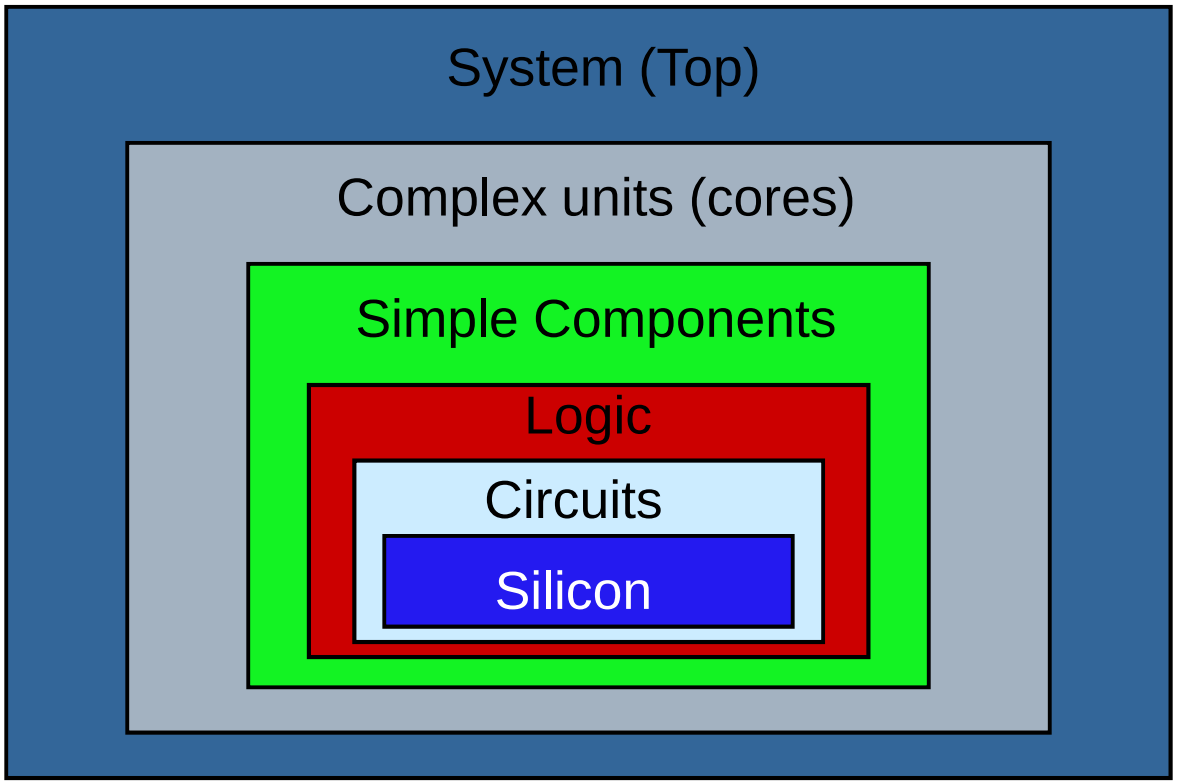
Chapter 3
VLSI Subsystem Design

Jin-Fu Li
Advanced Reliable Systems (ARES) Laboratory
Department of Electrical Engineering
National Central University
Jhongli, Taiwan

Outline

- Introduction
- Datapath Operators
- Control Structures

System-Level Hierarchy



Categories of Components

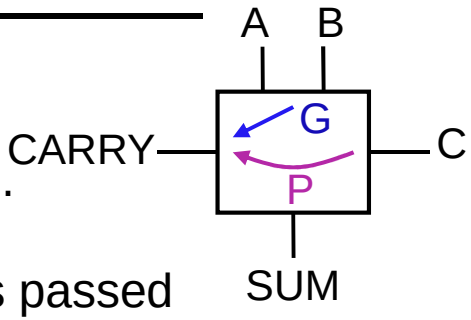
- Types of digital component
  - Datapath operators
  - Memory elements
  - Control structures
  - I/O cells
- Tradeoff of selection
  - Speed
  - Density
  - Programmability
  - Easy of design
  - etc

Datapath – *Adder*

Adder Truth Table

C	A	B	A.B(G)	A+B(P)	A⊕B	SUM	CARRY
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	0
0	1	0	0	1	1	1	0
0	1	1	1	1	0	0	1
1	0	0	0	0	0	1	0
1	0	1	0	1	1	0	1
1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1

Generate Signal G(A.B): occurs when a carry output (CARRY) is internally generated within the adder .



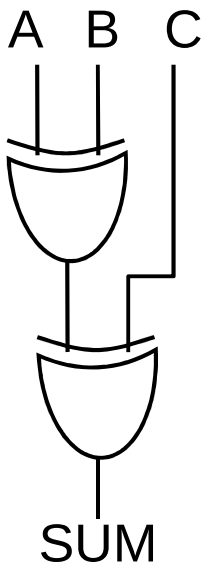
Propagate Signal P(A+B): when it is true, the carry in signal C is passed to the carry output (CARRY) when C is true

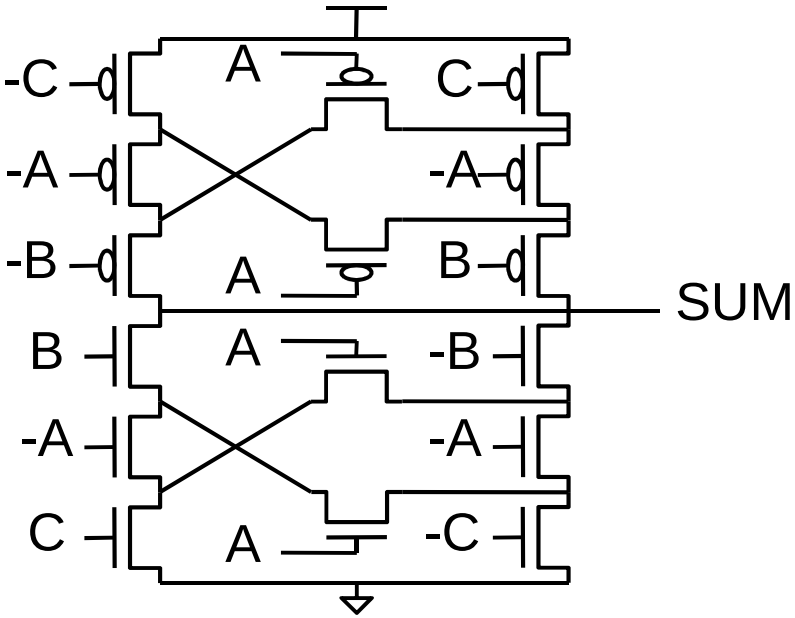
Datapath – *Adder*

$$SUM=A \oplus B \oplus C$$

$$CARRY=AB+AC+BC$$

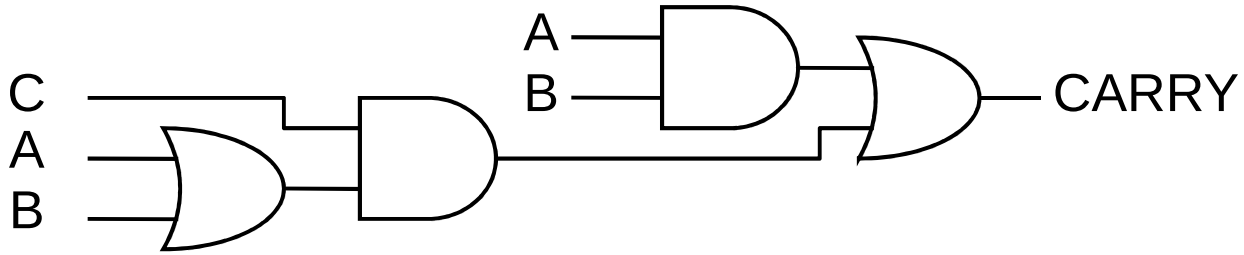
Single-bit schematic of SUM

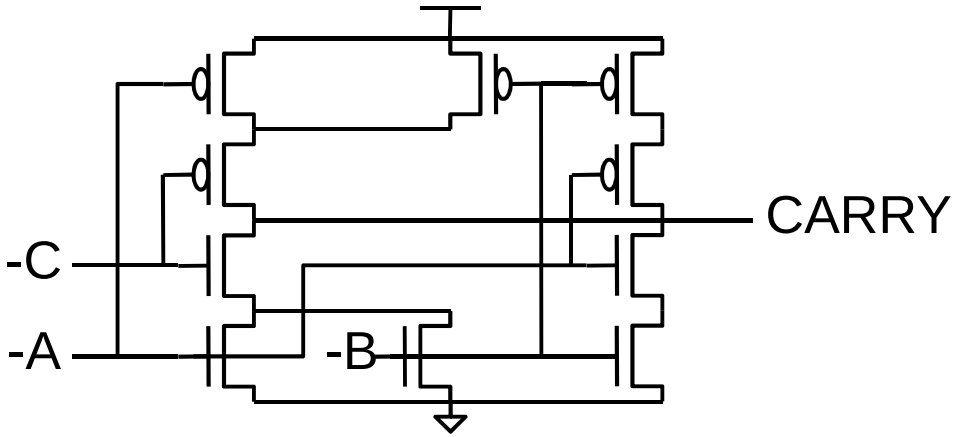




Datapath – *Adder*

Single-bit schematic of CARRY



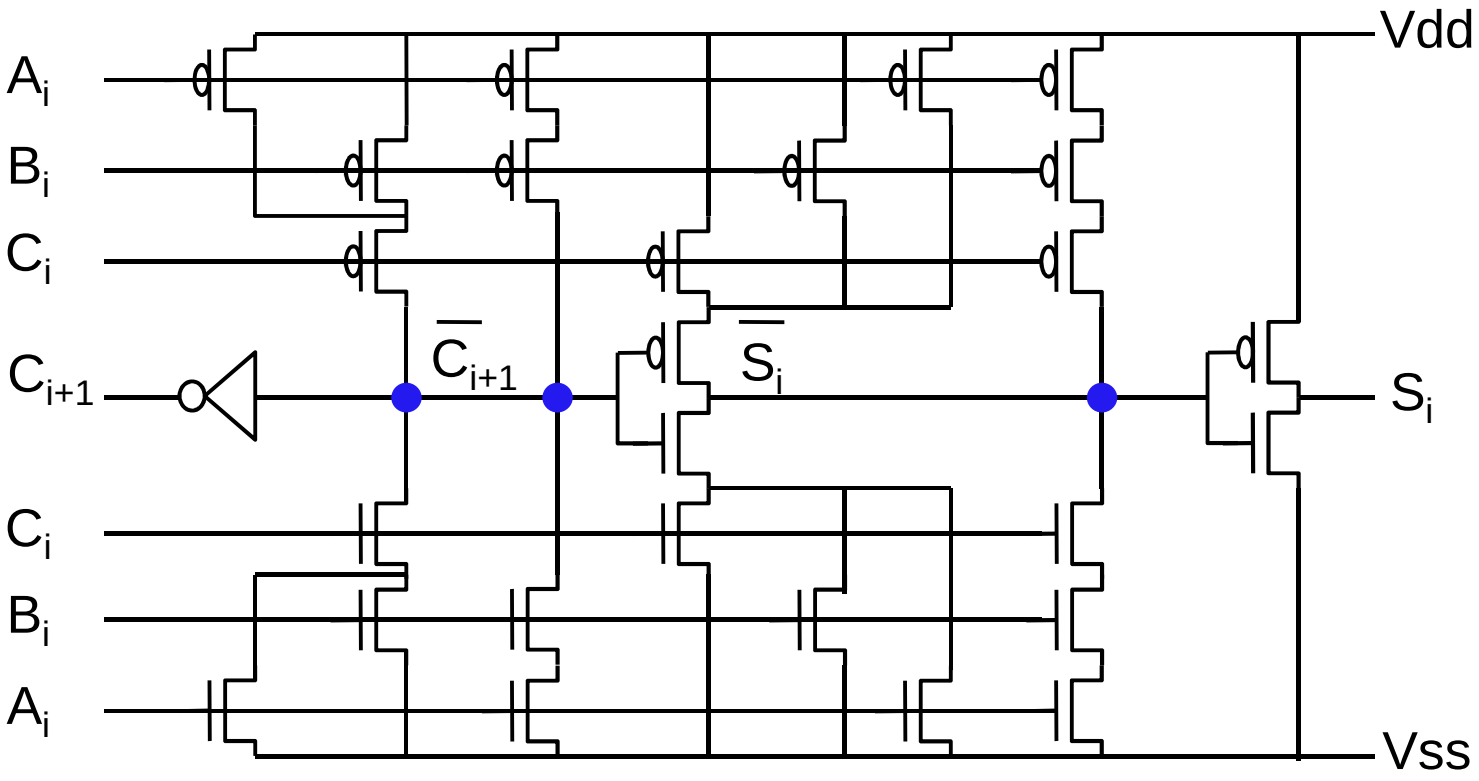


Datapath – *Adder*

Optimized combinational adder schematic

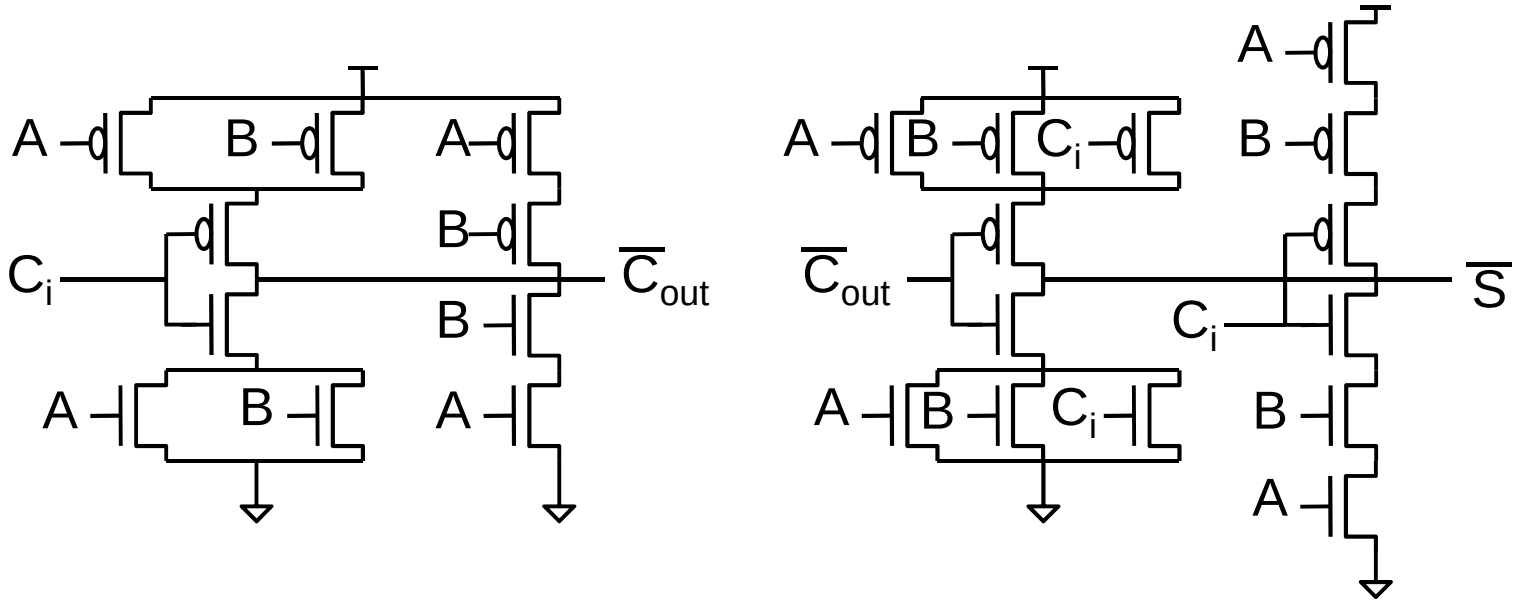
$$C_{i+1}=A_iB_i+A_iC_i+B_iC_i$$

$$S_i=(A_i+B_i+C_i).\overline{C}_{i+1}+A_iB_iC_i$$



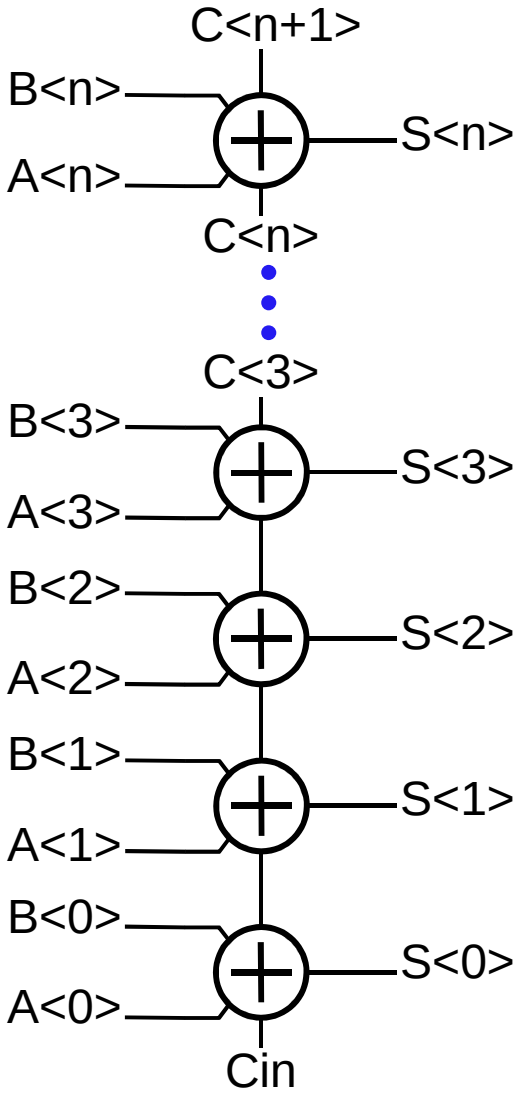
Datapath – *Adder*

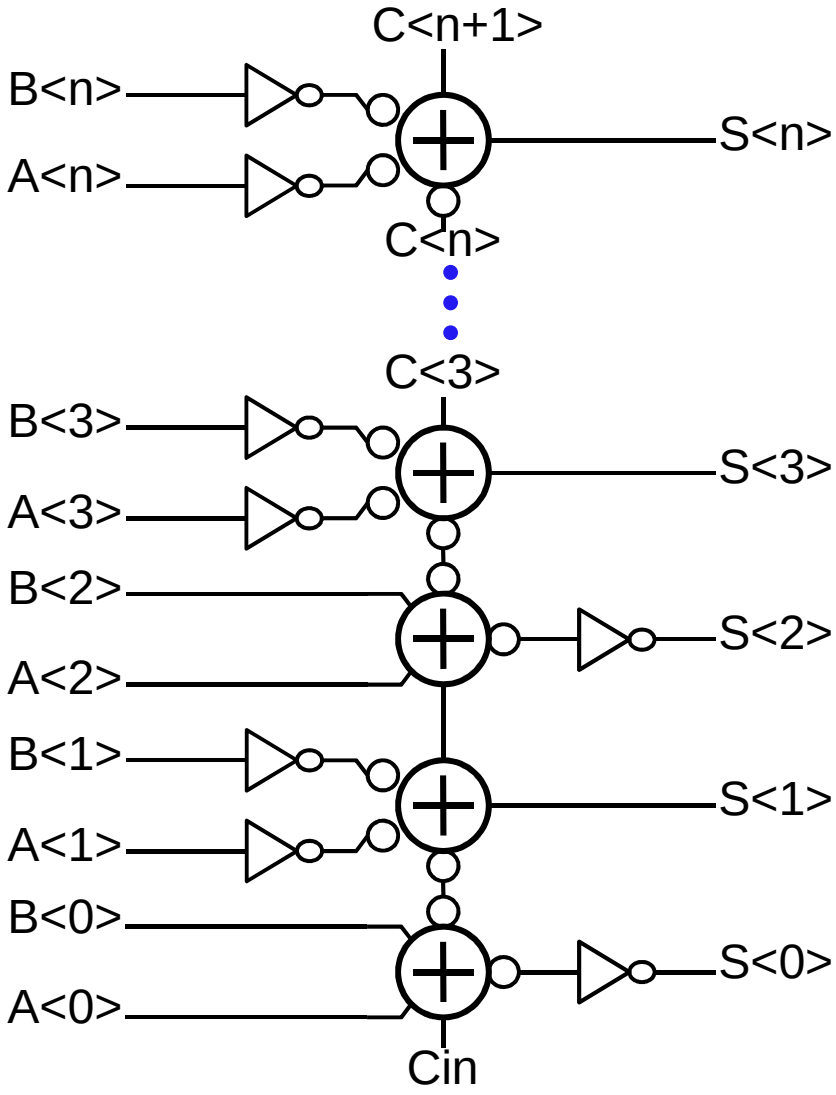
Symmetrical optimized combinational adder schematic



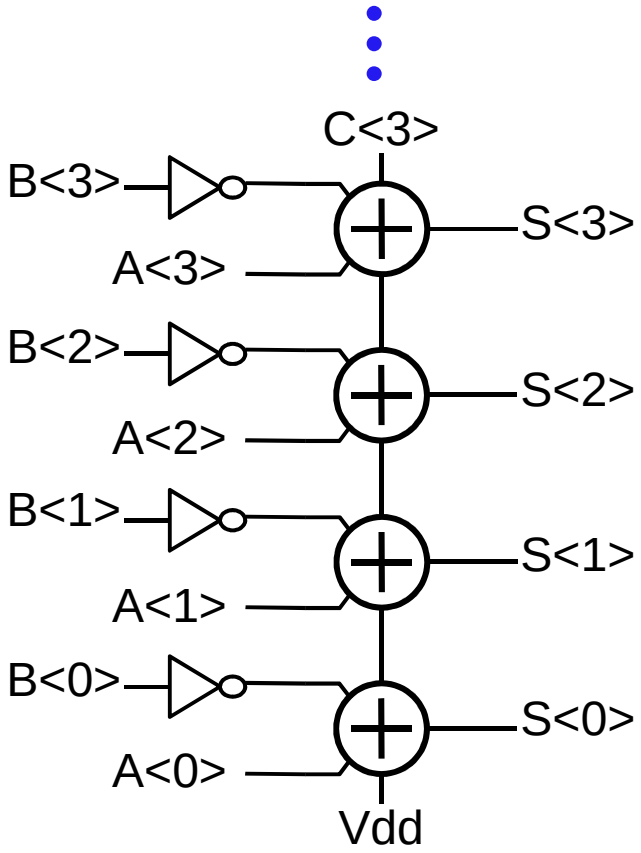
Datapath – *Bit-Parallel Adder*

Parallel adder implementations

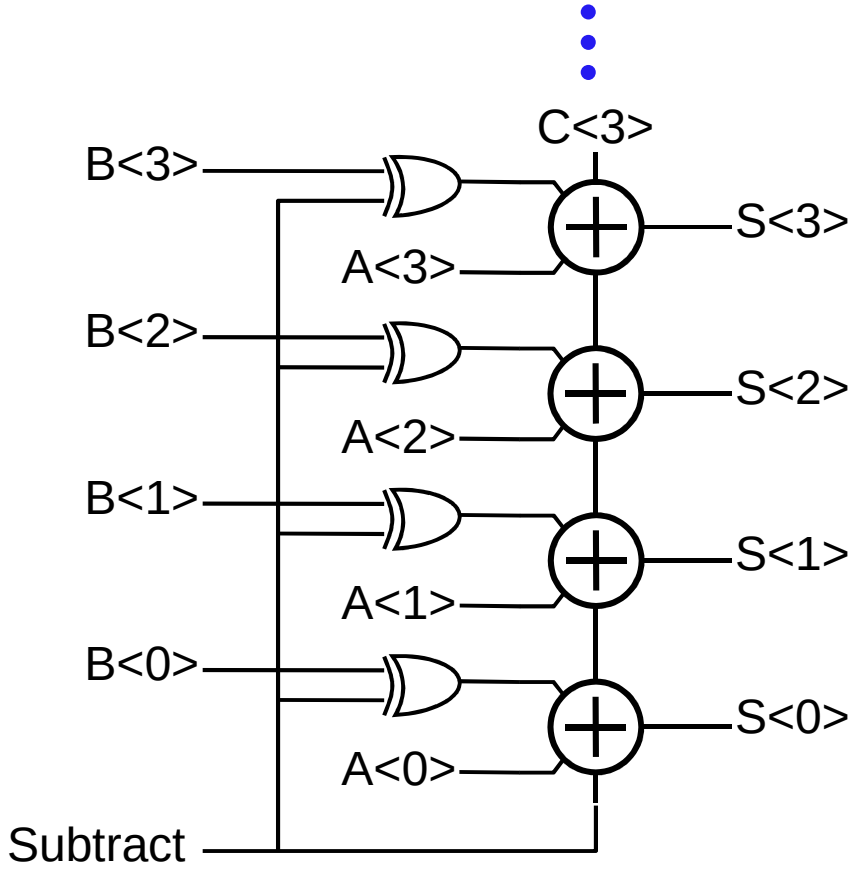




Datapath – *Bit-Parallel Adder*

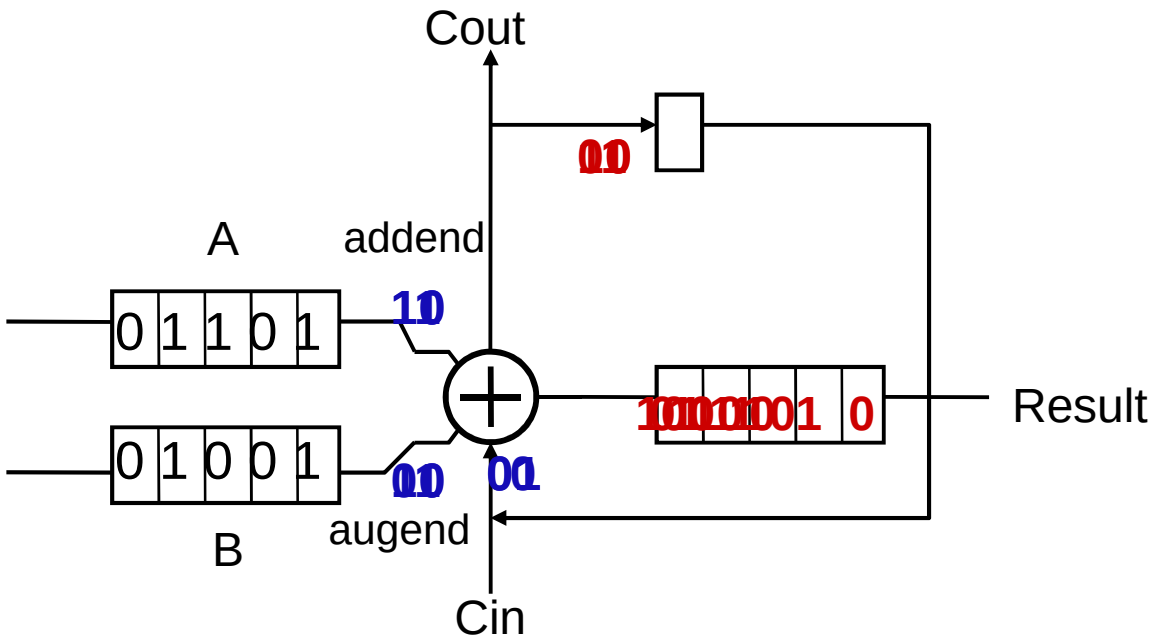


A-B



If (Subtract==0)
{S=A+B;}
else
{S=A-B;}

Datapath – *Bit-Serial Adder*



Datapath – *Carry Look-Ahead Adder (CLA)*

- Objective
  - To avoid the linear growth of the carry delay, we use a Carry Look-Ahead Adder (CLA) in which the carries can be generated in parallel
- Feature
  - The Carry of each bit is generated from the propagate and the generate signals as well as the input carry
  - The propagate and the generate signals are derived from the operand A_i and B_i by
  - $G_i=A_i.B_i$
  - $P_i=A_i+B_i$

Datapath – *Carry Look-Ahead Adder*

$C_{i+1}=A_iB_i+(A_i+B_i)C_i=G_i+P_iC_i$

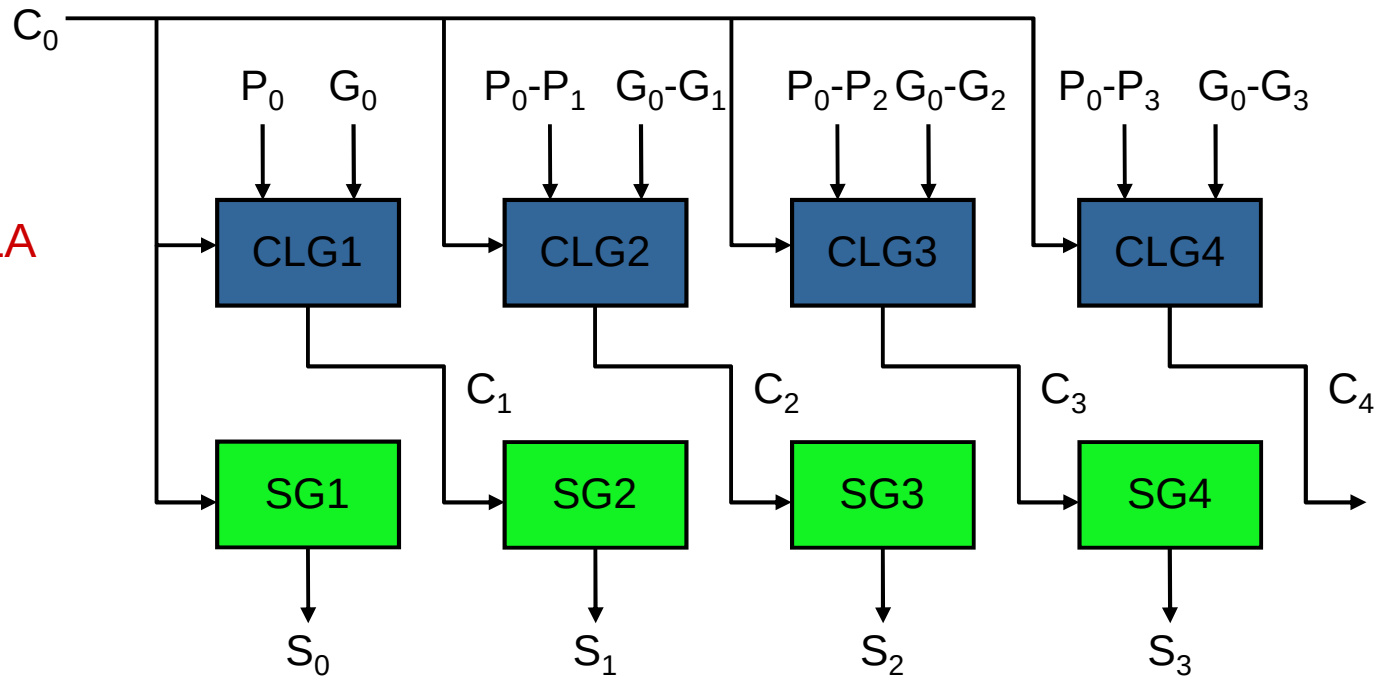
$C_1=G_0+P_0C_0$

$C_2=G_1+P_1G_0+P_1P_0C_0$

$C_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$

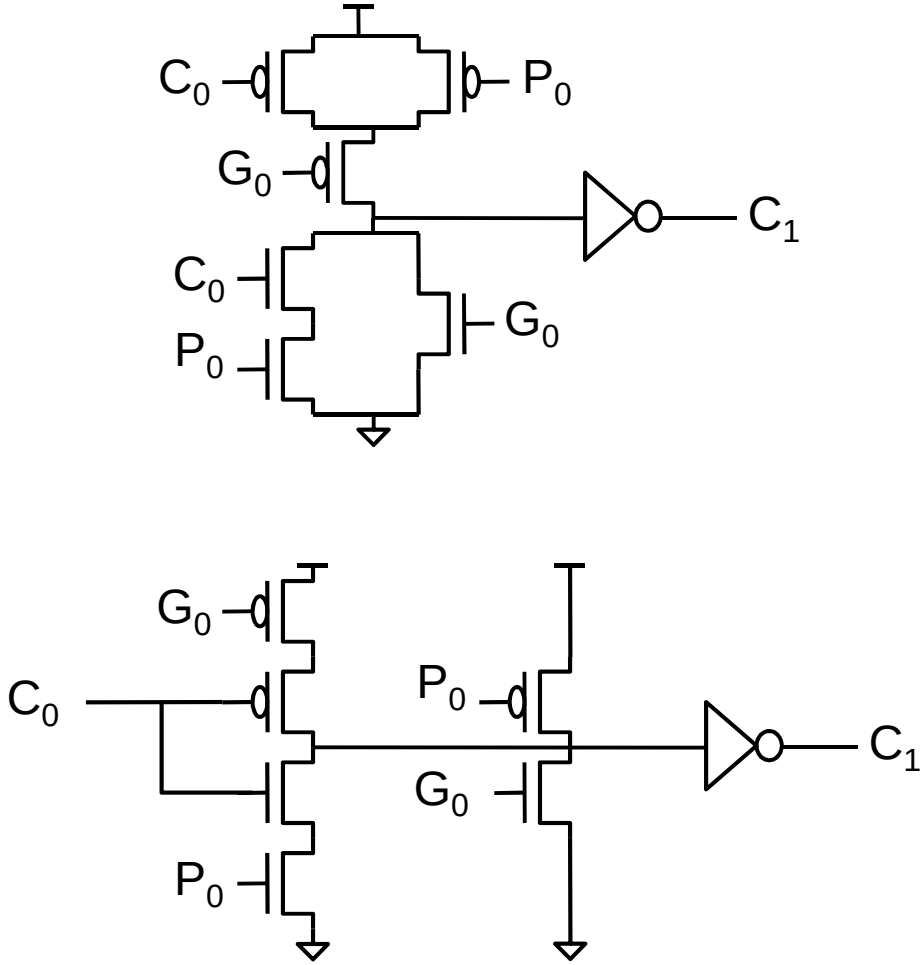
$C_4=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

4-bit CLA



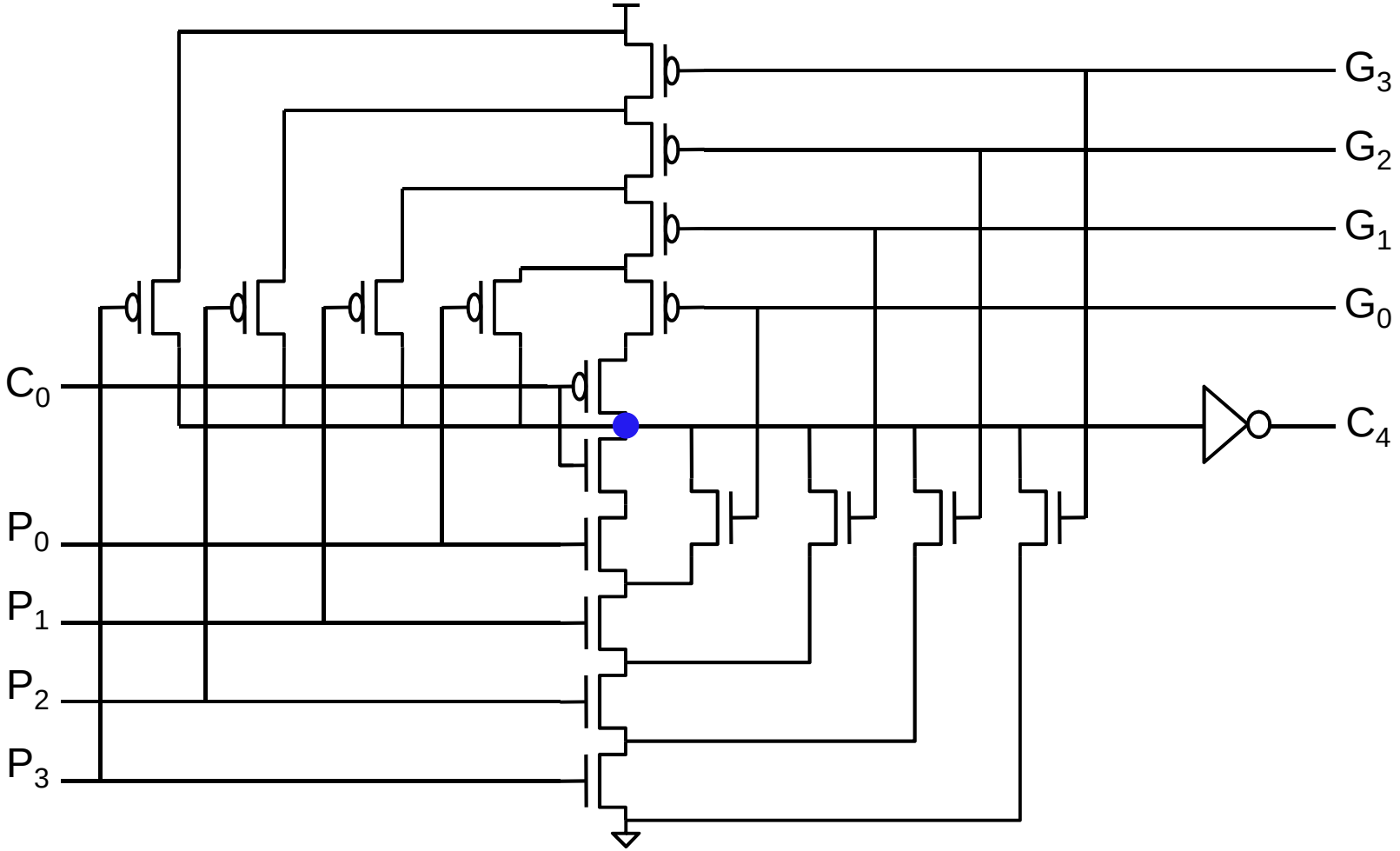
Datapath – *Carry Look-Ahead Adder*

CLG1



Datapath – *Carry Look-Ahead Adder*

CLG4



$C_4=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

Datapath – *Carry Look-Ahead Adder*

Manchester Carry Chain

$$C_{i+1}=G_i+P_iC_i$$

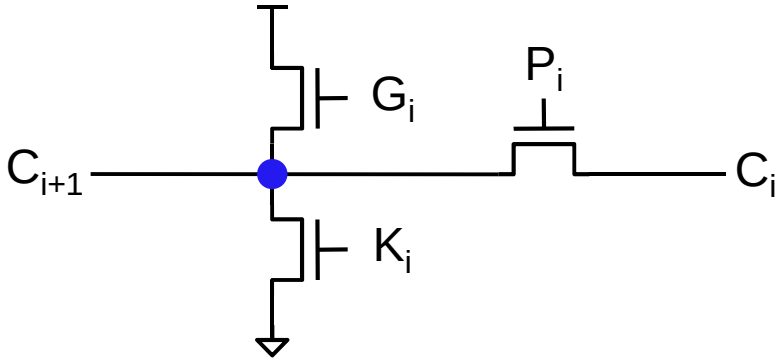
$$G_i=A_i.B_i$$

$$P_i=A_i+B_i$$

Introduce the carry-kill bit K_i, this term gets its name from the fact that if $K_i=1$, then $P_i=0$ and $G_i=0$, so that $C_{i+1}=0$; $K_i=1$ thus "kills" the carry-out bit.

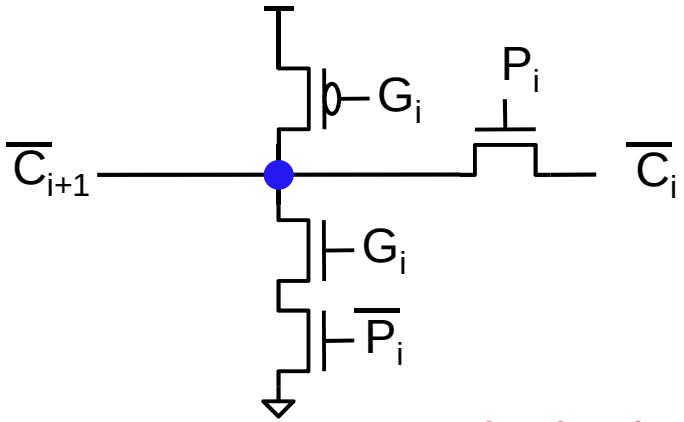
$$K_i=\overline{A}_i.\overline{B}_i$$

A_i	B_i	P_i	G_i	K_i
0	0	0	0	1
0	1	1	0	0
1	0	1	0	0
1	1	0	1	0

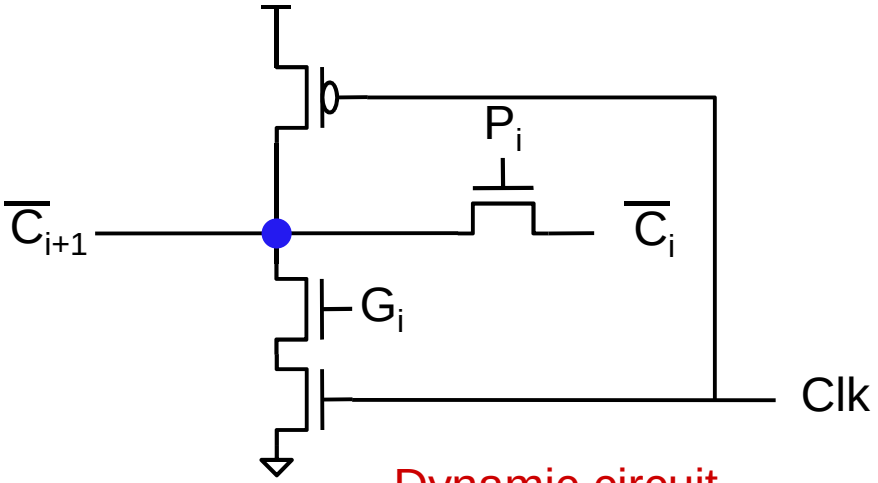


Datapath – *Carry Look-Ahead Adder*

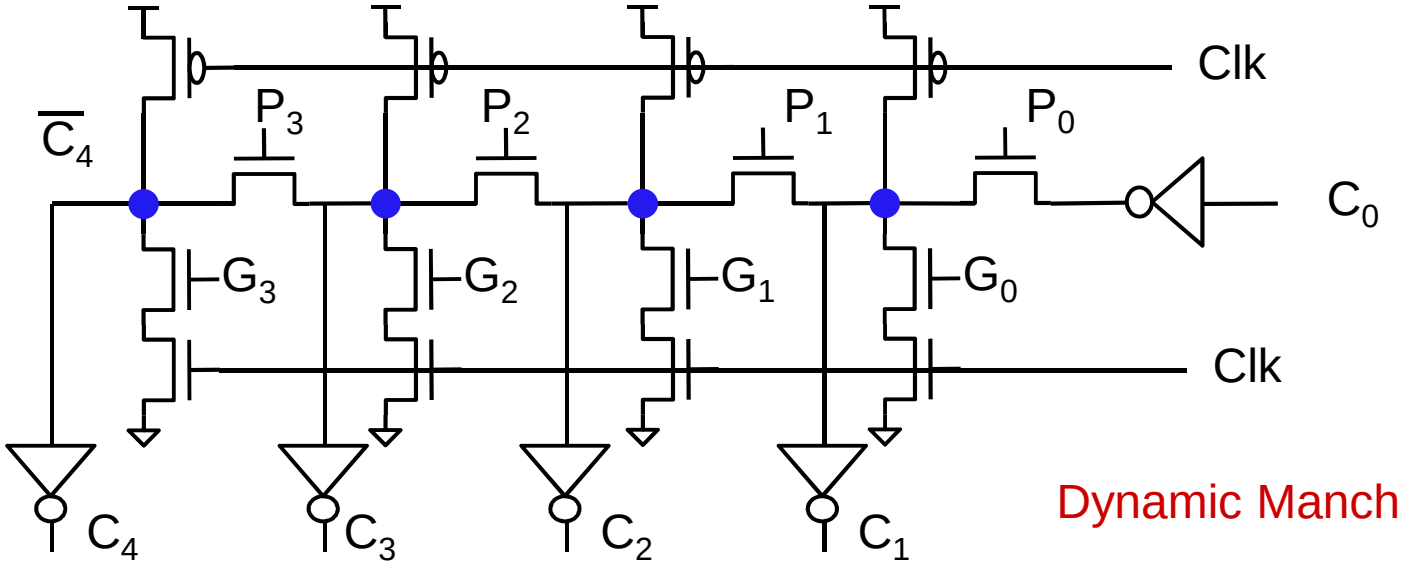
Manchester circuit styles



Static circuit



Dynamic circuit



Dynamic Manchester chain

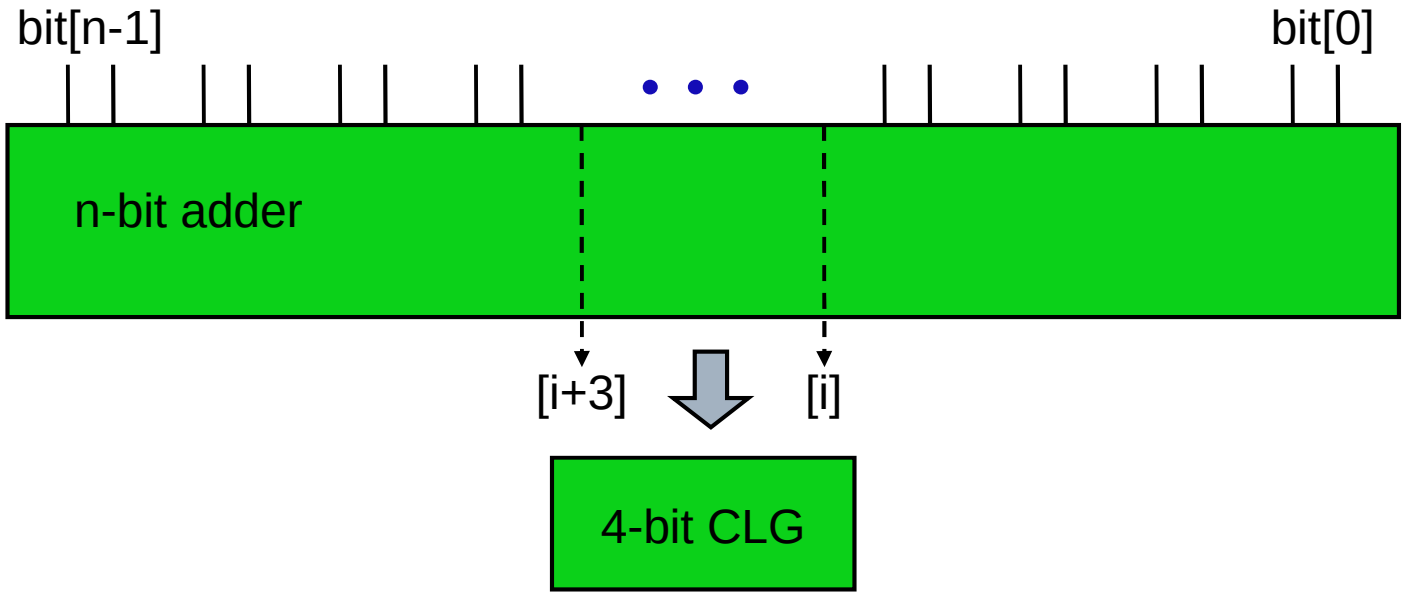
Datapath – *Carry Look-Ahead Adder*

Extension to wide adders

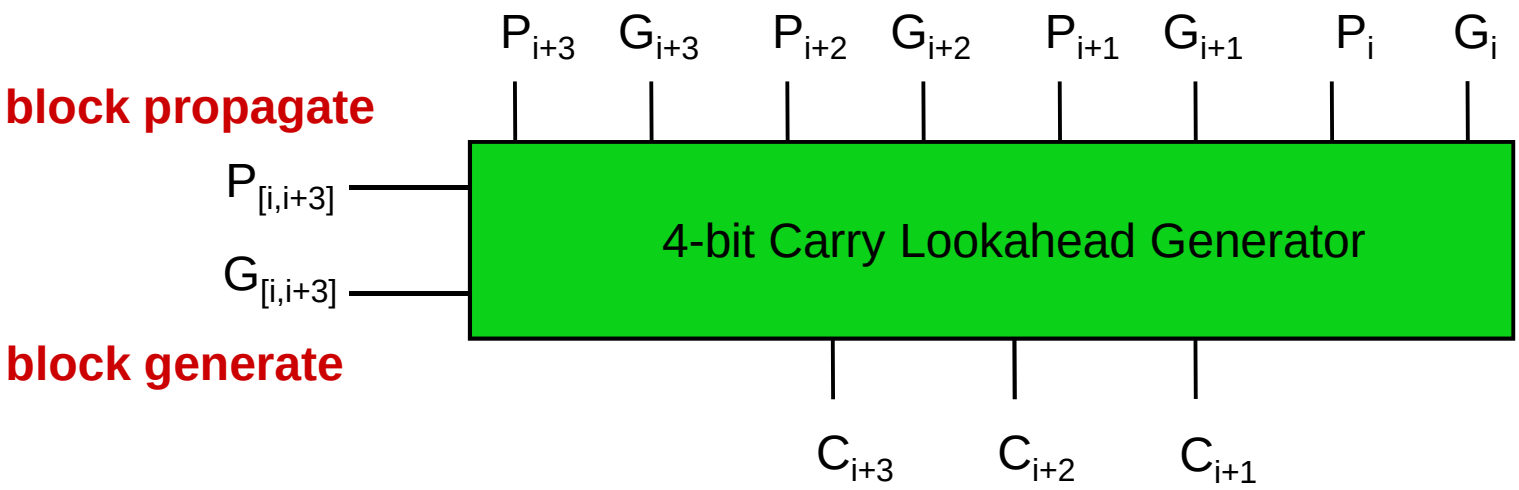
If we use a brute-force approach for an 8-bit design, then the carry-out bit C_8 would have a term of the form

$$P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 C_0$$

Multilevel CLA networks can improve this problem



Datapath – *Carry Look-Ahead Adder*

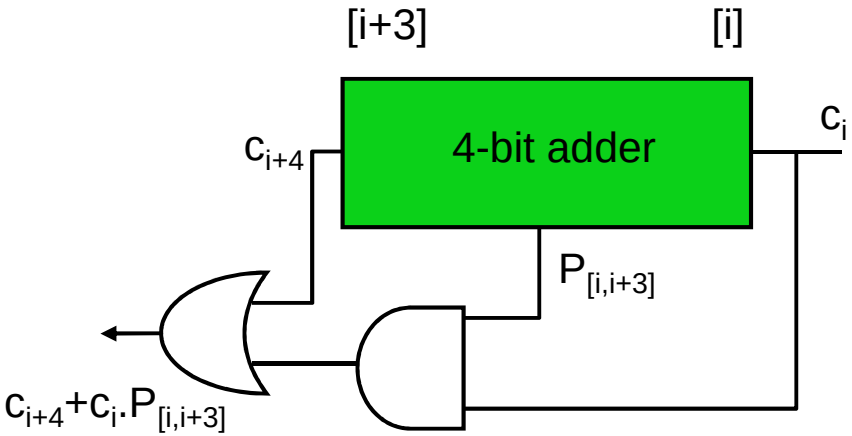


$$G_{[i,i+3]}=G_{i+3}+P_{i+3}G_{i+2}+P_{i+3}P_{i+2}G_{i+1}+P_{i+3}P_{i+2}P_{i+1}G_i$$

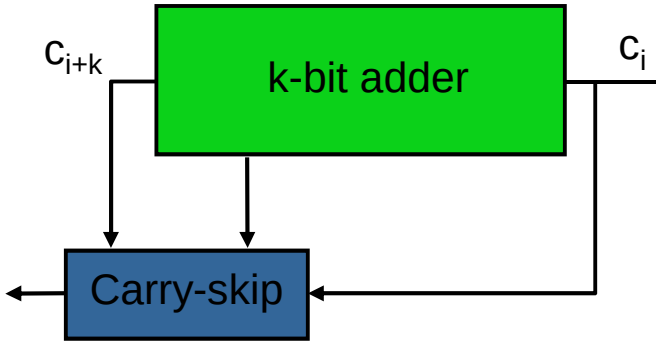
$$P_{[i,i+3]}=P_{i+3}P_{i+2}P_{i+1}P_i$$

Datapath – *Carry-Skip Adder*

A carry-skip adder is designed to speed up a wide adder by aiding the propagation of a carry bit around a portion of the entire adder.



Carry-skip logic

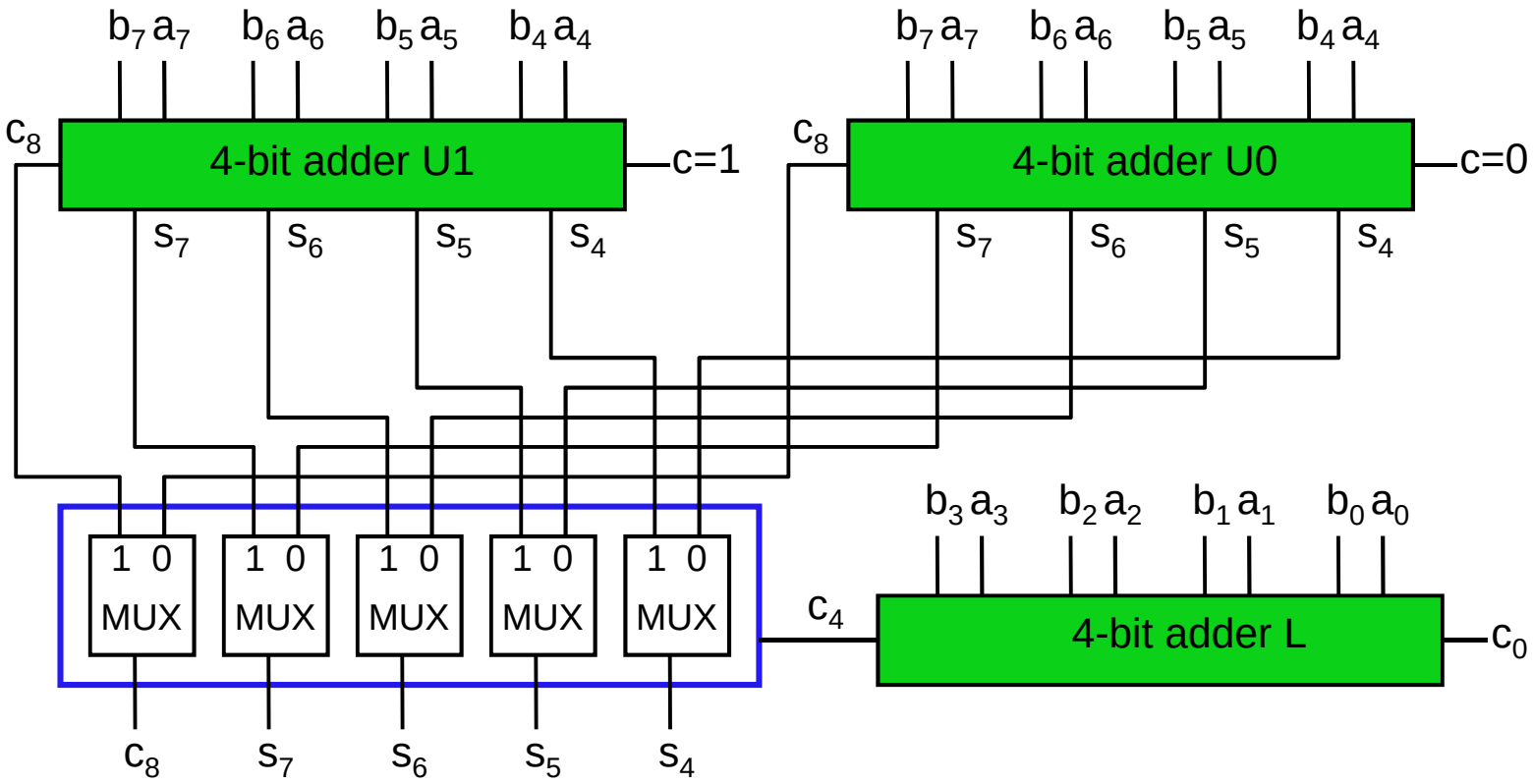


Generalization

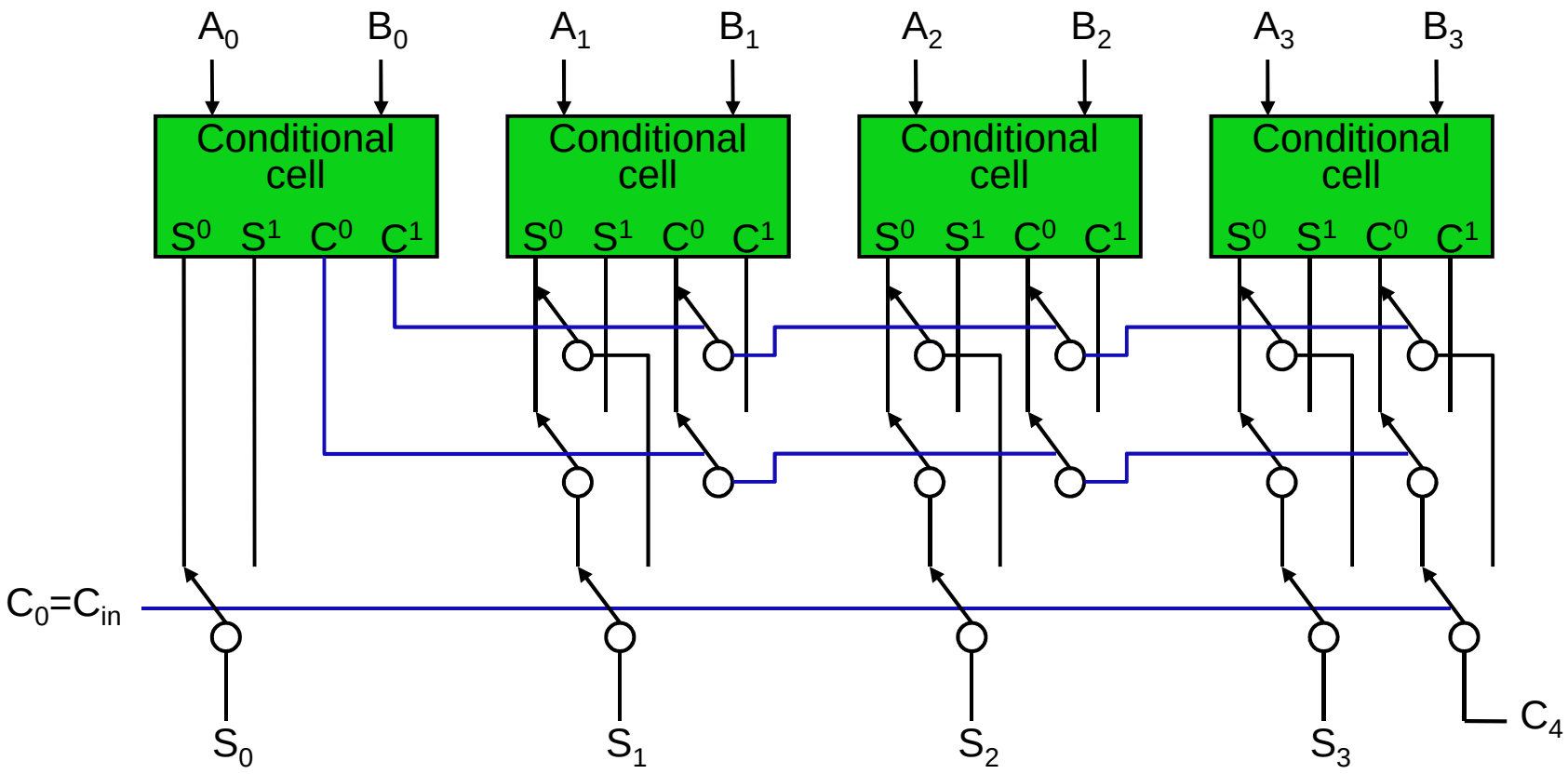
$$P_{[i,i+3]}=P_{i+3}P_{i+2}P_{i+1}P_i$$

$$Carry=C_{i+4}+P_{[i,i+3]}C_i$$

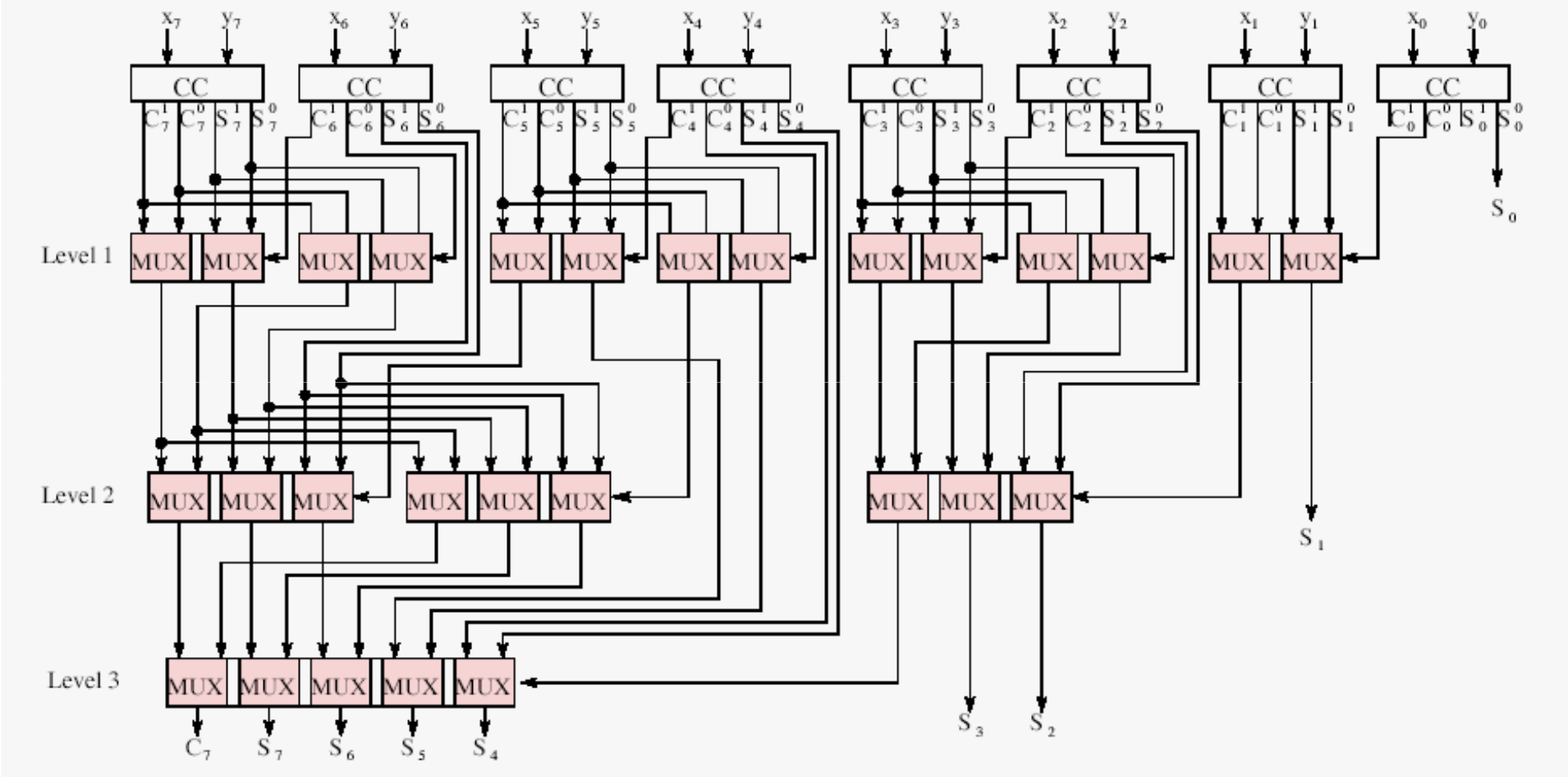
Datapath – *Carry-Select Adder*



Datapath – *Conditional-Sum Adder*



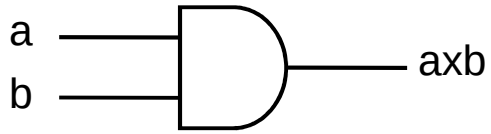
Datapath – *8-bit Conditional-Sum Adder*



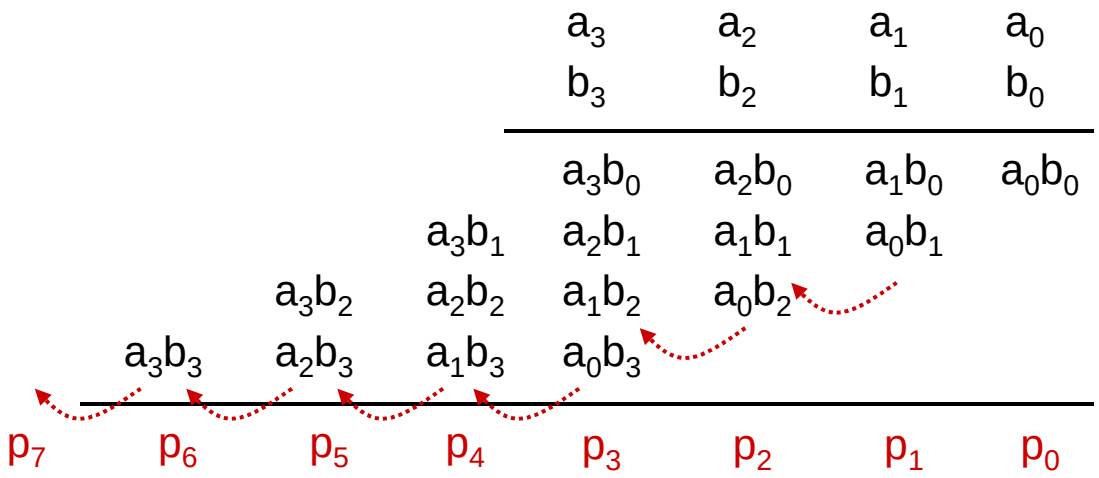
Datapath – *Multipliers*

Bit-level multiplier

a	b	axb
0	0	0
0	1	0
1	0	0
1	1	1



Multiplication of two 4-bit words



Datapath – *Multipliers*

The product axb is given by the 8-bit result

$p=p_7p_6p_5p_4p_3p_2p_1p_0$

The ith product term p_i can be expressed as

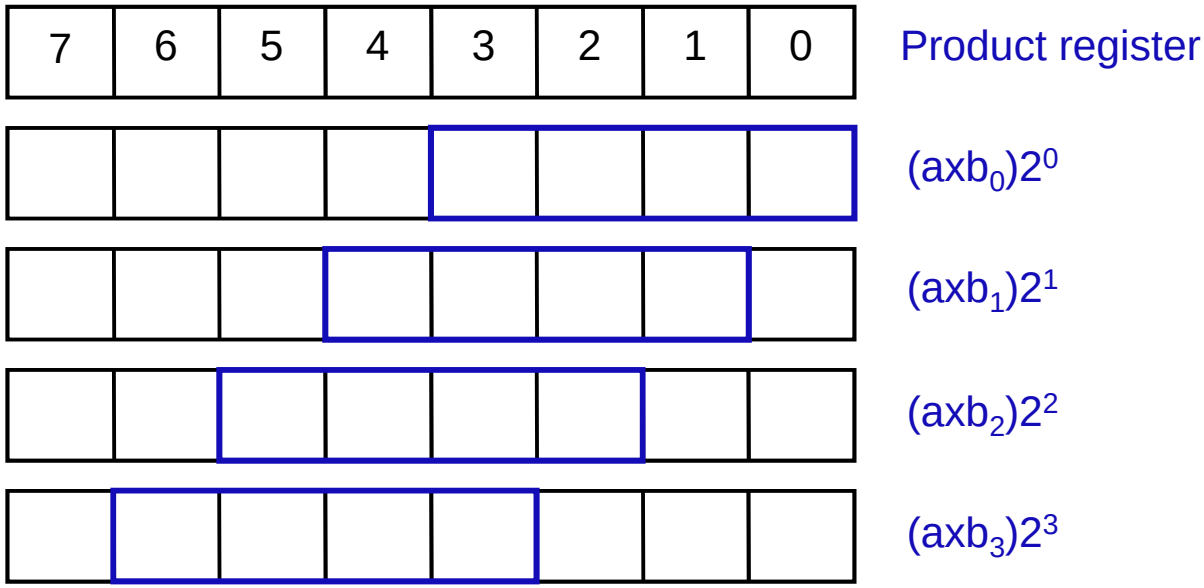
$$p_i = \sum_{i=j+k} a_j b_k + c_{i-1}$$

Alternate view of multiplication process

				a_3	a_2	a_1	a_0	
				b_3	b_2	b_1	b_0	
				$(a_3$	a_2	a_1	$a_0)xb_0$	$(axb_0)2^0$
			$(a_3$	a_2	a_1	$a_0)xb_1$		$(axb_1)2^1$
		$(a_3$	a_2	a_1	$a_0)xb_2$			$(axb_2)2^2$
	$(a_3$	a_2	a_1	$a_0)xb_3$				$(axb_3)2^3$
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

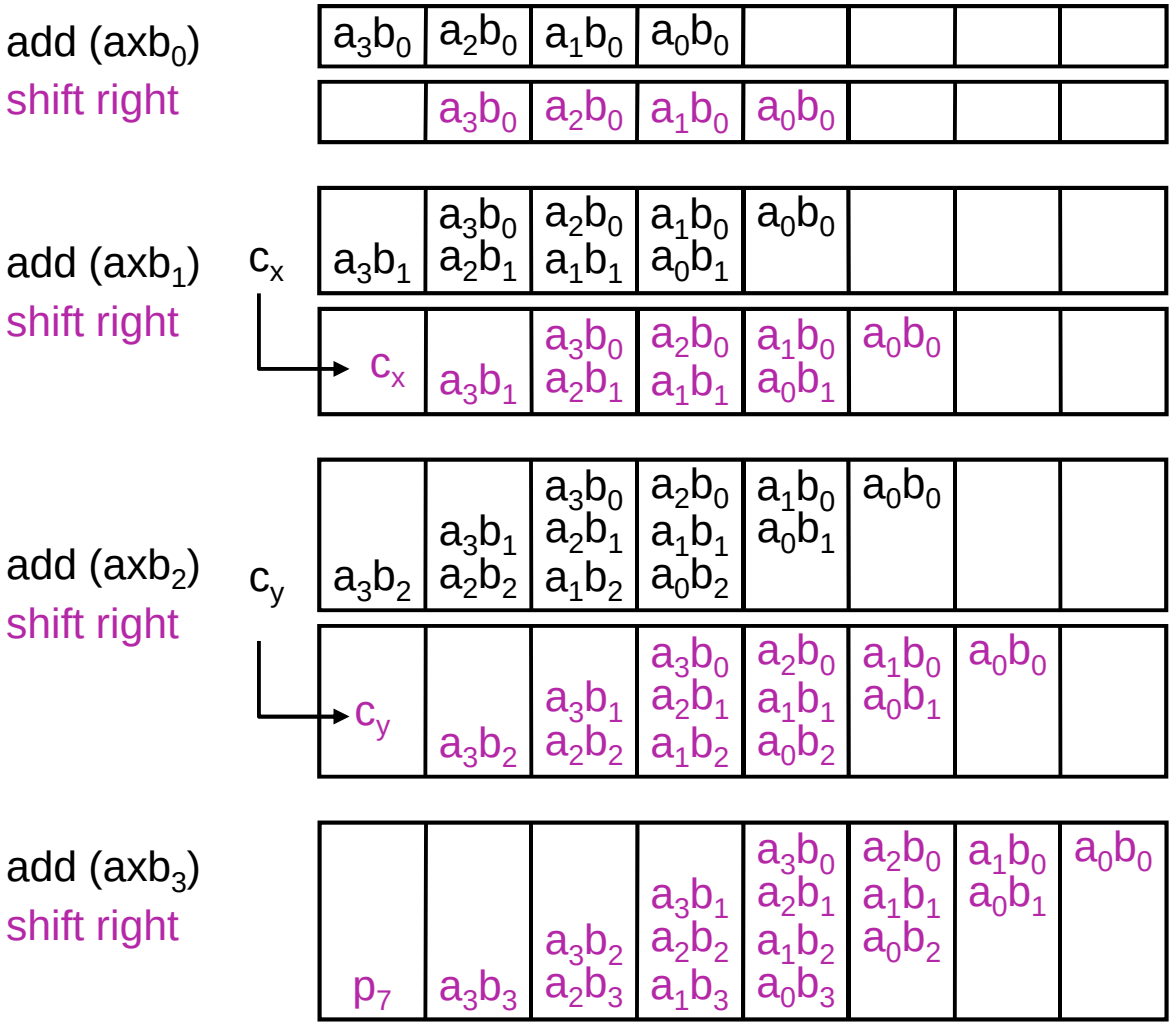
Datapath – *Multipliers*

Using a product register for multiplication

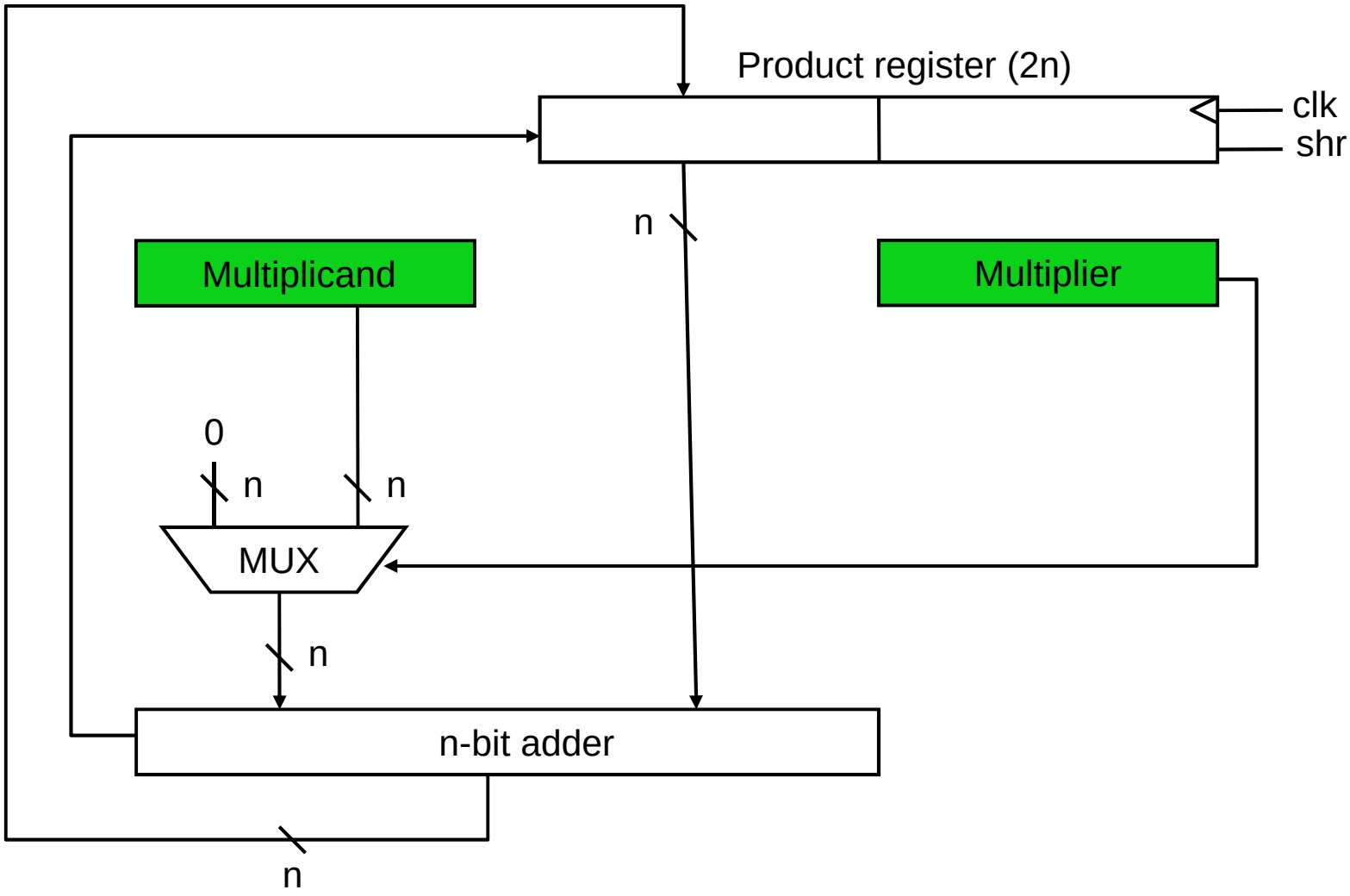


Datapath – *Multipliers*

Shift-right multiplication sequence



Datapath – *Register-Based Multiplier*



Datapath – *Array Multipliers*

- Consider two unsigned binary integers X and Y

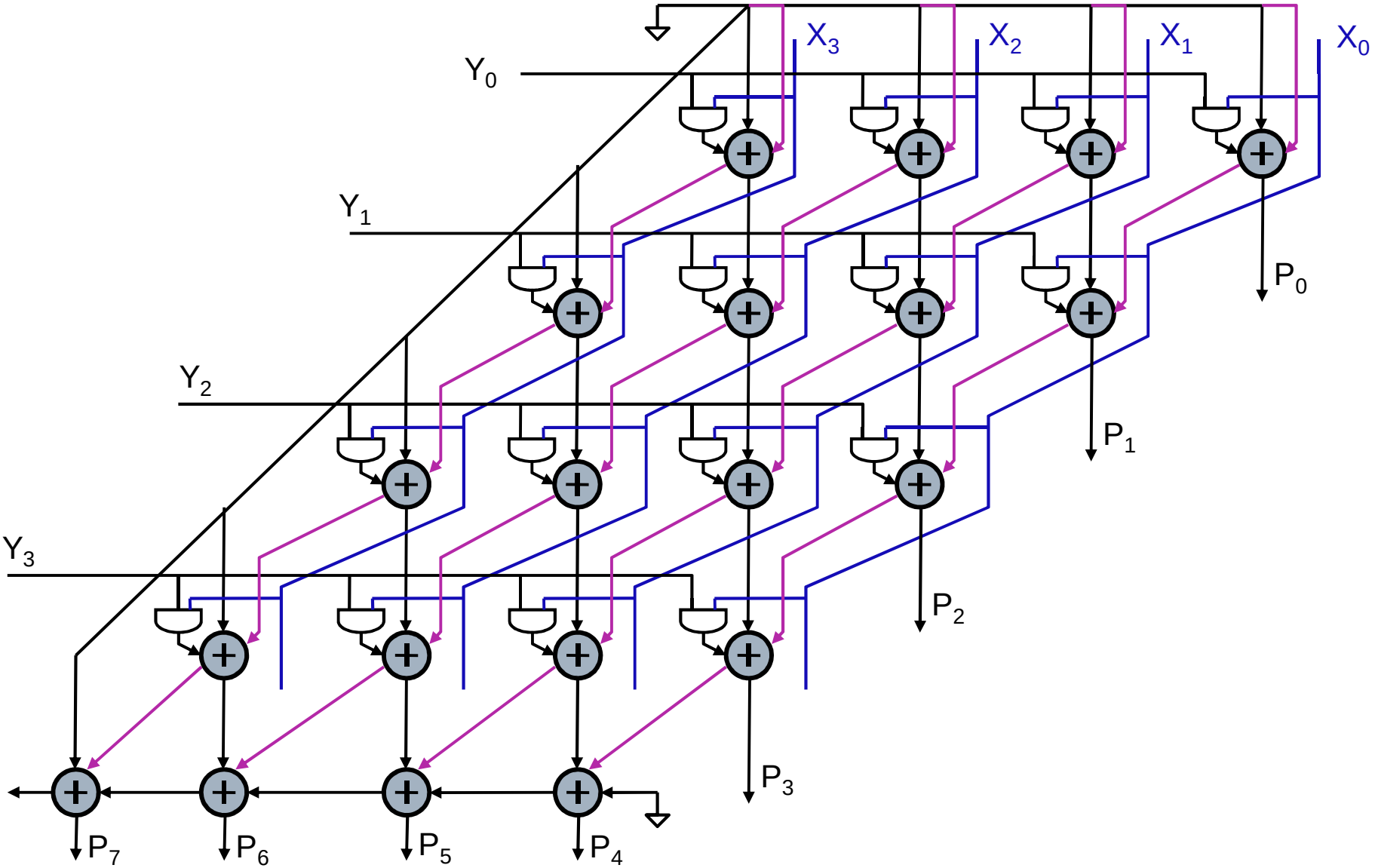
$$X = \sum_{i=0}^{n-1} X_i 2^i \qquad Y = \sum_{j=0}^{n-1} Y_j 2^j$$

$$P = X \times Y = \sum_{i=0}^{n-1} X_i 2^i \cdot \sum_{j=0}^{n-1} Y_j 2^j$$

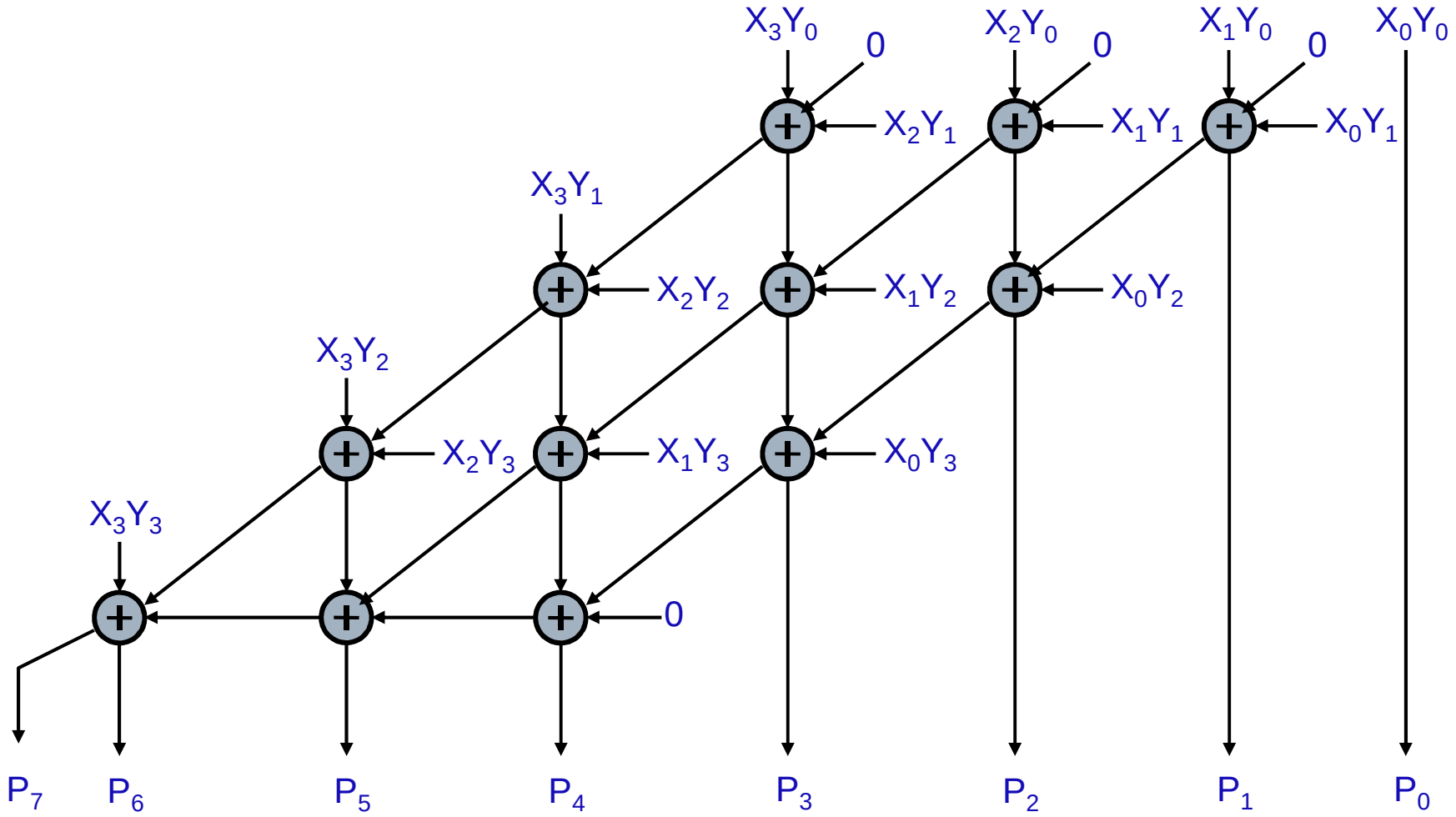
$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (X_i Y_j) 2^{i+j}$$

$$= \sum_{k=0}^{n+n-1} P_k 2^k$$

Datapath – *Array Multipliers*



Datapath – *Array Multipliers*



Datapath – *Booth Multiplier*

- Booth's algorithm takes advantages of the fact that an adder-substractor is nearly as fast and small as a simple adder
- Consider the two's complement representation of the multiplier y
  - $y = -2^n y_n + 2^{n-1} y_{n-1} + 2^{n-2} y_{n-2} + \cdots$
- The representation can be rewritten as
  - $y = 2^n (y_{n-1} - y_n) + 2^{n-1} (y_{n-2} - y_{n-1}) + 2^{n-2} (y_{n-3} - y_{n-2}) + \cdots$
- Extract the first two terms
  - $y = 2^n (y_{n-1} - y_n) + 2^{n-1} (y_{n-2} - y_{n-1})$
  - The right-hand term can be used to add x to partial product
  - The left-hand term add 2x

Datapath – *Booth Multiplier*

Actions during Booth multiplication

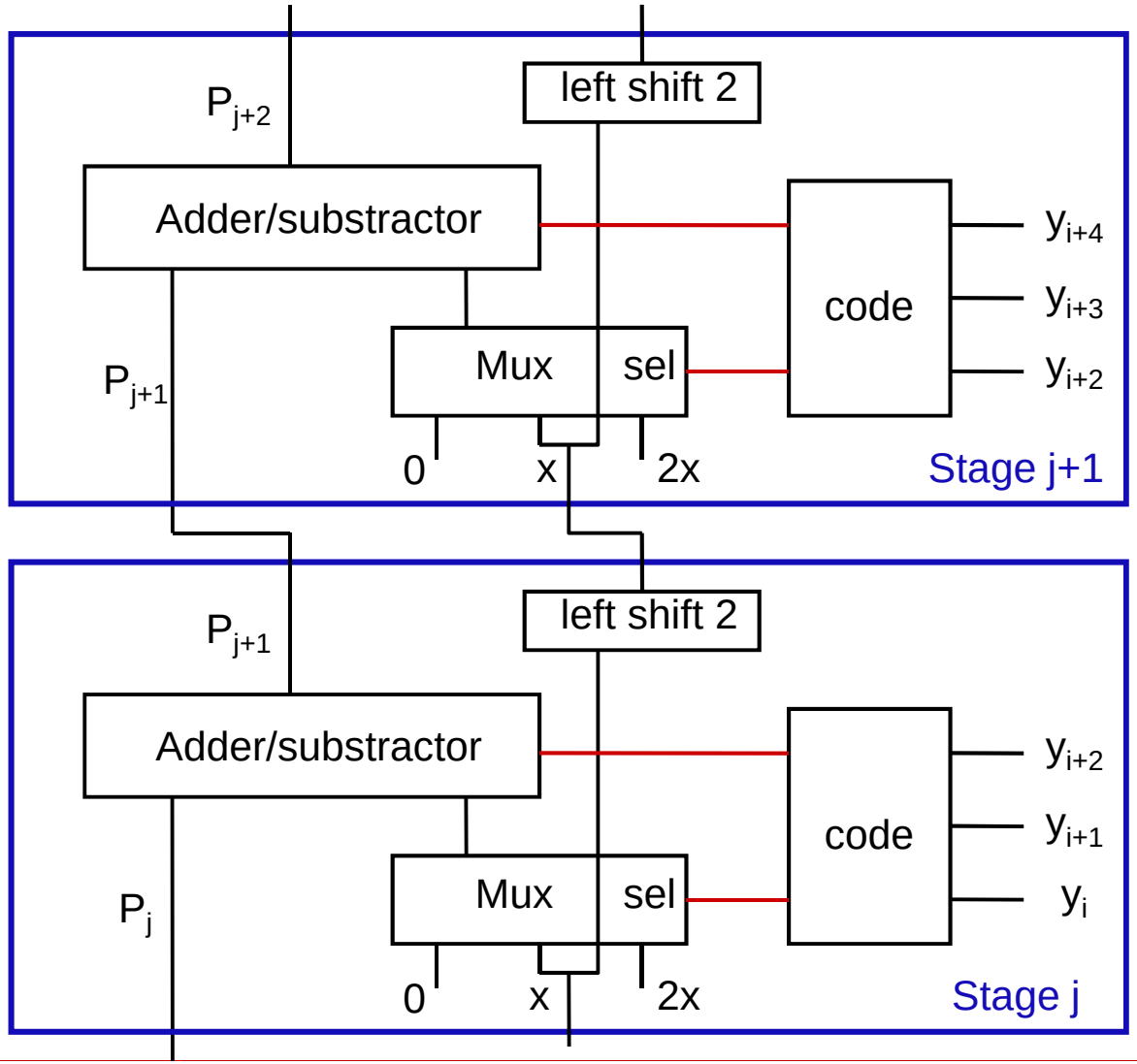
$y_i\ y_{i-1}\ y_{i-2}$	Operation
0 0 0	Add 0
0 0 1	Add x
0 1 0	Add x
0 1 1	Add 2x
1 0 0	Sub 2x
1 0 1	Sub x
1 1 0	Sub x
1 1 1	Add 0

For example, x=011001 (25_{10}), y=101110 (-18_{10})

1. $y_1y_0y_{-1}$=100, so $P_1=P_0$-2x.1=11111001110
2. $y_3y_2y_1$=111, so $P_2=P_1$+0.4=11111001110
3. $y_5y_4y_3$=101, so $P_3=P_2$-x.16=11000111110

Datapath – *Booth Multiplier*

Structure of a Booth multiplier



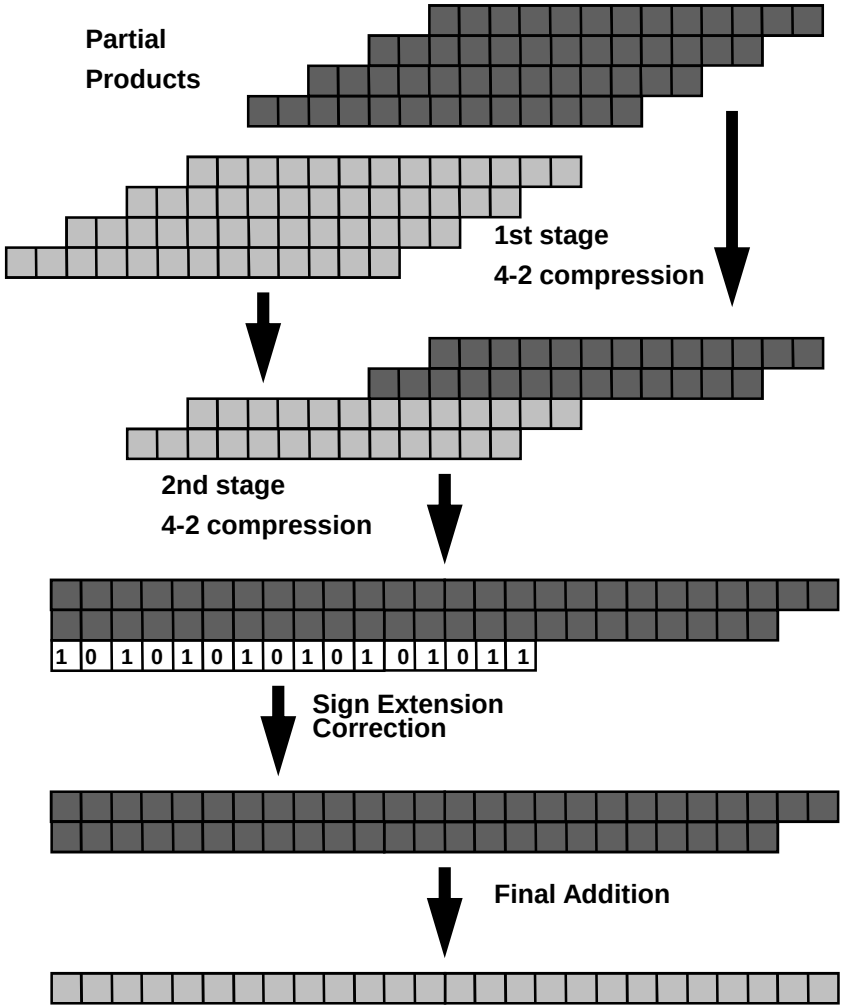
Datapath – *Wallace Tree Multiplier*

- A Wallace tree is a full adder tree structured specially for a quick addition of the partial products
- Example
  - A 16x16 Booth multiplier
  - 8 partial products are generated
  - Assume that all partial products are negative so all sign extension bits are 1's
  - Sign extension correction vector is 1010101010101011

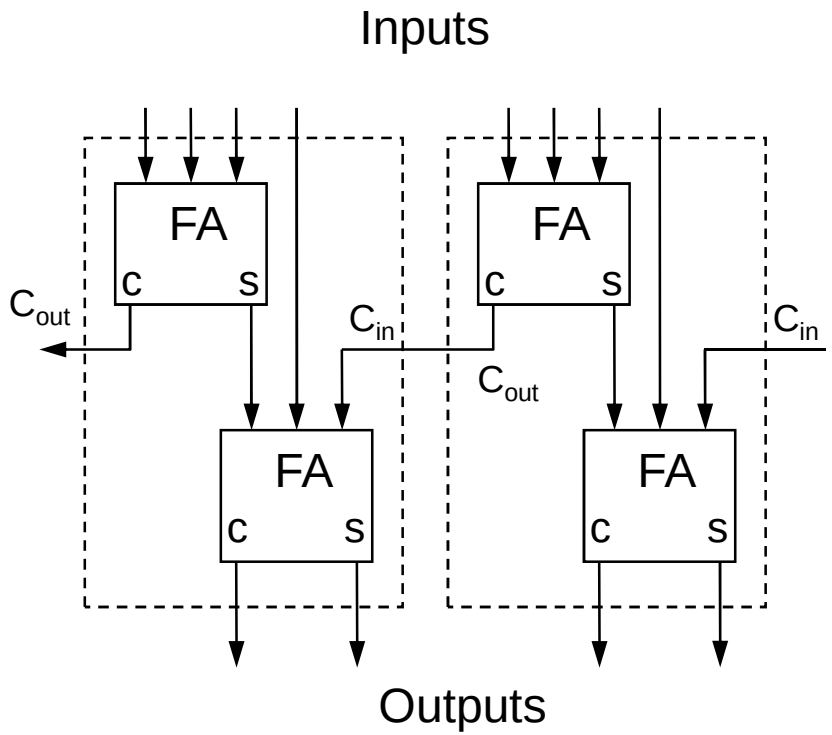
```
1111111111111111
 11111111111111
  111111111111
   1111111111
    11111111
     111111
      1111
       11
-----------------
1010101010101011
```

Datapath – *Wallace Tree Multiplier*

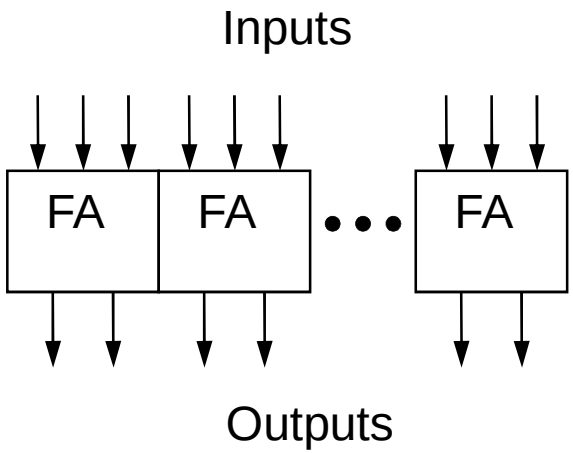
Wallace tree multiplication



Datapath – *Wallace Tree Multiplier*



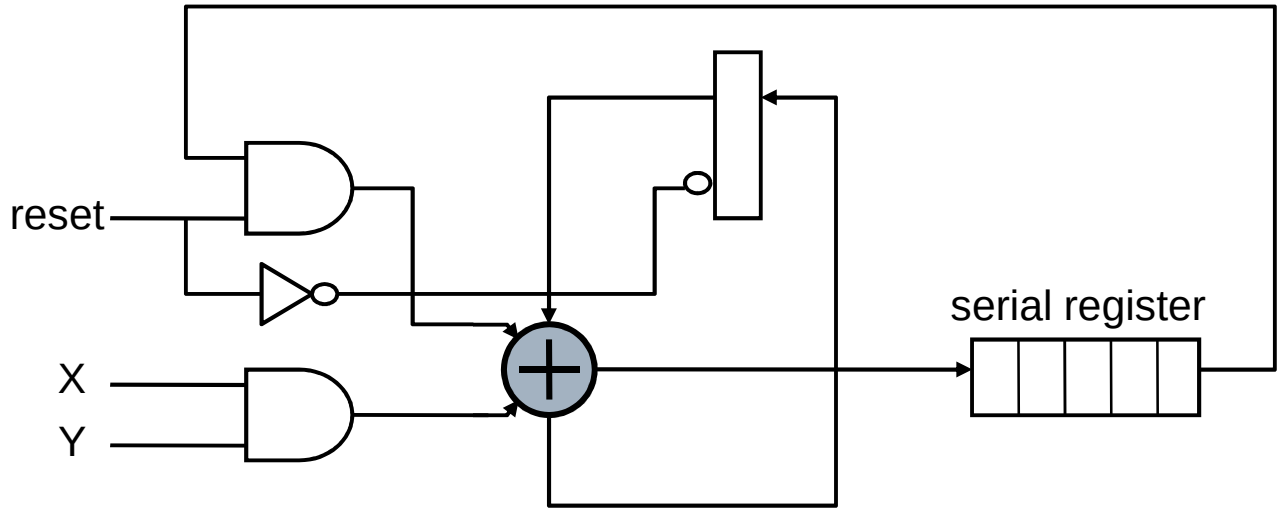
4-2 compressor



Carry-save adder

Datapath – *Serial Multiplication*

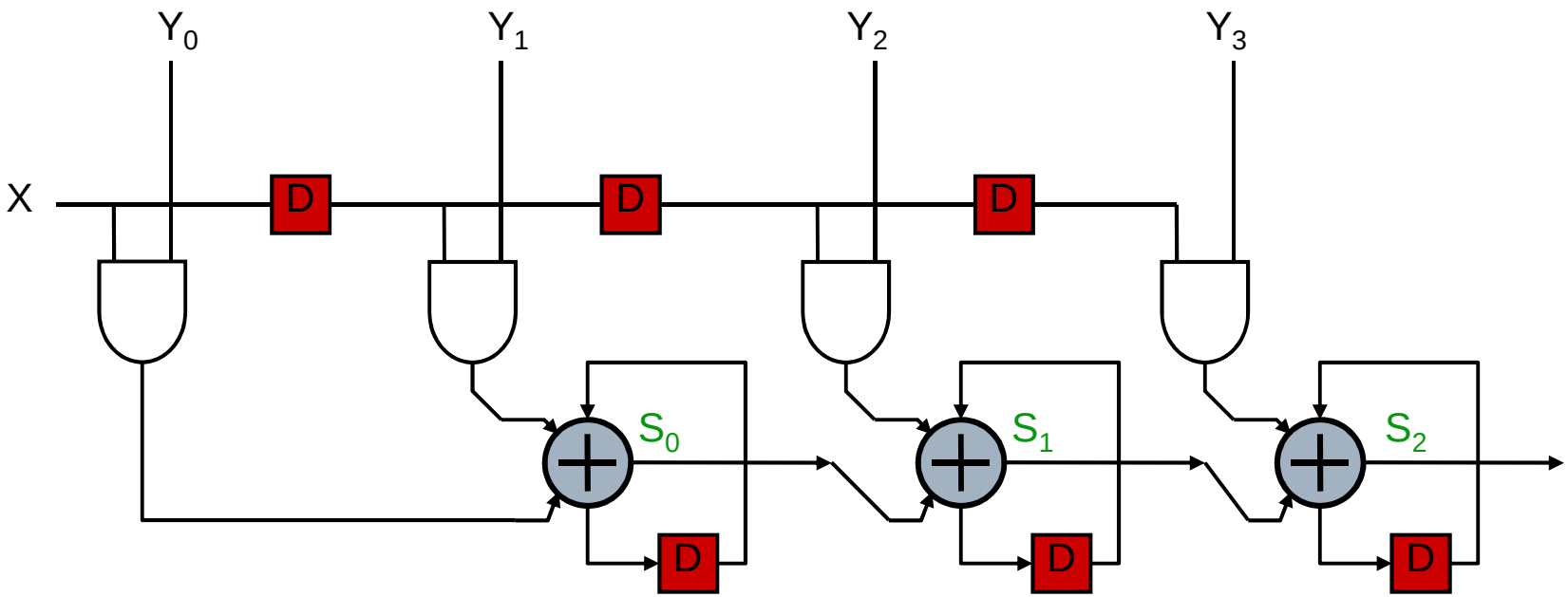
Serial multiplier



1. Require MN clock cycles to produce a product for an N-bit multiplier and a M-bit multiplicand

Datapath – *Serial Multiplication*

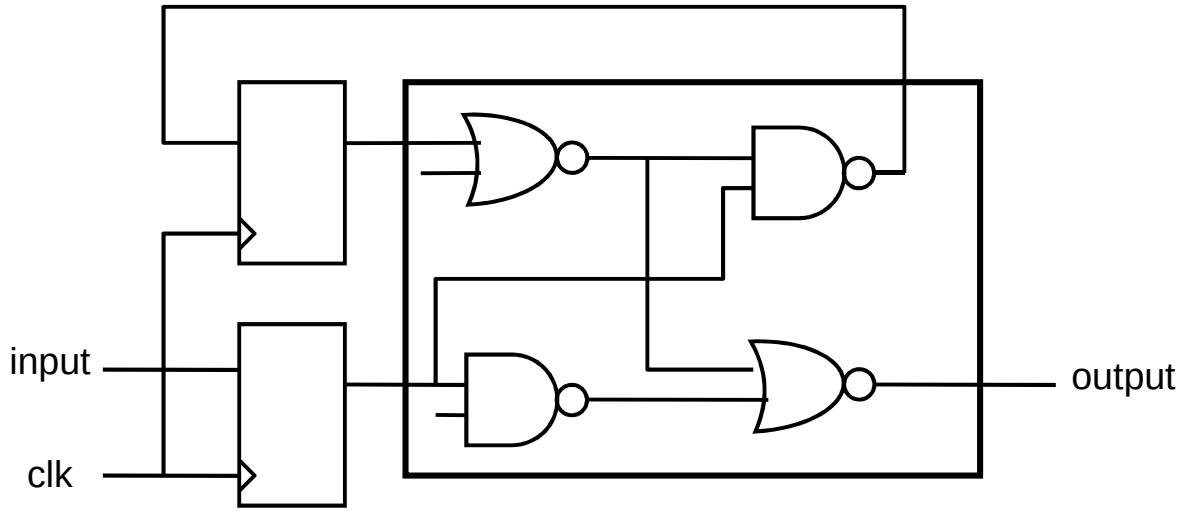
Serial/parallel multiplier



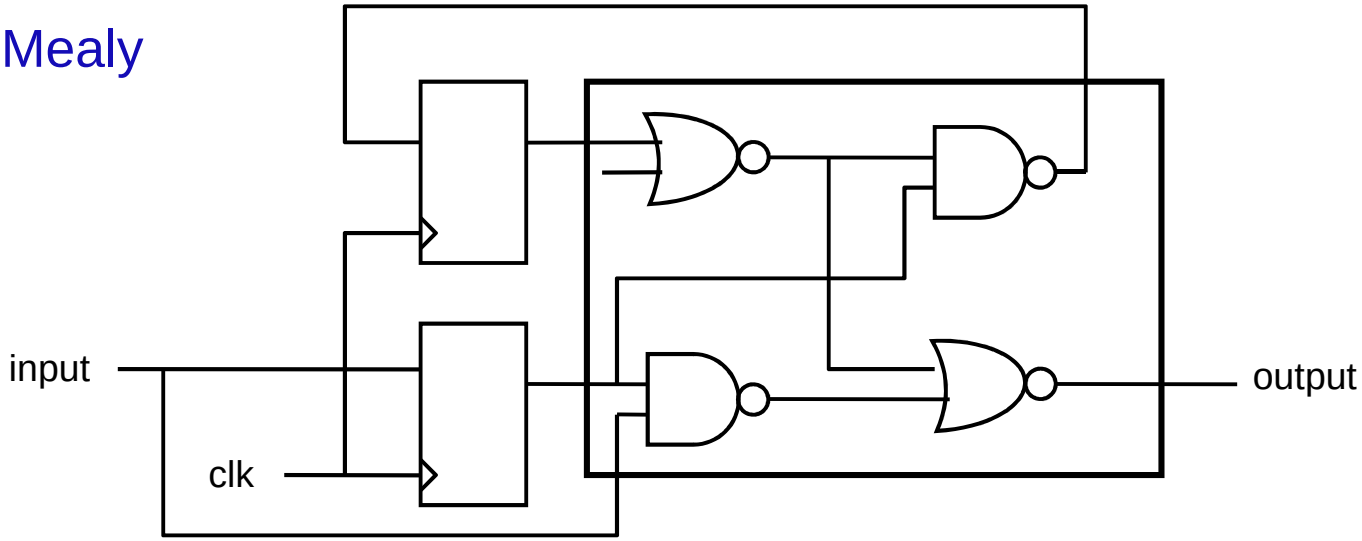
1. Require M+N clock cycles to produce a product for an N-bit multiplier and a M-bit multiplicand
2. The critical path consists of the adders

Control – *FSM*

Moore

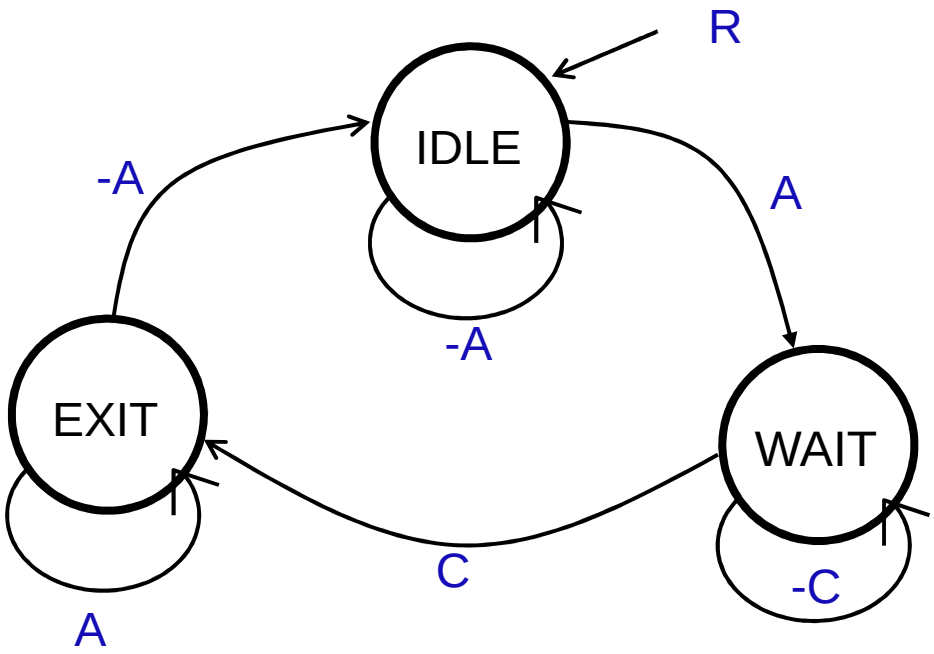


Mealy



Control – *FSM*

- FSM design procedure
  - Draw the state-transition diagram
  - Check the state diagram
  - Write state equations (Write HDL)
- An example of state-transition diagram



IDLE: (S1,S0)=(00)

WAIT: (S1,S0)=(01)

EXIT: (S1,S0)=(10)

A: car-in

C: change-ok

R: rst

Control – *FSM*

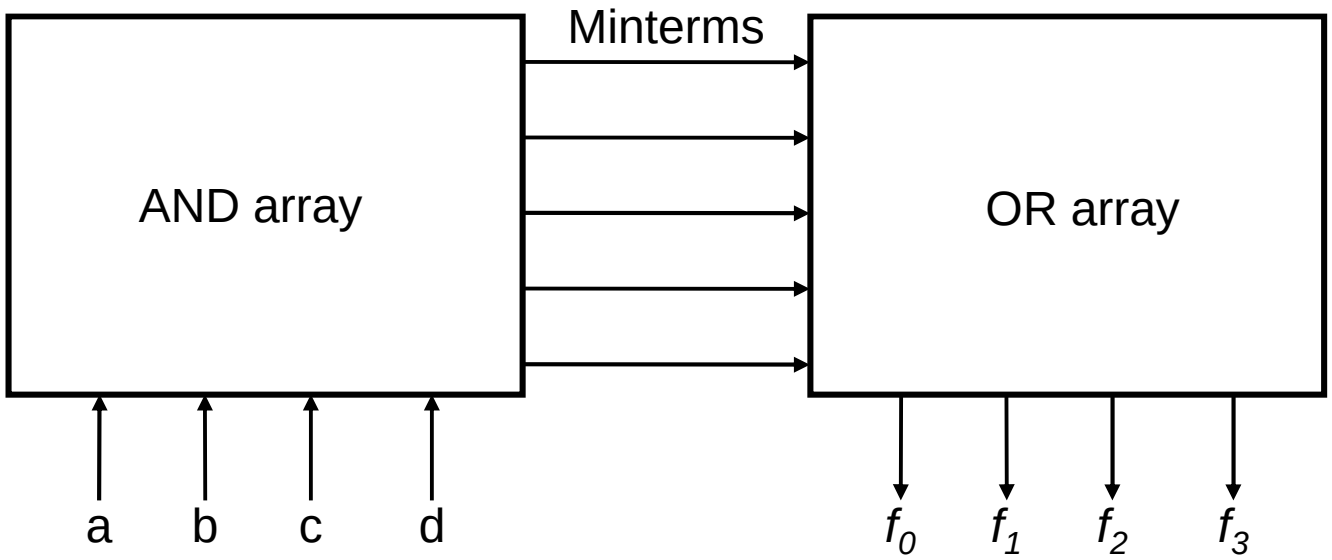
- Check the state-transition diagram
  - Ensure all states are represented, including the IDLE state
  - Check that the OR of all transitions leaving a state is TRUE. This is a simple method of determining that there is a way out of a state once entered.
  - Verify that the pairwise XOR of all exit transitions is TRUE. This ensures that there are not conflicting conditions that would lead to more than one exit-transition becoming active at any time.
  - Insert loops into any state if it is not guaranteed to otherwise change on each cycle.
- Formal FSM verification method
  - Perform conformance checking

Control – *Verilog Coding Style for FSMs*

```
module
      toll_booth(clk,rst,car_in,change_ok,green);
input          clk,rst,car_in,change_ok;
output         green;
reg[1:0]       state_reg, next_state;
parameter IDLE  = 2'b00;
parameter WAIT = 2'b01;
parameter EXIT  = 2'b11;
always @(posedge clk or posedge rst) begin
      If (rst==1'b1) state_reg<=IDLE;
      else state_reg<=next_state;
end
always @(state_reg or car_in or change_ok)
       begin
      case(state_reg):
            IDLE: if (car_in==1'1) begin
                        next_state=WAIT;
                        green=1'b0;
                     end else begin
                        next_state=IDEL;
                     end
             WAIT: if (change==1'b1) begin
                         next_state=EXIT;
                         green=1'b1;
                       end else begin
                          next_state=WAIT;
                          green=1'b0;
                       end
            EXIT: if (car_in==1'1) begin
                        next_state=EXIT;
                        green=1'b1;
                      end else begin
                        next_state=IDEL;
                         green=1'b0;
                      end
          default: begin
                         next_state=IDLE;
                         green=1'b0;
                       end
      endcase
end
endmodule
```

Control – *PLA*

Structure of a PLA



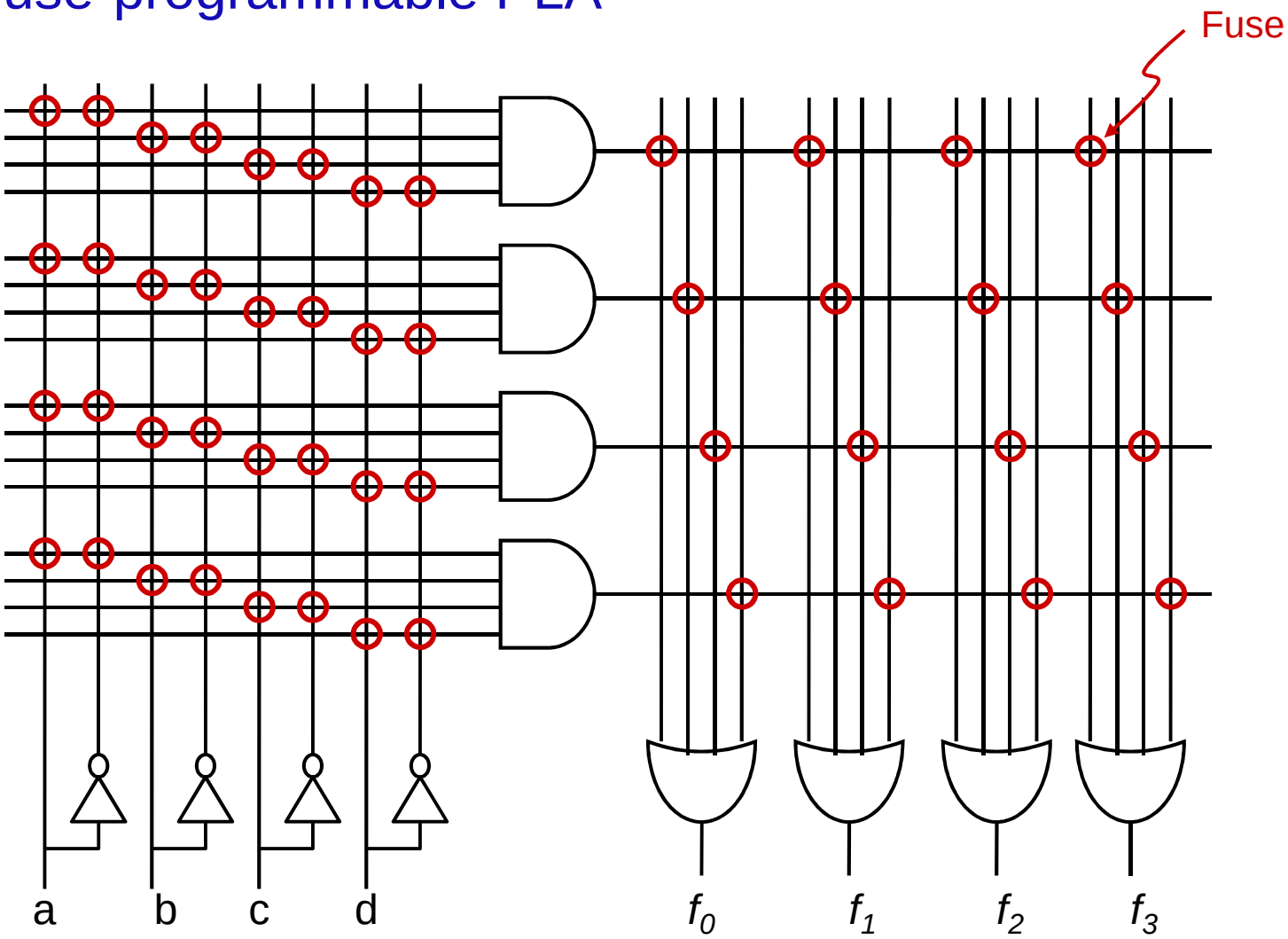
A PLA represents an expression of sum-of-product (SOP)

$$f_i = \sum_i m_i(a,b,c,d)$$

$$f_1 = \bar{a}\cdot\bar{b}\cdot c\cdot\bar{d} + \bar{a}\cdot b\cdot c\cdot\bar{d} + a\cdot b\cdot c\cdot d$$

Control – *PLA*

Fuse-programmable PLA



Control – *PLA*

Logic gate diagram of a PLA