# CS31001 COMPUTER ORGANIZATION AND ARCHITECTURE

Debdeep Mukhopadhyay,
CSE, IIT Kharagpur

# Datapath Elements and Their Designs

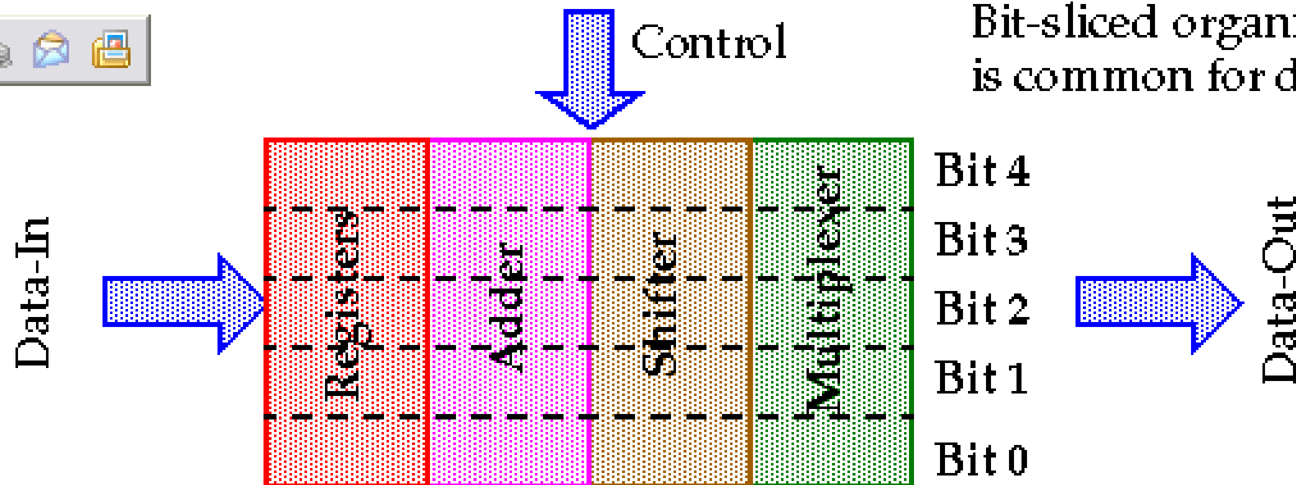
# Why Datapaths?

- *The speed of these elements often dominates the overall system performance so optimization techniques are important.*
- *However, as we will see, the task is non-trivial since there are multiple equivalent logic and circuit topologies to choose from, each with adv./disadv. in terms of speed, power and area.*
- *Datapath elements include shifters, adders, multipliers, etc.*

# Bit-slicing method of constructing ALU

- **Bit slicing** is a technique for constructing a processor from modules of smaller bit width.
- Each of these components processes one bit field or "slice" of an operand.
- The grouped processing components would then have the capability to process the chosen full word-length of a particular software design.

# Bit slicing



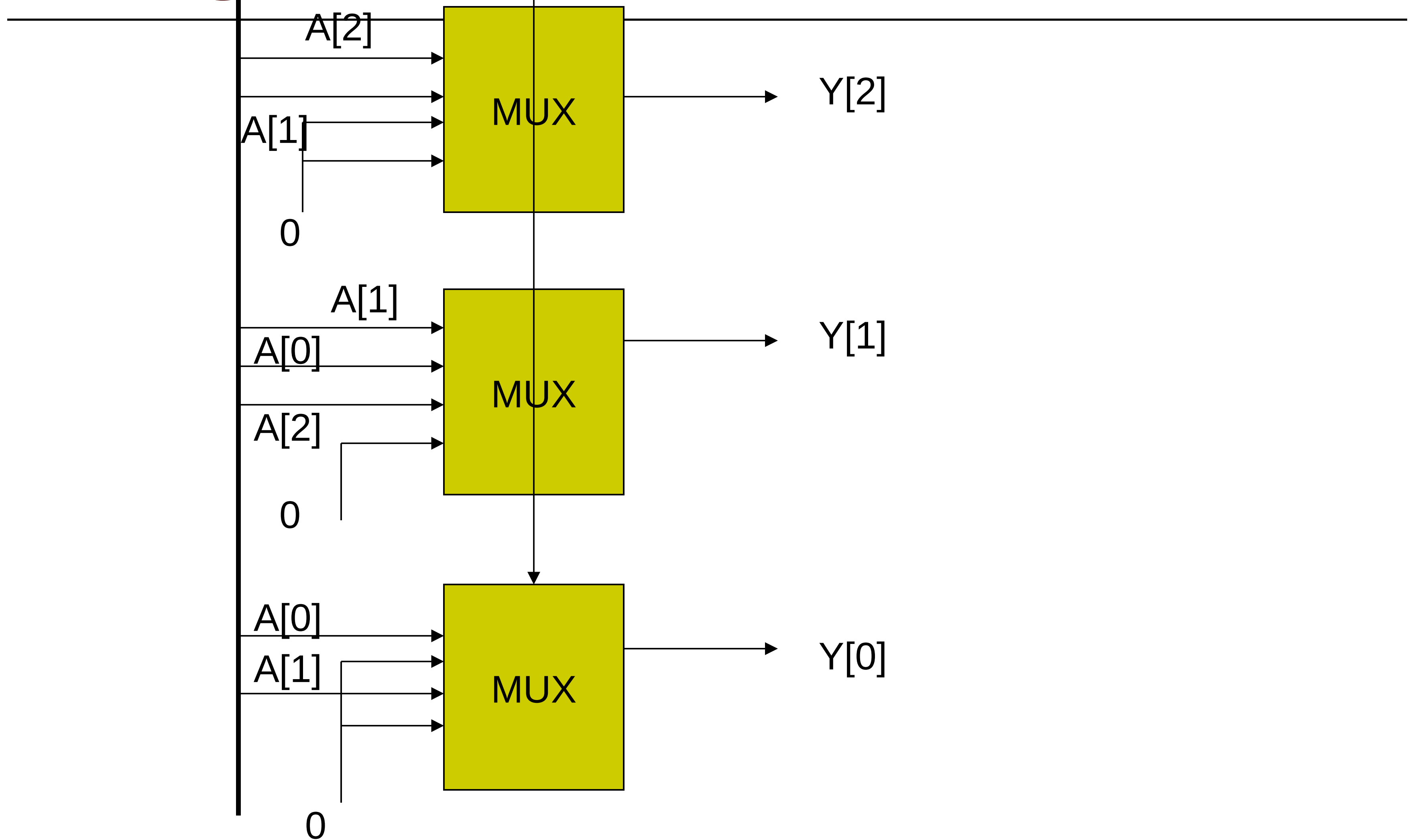

***How can we develop architectures which are bit sliced?***

# Shifters

| Sel1 | Sel0 | Operation | Function |
|---|---|---|---|
| 0 | 0 | Y<-A | No shift |
| 0 | 1 | Y<-shlA | Shift left |
| 1 | 0 | Y<-shrA | Shift right |
| 1 | 1 | Y<-0 | Zero outputs |

What would be a bit sliced architecture of this simple shifter?

# Using Muxes



# Verilog Code

```
module shifter(Con,A,Y);
        input [1:0] Con;
        input[2:0] A;
        output[2:0] Y;
        reg [2:0] Y;
        always @(A or Con)
        begin
             case(Con)
                0: Y=A;
                1: Y=A<<1;
                2: Y=A>>1;
                default: Y=3’b0;
             endcase
         end
endmodule
```

# Combinational logic shifters with shiftin and shiftout

| Sel | Operation | Function |
|---|---|---|
| 0 | Y<=A, ShiftLeftOut=0<br>ShiftRightOut=0 | No shift |
| 1 | Y<=shl(A),<br>ShiftLeftOut=A[5]<br>ShiftRightOut=0 | Shift left |
| 2 | Y<=shr(A),<br>ShiftLeftOut=0<br>ShiftRightOut=A[0] | Shift Right |
| 3 | Y<=0, ShiftLeftOut=0<br>ShiftRightOut=0 | Zero Outputs |

# Verilog Code

```
always@(Sel or A or ShiftLeftIn or ShiftRightIn);
begin
  A_wide={ShiftLeftIn,A,ShiftRightIn};
 case(Sel)
    0: Y_wide=A_wide;
    1: Y_wide=A_wide<<1;
    2: Y_wide=A_wide>>1;
    3:Y_wide=5'b0;
    default: Y_wide=A_wide;
 endcase
ShiftLeftOut=Y_wide[0];
Y=Y_wide[2:0];
ShiftRightOut=Y_wide[4];
end
```

# Combinational 6 bit Barrel Shifter

| Sel | Operation | Function |
|---|---|---|
| 0 | Y<=A | No shift |
| 1 | Y<-A rol 1 | Rotate once |
| 2 | Y<-A rol 2 | Rotate twice |
| 3 | Y<- A rol 3 | Rotate Thrice |
| 4 | Y<-A rol 4 | Rotate four times |
| 5 | Y<-A rol 5 | Rotate five times |

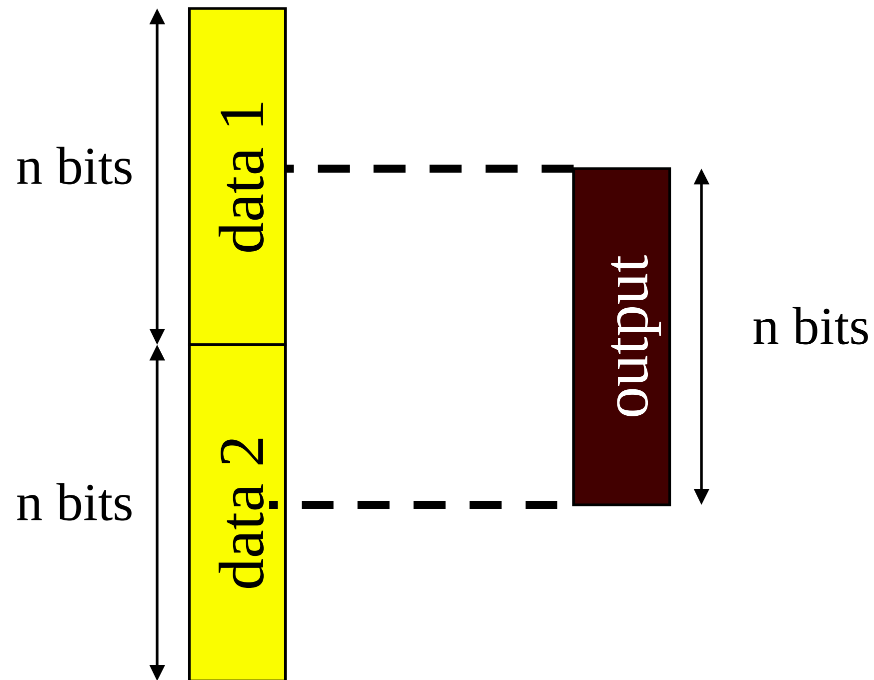
# Verilog Coding

```
function [2:0] rotate_left;
 input [5:0] A;
 input [2:0] NumberShifts;
 reg [5:0] Shifting;
 integer N;
 begin
   Shifting = A;
   for(N=1;N<=NumberShifts;N=N+1)
     begin
       Shifting={Shifting[4:0],Shifting[5]};
     end
   rotate_left=Shifting;
 end
endfunction
```

# Verilog

```
always @(Rotate or A)
begin
  case(Rotate)
    0: Y=A;
    1: Y=rotate_left(A,1);
    2: Y=rotate_left(A,2);
    3: Y=rotate_left(A,3);
    4: Y=rotate_left(A,4);
    5: Y=rotate_left(A,5);
    default: Y=6'bx;
  endcase
end
```

# Another Way

- 

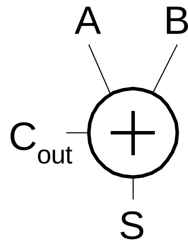

**Code is left as an exercise…**

# Single-Bit Addition

## Half Adder

$S =$

$C_{out} =$

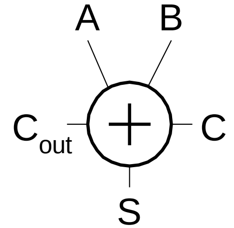



| A | B | $C_o$ | S |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

## Full Adder

$S =$

$C_{out} =$



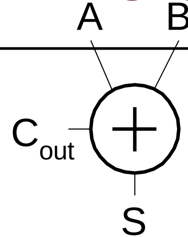

| A | B | C | $C_o$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

# Single-Bit Addition

## Half Adder

$$S = A \oplus B$$

$$C_{out} = A \cdot B$$

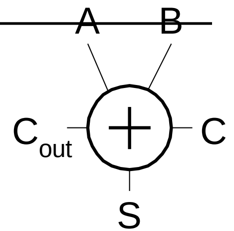



| A | B | $C_o$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

## Full Adder

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

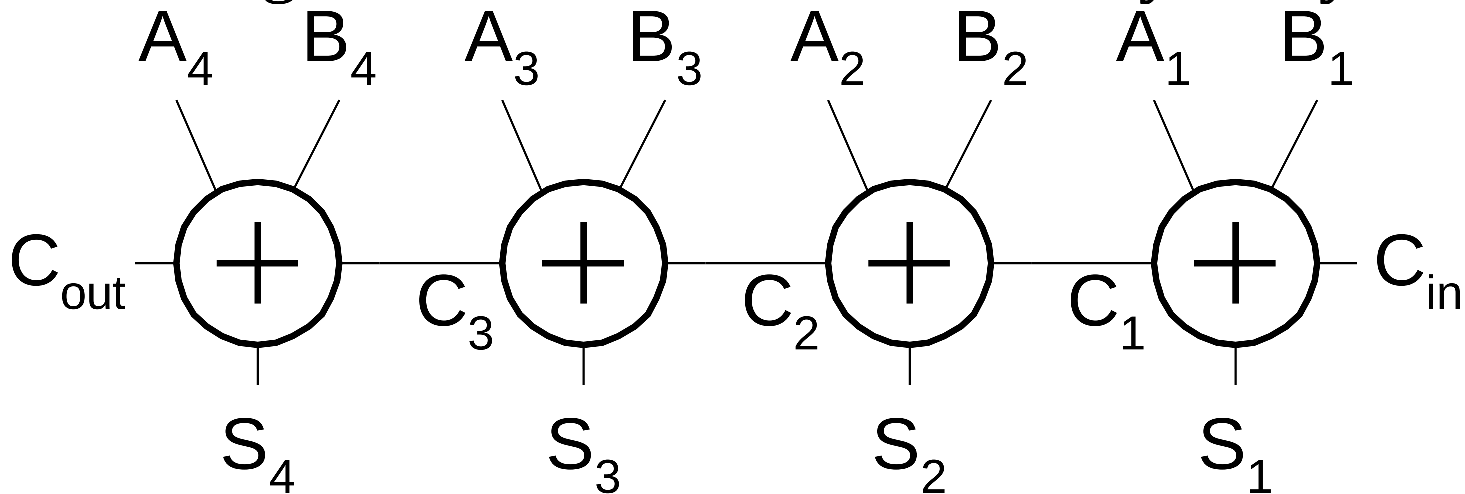
$$C_{out} = MAJ(A, B, C)$$




| A | B | C | $C_o$ | 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 |

# Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from Cin to Cout
  - Design full adder to have fast carry delay



# Full adder

- Computes one-bit sum, carry:
  - $s_i = a_i \text{ XOR } b_i \text{ XOR } c_i$
  - $c_{i+1} = a_i b_i + a_i c_i + b_i c_i$
- Half adder computes two-bit sum.
- Ripple-carry adder: n-bit adder built from full adders.
- Delay of ripple-carry adder goes through all carry bits.

# Verilog for full adder

```
module fulladd(a,b,carryin,sum,carryout);
    input a, b, carryin; /* add these bits*/
    output sum, carryout; /* results */

    assign {carryout, sum} = a + b + carryin;
                /*  compute the sum and carry */
    endmodule
```

# Verilog for ripple-carry adder

```
module nbitfulladd(a,b,carryin,sum,carryout)
        input [7:0] a, b; /* add these bits */
        input carryin; /* carry in*/
        output [7:0] sum; /* result */
        output carryout;
        wire [7:1] carry; /* transfers the carry between bits */

        fulladd a0(a[0],b[0],carryin,sum[0],carry[1]);
        fulladd a1(a[1],b[1],carry[1],sum[1],carry[2]);
…
        fulladd a7(a[7],b[7],carry[7],sum[7],carryout]);
endmodule
```

# Generate and Propagate

$$G[i] = A[i].B[i]$$
$$P[i] = A[i] \oplus B[i]$$
$$C[i] = G[i] + P[i].C[i-1]$$
$$S[i] = P[i] \oplus C[i-1]$$

$$G[i] = A[i].B[i]$$
$$P[i] = A[i] + B[i]$$
$$C[i] = G[i] + P[i].C[i-1]$$
$$S[i] = A[i] \oplus B[i] \oplus C[i-1]$$

Two methods to develop C[i] and S[i].

# Both are correct

- Because, A[i]=1 and B[i]=1 (which may lead to a difference is taken care of by the term A[i]B[i])
- How do we make an n bit adder?
- The delay of the adder chain needs to be optimized.

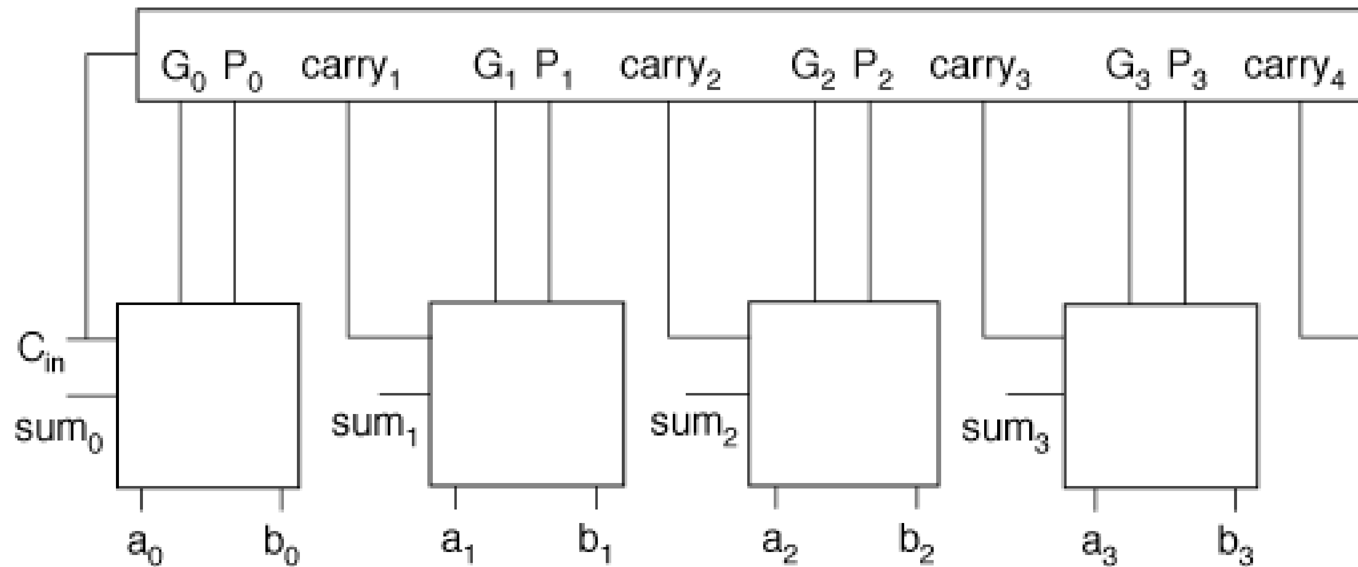
# Carry-lookahead adder

- First compute carry propagate, generate:
  - $P_i = a_i + b_i$
  - $G_i = a_i b_i$
- Compute sum and carry from P and G:
  - $s_i = c_i$ XOR $P_i$ XOR $G_i$
  - $c_{i+1} = G_i + P_i c_i$

# Carry-lookahead expansion

- Can recursively expand carry formula:
  - $c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}c_{i-1})$
  - $c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}(G_{i-2} + P_{i-1}c_{i-2})$
- Expanded formula does not depend on intermediate carries.
- Allows carry for each bit to be computed independently.

# Depth-4 carry-lookahead



# Analysis

- As we look ahead further logic becomes complicated.
- Takes longer to compute
- Becomes less regular.
- There is no similarity of logic structure in each cell.
- We have developed CLA adders, like Brent-Kung adder.

# Verilog for carry-lookahead carry block

```
module carry_block(a,b,carryin,carry);
        input [3:0] a, b; /* add these bits*/
        input carryin; /* carry into the block */
        output [3:0] carry; /* carries for each bit in the block */
        wire [3:0] g, p; /* generate and propagate */

        assign g[0] = a[0] & b[0]; /* generate 0 */
        assign p[0] = a[0] ^ b[0]; /* propagate 0 */
        assign g[1] = a[1] & b[1]; /* generate 1 */
        assign p[1] = a[1] ^ b[1]; /* propagate 1 */
…
        assign carry[0] = g[0] | (p[0] & carryin);
        assign carry[1] = g[1] | p[1] & (g[0] | (p[0] & carryin));
        assign carry[2] = g[2] | p[2] &
                (g[1] | p[1] & (g[0] | (p[0] & carryin)));
        assign carry[3] = g[3] | p[3] &
                (g[2] | p[2] & (g[1] | p[1] & (g[0] | (p[0] & carryin))));

        endmodule
```

ci+1 = Gi + Pi(Gi-1 + Pi-1ci-1)

# Verilog for carry-lookahead sum unit

```
module sum(a,b,carryin,result);
    input a, b, carryin; /* add these bits*/
    output result; /* sum */

    assign result = a ^ b ^ carryin;
                /*  compute the sum  */
endmodule
```

# Verilog for carry-lookahead adder

```
module carry_lookahead_adder(a,b,carryin,sum,carryout);
input [15:0] a, b; /* add these together */
input carryin;
output [15:0] sum; /* result */
output carryout;
wire [16:1] carry; /* intermediate carries */

assign carryout = carry[16]; /* for simplicity */
/* build the carry-lookahead units */
carry_block b0(a[3:0],b[3:0],carryin,carry[4:1]);
carry_block b1(a[7:4],b[7:4],carry[4],carry[8:5]);
carry_block b2(a[11:8],b[11:8],carry[8],carry[12:9]);
carry_block b3(a[15:12],b[15:12],carry[12],carry[16:13]);
/* build the sum */
sum  a0(a[0],b[0],carryin,sum[0]);
sum  a1(a[1],b[1],carry[1],sum[1]);
…
sum a15(a[15],b[15],carry[15],sum[15]);
endmodule
```

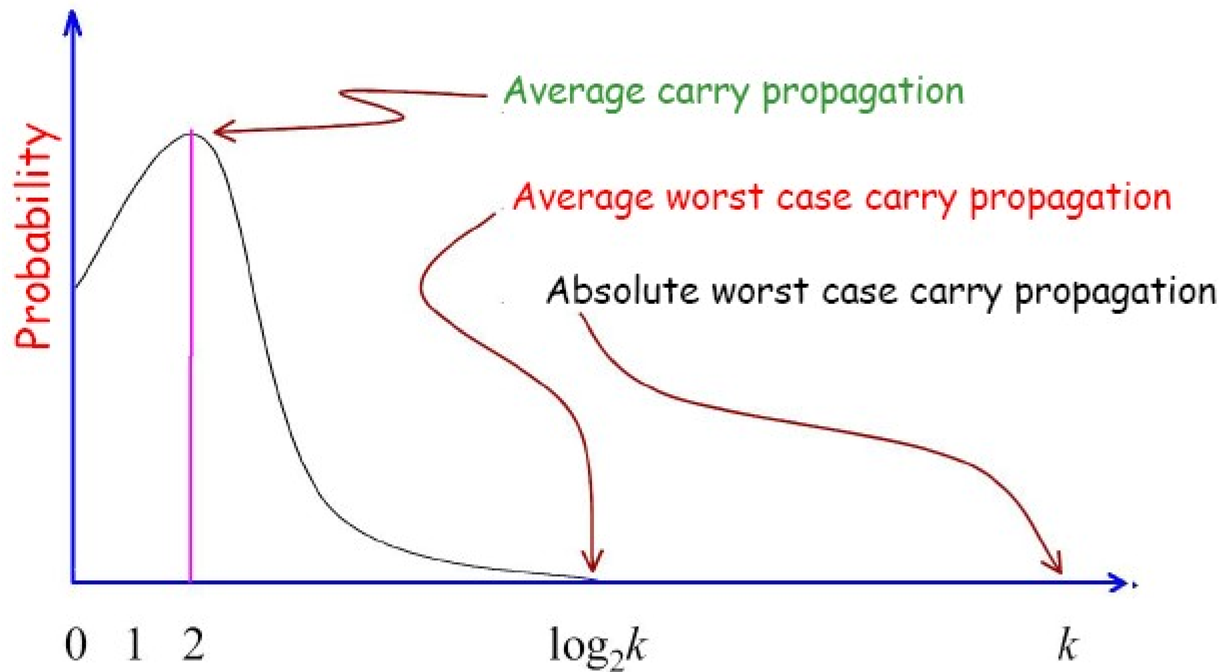
# Dealing with the problem of carry propagation

1. Reduce the carry propagation time.

2. To detect the completion of the carry propagation time.

*We have seen some ways to do the former. How do we do the second one?*

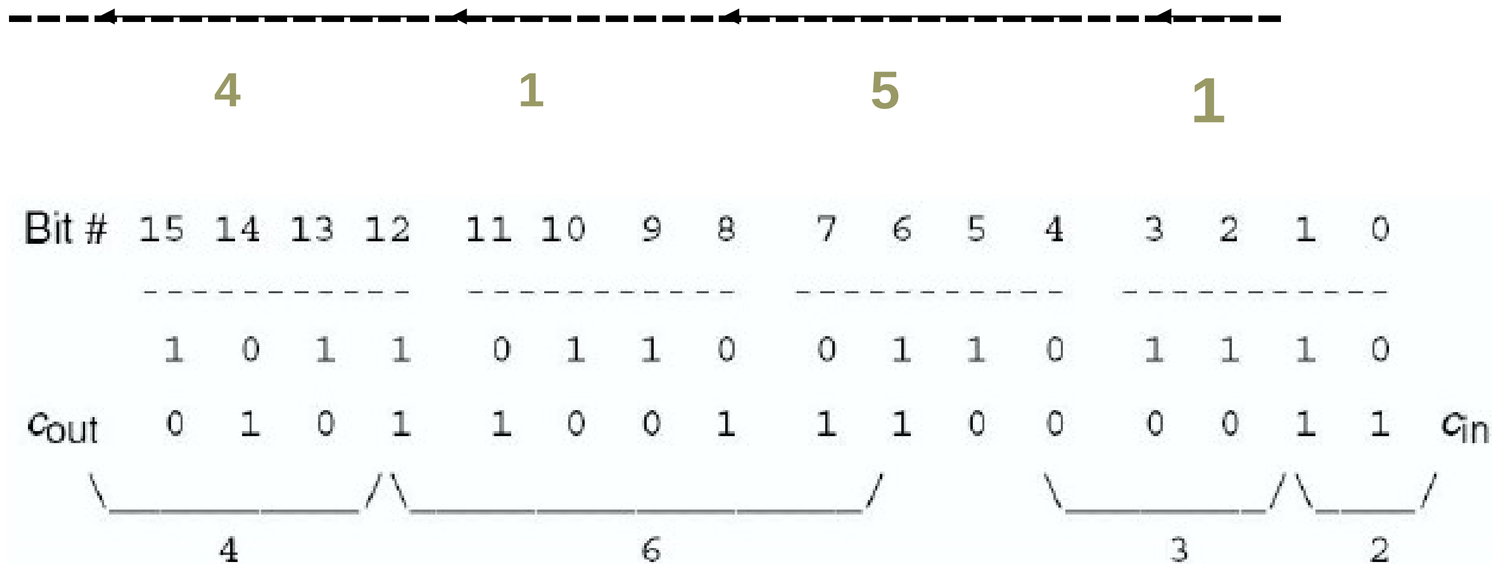
# Motivation



# Carry Completion Sensing

A=**0** 0 1 1 **1** 0 **1** 1 **0** 1 1 0 1 **1** **0** **1**

B=**0** 1 0 0 **1** 1 **1** 0 **0** 0 0 1 0 **1** **0** **1**



# Can we compute the average length of carry chain?

- What is the probability that a chain generated at position i terminates at j?
  - It terminates if both the inputs A[j] and B[j] are zero or 1.
  - From i+1 to j-1 the carry has to propagate.
  - $p=(1/2)^{j-i}$
  - So, what is the expected length?
  - Define a random variable L, which denotes the length of the chain.

# Expected length

- The chain can terminate at j=i+1 to j=k (the MSB position of the adder)
- Thus L=j-i for a choice of j.
- Thus expected length is: **approximately 2!**

$$\sum_{j=i+1}^{k-1}(j-i)2^{-(j-i)}+(k-i)2^{-(k-1-i)}$$

(the carry definitely ends at position k, so we do not multiply $2^{-(k-1-i)}$ with 1/2.)

$$=\sum_{l=1}^{k-1-i} l2^{-l}+(k-i)2^{-(k-1-i)}=2-(k-i+1)2^{-(k-1-i)}+(k-i)2^{-(k-1-i)}$$

$$=2-2^{-(k-1-i)}$$

[Using, $\sum_{l=1}^{p} l2^{-l}=2-(p+2)2^{-p}$]

# Carry completion sensing adder

A=0111011011101101

B=1001110000010101

------------------------------

C=0000000000000000

N=0000000000000000

------------------------------

C=0001010000000101

N=0000000010000010

A=0111011011101101

B=1001110000010101

------------------------------

C=0001010000000101

N=0000000010000010

------------------------------

C=0011110000001101

N=0000000110000010

# Carry completion sensing adder

A=011101101101101

B=100111000010101

------------------------------

C=001111000001101

N=000000110000010

------------------------------

C=011111000011101

N=000000110000010

A=011101101101101

B=100111000010101

------------------------------

C=011111000011101

N=000000110000010

------------------------------

C=111111000111101

N=000000110000010

# Carry completion sensing adder

A=011101101101101

B=100111000010101

------------------------------

C=111111000111101

N=000000110000010

-----------------------------

-

C=111111001111101

N=000000110000010

# Carry completion sensing adder

- (A[i],B[i])=(0,0)=>(Ci,Ni)=(0,1)
- (A[i],B[i])=(1,1)=>(Ci,Ni)=(1,0)
- (A[i],B[i])=(0,1)=>(Ci,Ni)=(Ci-1,Ni-1)
- (A[i],B[i])=(1,0)=>(Ci,Ni)=(Ci-1,Ni-1)
- Stop, when for all i, Ci V Ni = 1

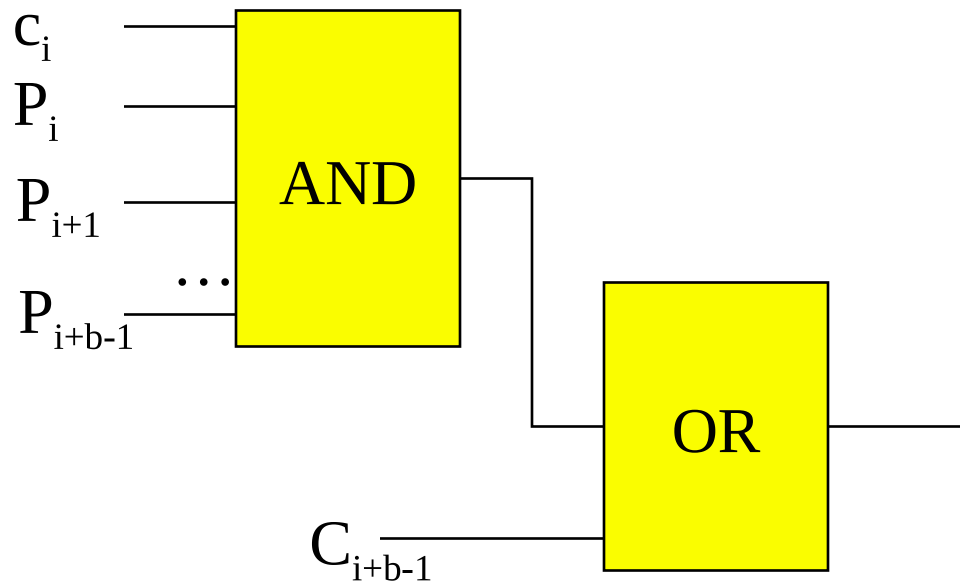
# Justification

- Ci and Ni together is a coding for the carry.
- When Ci=1, carry can be computed. Make Ni=0
- When Ci=0 is the final carry, then indicate by Ni=1
- The carry can be surely stated when both Ai and Bi are 1’s or 0’s.

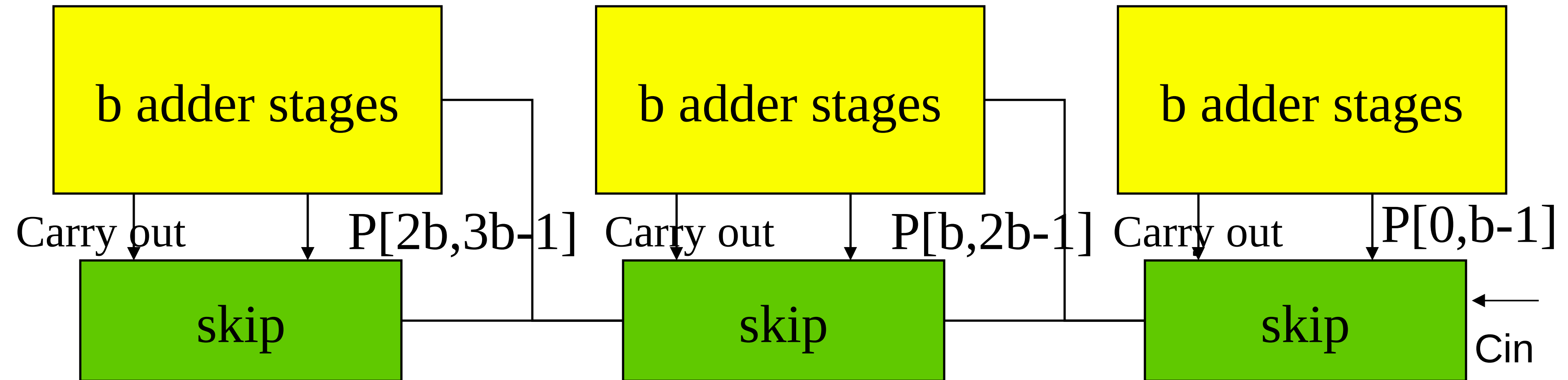
# Carry-skip adder

- Looks for cases in which carry out of a set of bits is identical to carry in.
- Typically organized into *b*-bit stages.
- Can bypass carry through all stages in a group when all propagates are true: $P_i \, P_{i+1} \ldots P_{i+b-1}$.
  - Carry out of group when carry out of last bit in group or carry is bypassed.

# Carry-skip structure

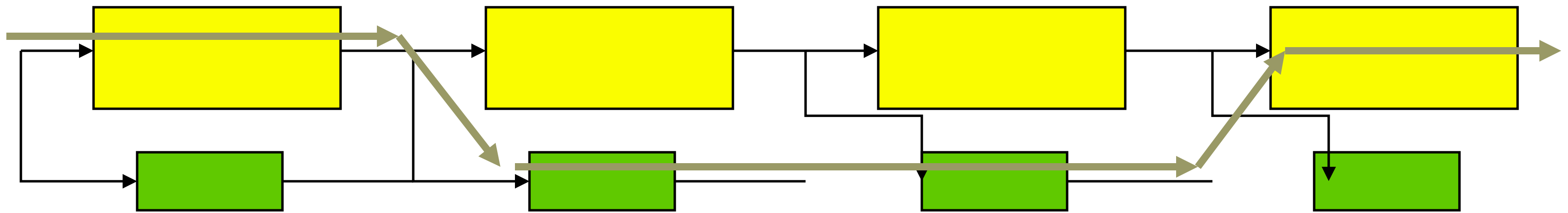


# Carry-skip structure



# Worst-case carry-skip

- Worst-case carry-propagation path goes through first, last stages:



# Verilog for carry-skip add with P

```
module fulladd_p(a,b,carryin,sum,carryout,p);
    input a, b, carryin; /* add these bits*/
    output sum, carryout, p; /* results including propagate */

    assign {carryout, sum} = a + b + carryin;
                    /*  compute the sum and carry */
   assign p = a ^ b;
endmodule
```

# Want to use ripple carry adder for the blocks

```
module fulladd_p(a,b,carryin,sum,carryout,p);
    input a, b, carryin; /* add these bits*/
    output sum, carryout, p; /* results including propagate */
  $rtl_binding="ADD3_RPL";
    assign {carryout, sum} = a + b + carryin;
                /*  compute the sum and carry */
    assign p = a ^ b;
endmodule
```

# Verilog for carry-skip adder

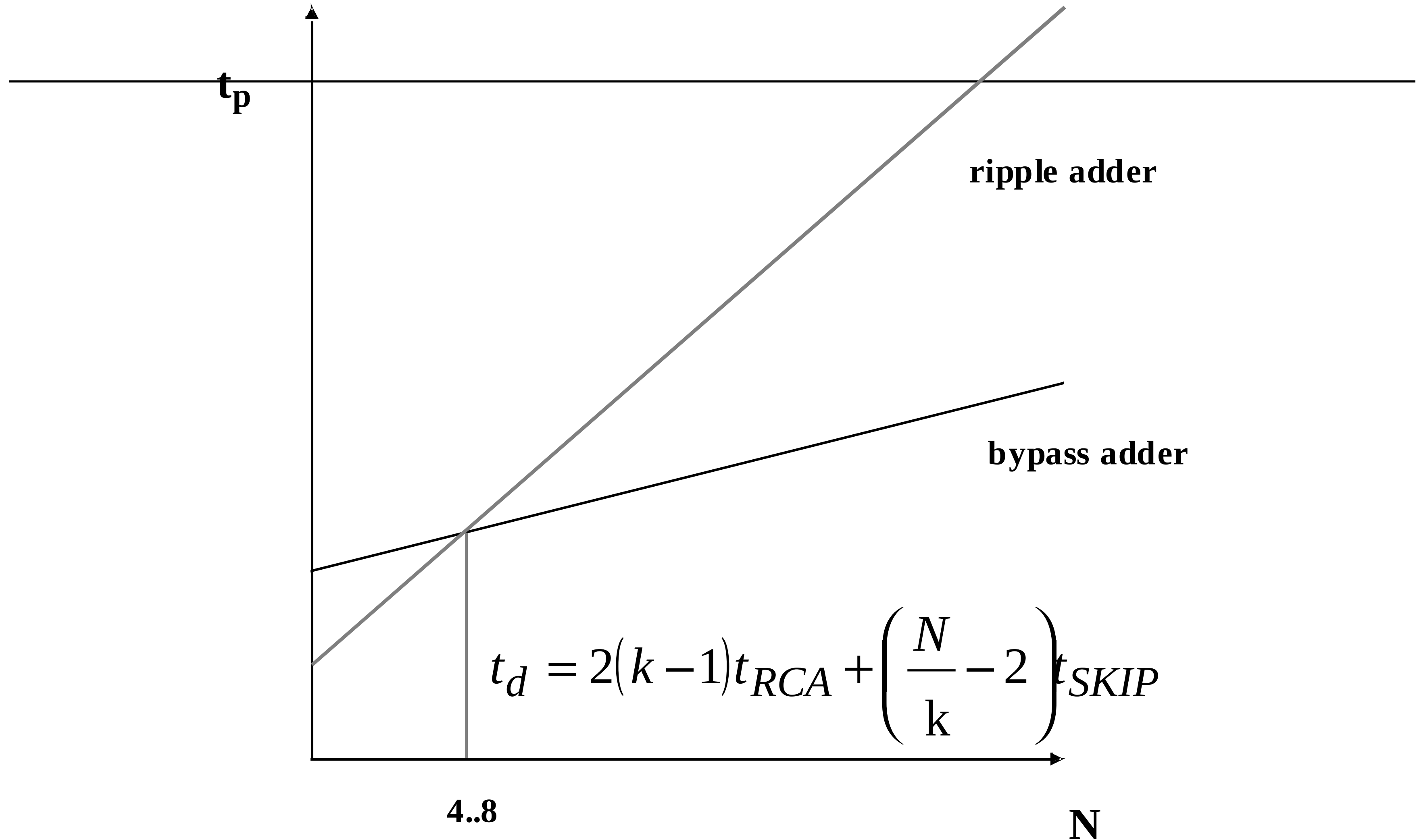
```
module carryskip(a,b,carryin,sum,carryout);
	input [7:0] a, b; /* add these bits */
	input carryin; /* carry in*/
	output [7:0] sum; /* result */
	output carryout;
	wire [8:1] carry; /* transfers the carry between bits */
	wire [7:0] p; /* propagate for each bit */
	wire cs4; /* final carry for first group */

	fulladd_p a0(a[0],b[0],carryin,sum[0],carry[1],p[0]);
	fulladd_p a1(a[1],b[1],carry[1],sum[1],carry[2],p[1]);
	fulladd_p a2(a[2],b[2],carry[2],sum[2],carry[3],p[2]);
	fulladd_p a3(a[3],b[3],carry[3],sum[3],carry[4],p[3]);
	assign cs4 = carry[4] | (p[0] & p[1] & p[2] & p[3] & carryin);
	fulladd_p a4(a[4],b[4],cs4,	sum[4],carry[5],p[4]);
…
	assign carryout = carry[8] | (p[4] & p[5] & p[6] & p[7] & cs4);
endmodule
```

# Delay analysis

- Assume that skip delay = 1 bit carry delay.
- Delay of k-bit adder with block size b:
  - T = (b-1) + 0.5 + (k/b –2) + (b-1)

    *block 0  OR gate  skips  last block*
- For equal sized blocks, optimal block size is sqrt(k/2).

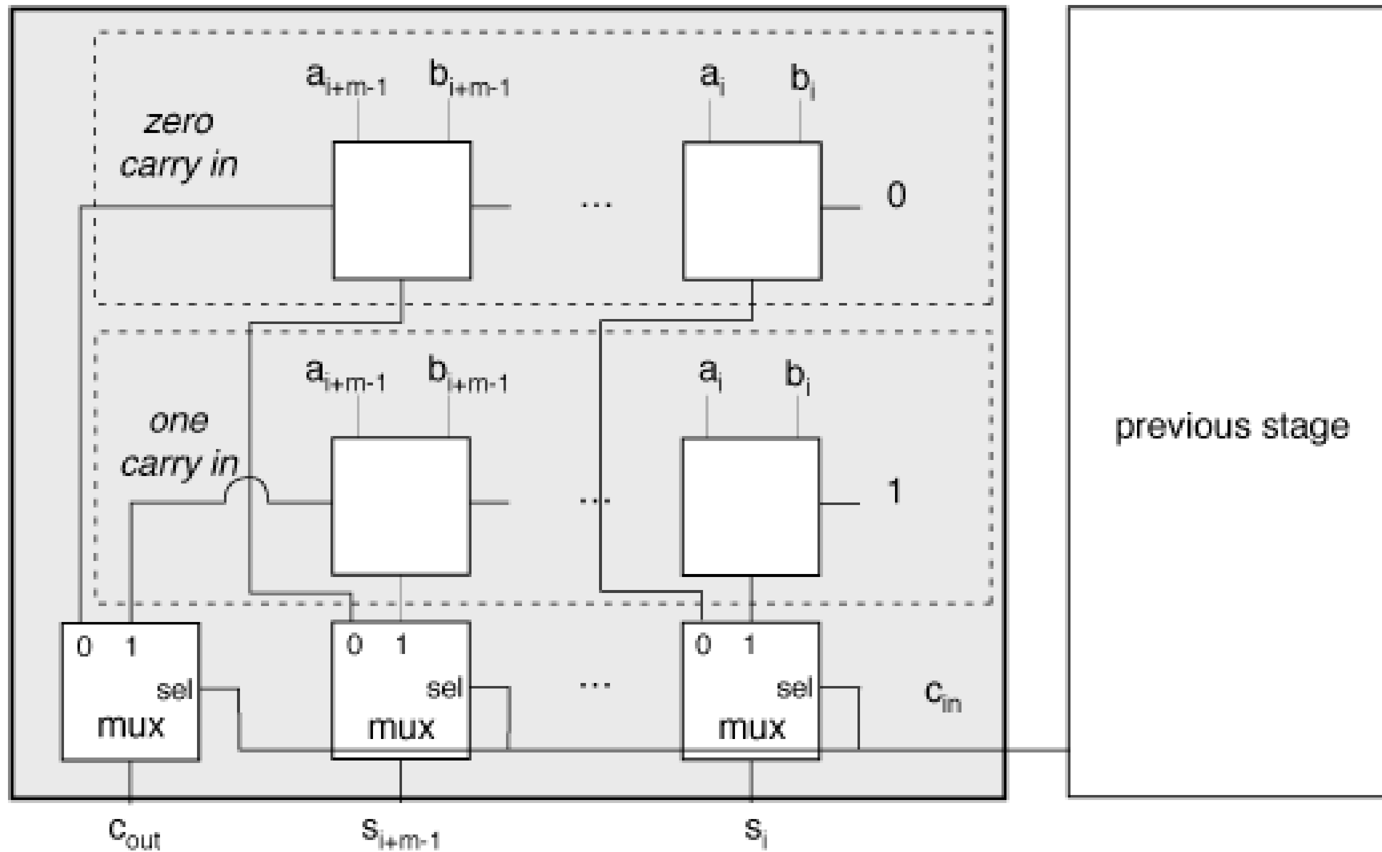
# Delay of Carry-Skip Adder



# Carry-select adder

- Computes two results in parallel, each for different carry input assumptions.
- Uses actual carry in to select correct result.
- Reduces delay to multiplexer.

# Carry-select structure

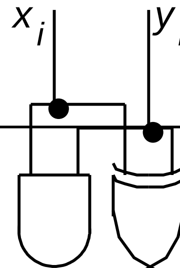


# Carry-save adder

- Useful in multiplication.
- Input: 3 n-bit operands.
- Output: n-bit partial sum, n-bit carry.
  - Use carry propagate adder for final sum.
- Operations:
  - $s = (x + y + z) \bmod 2$.
  - $c = [(x + y + z) - 2] / 2$.

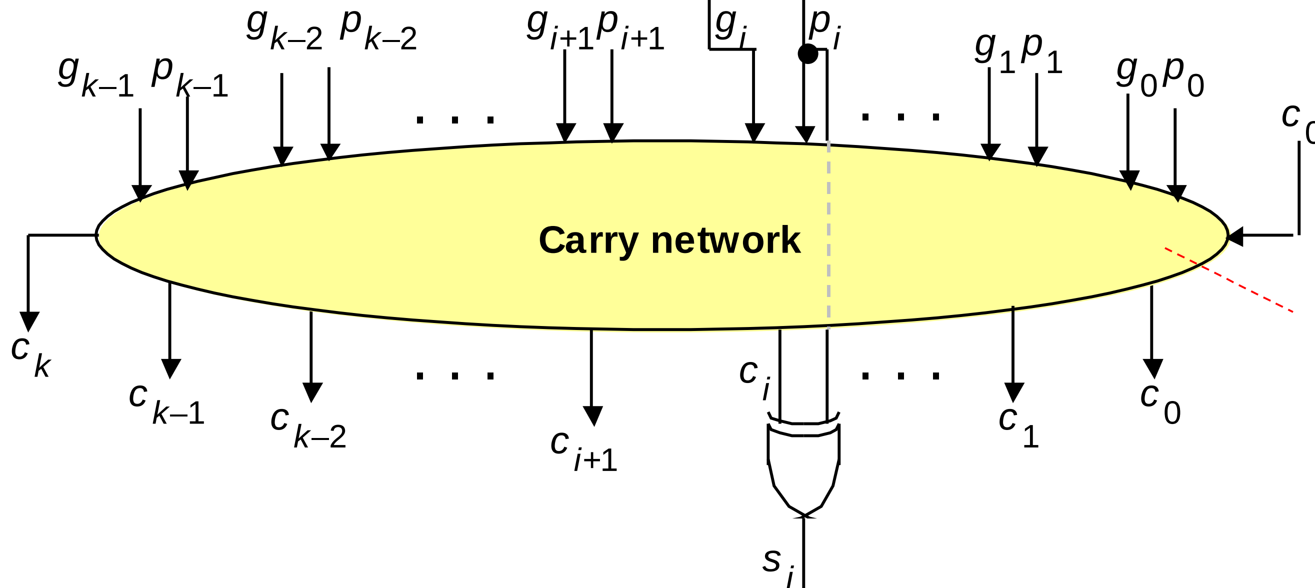
# Carry Network is the Essence of a Fast Adder

| $g_i$ | $p_i$ | Carry is: |
|---|---|---|
| 0 | 0 | annihilated or killed |
| 0 | 1 | propagated |
| 1 | 0 | generated |
| 1 | 1 | (impossible) |




$g_i = x_i \, y_i$

$p_i = x_i \oplus y_i$




Ripple; Skip; Lookahead; Parallel-prefix

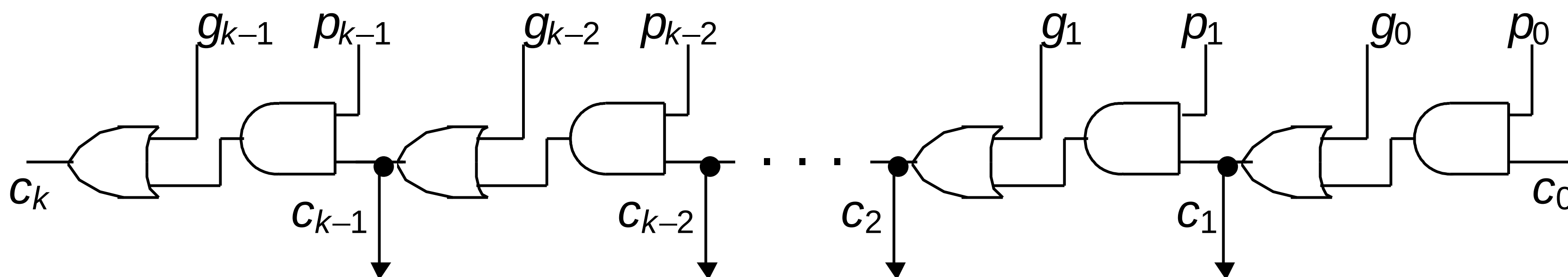
Generic structure of a binary adder, highlighting its carry network.

# Ripple-Carry Adder Revisited

The carry recurrence: $c_{i+1} = g_i \vee p_i c_i$

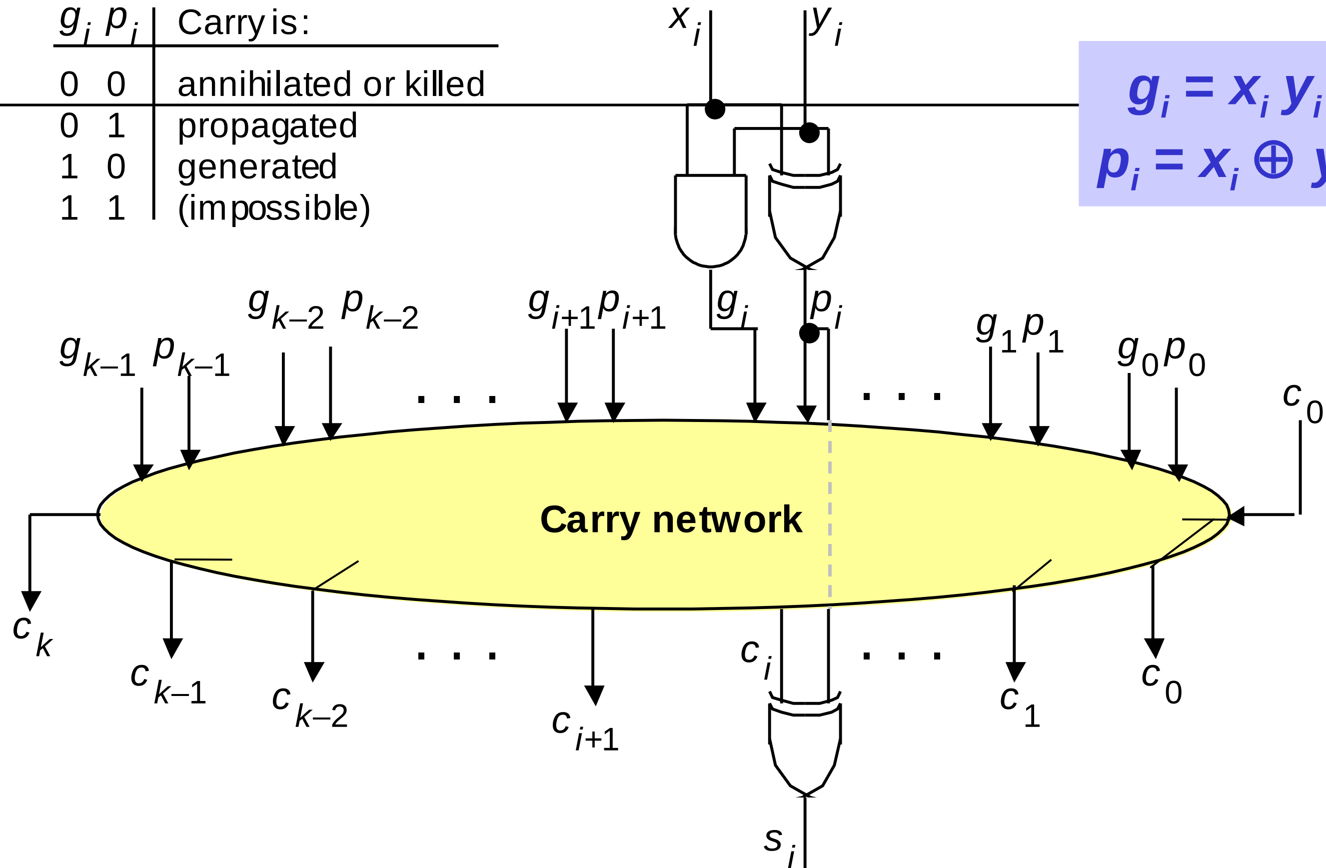
Latency of *k*-bit adder is roughly 2*k* gate delays:

1 gate delay for production of *p* and *g* signals, plus
2(*k* – 1) gate delays for carry propagation, plus
1 XOR gate delay for generation of the sum bits




Alternate view of a ripple-carry network in connection with the generic adder structure shown in Fig. 5.14.

# The Complete Design of a Ripple-Carry Adder



# 6.1 Unrolling the Carry Recurrence

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

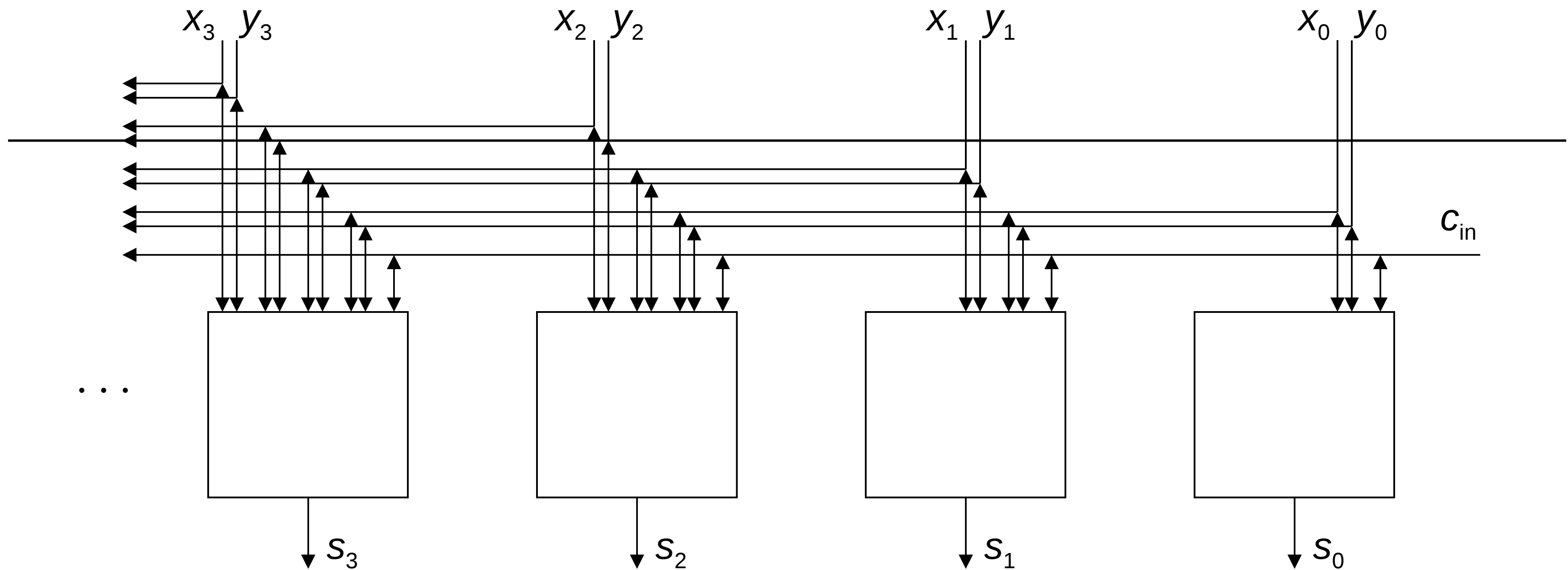
| Signal | Radix $r$ | Binary |
|---|---|---|
| $g_i$ | is 1 iff $x_i + y_i \geq r$ | $x_i y_i$ |
| $p_i$ | is 1 iff $x_i + y_i = r - 1$ | $x_i \oplus y_i$ |
| $a_i$ | is 1 iff $x_i + y_i < r - 1$ | $x_i' y_i' = (x_i \vee y_i)'$ |
| $t_i$ | is 1 iff $x_i + y_i \geq r - 1$ | $x_i \vee y_i$ |
| $s_i$ | $(x_i + y_i + c_i) \bmod r$ | $x_i \oplus y_i \oplus c_i$ |

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

**Note:**
**Addition symbol vs logical OR**

$$
\begin{aligned}
c_i &= g_{i-1} \vee c_{i-1} p_{i-1} \\
&= g_{i-1} \vee (g_{i-2} \vee c_{i-2} p_{i-2}) p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee c_{i-2} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee c_{i-3} p_{i-3} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee g_{i-4} p_{i-3} p_{i-2} p_{i-1} \vee c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} \\
&= \ldots
\end{aligned}
$$

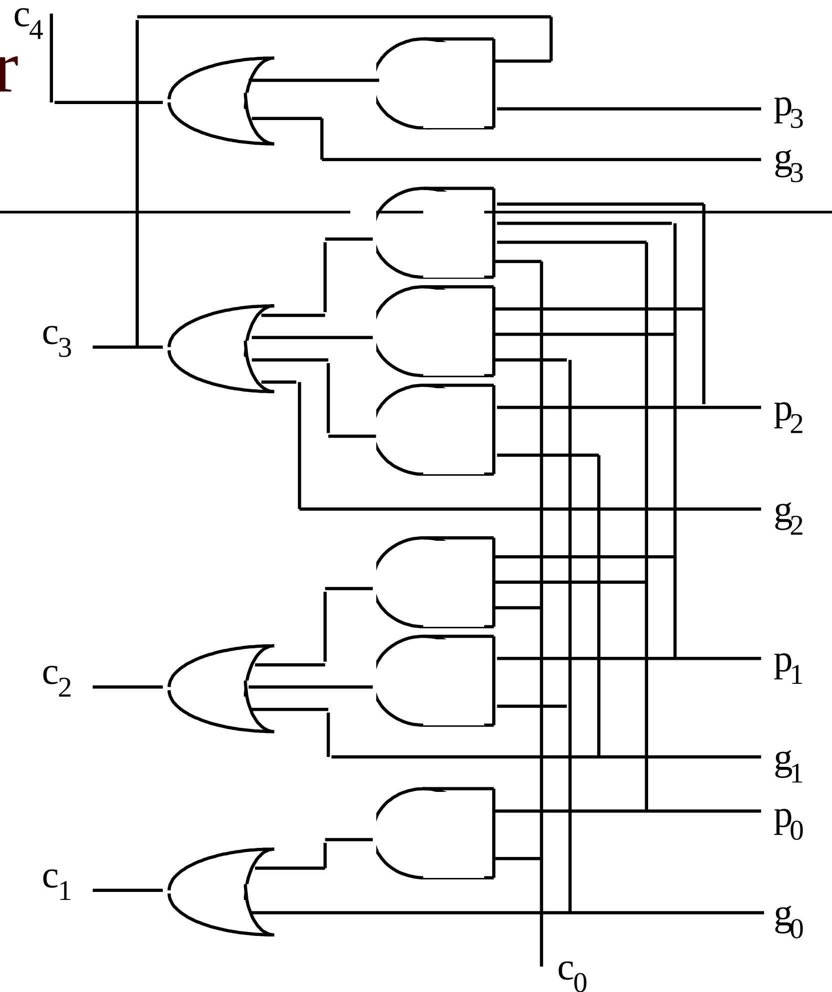
# Full Carry Lookahead




Theoretically, it is possible to derive each sum digit directly from the inputs that affect it

Carry-lookahead adder design is simply a way of reducing the complexity of this ideal, but impractical, arrangement by hardware sharing among the various lookahead circuits

# Four-Bit Carry-Lookahead Adder




Complexity reduced by deriving the carry-out indirectly

Full carry lookahead is quite practical for a 4-bit adder

$$c_1 = g_0 \vee c_0 p_0$$

$$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$$

$$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$$

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

Four-bit carry network with full lookahead.

# Carry Lookahead Beyond 4 Bits

Consider a 32-bit adder

$c_1 = g_0 \vee c_0 p_0$

$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$

$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$

.
.
.

$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ . \ . \ . \ \vee c_0 p_0 p_1 p_2 p_3 \ldots p_{29} p_{30}$

No circuit sharing:
Repeated computations

32-input AND

32-input OR

High fan-ins necessitate tree-structured circuits

# Solution to the Fan-in Problem

High-radix addition (i.e., radix $2^h$)

Increases the latency for generating $g$ and $p$ signals and sum digits, but simplifies the carry network (optimal radix?)

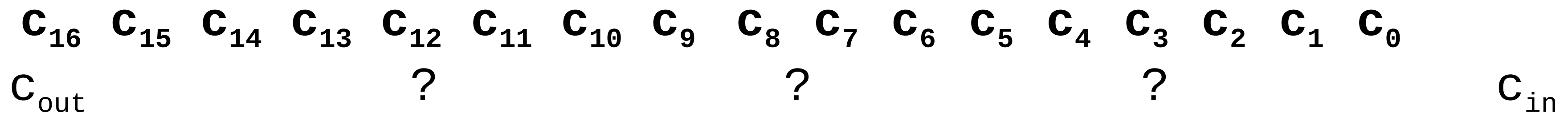
Multilevel lookahead

Example: 16-bit addition

Radix-16 (four digits)

Two-level carry lookahead (four 4-bit blocks)

Either way, the carries $c_4$, $c_8$, and $c_{12}$ are determined first

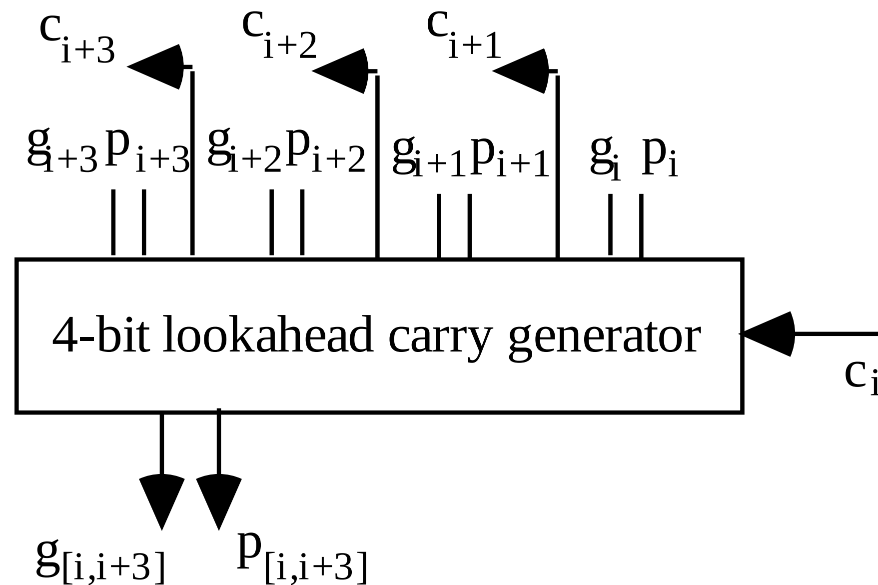


# Carry-Lookahead Adder Design

Block *generate* and *propagate* signals

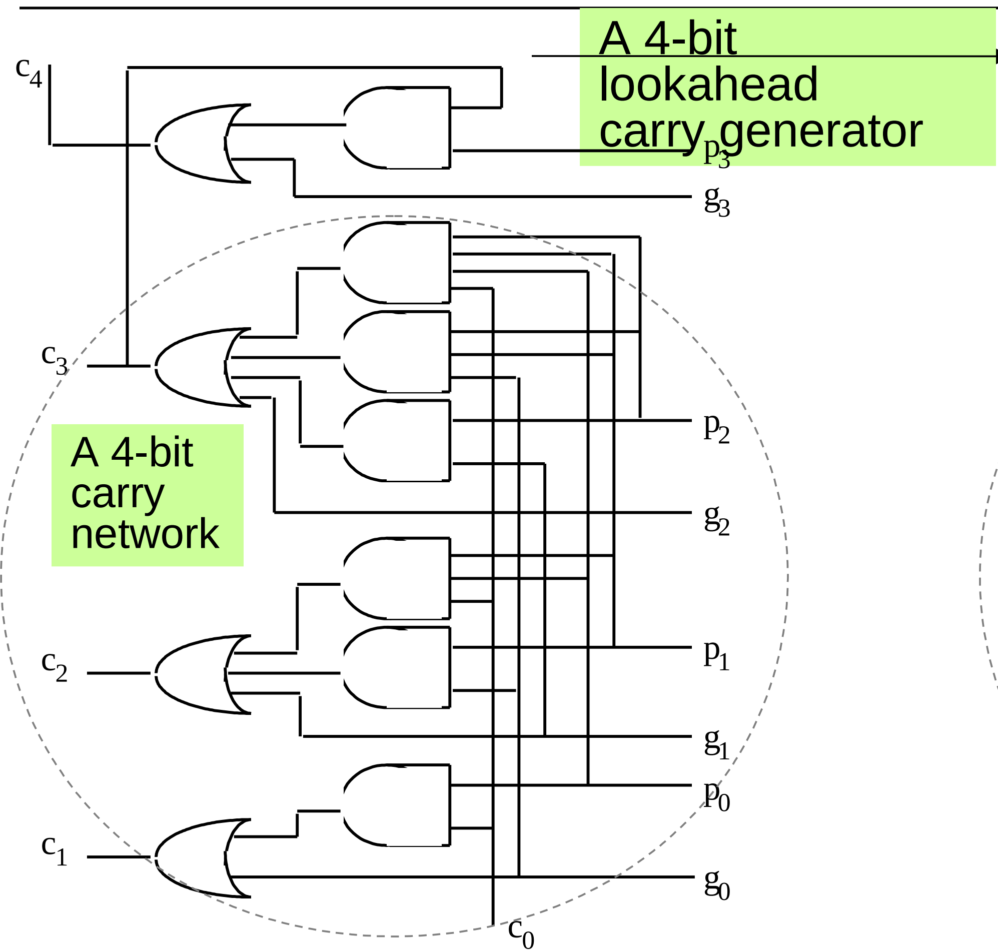
$$g_{[i,i+3]} = g_{i+3} \vee g_{i+2}p_{i+3} \vee g_{i+1}p_{i+2}p_{i+3} \vee g_i\, p_{i+1}p_{i+2}p_{i+3}$$

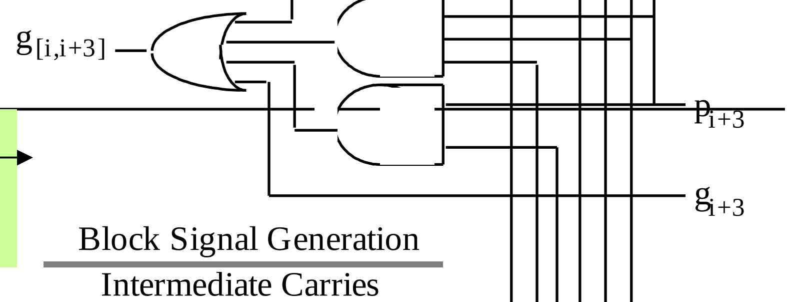
$$p_{[i,i+3]} = p_i\, p_{i+1}p_{i+2}p_{i+3}$$




Schematic diagram of a 4-bit lookahead carry generator.

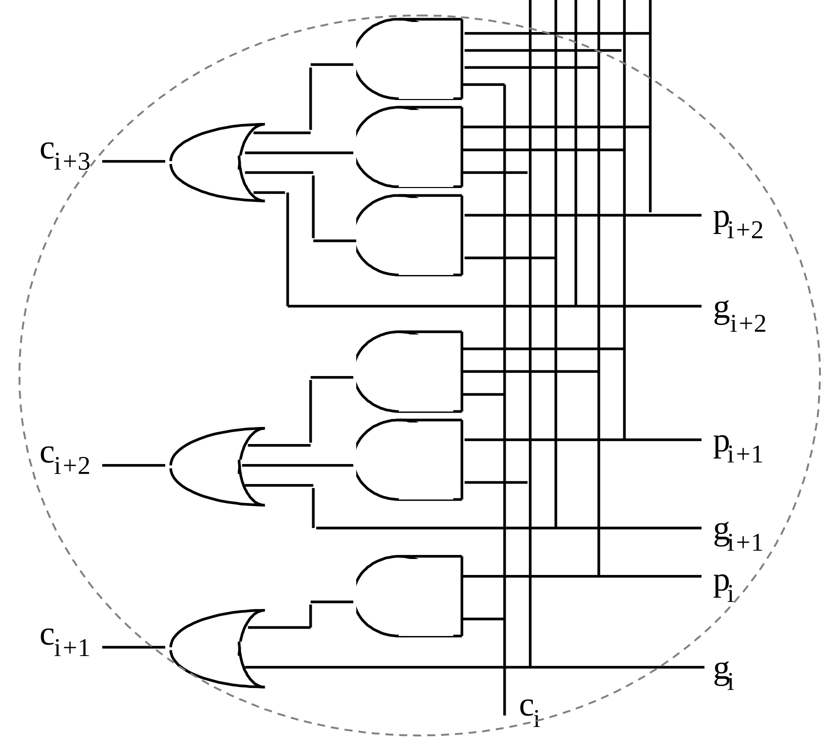
# A Building Block for Carry-Lookahead Addition



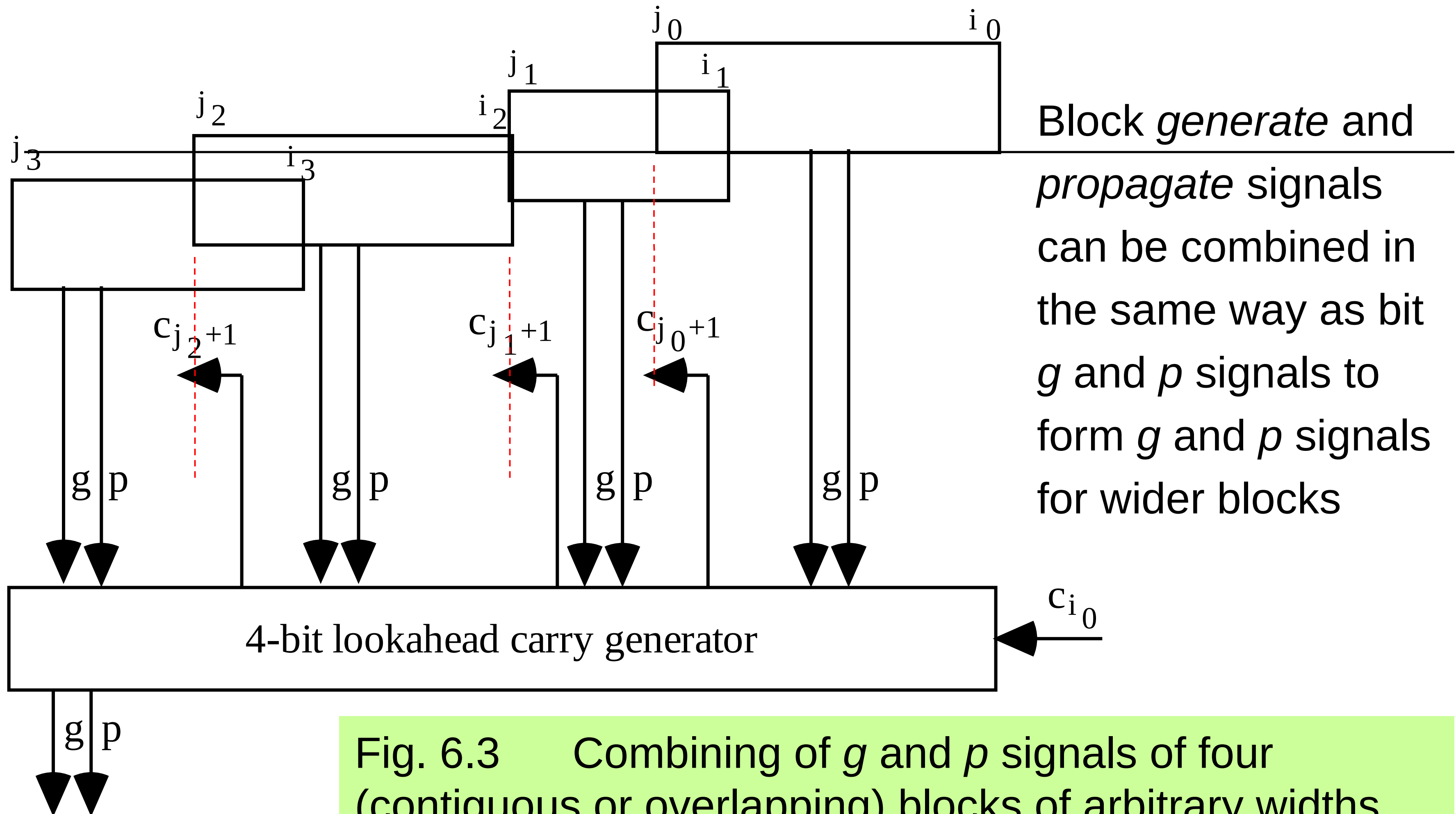




Block Signal Generation

Intermediate Carries



# Combining Block $g$ and $p$ Signals




Block *generate* and *propagate* signals can be combined in the same way as bit $g$ and $p$ signals to form $g$ and $p$ signals for wider blocks

Fig. 6.3 Combining of $g$ and $p$ signals of four (contiguous or overlapping) blocks of arbitrary widths into the $g$ and $p$ signals for the overall block $[i_0, j_3]$.

# A Two-Level Carry-Lookahead Adder

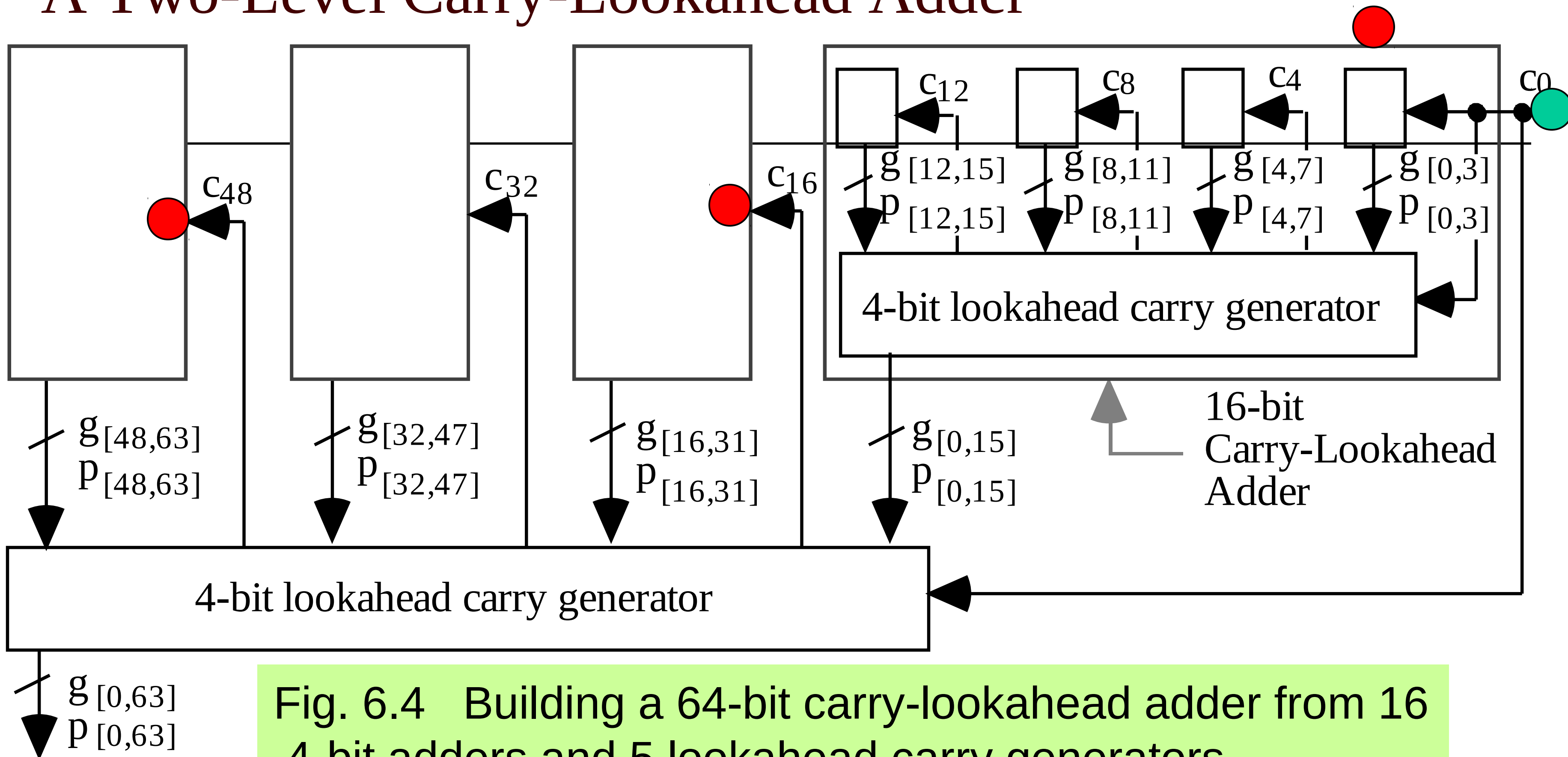



Fig. 6.4 Building a 64-bit carry-lookahead adder from 16 4-bit adders and 5 lookahead carry generators.

Carry-out: $c_{out} = g_{[0,k-1]} \vee c_0 \, p_{[0,k-1]} = x_{k-1}y_{k-1} \vee s_{k-1}' (x_{k-1} \vee y_{k-1})$

# Latency of a Multilevel Carry-Lookahead Adder

Latency through the 16-bit CLA adder consists of finding:

| | |
|---|---|
| $g$ and $p$ for individual bit positions | 1 gate level |
| $g$ and $p$ signals for 4-bit blocks | 2 gate levels |
| Block carry-in signals $c_4$, $c_8$, and $c_{12}$ | 2 gate levels |
| Internal carries within 4-bit blocks | 2 gate levels |
| Sum bits | 2 gate levels |

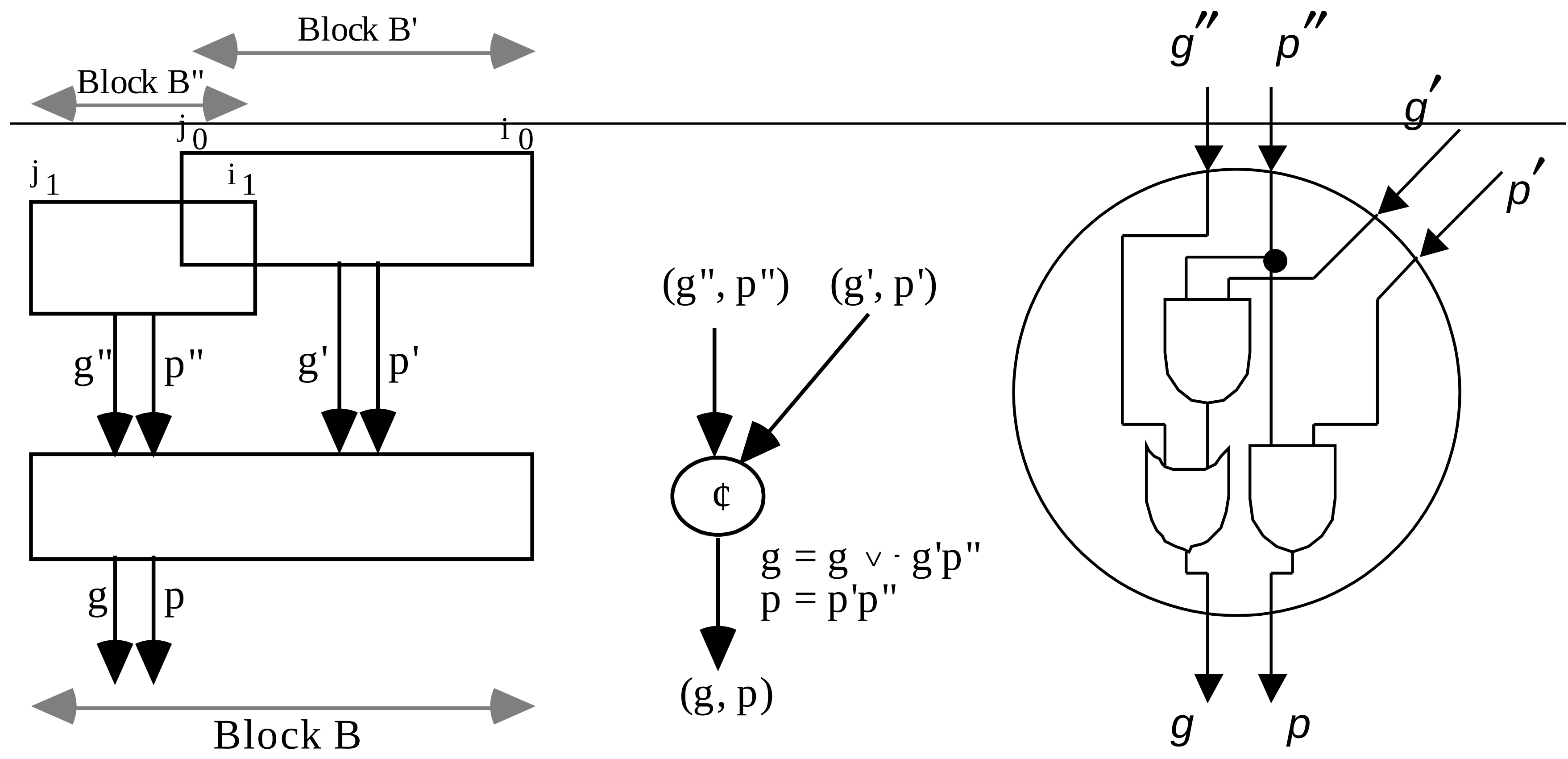
Total latency for the 16-bit adder 9 gate levels

(compare to 32 gate levels for a 16-bit ripple-carry adder)

Each additional lookahead level adds 4 gate levels of latency

Latency for $k$-bit CLA adder: $T_{\text{lookahead-add}} = 4 \log_4 k + 1$ gate levels

# Carry Determination as Prefix Computation




Combining of *g* and *p* signals of two (contiguous or overlapping) blocks B' and B" of arbitrary widths into the *g* and *p* signals for block B.

# Formulating the Prefix Computation Problem

The problem of carry determination can be formulated as:

Given $(g_0, p_0)(g_1, p_1)$ . . . $(g_{k-2}, p_{k-2})$ $(g_{k-1}, p_{k-1})$

Find $(g_{[0,0]}, p_{[0,0]})$ $(g_{[0,1]}, p_{[0,1]})$ . . . $(g_{[0,k-2]}, p_{[0,k-2]})$ $(g_{[0,k-1]}, p_{[0,k-1]})$

$c_1$ $c_2$ . . . $c_{k-1}$ $c_k$

Carry-in can be viewed as an extra (–1) position: $(g_{-1}, p_{-1}) = (c_{in}, 0)$

The desired pairs are found by evaluating all prefixes of

$$(g_0, p_0) ¢ (g_1, p_1) ¢ \ldots ¢ (g_{k-2}, p_{k-2}) ¢ (g_{k-1}, p_{k-1})$$

The carry operator ¢ is associative, but not commutative

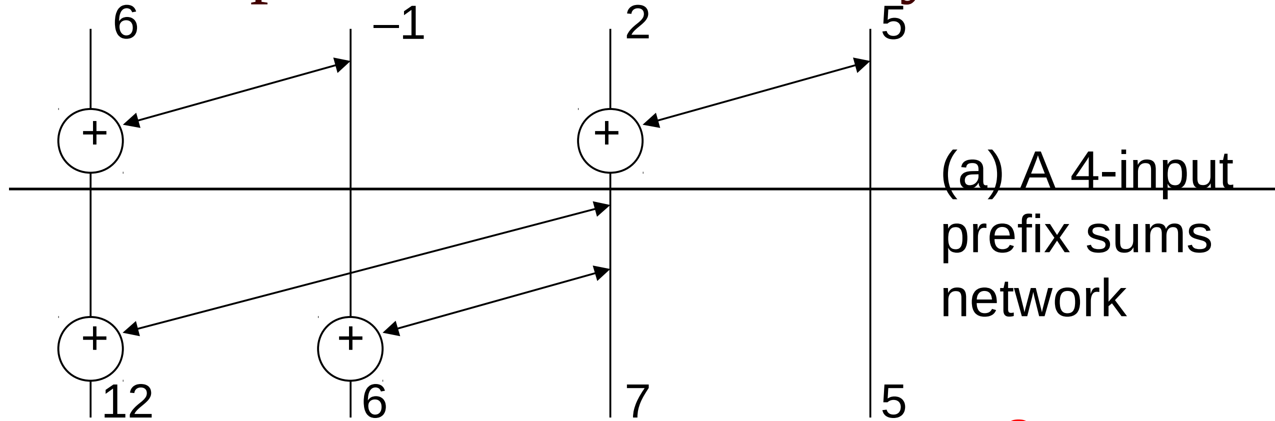
$$[(g_1, p_1) ¢ (g_2, p_2)] ¢ (g_3, p_3) = (g_1, p_1) ¢ [(g_2, p_2) ¢ (g_3, p_3)]$$

Prefix sums analogy:

Given $x_0$ $x_1$ $x_2$ . . . $x_{k-1}$

Find $x_0$ $x_0+x_1$ $x_0+x_1+x_2$ . . . $x_0+x_1+\ldots+x_{k-1}$

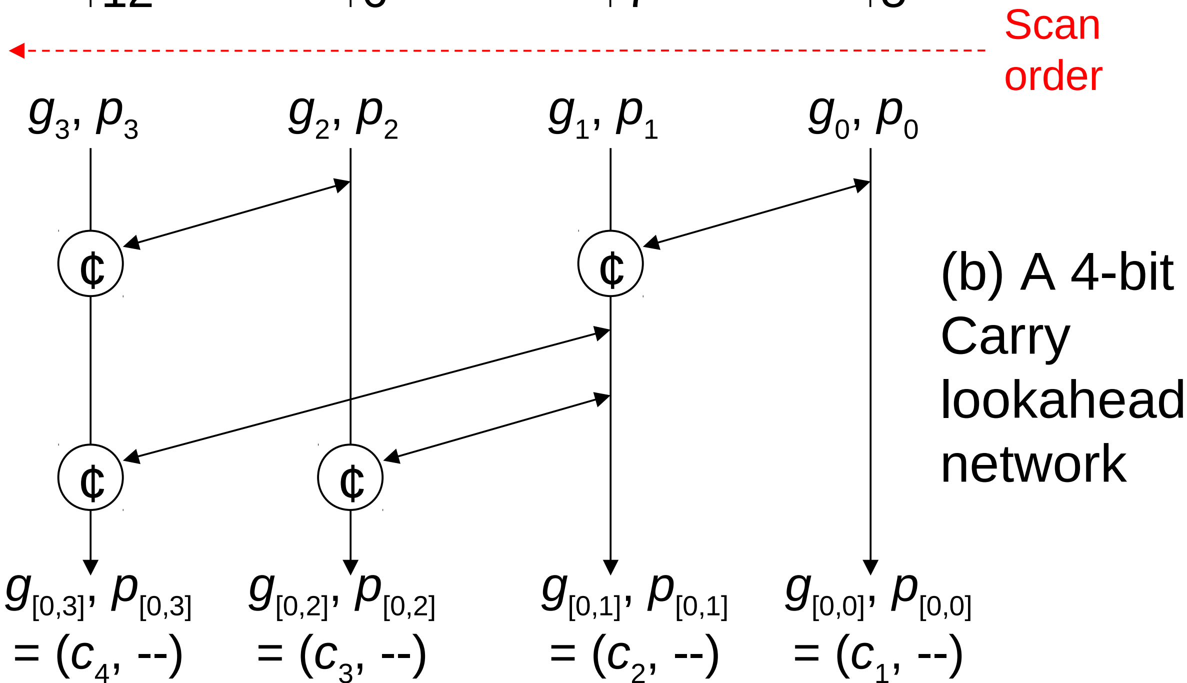
# Example Prefix-Based Carry Network



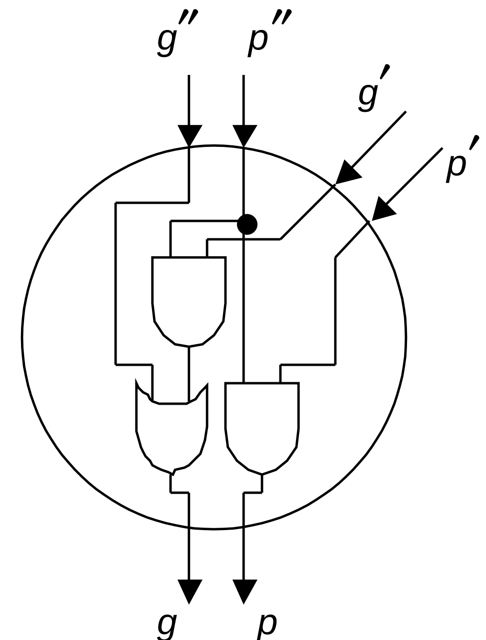

(a) A 4-input prefix sums network

Fig. 6.6 Four-input parallel prefix sums network and its corresponding carry network.

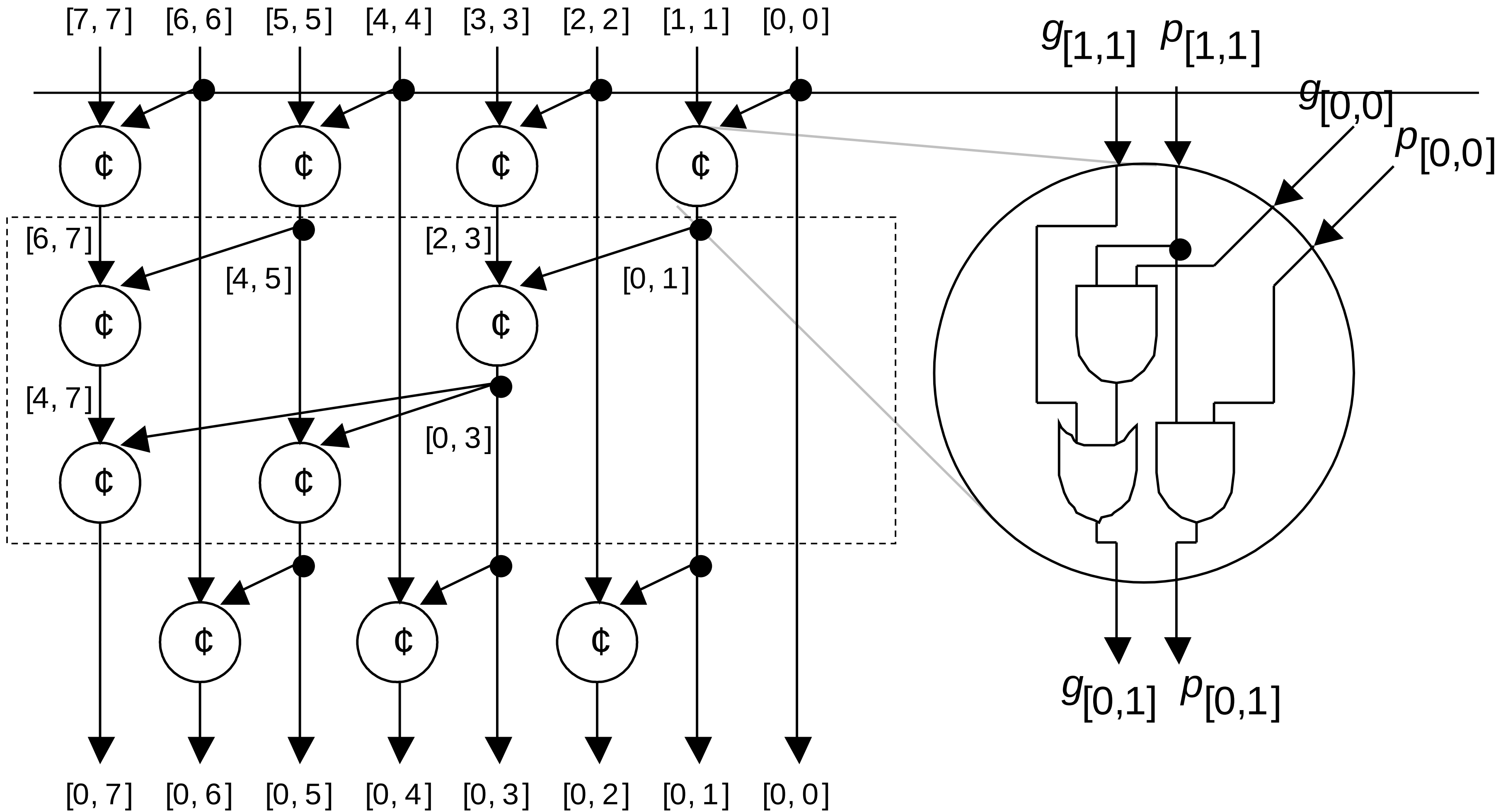
Scan order



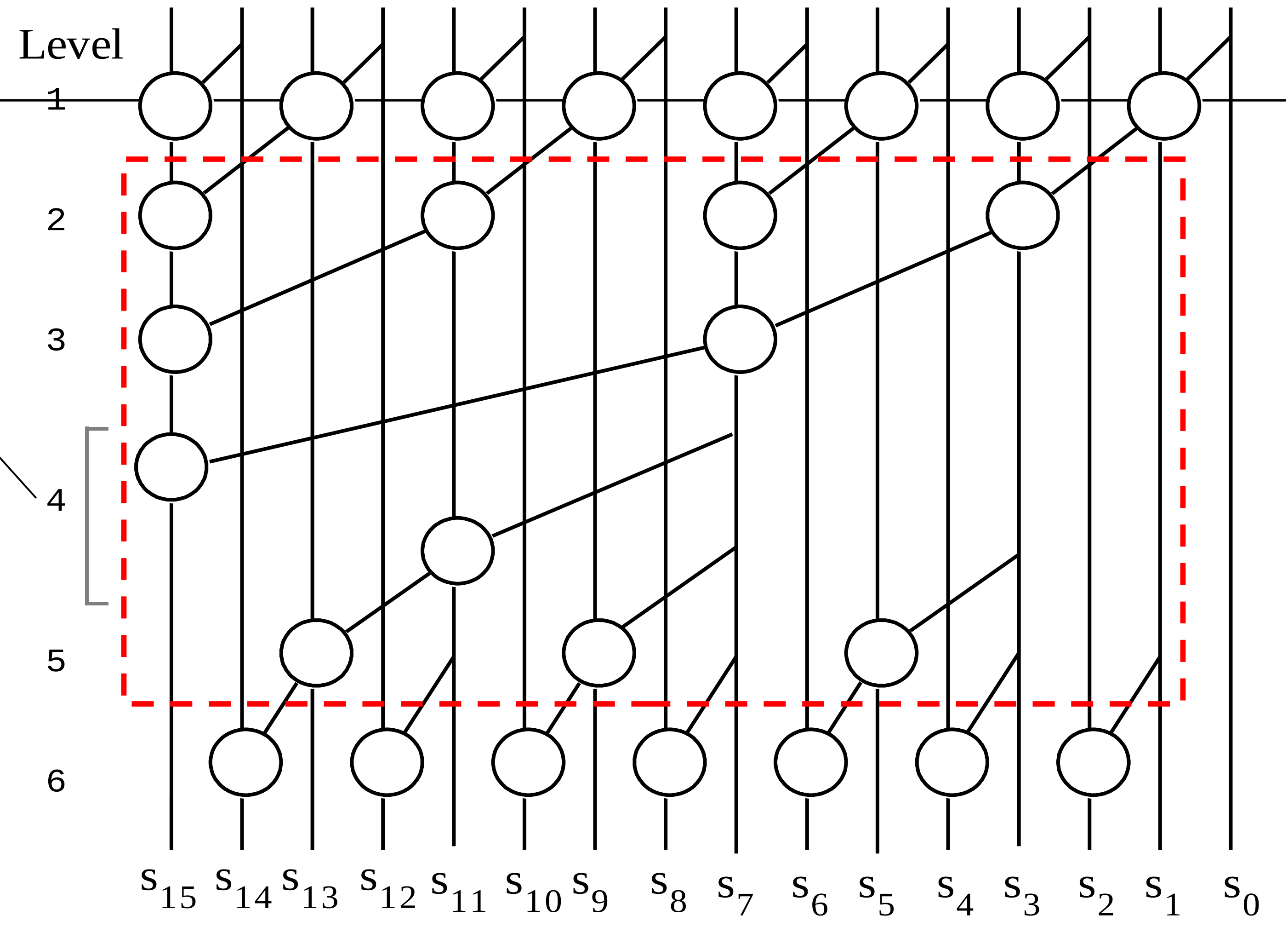

(b) A 4-bit Carry lookahead network



# Brent-Kung Carry Network (8-Bit Adder)



# Brent-Kung Carry Network (16-Bit Adder)




Reason for latency being $2 \log_2 k - 2$

Brent-Kung parallel prefix graph for 16 inputs.

# Adder comparison

- Ripple-carry adder has highest performance/cost.
- Optimized adders are most effective in very long bit widths (> 48 bits).

# ALUs

- ALU computes a variety of logical and arithmetic functions based on **opcode**.
- May offer complete set of functions of two variables or a subset.
- ALU built around adder, since carry chain determines delay.