a,b,cin);

output cout;
output [5:0] s;
input  [5:0] a,b;
input  cin;
reg [5:0] s,c;
reg cout;
always@(a or b or cin)
begin
      {c[0],s[0]}= a[0] + b[0] + cin;
     for(i=1; i<6; i=i+1)
              {c[i],s[i]}= a[i] + b[i] + c[i-1]; // through data flow modeling.
     cout = c[5];
 end
endmodule
```

Important Observation

- Do we have to wait for the carry to show up to begin doing useful work?
  - We do have to know the carry to get the right answer.
  - But, it can only take on two values

Non-uniform Group 12-Bit Carry Select Adder

- Three partitions of 3-bits, 4-bits, 5-bits are made
- The cout of the first block is ready earlier making it faster in functionality than the uniform group 12- bit carry select adder
- So non-uniform group carry select adder is faster than the uniform group carry select adder

Carry Generate and Propagate Logic

$$g_i = a_i b_i$$

$$p_i = a_i \oplus b_i$$

$$c_{i+1} = g_i + p_i c_i$$

$$s_i = c_i \oplus p_i$$

Group Carry and Group Propagate

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1$$

$$= g_1 + p_1 (g_0 + p_0 c_0)$$

$$= g_1 + p_1 g_0 + p_0 p_1 c_0$$

$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

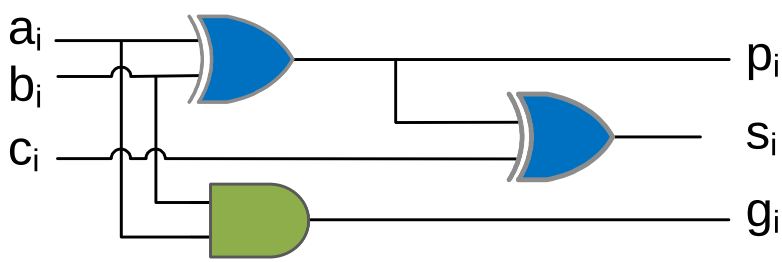
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

let $G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

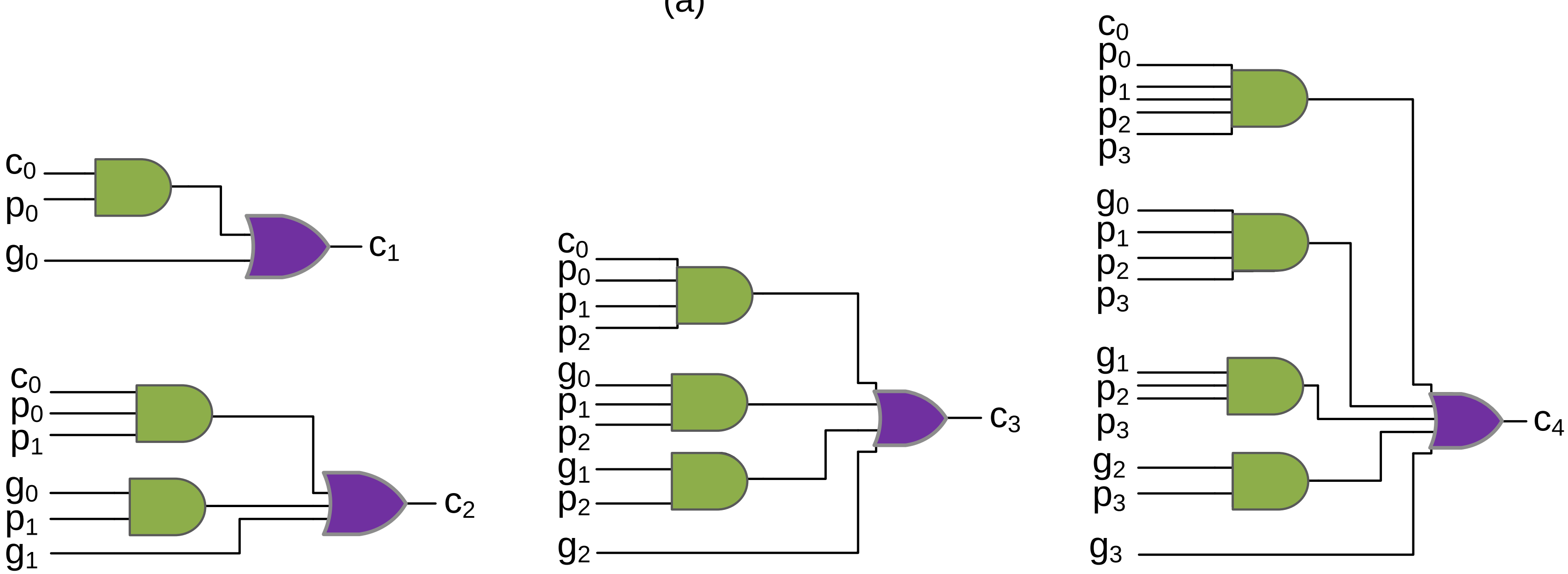
and $P_0 = p_3 p_2 p_1 p_0$

we can write $c_4 = G_0 + P_0 c_0$

CLA logic for computing carries in two-Gate delay time

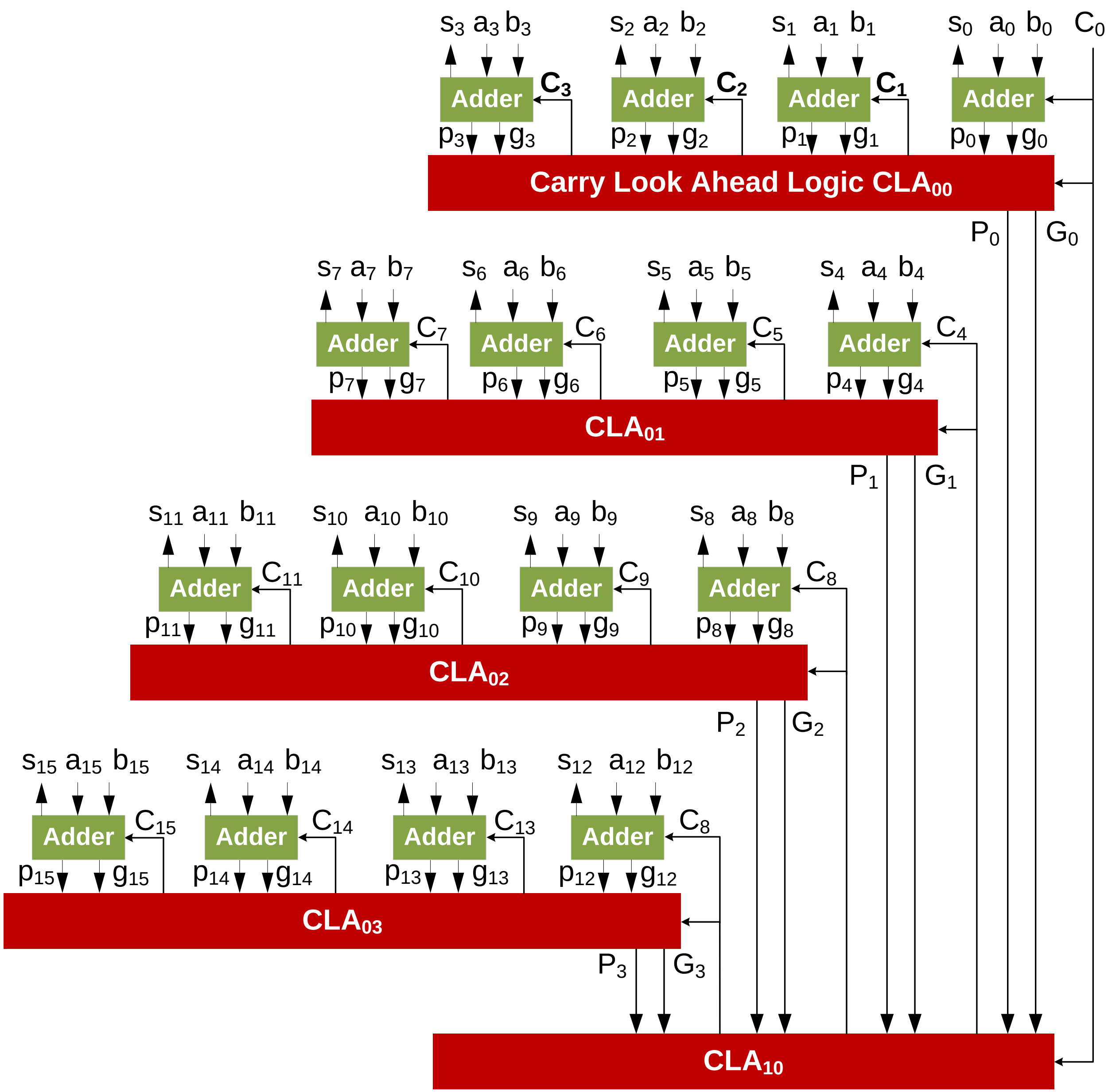


(a)

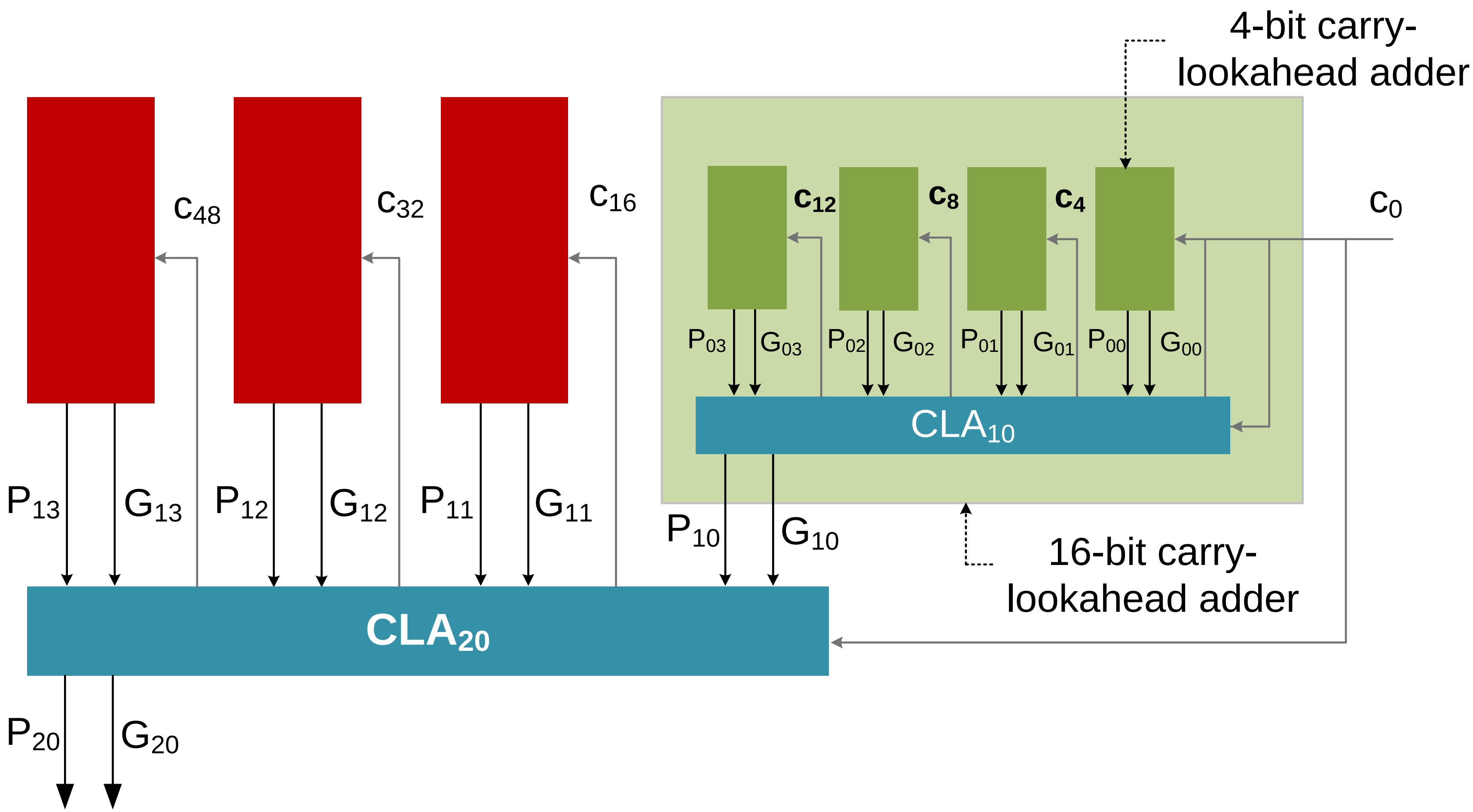


(b)

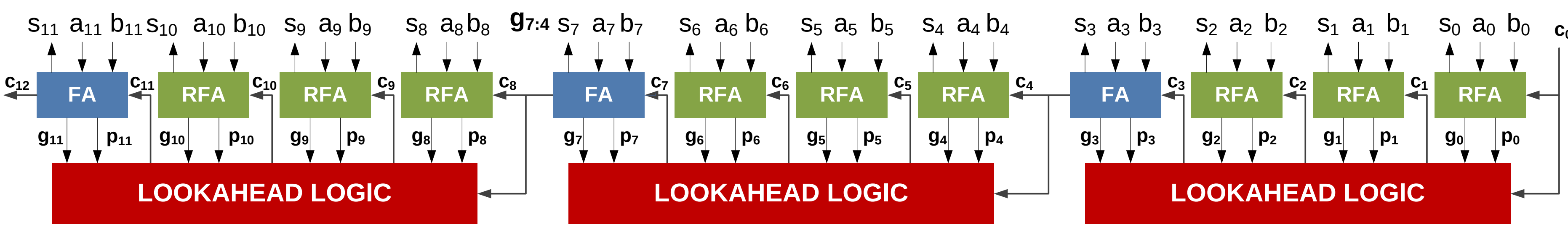
A 16-bit carry look-ahead adder using two levels of CLA logic



A 64-bit carry look-ahead adder using three levels of CLA logic



A 12-bit Hybrid Ripple Carry and Carry Look-ahead Adder



Binary Carry Look-ahead Adder (BCLA)

$g_{i=}a_i b_i$

$p_{i=}a_i \oplus b_i$

and

$(G_i, P_i) = (g_i, p_i) \bullet (g_{i-1}, p_{i-1}) \bullet ...(g_i, p_i) \bullet (g_0, p_0)$

$Eq1$

And the problem can be recursively solved as

$(G_0, P_0) = (g_0, p_0)$

for $i = 1$ to N-1

$(G_i, P_i) = (g_i, p_i) \bullet (G_{i-1}, P_{i-1})$

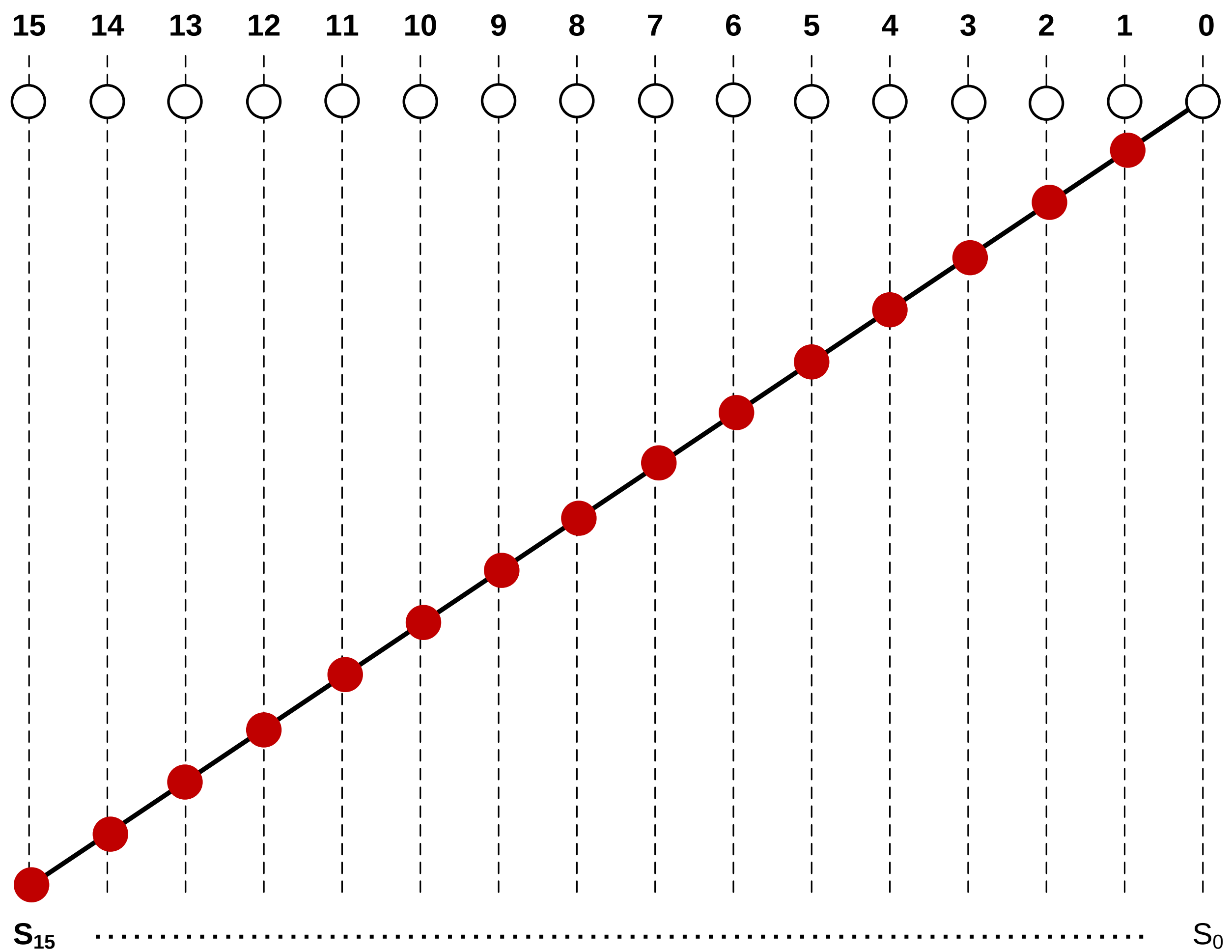
$c_i = G_i + P_i c_0$

end

Where the dot operator $\bullet$ is given as :

$(G_i, P_i) = (g_i, p_i) \bullet (G_{i-1}, P_{i-1}) = (G_{i-1} + p_i G_{i-1}, p_i P_{i-1})$

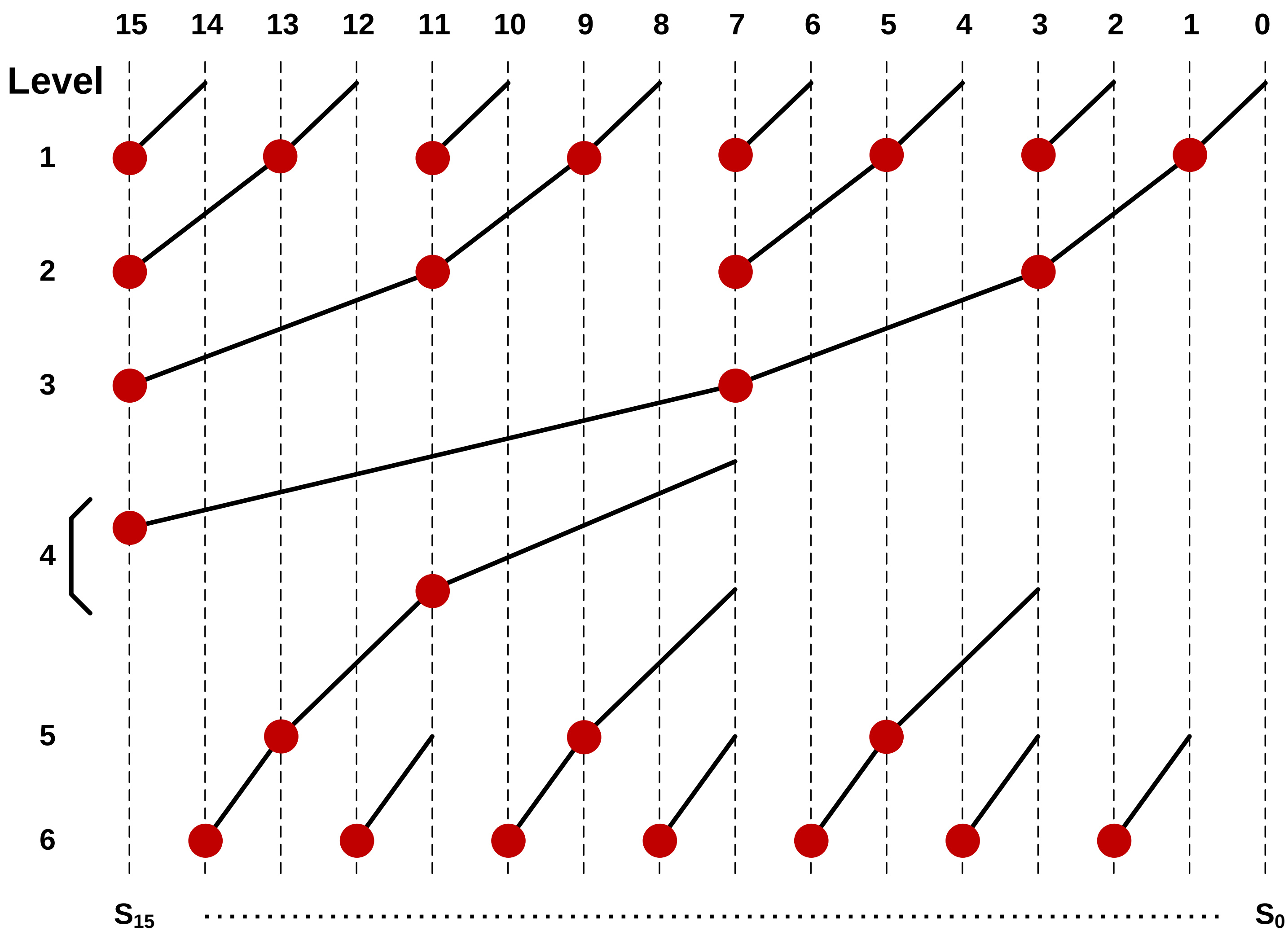
Binary carry look-ahead adder Serial Implementation



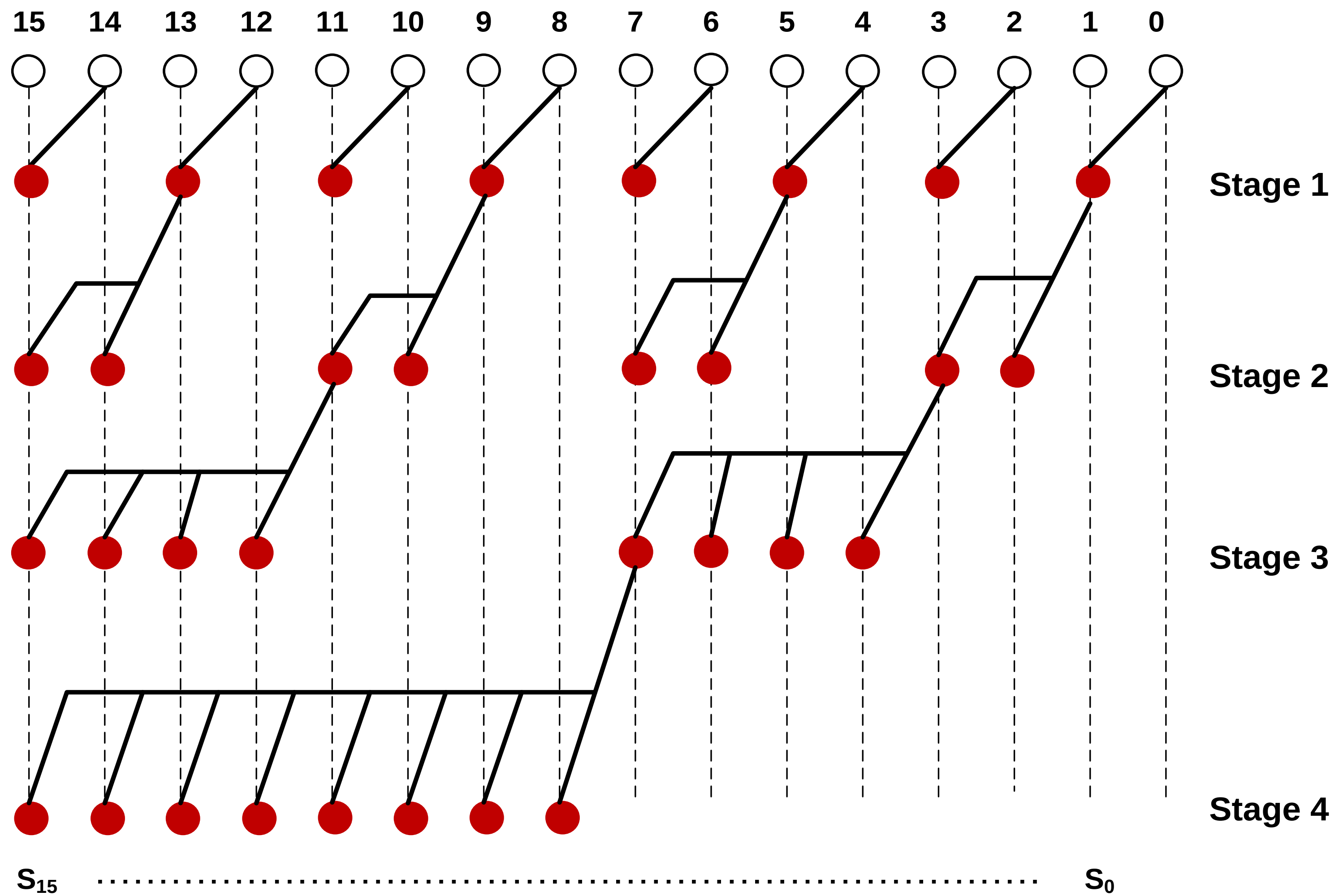
```
module BinaryCarryLookaheadAdder
# (parameter N = 16)
(input [N-1:0] a,b,
 input c_in,
 output reg [N-1:0] sum,
 output reg c_out);
 reg [N-1:0] p, g, P, G;
reg [N:0] c;
integer i;
always@(*)
```

```
         begin
        for (i=0;i<N;i=i+1)
        begin
       //generate all ps and gs
      p[i]= a[i] ^ b[i];
      g[i]= a[i] & b[i];
      end
  End
```

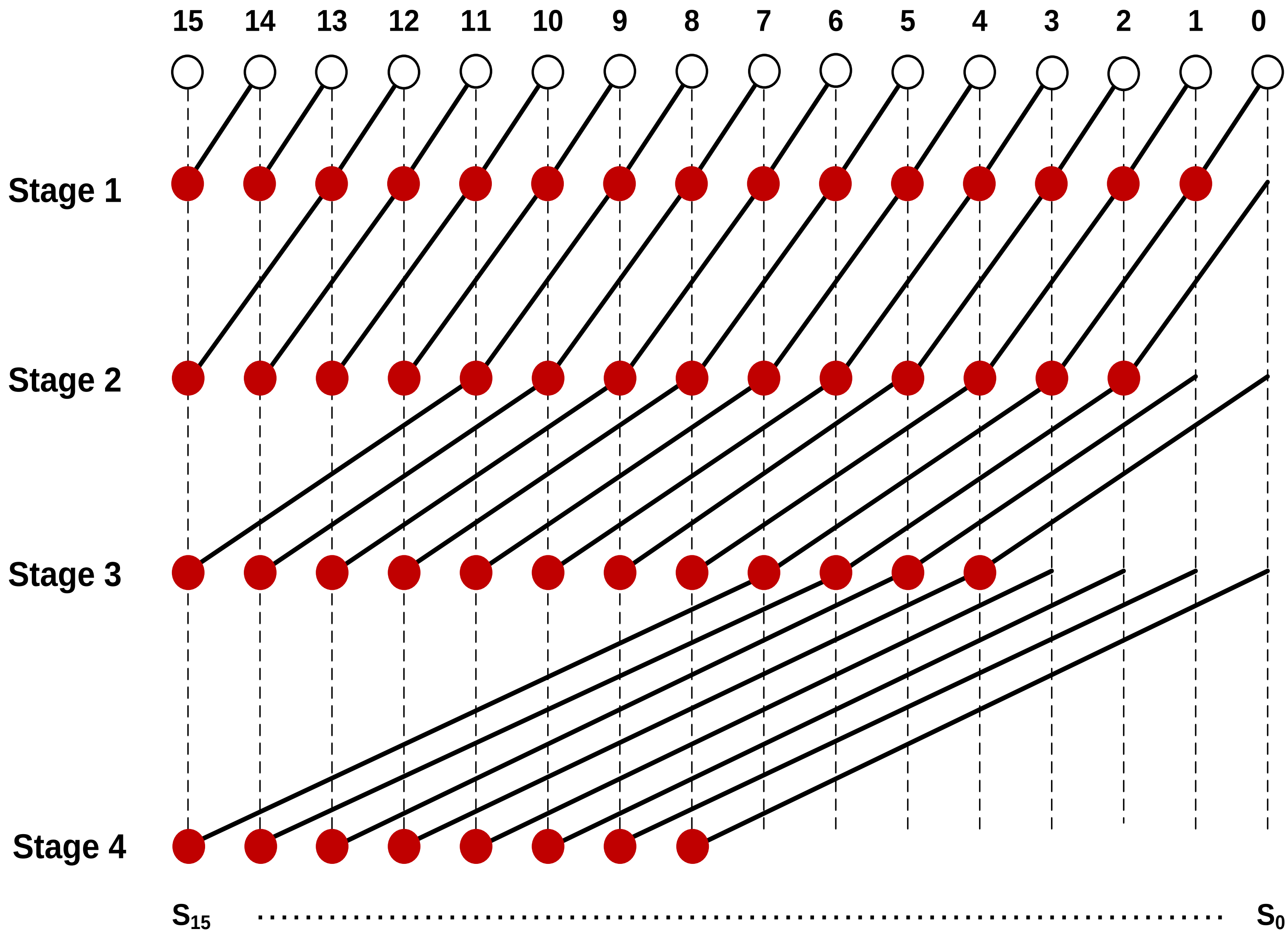
Brent–Kung adder



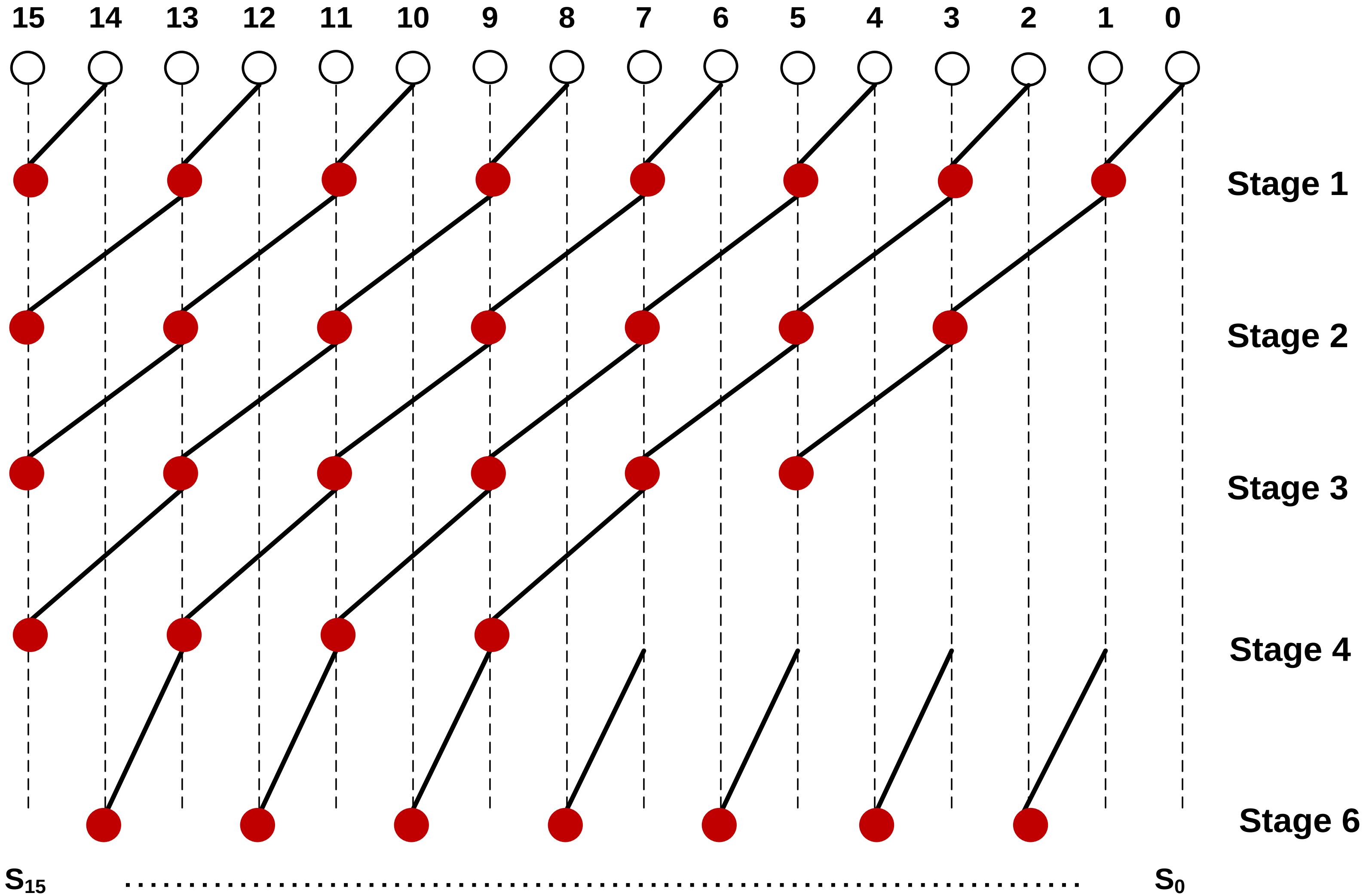
Ladner–Fischer parallel prefix adder



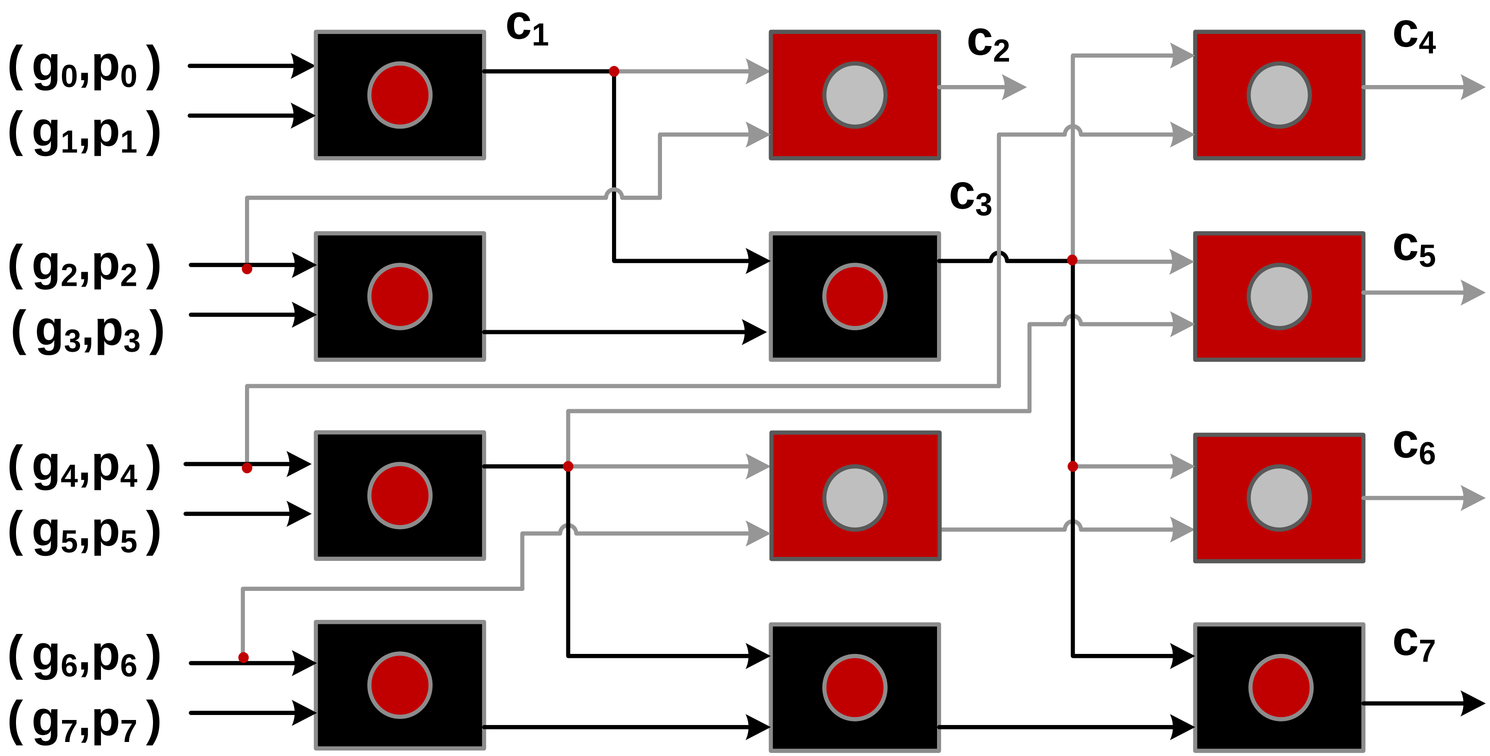
Kogge–Stone parallel prefix adder



Han–Carlson parallel prefix adder



Regular layout of an 8-bit Brent-Kung Adder



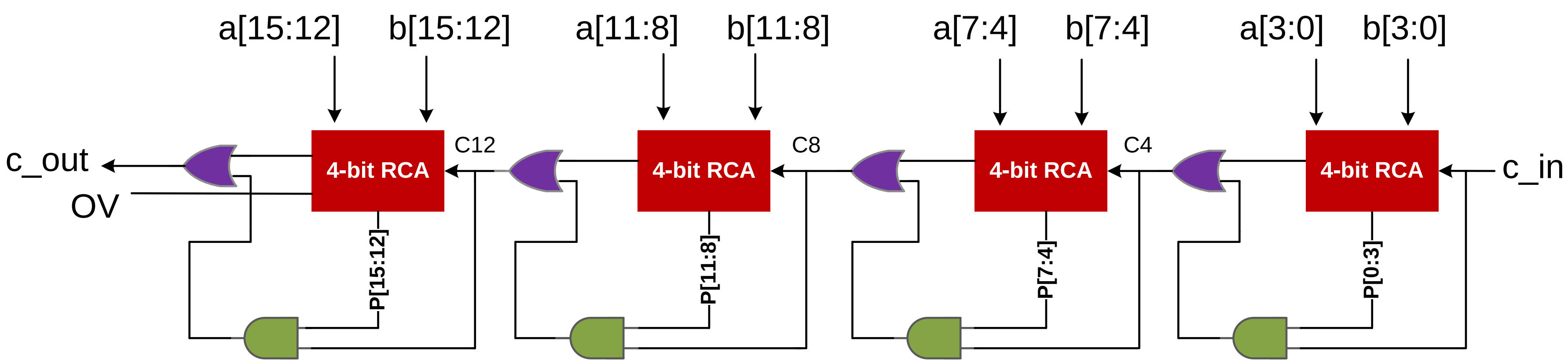
Carry Skip Adder

- If any group generates a carry, it passes it to the next group
- In case the group does not generate its own carry then it simply bypasses the carry from the previous block to its next block

$$P_i = p_i p_i + 1 p_i + 2 \ldots p_i + k - 1$$

$$p_i = a_i \oplus b_i$$

A 16-bit equal-group carry skip adder



Conditional Sum Adder

- The process that led to the two-level carry select adder can be continued . . .
- A logarithmic time *conditional-sum adder* results if we proceed to the extreme:
  - single bit adders at the top
- A conditional-sum adder is actually a (log2 *k*)-level carry-select adder
- Implemented in multiple levels
- Built using Conditional Cells (CC) and MUX(s)

Principle

- The conditional cell generates a pair of sum and carry bits i at each bit position (**si0, ci0,si1,ci1)**
- One pair assumes carry_in of one (**si1, ci1)** and the other assumes a carry_in of zero (**si0, ci0)**
- The correct sums and carries are then selected using a tree of multiplexers
- All level one bits are paired up
- The sum and carry of the next bit position, brought down to level 2 are selected by the least significant carry
- This continues until all the sums and carries are resolved in the last level

Example

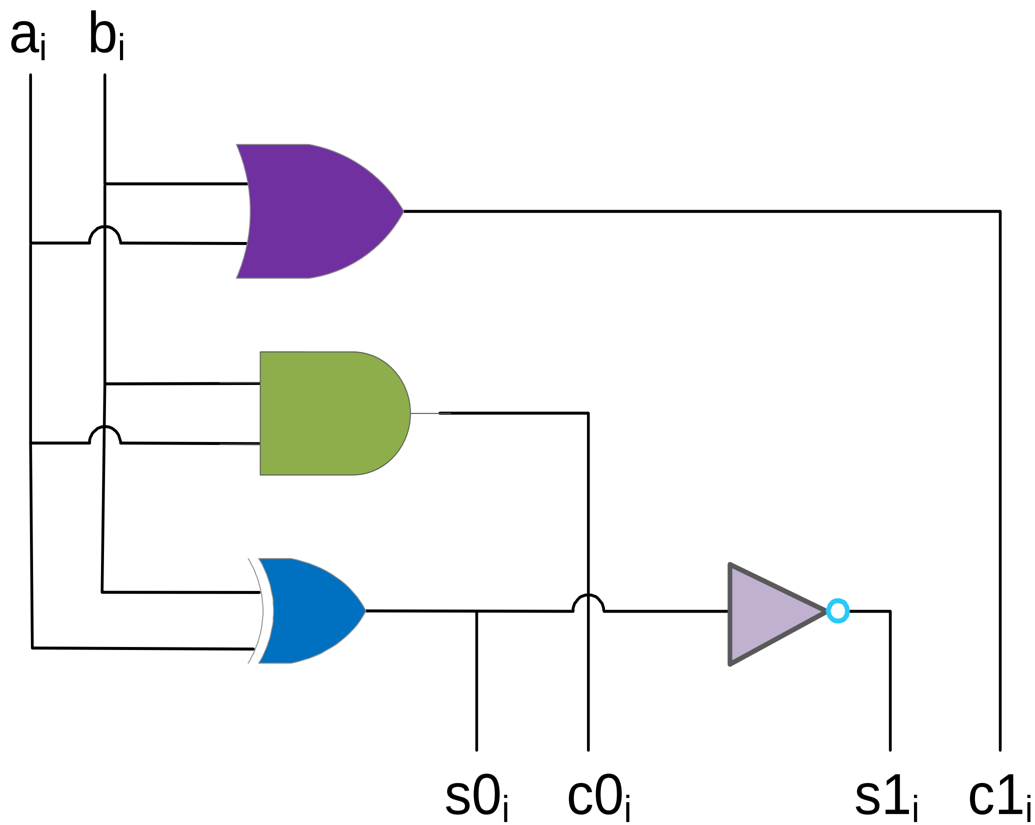
$$s0_i = a_i \oplus b_i$$

$$s1_i = a_i \sim \oplus b_i$$

$$c0_i = a_i b_i$$

$$c1_i = a_i + b_i$$

Conditional Cell (CC)



Addition of three bit numbers using a conditional sum adder

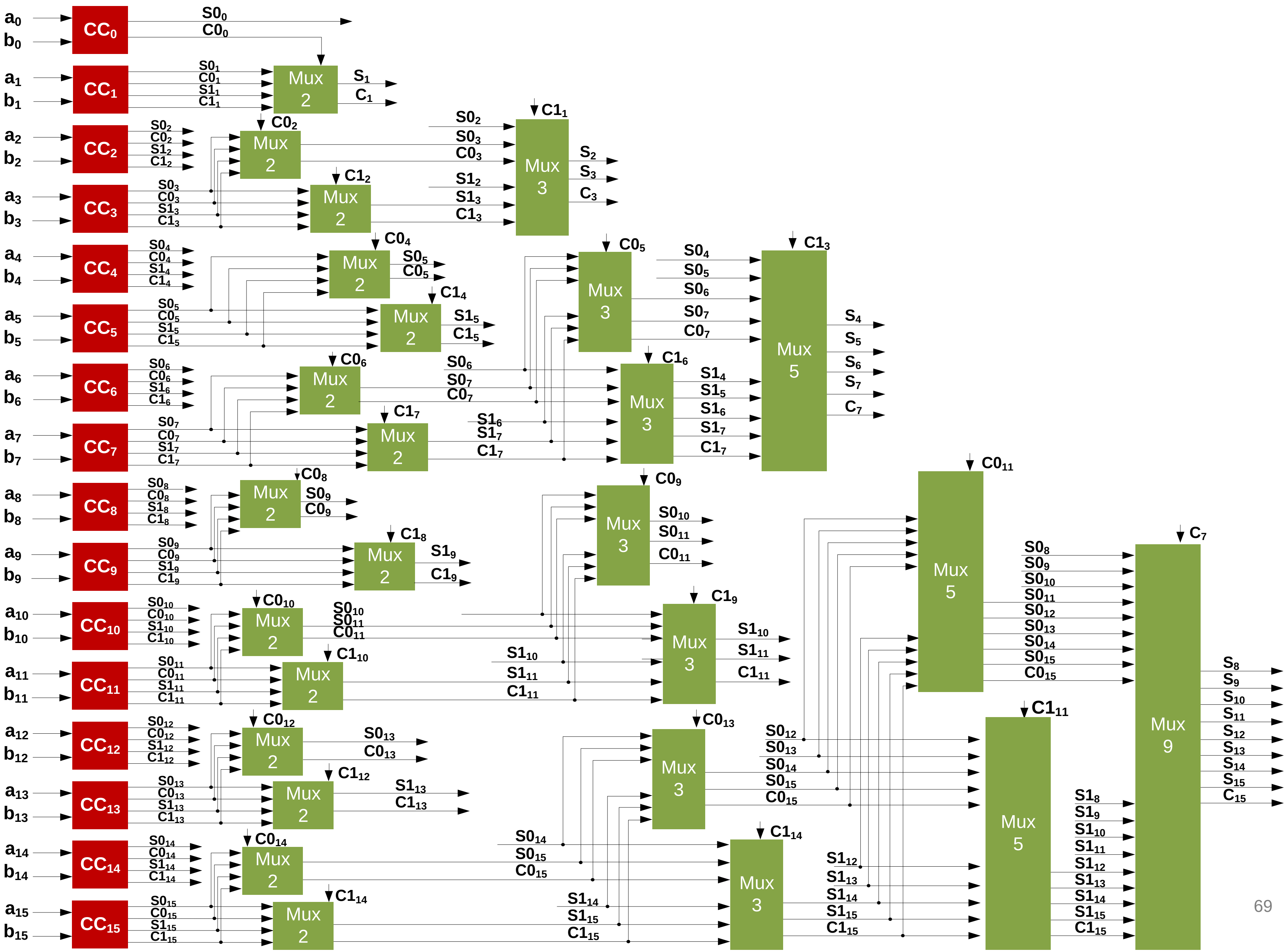
(Here we are assuming actual c_{in}=0)

1	1	1		
1	0	1		
0	1	0	$s0_i$	$C_{in}=0$
1	0	1	$c0_i$	
1	0	1	$s1_i$	$C_{in}=1$
1	1	1	$c1_i$	
0	1	0		
1		1		
1	0			
1				
1	0	0		
1				

Example: Conditional Sum Adder

Group width	Group carry-in	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	a_i	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1	a_i
	b_i	0	0	1	1	0	1	1	0	1	1	0	1	0	1	1	0	b_i
Group width	Group carry-in	Group sum and block carry out																
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	i
1	0	1	0	1	0	1	1	1	1	0	0	0	1	1	0	1	1	$S0_i$
		0	0	0	1	0	0	0	0	1	1	0	0	0	1	0	0	$C0_i$
	1	0	1	0	1	0	0	0	0	1	1	1	0	0	1	0		$S1_i$
		1	0	1	1	1	1	1	1	1	1	0	1	1	1	1		$C1_i$
2	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	1	1	
		0		1		0		0		1		0		1		0		
	1	1	1	0	1	0	0	0	0	1	1	1	0	0	1			
		0		1		1		1		1		0		1				
4	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	1	1	
		0				0				1				1				
	1	1	1	0	1	0	0	0	0	1	0	1	0					
		0				1				1								
8	0	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	
		0								1								
	1	1	1	0	1	0	0	0	0									
		0																
16	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	1	1	
		0																
	1																	

A 16-bit Conditional Sum Adder



Hybrid Adder Designed

- Hybrids are obtained by combining elements of:
  - Ripple-carry adders
  - Carry-lookahead (generate-propagate) adders
  - Carry-skip adders
  - Carry-select adders
  - Conditional-sum adders

- You can obtain adders with
  - higher performance
  - greater cost-effectiveness
  - lower power consumption

Example

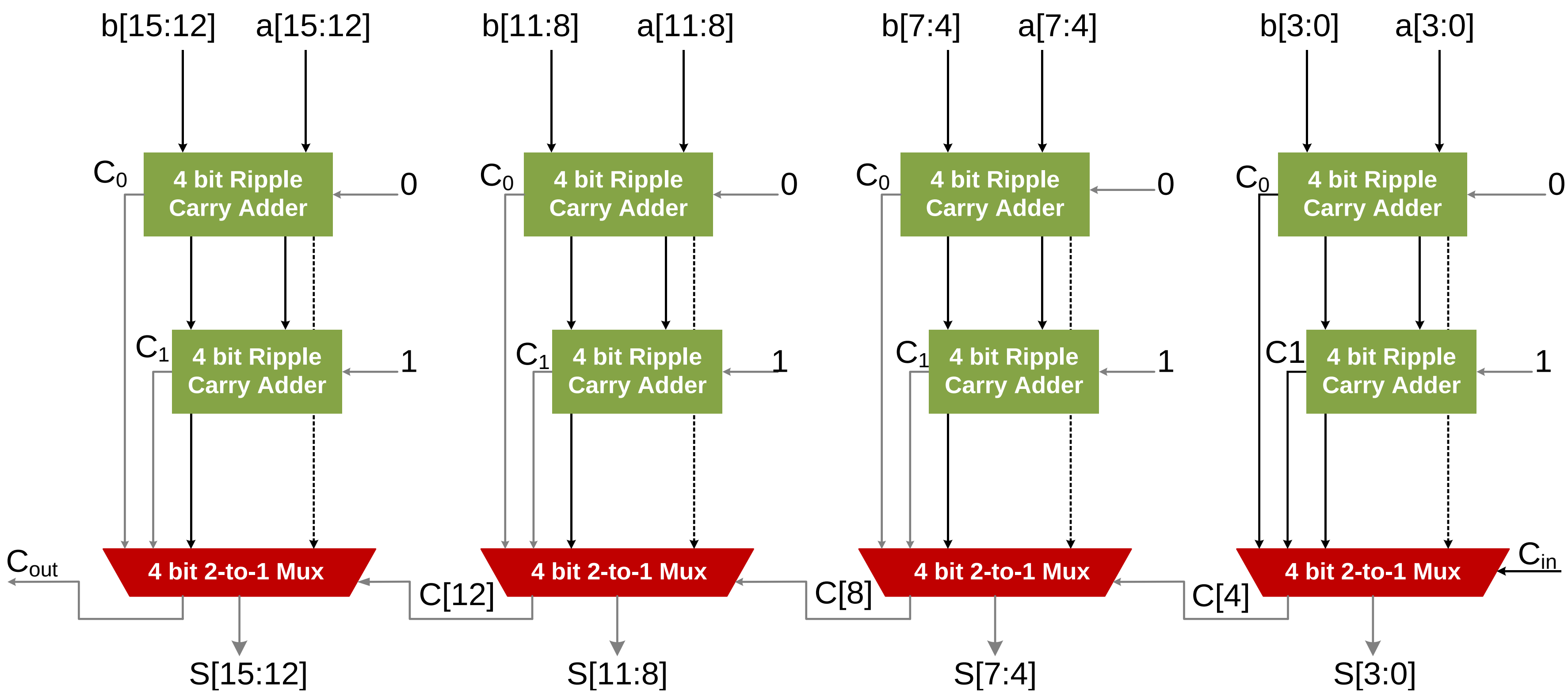
$$s0_i = a_i \oplus b_i$$

$$s1_i = a_i \sim \oplus b_i$$

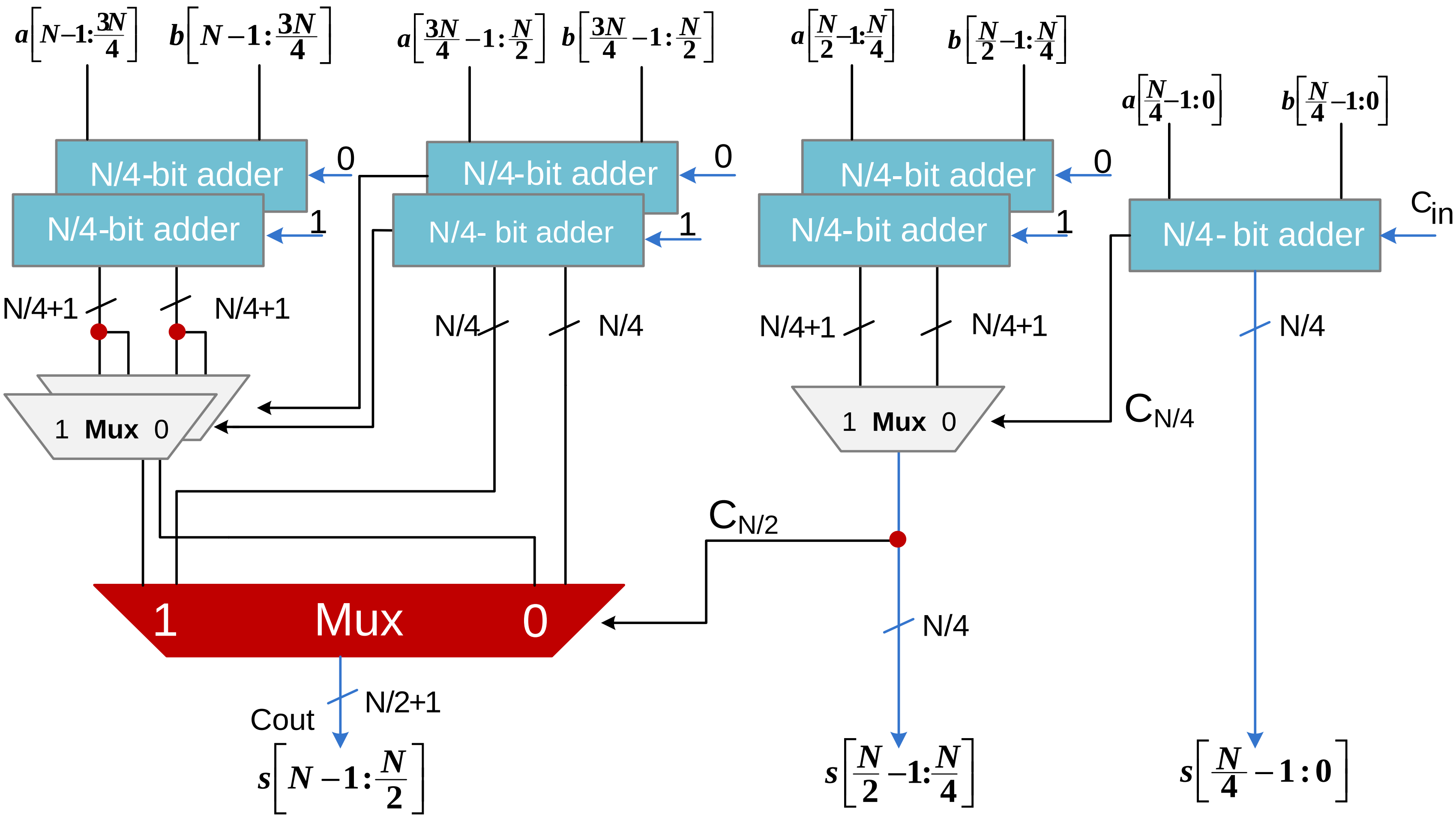
$$c0_i = a_i b_i$$

$$c1_i = a_i + b_i$$

A 16 –bit uniform-groups carry select adder

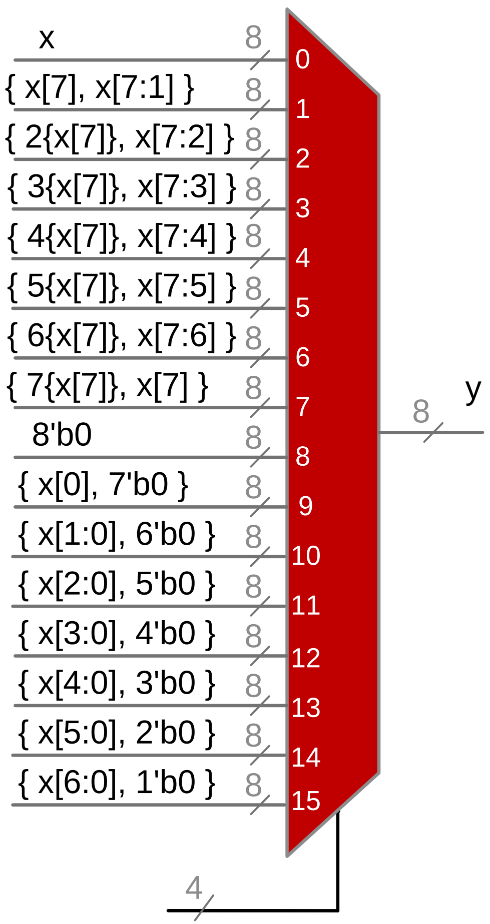


Hierarchical CSA

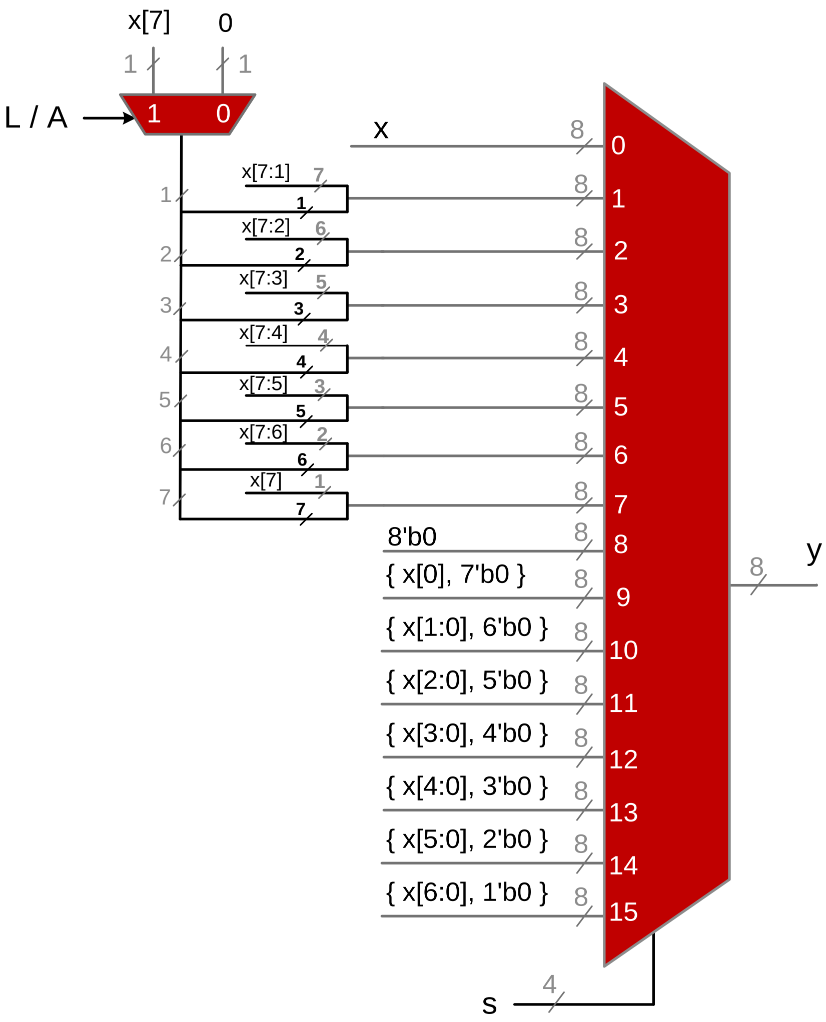


Barrel Shifter

(a) Design of a logic shifter for an 8-bit Operand (b) Design of logic and arithmetic shifter for an 8-bit signed operand



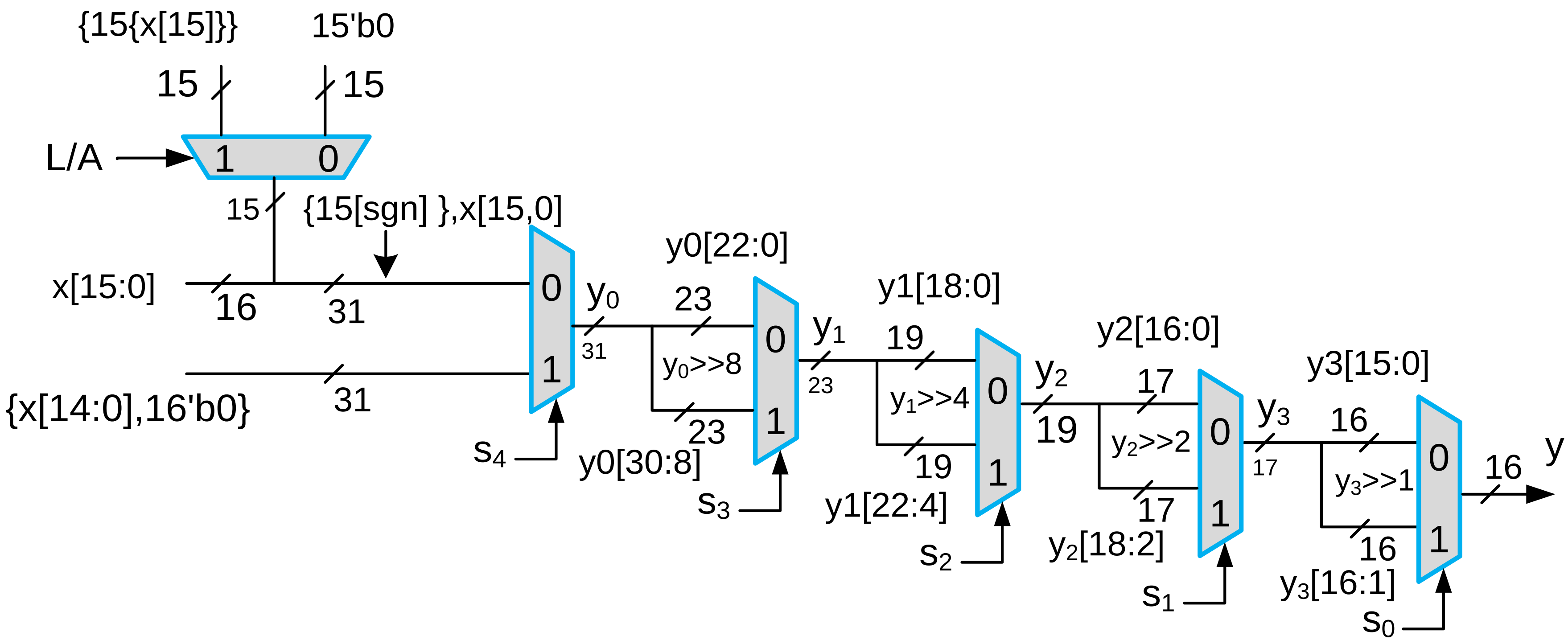
(a)



(b)

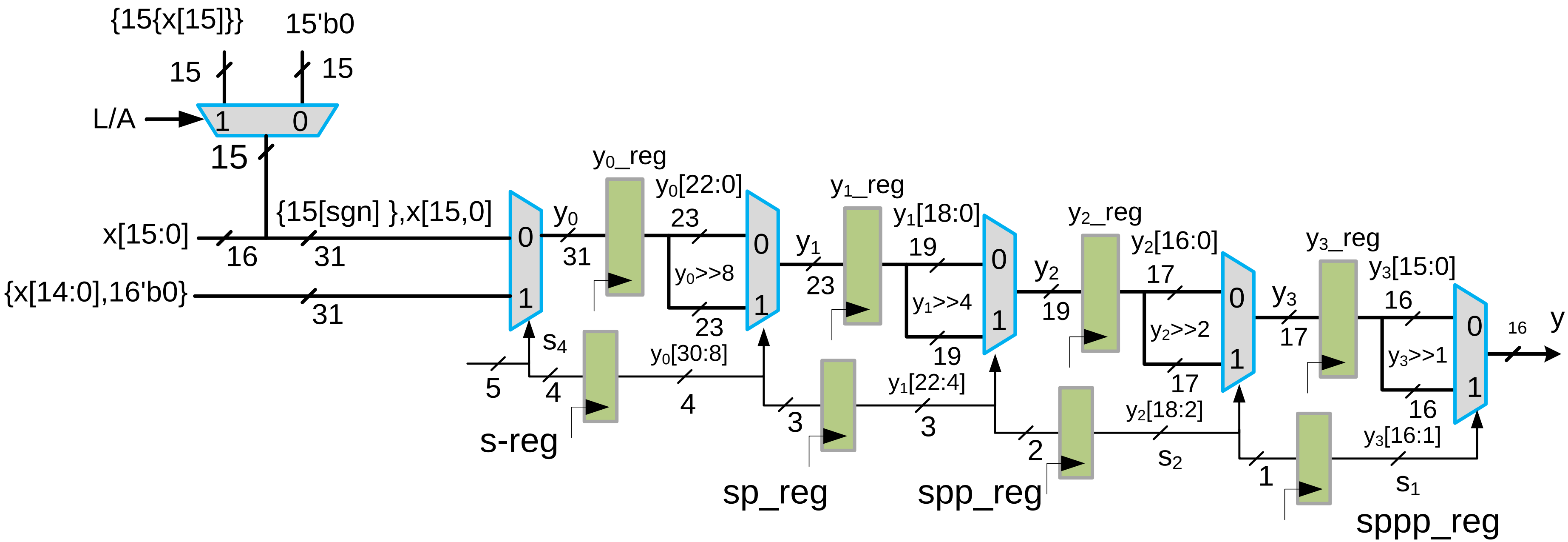
Design of a Barrel Shifter performing shifts in multiple stages (a)

Single cycle design



(a)

Design of a Barrel Shifter performing shifts in multiple (b) Pipelined design



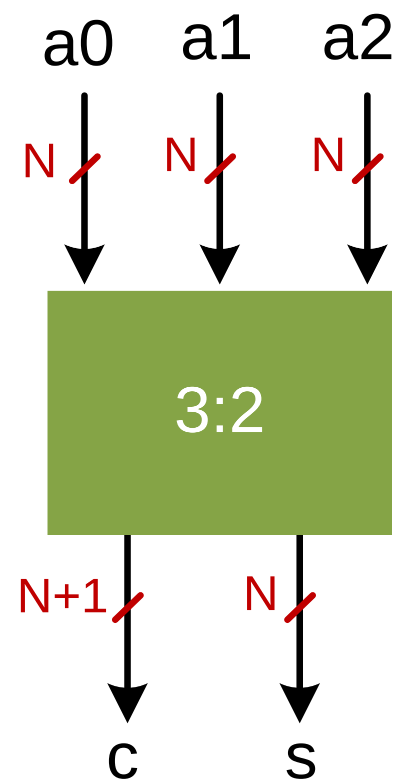
(b)

Carry Save Adders and Compressors

Carry Save Addition saves the carry at next bit location

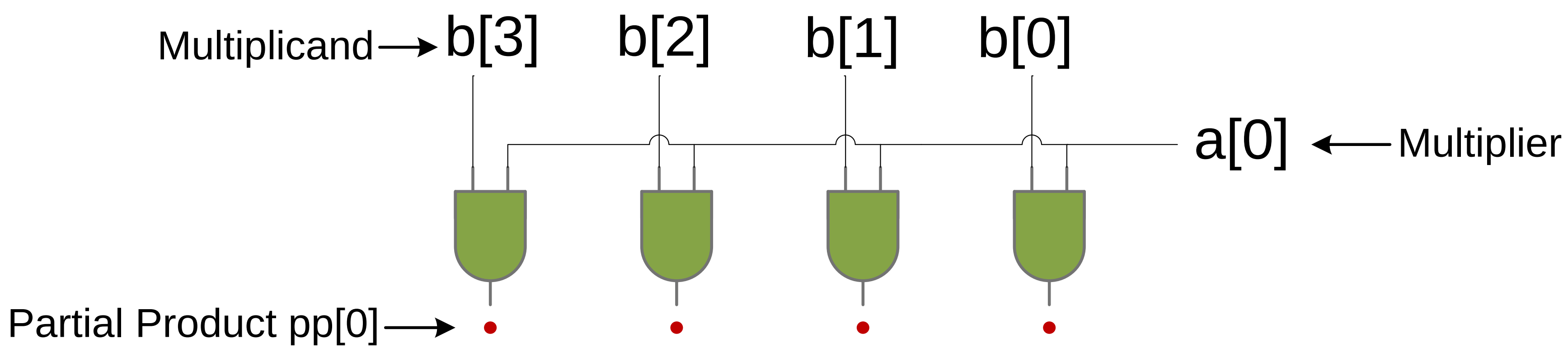
- The CSA does not ripple any carry
- It has a delay of one FA
- The concept of CSA is effective in designing partial products compression/ reduction logic

a0	=		0	0	1	0	1	1
a1	=		0	1	0	1	0	1
a2	=		1	1	1	1	0	1
s	=		1	0	0	0	1	1
c	=	0	1	1	1	0	1	



Dots are used to represent each bit of the partial product

- Dot notation facilitates description of different reduction schemes
- Dots are used to represent each bit of the partial product



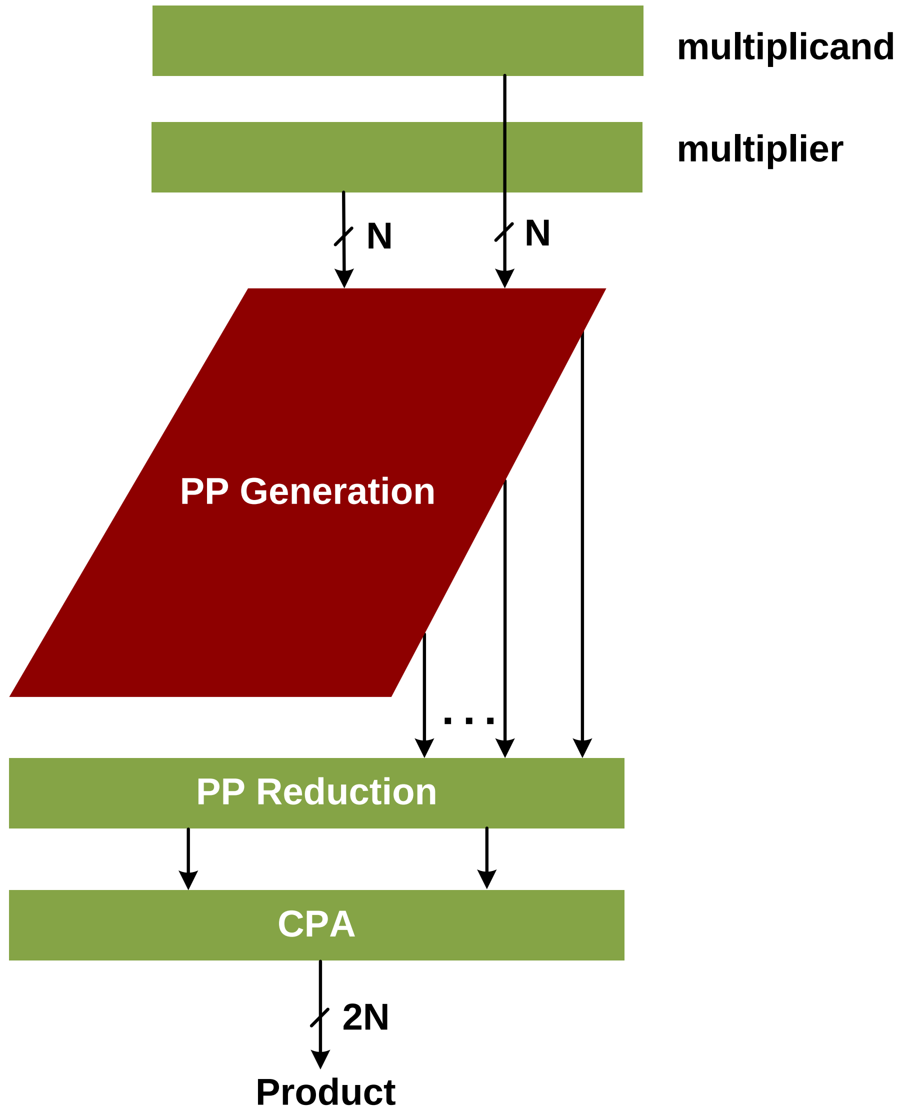
Parallel multiplier architecture

Designing Customized Multipliers

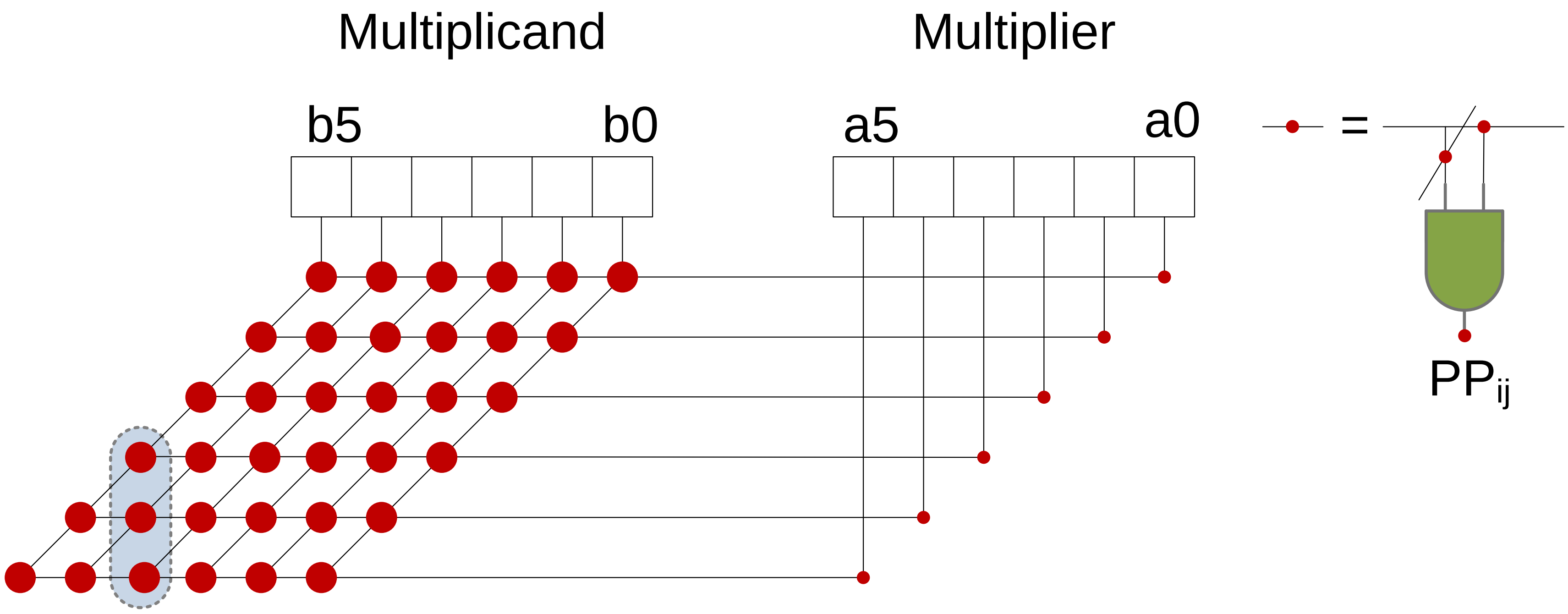
Three components of a multiplier

- N-bit inputs operands
- Partial Product Array Generation = N shifted binary numbers
- Partial Product Array Reduction= reduction to 2 binary numbers
- Final addition = 2N-bit final product

Three components of a multiplier



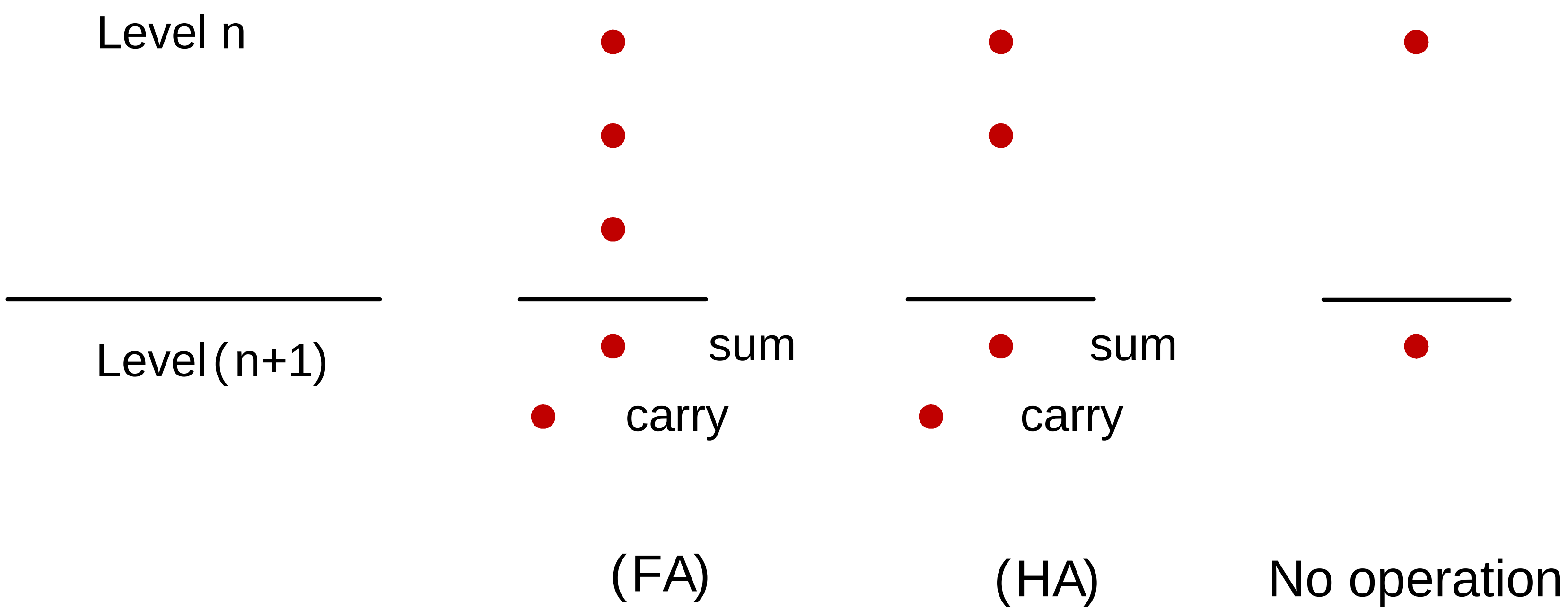
Partial Product Generation for a 6x6 Multiplier



Partial Product Generation Verilog Code

```
module multiplier (
input [5:0] a,b,
output [11:0] prod);

integer i;

reg [5:0] pp [0:5]; // 6 partial products

always@*
begin
        for(i=0; i<6; i=i+1)
        begin
                pp[i] = b & {6{a[i]}};
        end
end

assign prod = pp[0]+{pp[1],1'b0}+{pp[2],2'b0}

        +{pp[3],3'b0}+{pp[4],4'b0}+{pp[5],5'b0};

endmodule
```

Reducing number of dots in a column



- Three dots are shown
- Each symbolizes a partial product
- Using FA reduces these to two bits
- One has the weight of 2^0(sum)
- The other has the weight of 2^1(carry)
- This type of reduction is known as 3 to 2 reduction or carry saves reduction
- The two dots are reduced to 2 using a HA

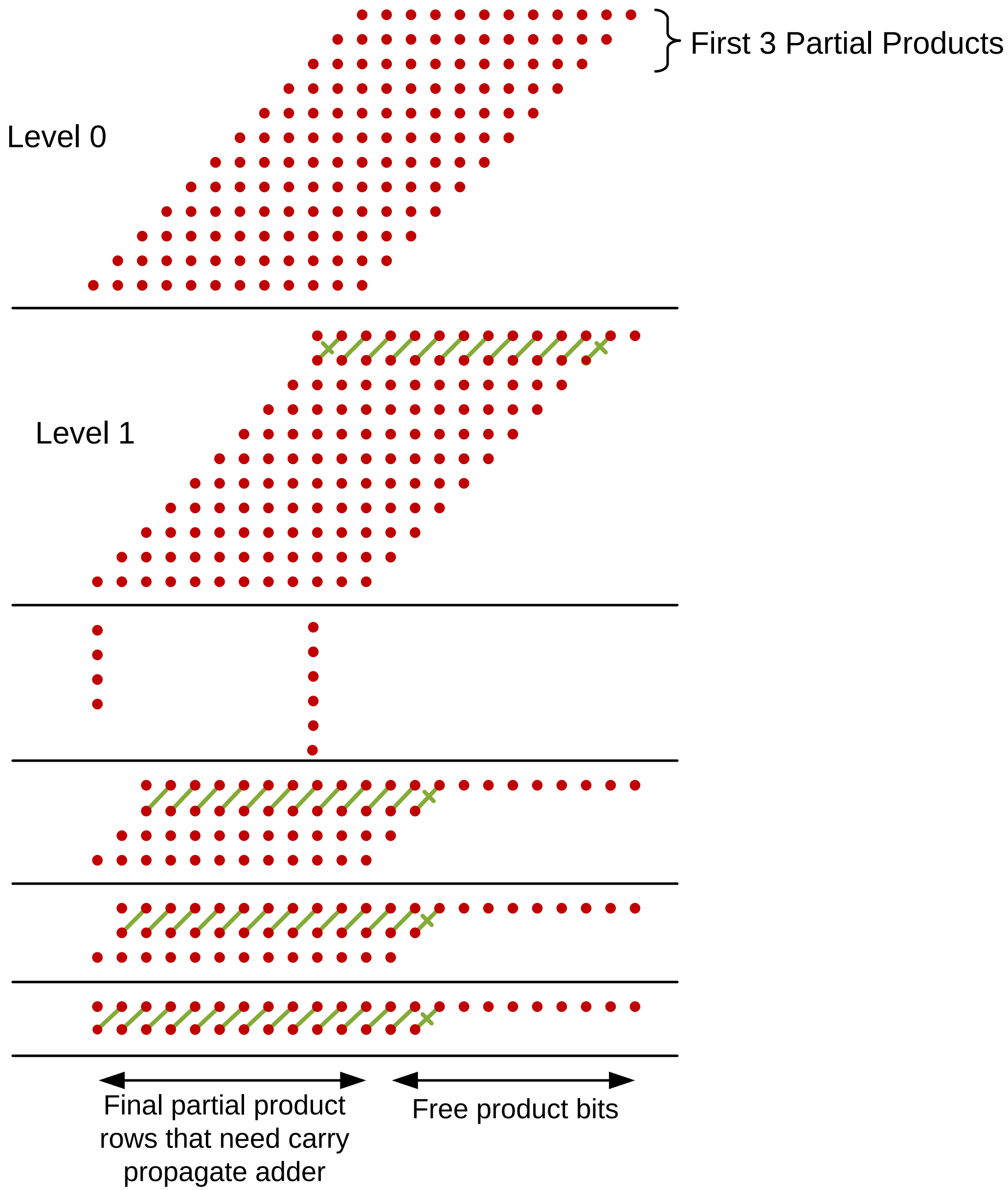
Partial Products Reduction Schemes

- Carry Save Reduction Scheme
- Dual Carry Save Reduction Scheme
- Wallace Tree Reduction Scheme
- Dadda Tree Reduction Scheme

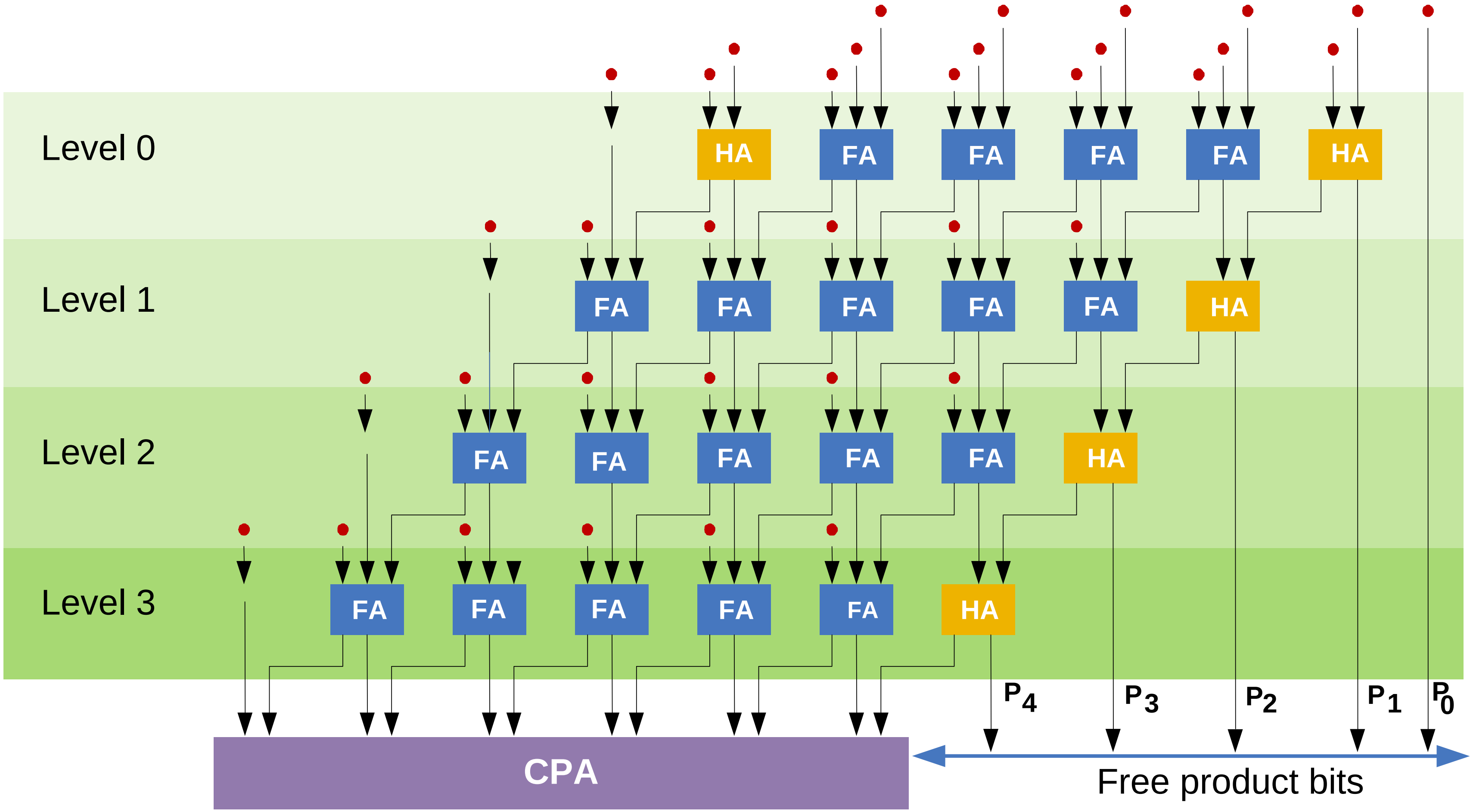
12x12 Carry Save Reduction Scheme

- Considers three rows at a time
- Take first three rows use CSA to reduce them to two
- Iteratively take two layers from previous reduction and a new from PP layer and reduce them to two using a CSA
- Finally produces two layers
- Also produces free product bits
- The two layers are added using any CPA

PP reduction for a 12x12 Multiplier using Carry Save Reduction Scheme



Carry Save Reduction Scheme Layout for a 6x6 Multiplier



Dual Carry Save Reduction

- The partial products are divided into 2 equal size groups
- The carry save reduction scheme is applied on both the groups simultaneously
- This results into two partial product layers in each group
- The four layers are then reduced using Carry Save Reduction
- The last two layers are added using any CPA

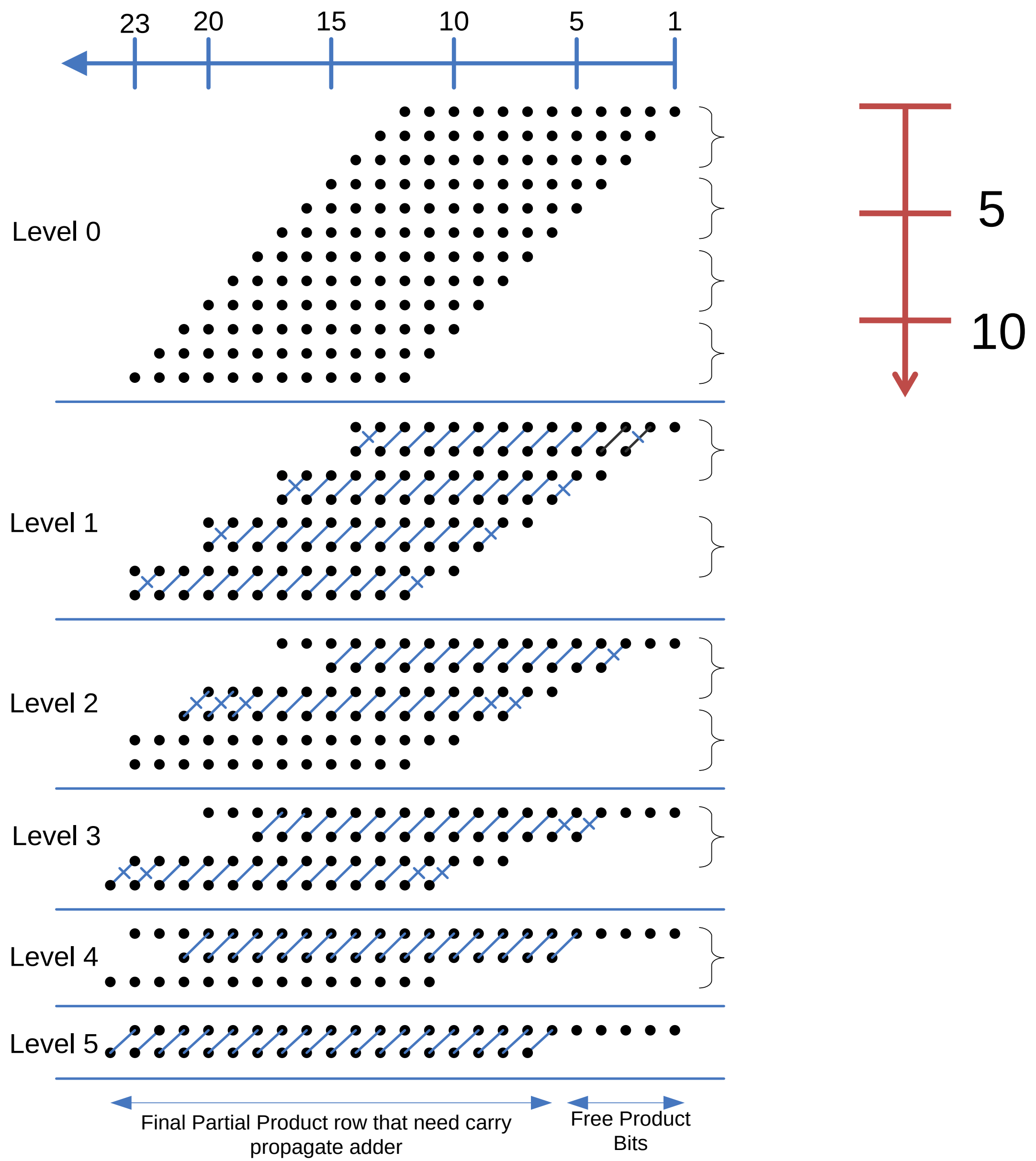
Wallace Tree Multipliers

- One of the most commonly used multiplier architecture
- It is log time array multiplier
- The number of adder levels increases logarithmically as the partial product rows increase

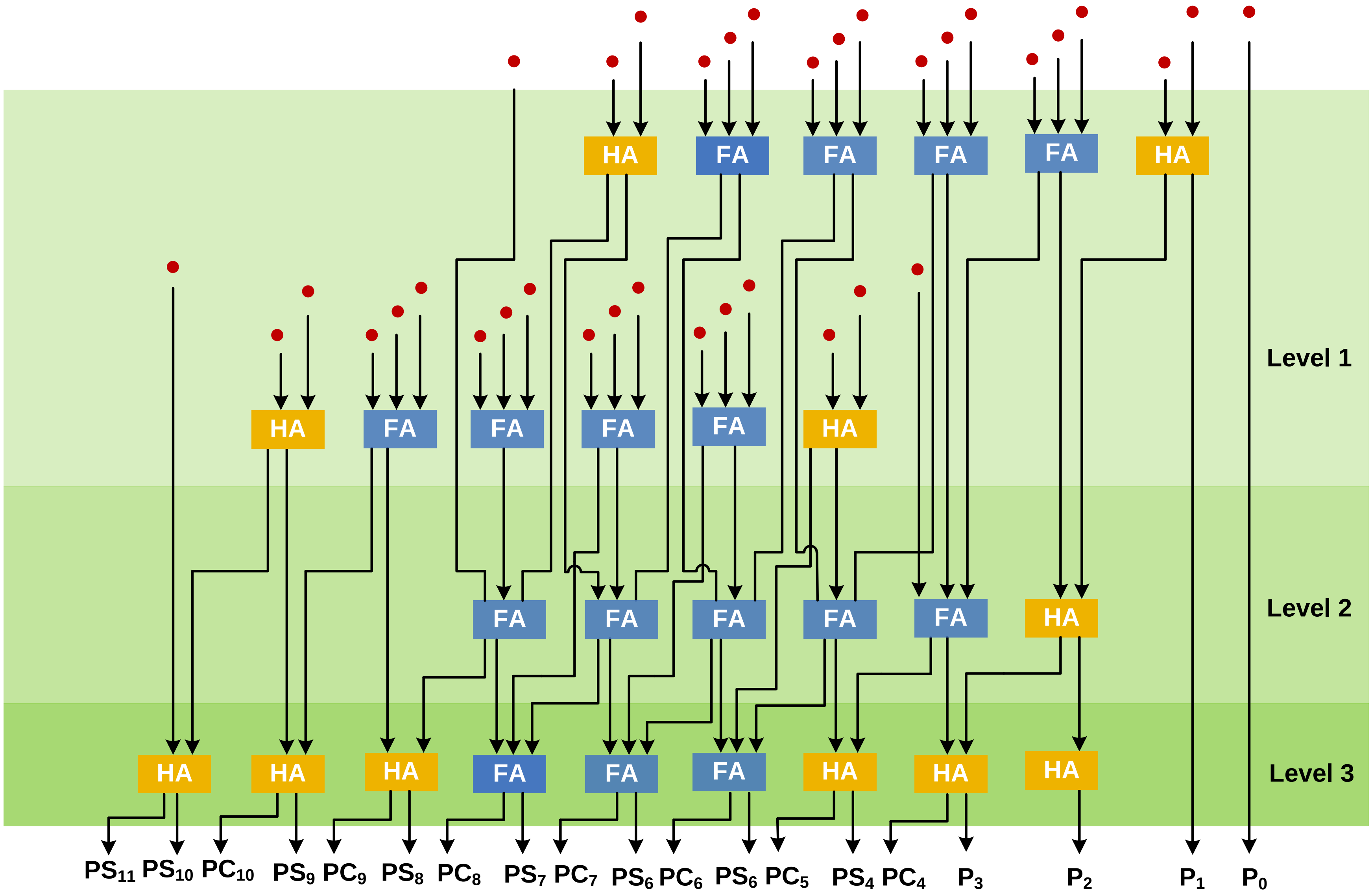
Wallace Tree Multipliers

- Make group of threes and apply CSA reduction in parallel
- Each CSA layer produces two rows
- These rows then, with other rows from other partial product groups, form a new reduced matrix
- Iteratively apply Wallace reduction on the new generated matrix
- This process continues until only two rows are left
- The final rows are added together for the final product

Wallace Reduction Tree applied on 12 PPs



Wallace Reduction layout for a 6x6 array of PPs



Dada Reduction uses the Wallace Reduction Table

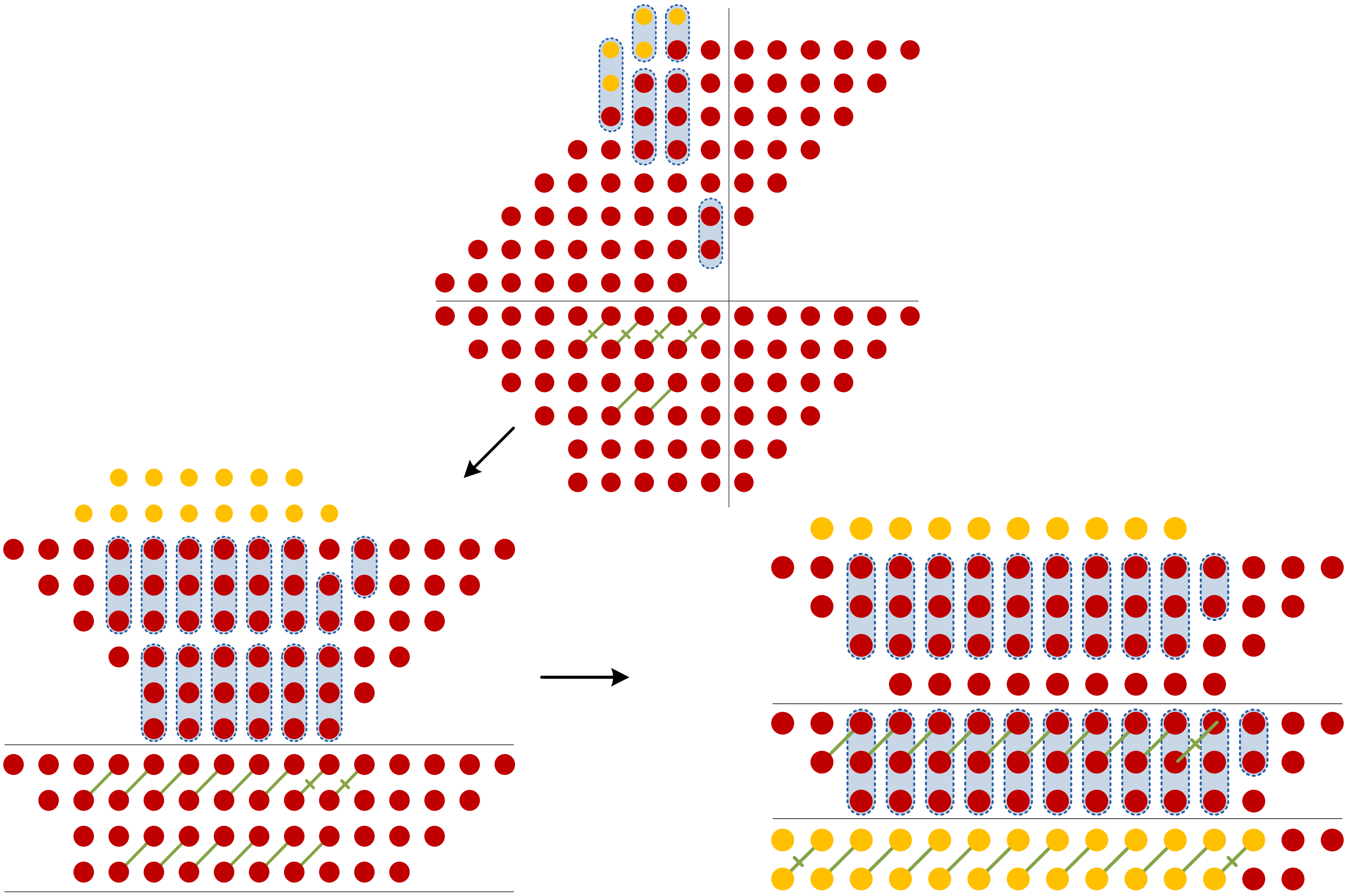
Adder Levels in Wallace Tree Reduction Scheme

Number of partial Products	Number of full adder Levels
3	1
4	2
$5 \leq n \leq 6$	3
$7 \leq n \leq 9$	4
$10 \leq n \leq 13$	5
$14 \leq n \leq 19$	6
$20 \leq n \leq 28$	7
$29 \leq n \leq 42$	8
$43 \leq n \leq 63$	9

Dada Reduction

- Minimizes the number of HAs and FAs
- Reduction considers each column separately
- Reduces the number of dots in each column to the maximum number of layers in the next level in Wallace Reduction Table

Dadda reduction levels for reducing eight PPs to two



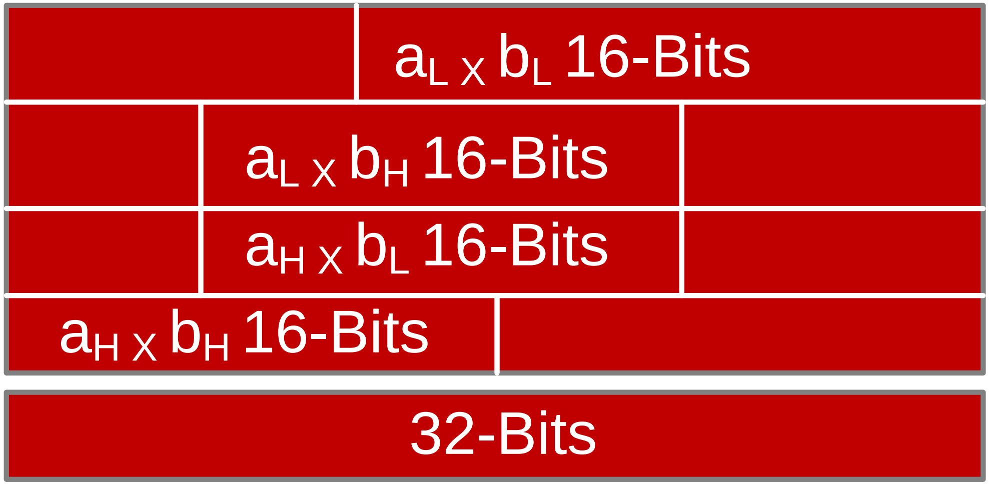
A Decomposed Multiplier

- Four Multipliers of size NxN can be combined to make a 2N x 2N multiplier

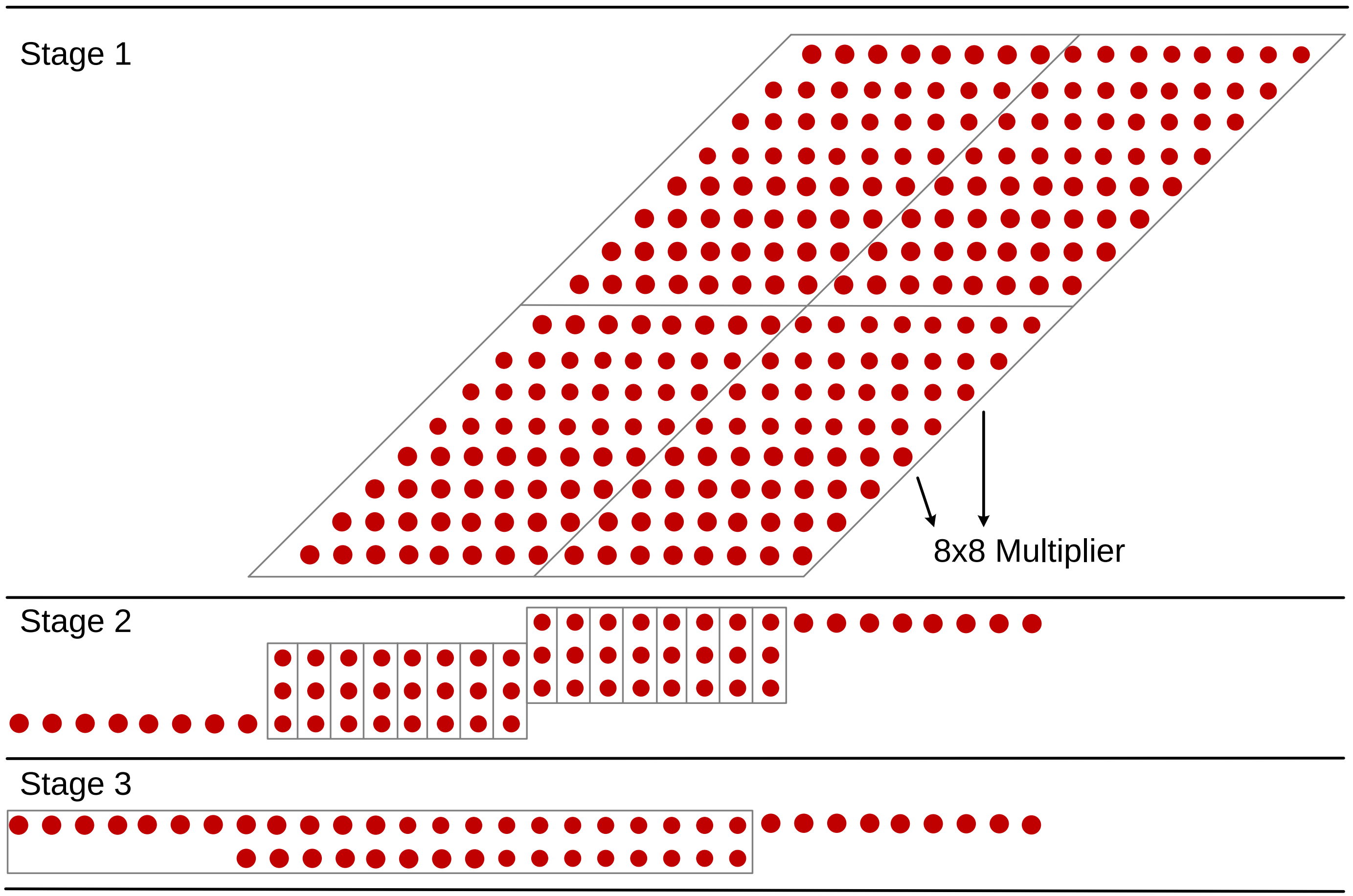
$$(a_L + 2^8 a_H) \times (b_L + 2^8 b_H)$$

$$= (a_L \times b_L + a_L b_H 2^8 + a_H b_L 2^8 + a_H b_H 2^{16}$$

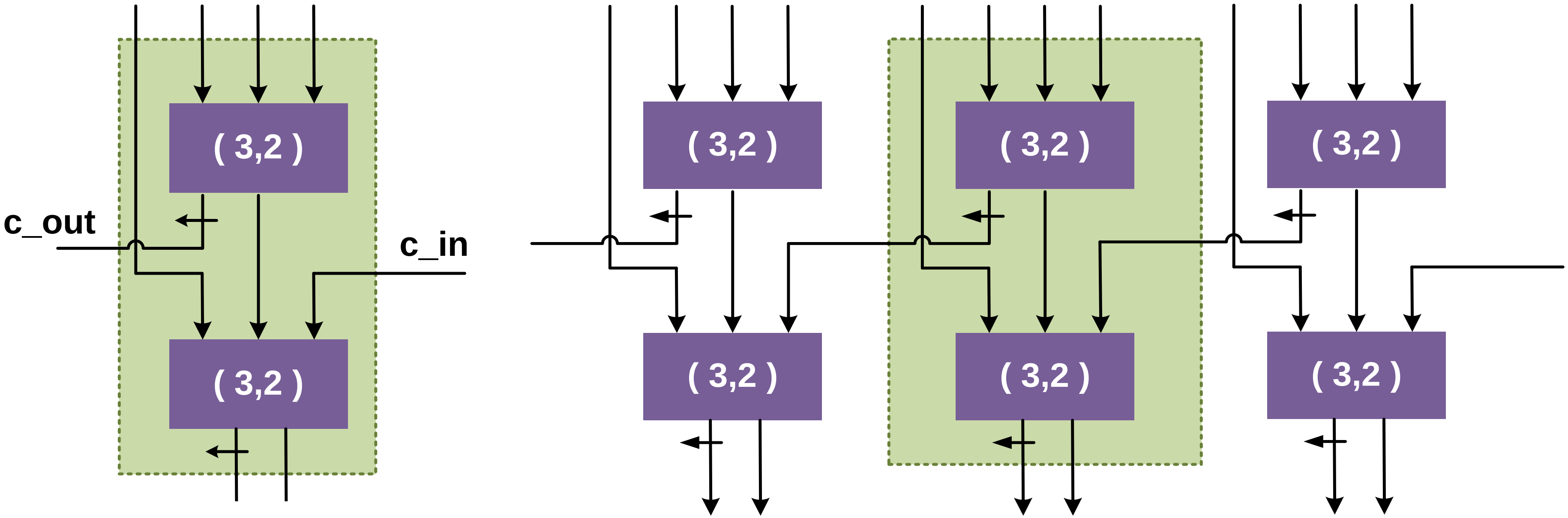
A 16x16 bit Multiplier decomposed into four 8x8 multipliers



The results of these multipliers are appropriately added to get the final product



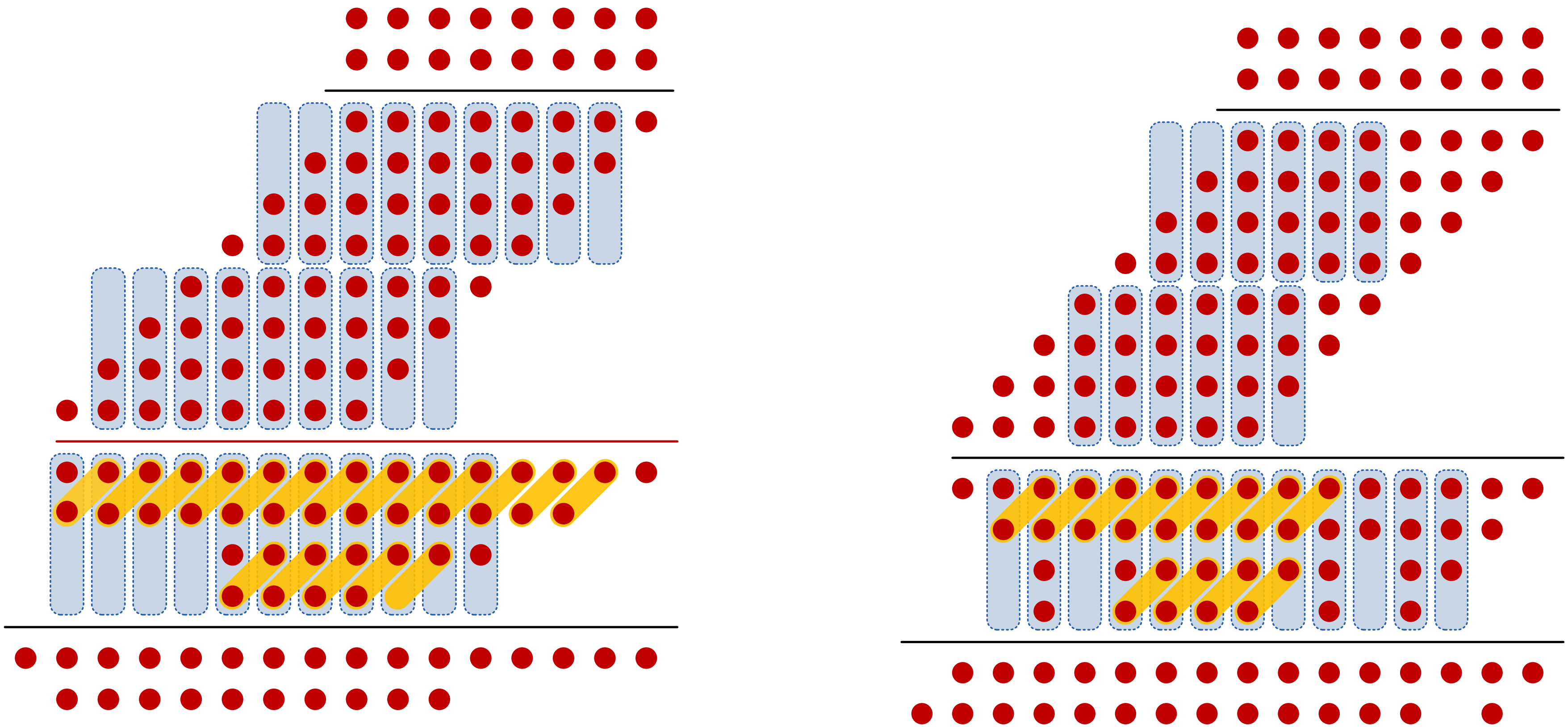
Optimized Compressors



(a)
Candidate implementation of 4:2 compressor

(b)
Concatenation of 4:2 compression to create wider tiles

Contd…

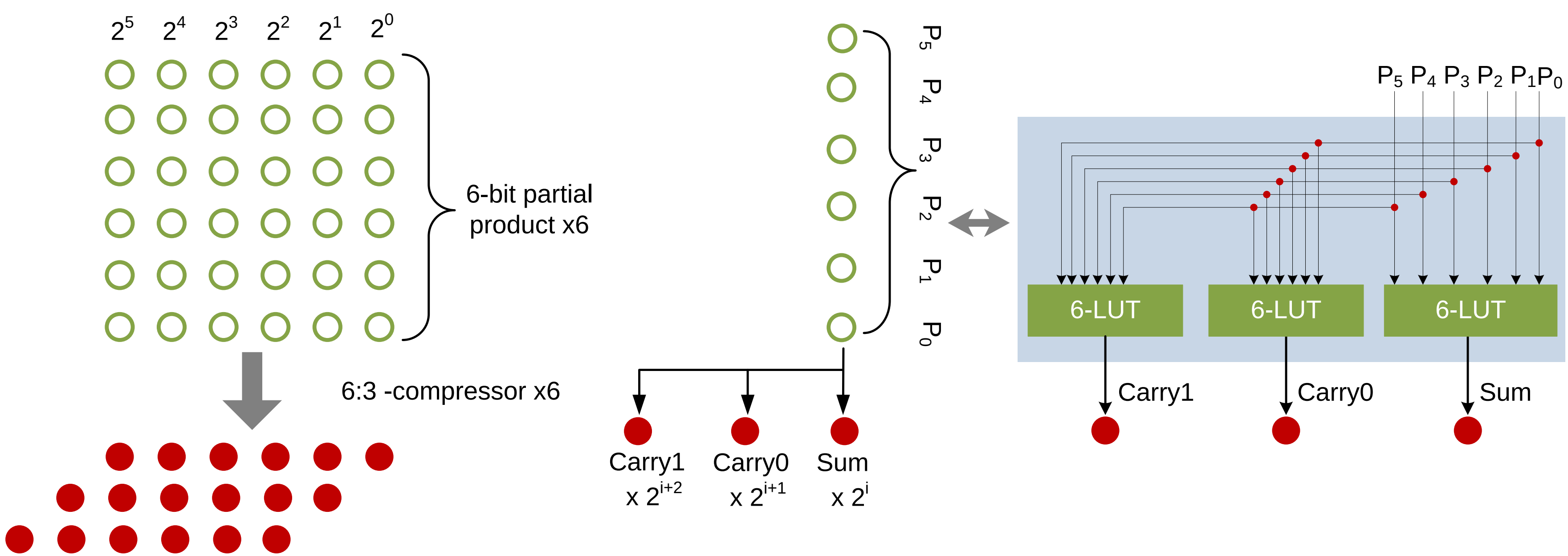


(c)
Use of 4:2 compressor in Wallace tree reduction of an 8x8 multiplier

(d)
Use of 4:2 compressor in an 88 multiplier in Dadda reduction

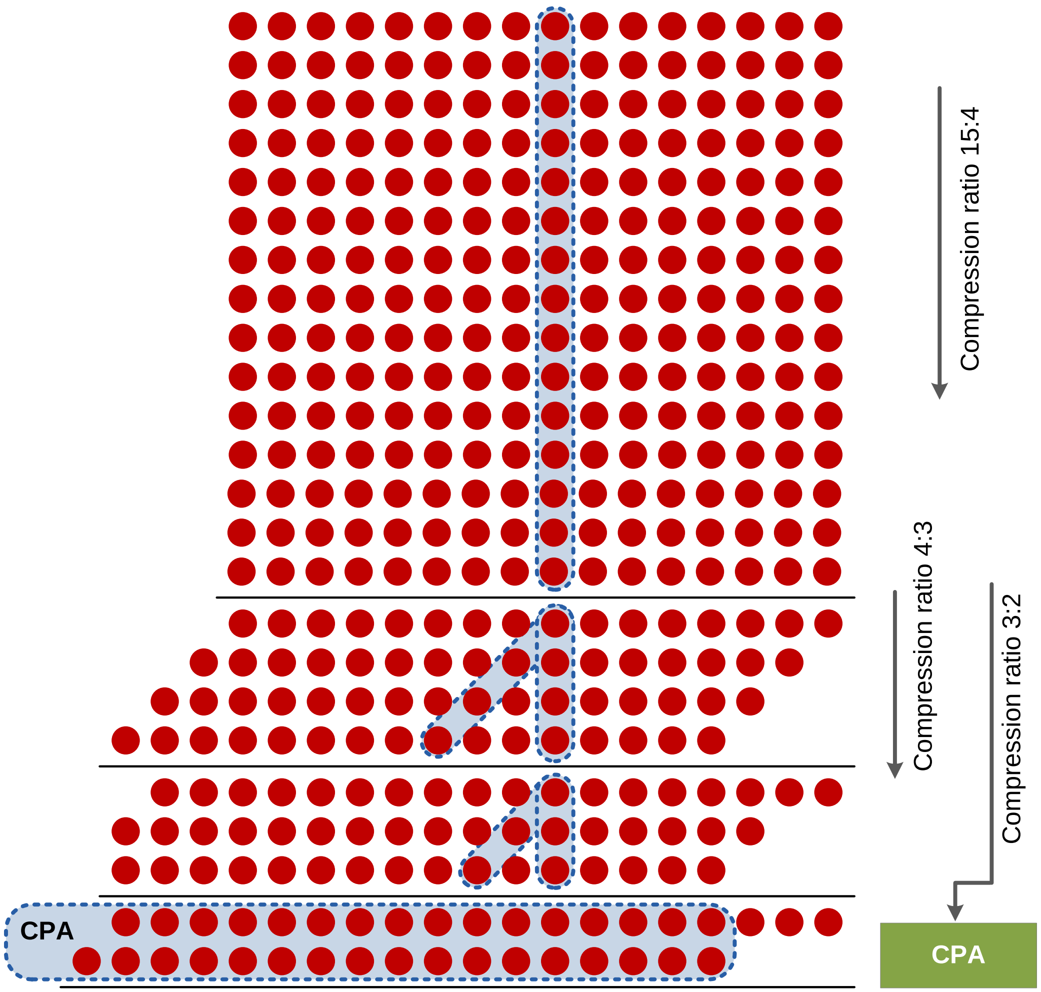
Single- and Multiple-column Counters

- A 6:3 counter reducing six layers of multiple operands to three
- A 6:3 counter is mapped on three 6-input LUTs

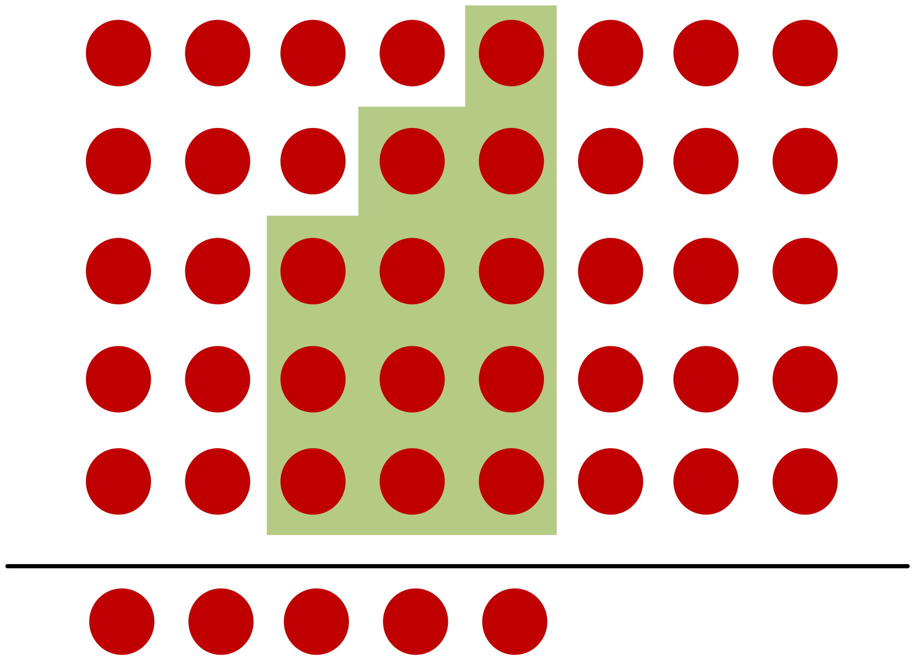


Counters compressing a 15x15 matrix

- 15:4, 4:3 and 3:2 counters working in cascade to compress a 15x15 matrix

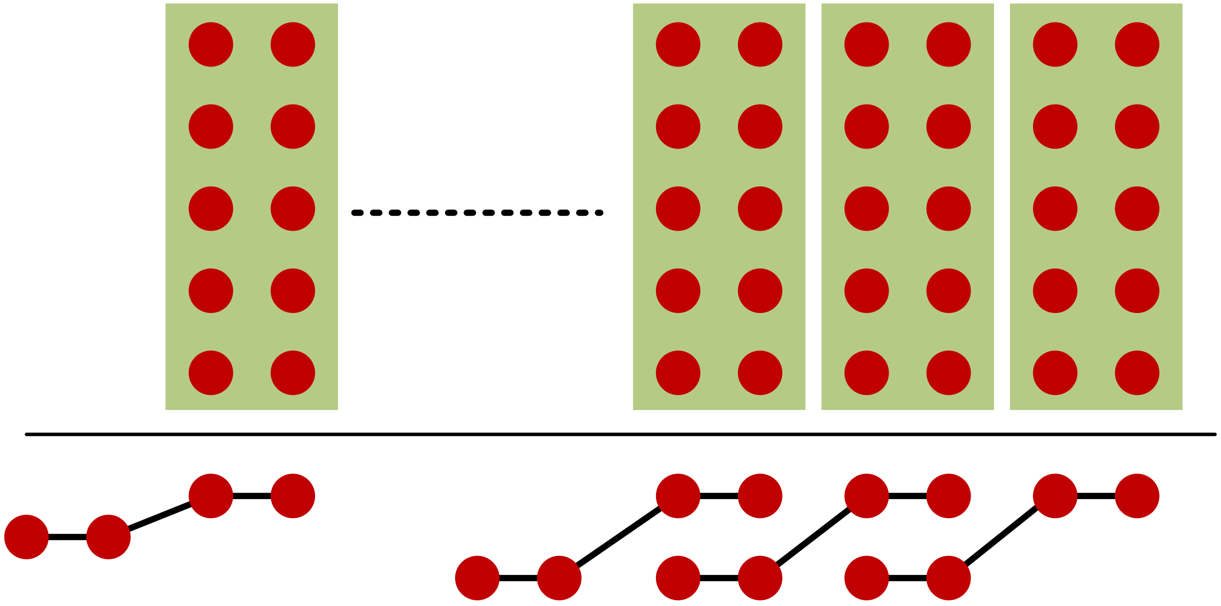


A (3,4,5:5) GPC compressing three columns with 3, 4, and 5 bits to 5 bits in different columns

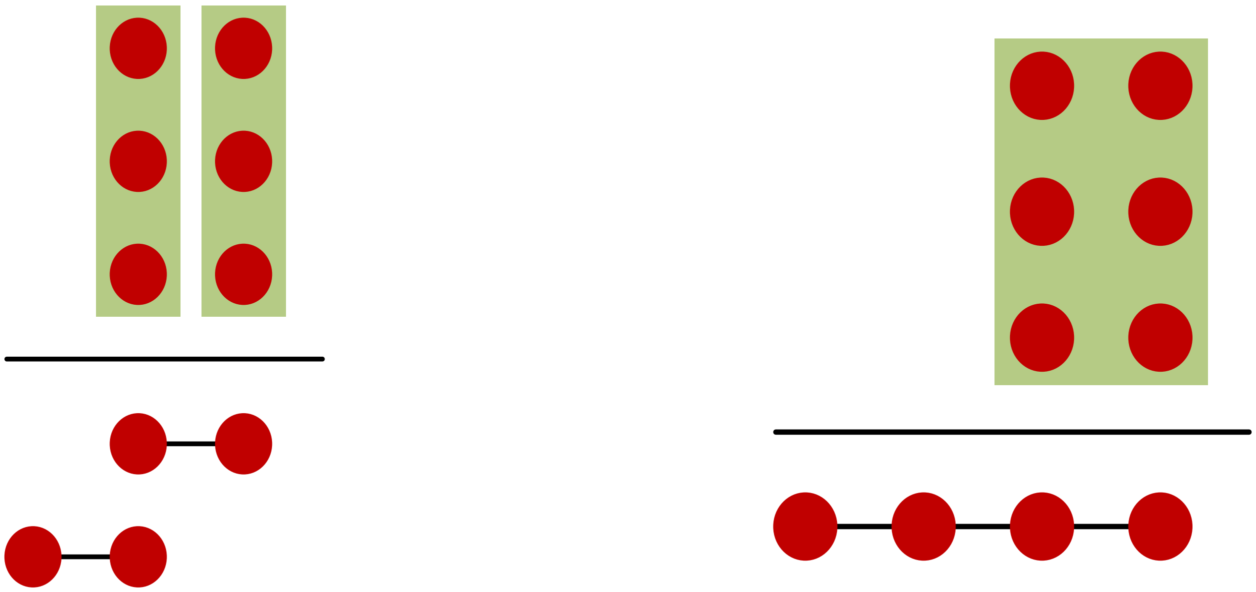


Compressor Tree Synthesis using compression of two columns of 5 bits each into 4 bit (5, 5; 4) GPCs

- Two columns of 5 bits each results into 4-bit
- This GPC is represented as (5,5;4)



Compressor tree mapping by (a) 3:2 counters (b) and a (3, 3; 4) GPC



(a) (b)

Two's Complement Signed Multiplier

- The sign bit in 2's complement representation plays a critical role in signed multiplier

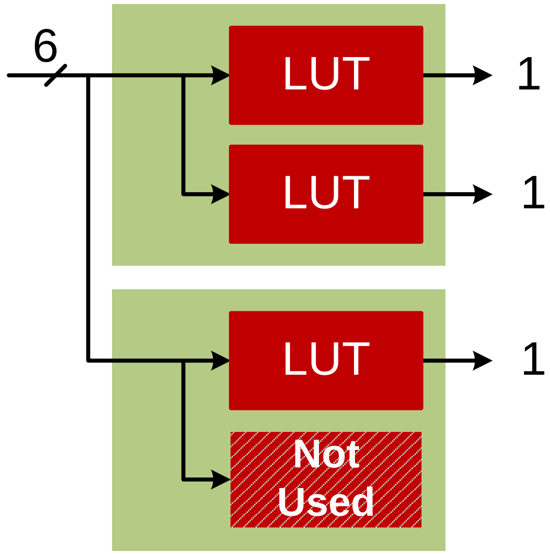
$$x = -x_{n-1} 2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i$$

$$PP[i] = (a_i 2^i)\left(-b_{n-1} 2^{N_2 - 1} + \sum_{i=0}^{N_2 - 2} b_i 2^i \right) \qquad \text{for } i=0,1,\ldots,N_1\text{-}2$$

$$PP[N_1 - 1] = (-a_{N_1 - 1} 2^{N_1 - 1})\left(-b_{n-1} 2^{N_2 - 1} + \sum_{i=0}^{N_2 - 2} b_i 2^i \right)$$

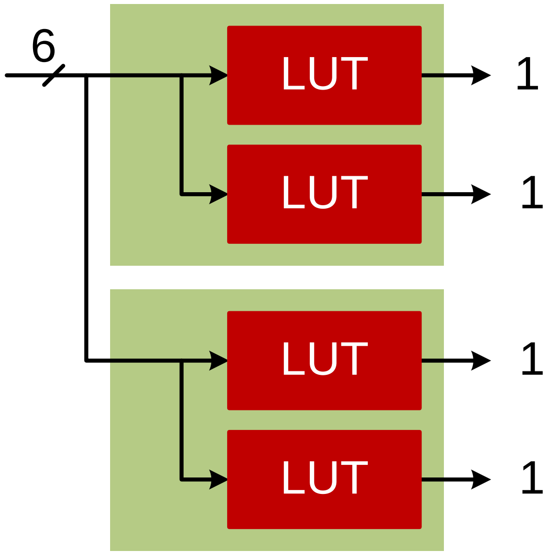
Optimized GPC for FPGA Implementation

- FGPAs are best suited for counters and GPC-based compression trees
- LUTs in many FPGAs come in groups of two with shared 6-bit input
- A GPC (3,3;4) best utilize 6-LUT-based FPGAs



(a)

An Altera FPGA Adaptive Logic Module (ALM) contains two 6-LUTs with shared inputs, 6 inputs, 3 outputs GPC has 3/4 logic utilization

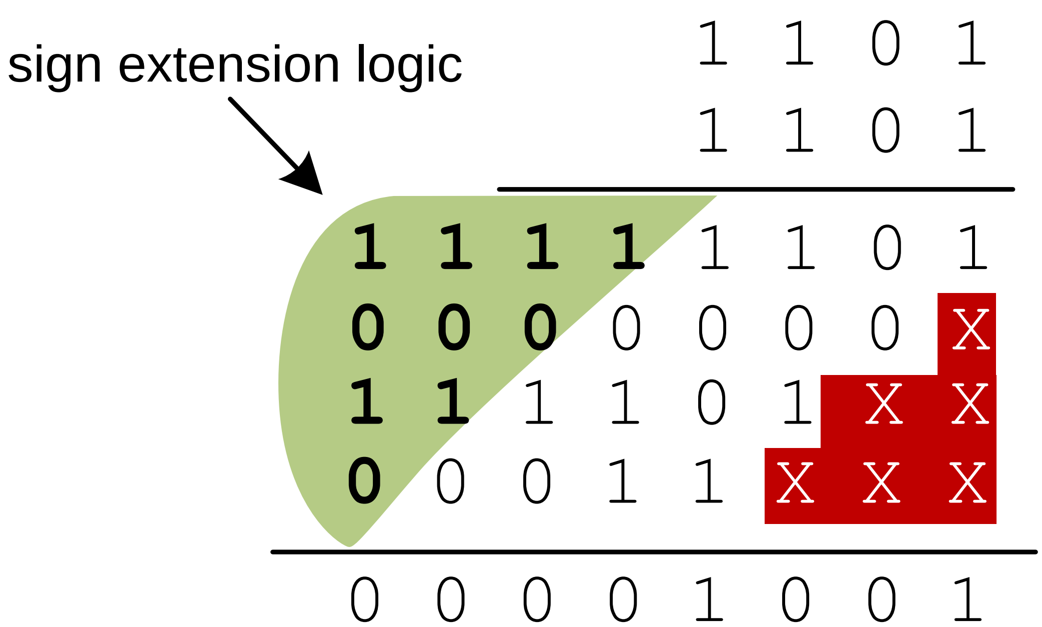


(b)

A 6 inputs, 4 outputs GPC has full logic utilization.

Showing 4 x 4-bit signed by signed multiplication

- To cater for the sign bit
  - The sign bits of the first three PPs are extended
  - Two’s complement of the last PP is taken
- HW implementation results in additional logic



Sign - extension Elimination

extend all 1s

flip the sign bit

$$
\begin{array}{rl}
B= & 0\ 0\ 0\ 0\ 0\ \ 0.1\ 1\ 0\ 1\ 0\ 1\ 1 \\
B= & \boxed{1\ 1\ 1\ 1\ 1}\ \ \bar{0}.1\ 1\ 0\ 1\ 0\ 1\ 1 \\
+ & \ \ \ 1 \\
\hline
 & 0\ 0\ 0\ 0\ 0\ \ 0.1\ 1\ 0\ 1\ 0\ 1\ 1 \\
\hline
\end{array}
$$

add 1 at the location of sign bit

(a)

extend all 1s

flip the sign bit

$$
\begin{array}{rl}
B= & 1\ 1\ 1\ 1\ 1\ \ 1.1\ 1\ 0\ 1\ 0\ 1\ 1 \\
B= & \boxed{1\ 1\ 1\ 1\ 1}\ \ \bar{1}.1\ 1\ 0\ 1\ 0\ 1\ 1 \\
+ & \ \ \ 1 \\
\hline
 & 1\ 1\ 1\ 1\ 1\ \ 1.1\ 1\ 0\ 1\ 0\ 1\ 1 \\
\hline
\end{array}
$$

add 1 at the location of sign bit

(b)

Sign-Extension Elimination and CV Formulation for signed by signed Multiplication

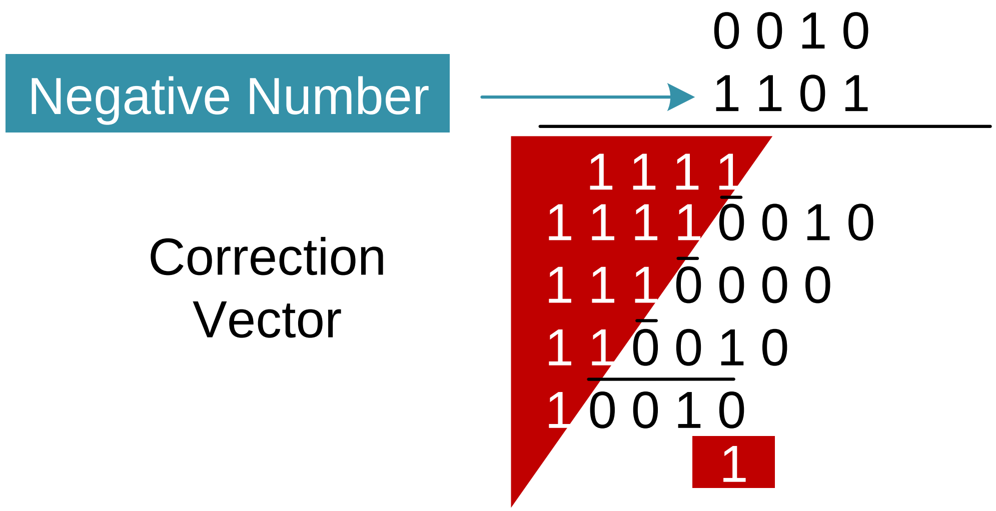
```
111111
11111SXXXXXXXXXX
1111SXXXXXXXXXX
111SXXXXXXXXXX
11SXXXXXXXXXX
1SXXXXXXXXXX
SXXXXXXXXXX
          1
```

1's compliment and adding 1 at LSB

```
1 1 1 1 1 1           1
1 1 1 1 1
1 1 1 1
1 1 1
1 1
1
-------------------------------
0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0
-------------------------------
```

Multiplying two numbers, 0011 and 1101

- All the 1s in red area are added to get CV
- The CV is 8’b0001_0000



(a)

Contd…

- CV is simply added as one of PP
- In case of NxN multiplier, CV is always a 1 at N+1 bit location

```
                        0 0 1 0
                        1 1 0 1
                     ____________
Correction Vector     1 1 0 1 0
                      1 0 0 0 X
                    1 0 1 0 X X
                  0 1 0 1 X X X
              ___________________
                  1 1 1 1 0 1  0
```

(b)

Contd…

- The MSB of all the PPs except the last one are flipped and a 1 is added at the sign-bit location, and the number is extended by all 1s
- For the last PP, the two's complement is computed
  - Flip all the bits and adding 1 to the LSB position
  - The MSB of the last PP is flipped again and 1 is added to this bit location for sign extension.
- All these 1s are added to find a correction vector (CV)

Application of the string property

$$0\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ \boxed{1\ 1\ 1}\ 0\ 1$$

String

$$0\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ \boxed{1\ 1\ 1\ 1\ 1}\ 0\ 0\ \bar{1}\ 0\ 1$$

String

$$0\ 0\ 1\ 1\ 1\ 1\ 0\ \boxed{1\ 1\ 1}\ 0\ 0\ 0\ 0\ \bar{1}\ 0\ 0\ \bar{1}\ 0\ 1$$

String

$$0\ 0\ \boxed{1\ 1\ 1\ 1\ 1}\ 0\ 0\ \bar{1}\ 0\ 0\ 0\ 0\ \bar{1}\ 0\ 0\ \bar{1}\ 0\ 1$$

String

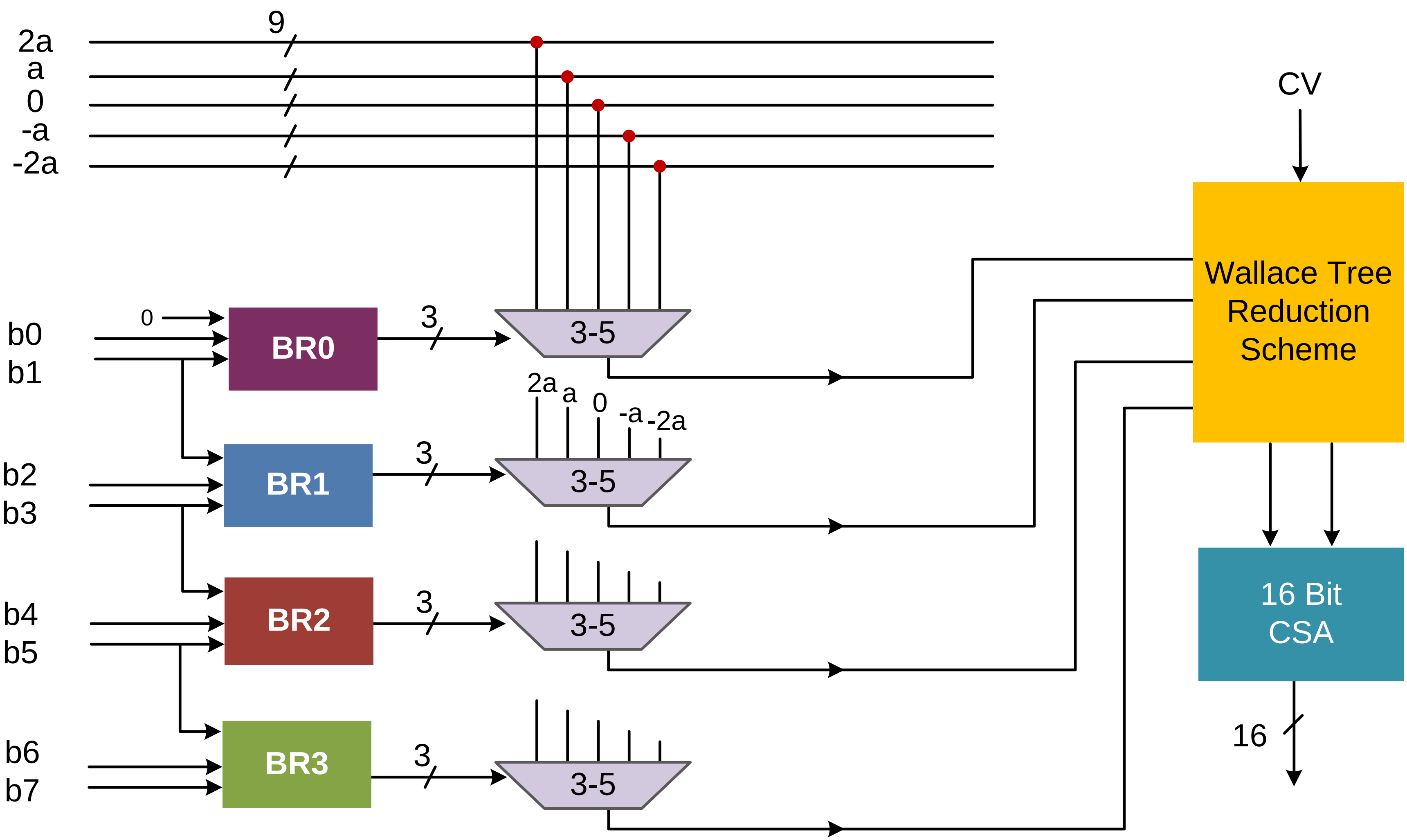
$$0\ 1\ 0\ 0\ 0\ 0\ \bar{1}\ 0\ 0\ \bar{1}\ 0\ 0\ 0\ 0\ \bar{1}\ 0\ 0\ \bar{1}\ 0\ 1$$

Hence the number of 1(s) has reduced from 14 to 6. Both have the same value.

Generation of four PPs

	10	10	11	01
	−2	+1	−1	1
11111111	10	10	11	01
00000001	01	00	11	
11111010	11	01		
00101001	1			
00100101	01	00	10	01

An 8 x 8 bit modified Booth recoder multiplier



Pre-calculated part of the CV

1	0	1	0	1
1	1	1	1	$\bar{S}$
1	1	$\bar{S}$		
$\bar{S}$				
0	1	0	1	1

Algorithm Transformations for CSA

$$sum1 = op1 + op2;$$

$$sum2 = op3 + op4;$$

$$if\,(sum1 > sum2)$$

$$sel = 0;$$

$$else$$

$$sel = 1;$$

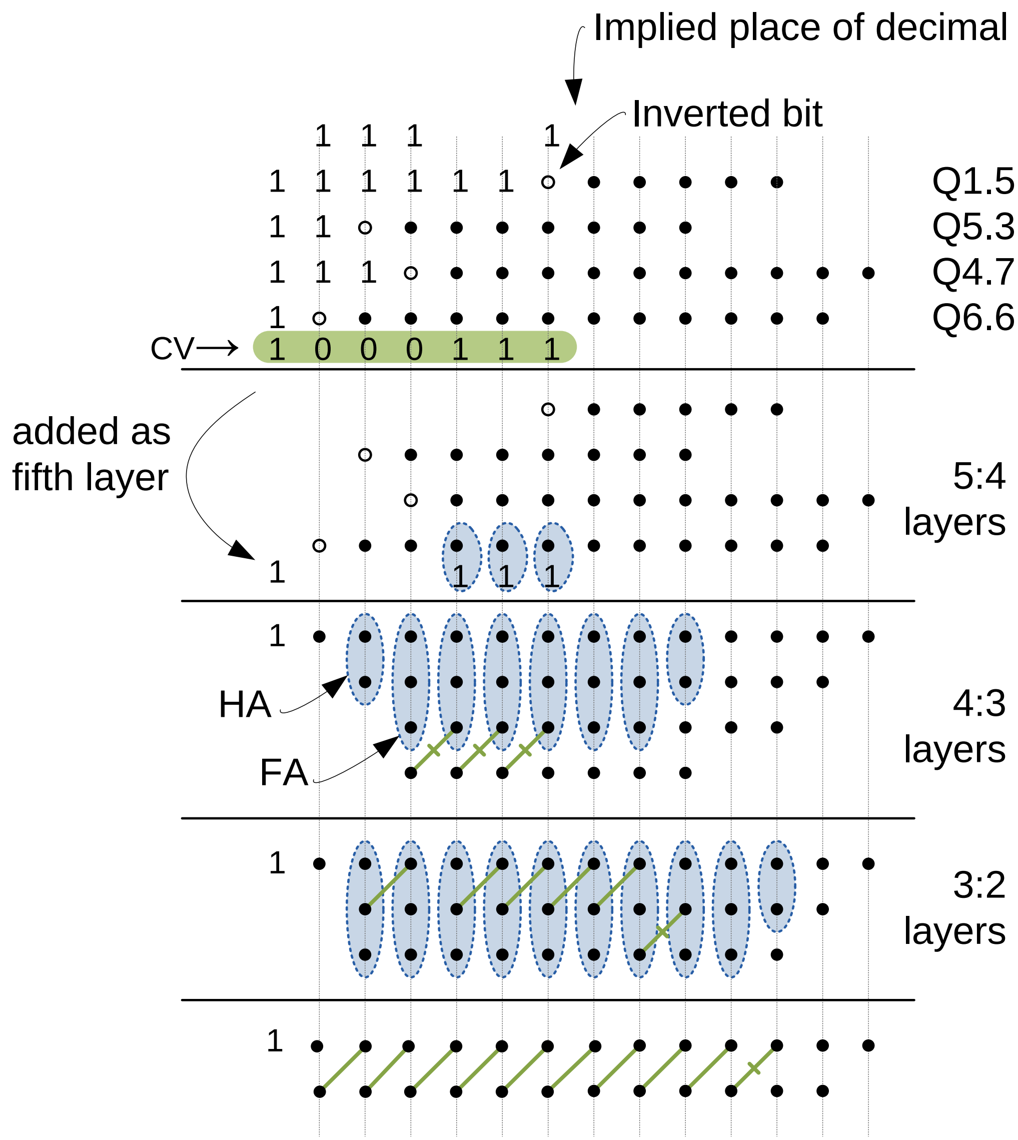
To transform the logic for optimal use of compression tree the algorithm is modified as:

$$sign(op1 + op2 - (op3 + op4)) = sign(op1 + op2 - op3 - op4)$$

Example: Multi Operands addition

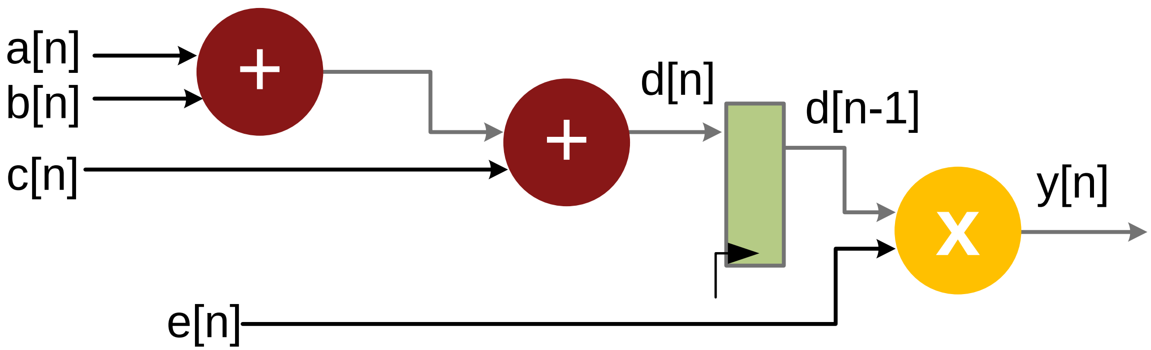
- Multiple operands addition should use compression tree
  - Avoid multiple instantiations of CPA
- The example adds Q1.5, Q5.3, Q4.7, and Q6.6 format sign numbers
- Compute CV using sign extension elimination technique
- Add it as 5th partial product
- Compress using dadda tree
- The last two rows can be added using any CPA

Example illustrating use of compression tree in multi-operand addition

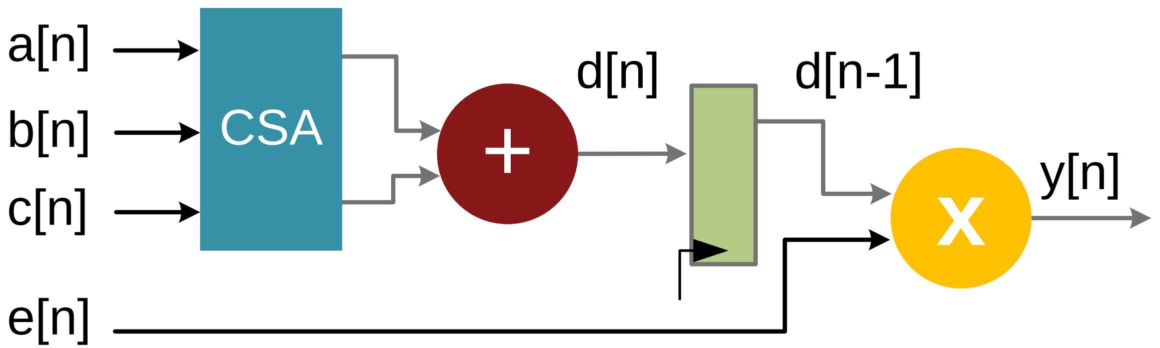


Algorithm Transformations for CSA

- Multi operands addition should use compression tree and one CPA



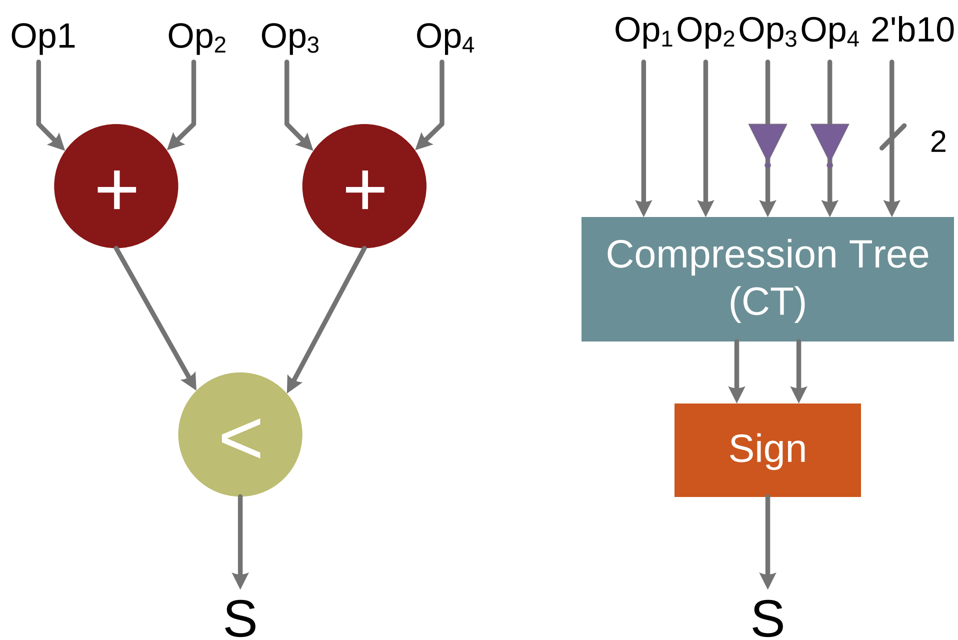
(a) FSFG with multi operand addition



(b) Modified FSFG reducing three operands to two

Compression tree replacement for an Add Compare and Select Operation

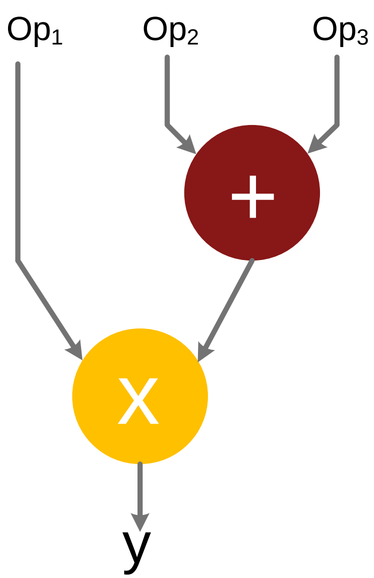
- In many applications multi operands addition is hidden and can be extracted
- This example performs an Add-Compare-Select operation
- The operation requires three CPAs
- The statements can be transformed to exploit compression tree

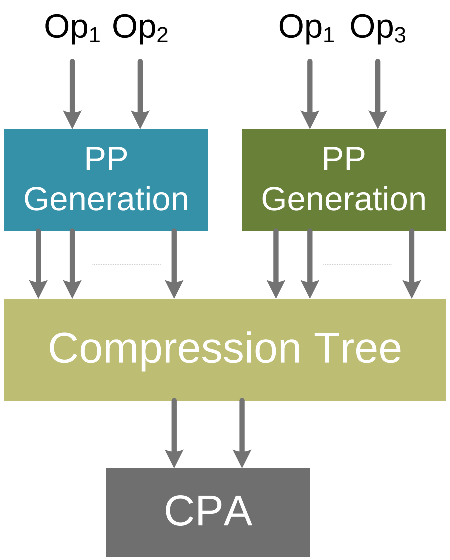


Transforming the add and multiply operations to use one CPA and a compression tree

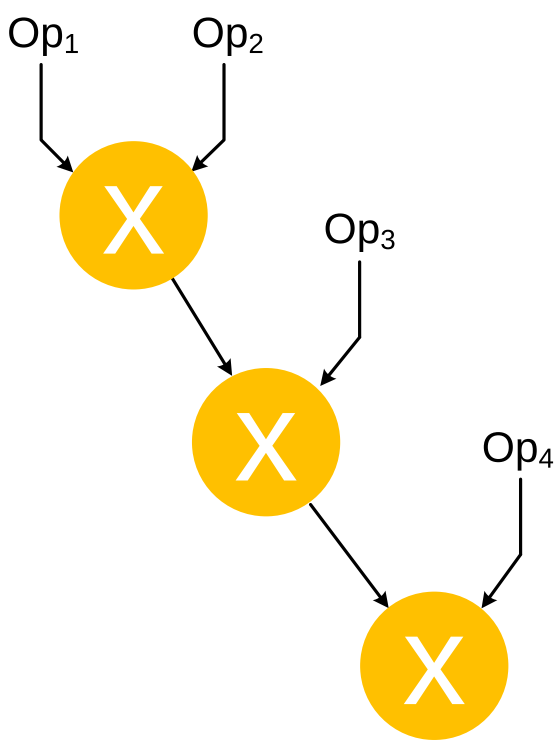
- Apply distributive property of multiplication
- Generate PPs for the two multiplications
- Use one compression tree to reduce all PPs to two layers
- Use one CPA to add these two layers

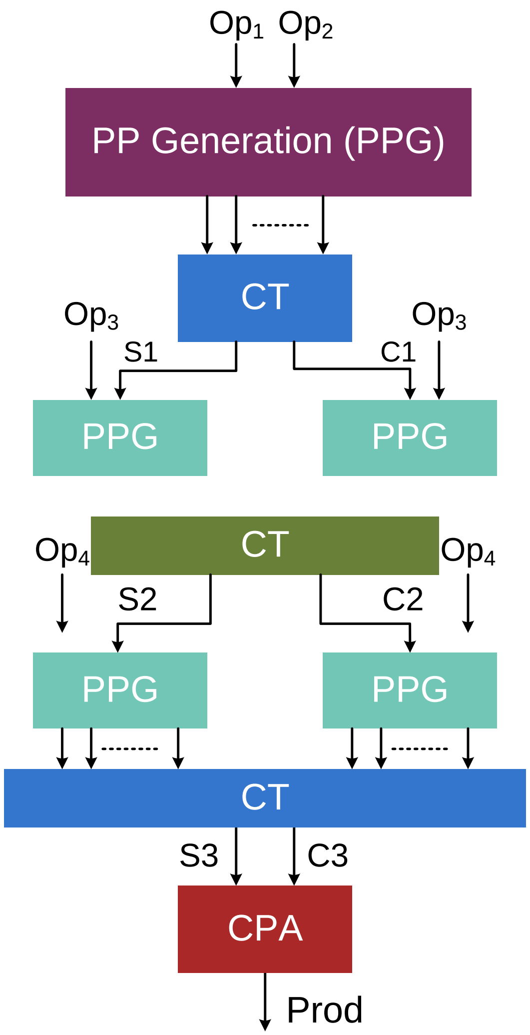
$$op1 \text{ x } (op2 + op3) = op1 \text{ x } op2 + op1 \text{ x } op3$$





Transformation to use compression trees and single CPA to implement a cascade of multiplication operations





String Property

- $7=111=8-1=100\overline{1}$
- 31= 1 1 1 1 1 =32-1

Or $1\ 0\ 0\ 0\ \overline{0}\ 1=32-1=31$

- Replace string of 1s in multiplier with
  - In a string when ever we have the least significant 1, we put a bar on it
  - We go to the end of the string
  - We replace all the 1(s) with 0
  - We put a 1 where the string ends

- Instead of multiplying with a single bit
- We multiply with two bits hence making the partial products half in No.

Booth Recoding Basic Idea

A = 1 0 1 0 1 1 0 1

B = 1 0 1 0 1 1 0 1

For these two bits Booth's algorithm restricts the value to be (-2, -1, 0, +1,+2)

+2 means Shift left A by one

+1 means Copy A in the answer

0 means copy all 0's

-1 means 2's complement and then copy

-2 means 2's complement and then shift left

Booth's Algorithm

- Form pairs using string property

1 0 1 0 1 1 0 1   0

- Use the MSB of the previous group to check for the string property on the pair, use 0 for the first pair

As the string property is applied on three bits, there are following eight possibilities:

$2^1=2$	$2^0=1$		
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0