MODULE- 1

Introduction to ASICs: Types of ASICs-Full Custom with ASIC, Semi-custom ASICS, Standard Cell based ASIC, Gate array based ASIC, Channeled gate array, Channel less gate array, structured get array, Programmable logic device, FPGA, design flow, ASIC cell libraries. (1.1 ,1.2 &1.5)

Data Logic Cells: Data Path Elements, Adders, Multiplier, Arithmetic Operator, I/O cell, Cell Compilers. (2.6,2.7 &2.8)

by,

Dr.P.Vimala
Department of ECE
Dayananda Sagar College of Engineering
Bangalore

Figures from M.J.S .Smith, - “**Application - Specific Integrated Circuits**”

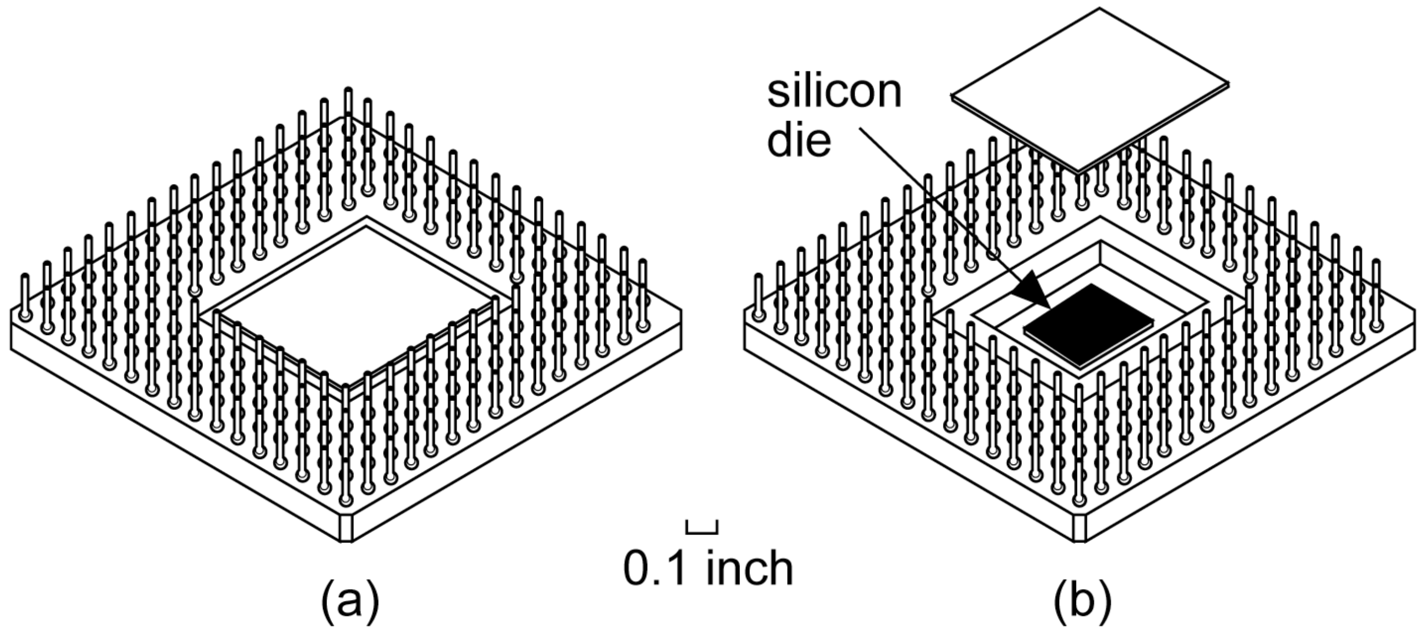
ASICs

- ASIC - Application Specific Integrated Circuit
  - In Integrated Circuit (IC) designed to perform a specific function for a specific application

- Levels of integration:
  - SSI - Small scale integration
  - MSI - Medium scale integration
  - LSI - Large scale integration
  - VLSI - Very large scale integration
  - USLI - Ultra large scale integration

- Implementation technology
  - Bipolar technology
    – used TTL, ECL
  - MOS technology
    - NMOS, PMOS
  - BiCMOS technology
  - Custom IC

An Integrated Circuit



A packaged Integrated Circuit (IC)
(a) Pin-Grid Array (PGA) Package
(b) Silicon die is under the package lid

Types of ASICs

- Full-Custom ASICs
- Standard-Cell–Based ASICs
- Gate-Array–Based ASICs
    - Channeled Gate Array
    - Channel less Gate Array
    - Structured Gate Array
- Programmable Logic Devices
- Field-Programmable Gate Arrays

Full-Custom ASICs

- Full-custom design are usually laid out completely by hand
- All mask layers are customized in a full-custom ASIC
  - Generally, the designer lays out all cells by hand
    Some automatic placement and routing may be done
  - Critical (timing) paths offers the highest performance and lowest part cost (smallest die size) for a given design
- The disadvantages of full-custom design include increased design time, complexity, design expense, and highest risk
- Microprocessors (strategic silicon) were exclusively full-custom, but designers are increasingly turning to semicustom ASIC techniques in this area as well
- Other examples of full-custom ICs or ASICs are requirements for high-voltage (automobile), analog/digital (communications), sensors and actuators, and memory (DRAM)

Standard-Cell-Based ASICs

- A cell-based ASIC (CBIC) are built of row of standard cells (wall built of bricks)
- Standard cells areas. The MP or MC are called mega cells (mega functions , full-custom blocks , system-level macros(SLMs), fixed blocks , cores , or Functional Standard Blocks (FSBs))
- The important features of CBICs are:
  - All mask layers are customized - transistors and interconnect. Automated buffer sizing, placement and routing
  - Custom blocks can be embedded
  - Manufacturing lead time is about eight weeks.

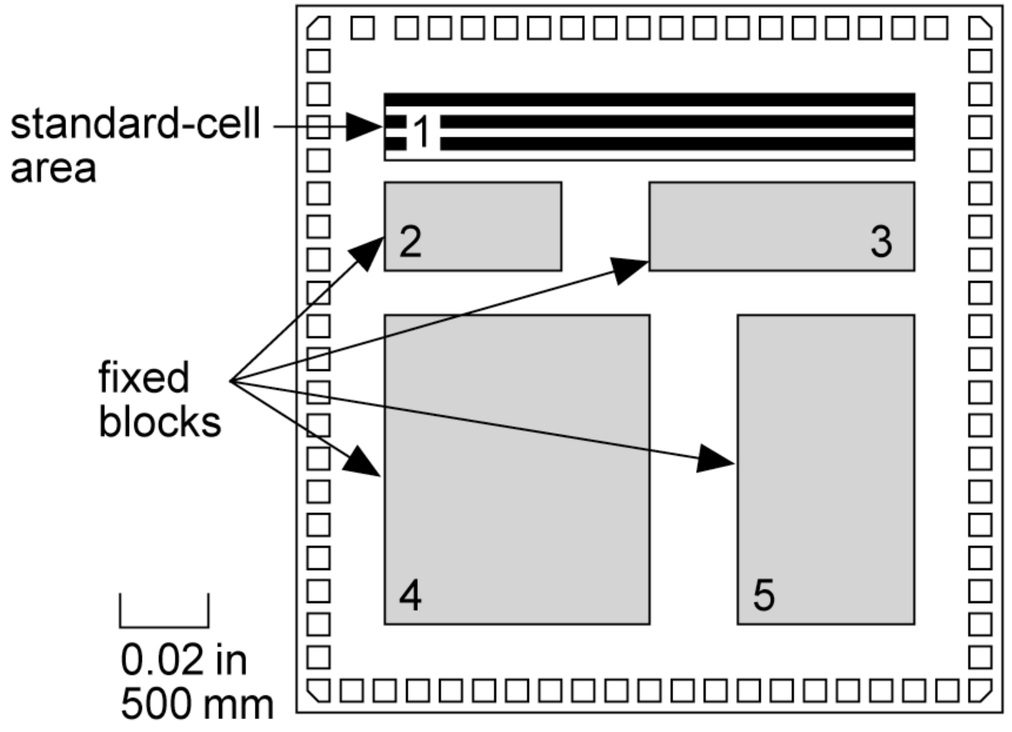


Figure 1.2 A cell-based ASIC (CBIC)

Standard Cell Layout

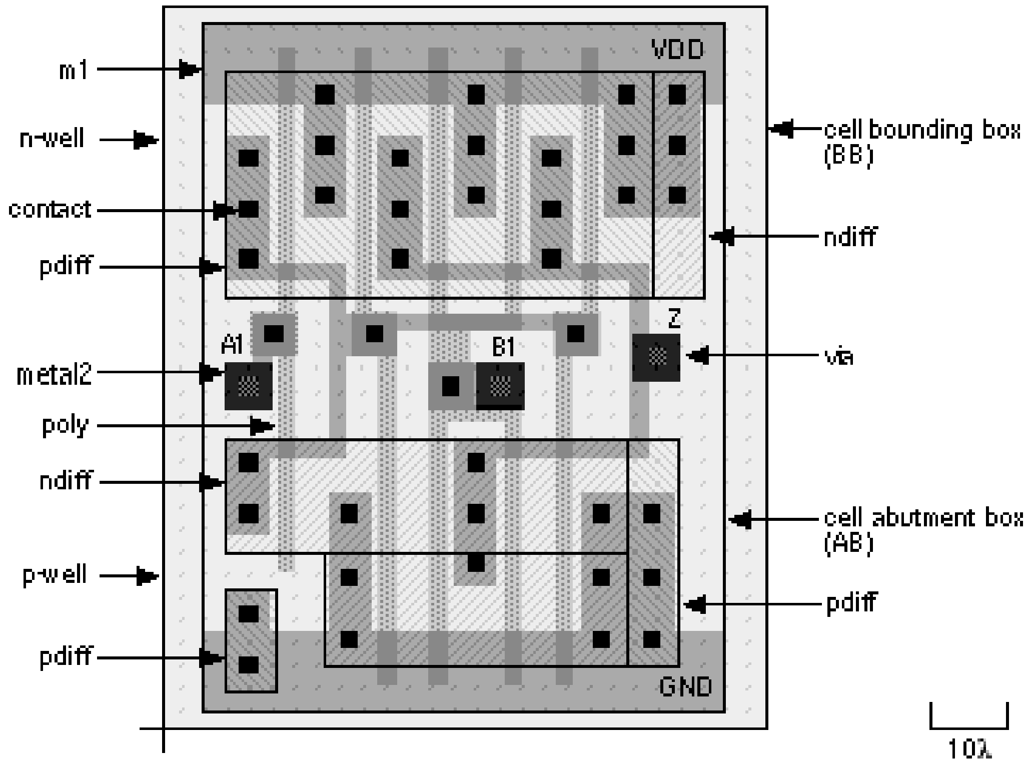


Figure 1.3 Layout of a standard cell

- This cell would be approximately 25 microns wide on an ASIC with l (lambda) = 0.25 microns (a micron is 10^{-6} m).
- Standard cells are stacked like bricks in a wall; the abutment box (AB) defines the "edges" of the brick.
- The difference between the bounding box (BB) and the AB is the area of overlap between the bricks.
- Power supplies (labeled VDD and GND) run horizontally inside a standard cell on a metal layer that lies above the transistor layers.
- Each different shaded and labeled pattern represents a different layer. This standard cell has center connectors (the three squares, labeled A1, B1, and Z) that allow the cell to connect to others.

Standard Cell ASIC Routing

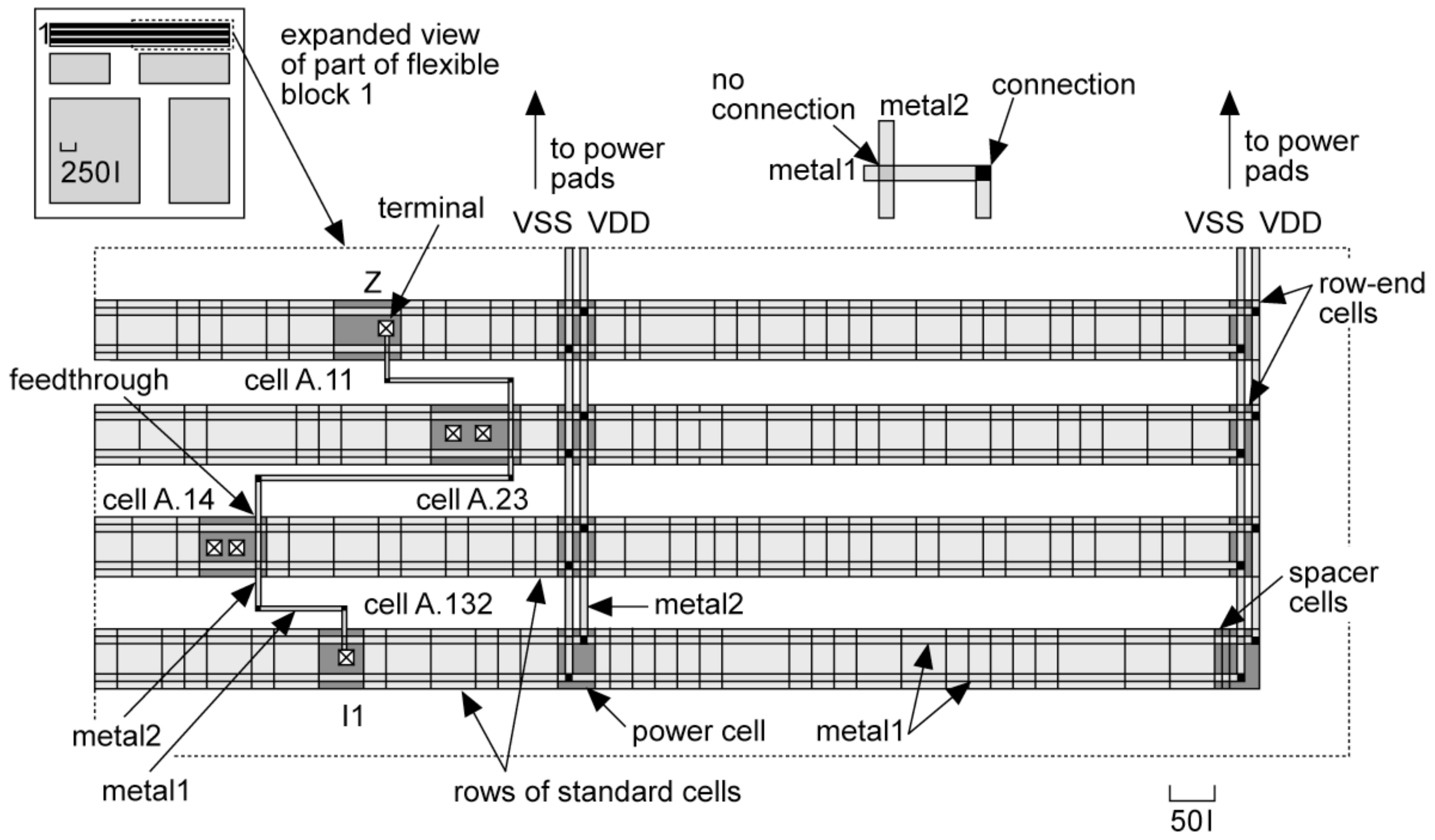
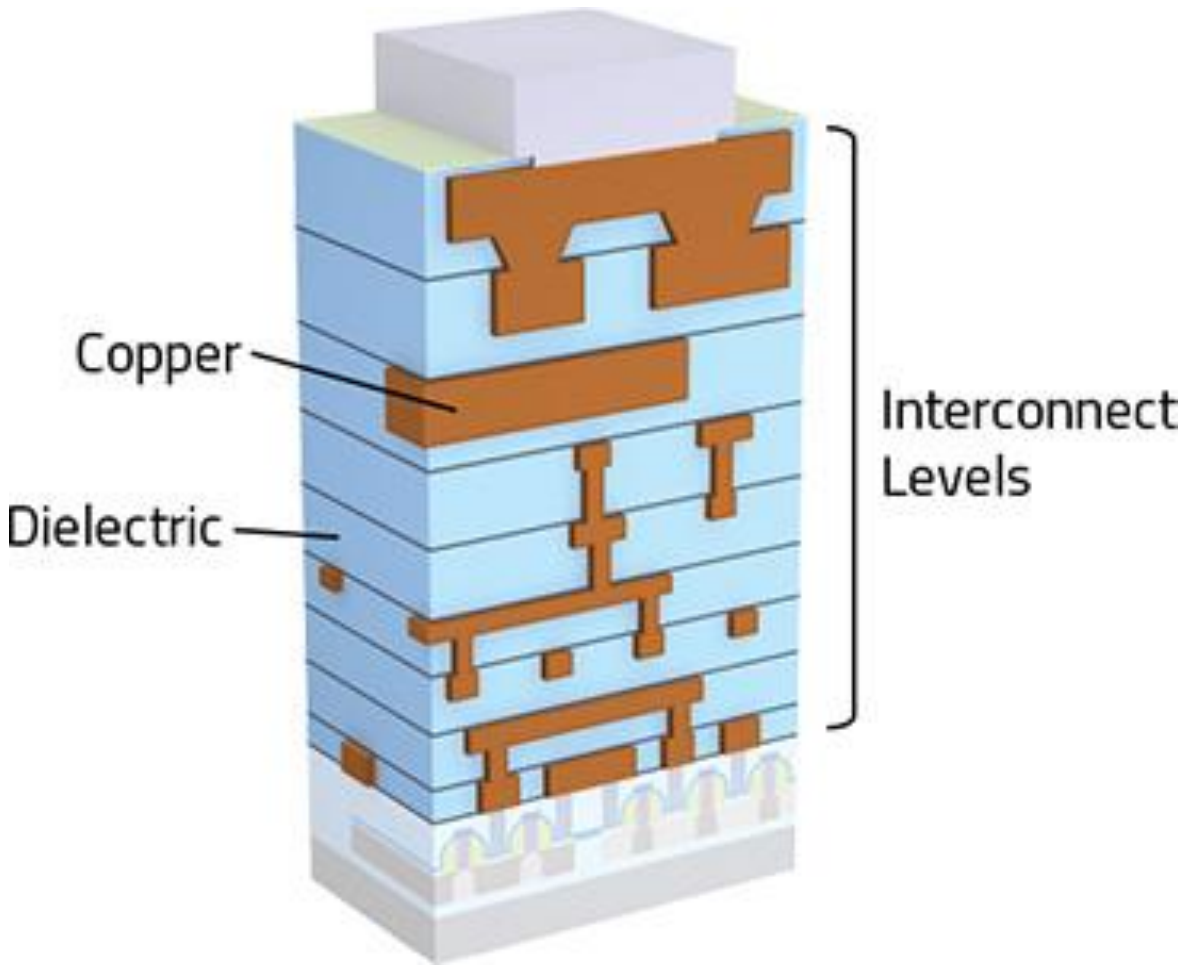
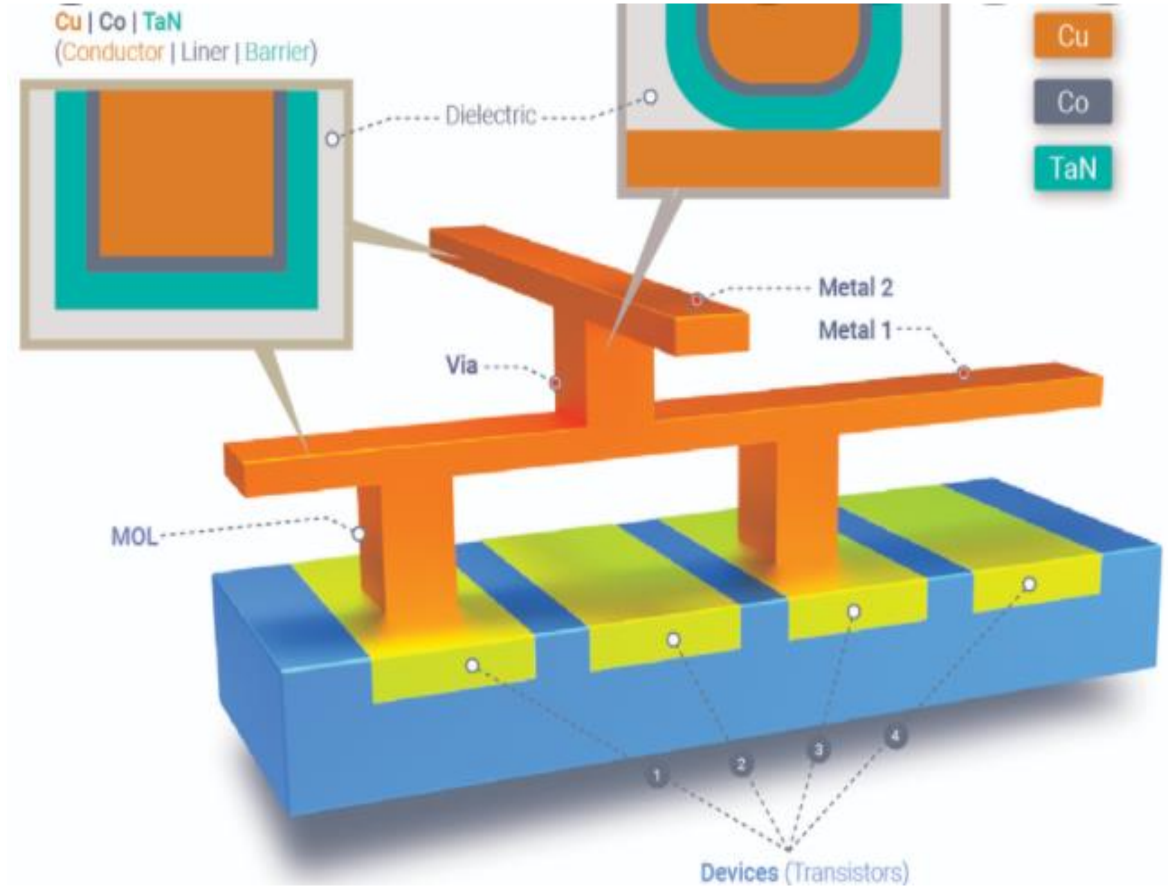


Figure 1.4 Routing the CBIC

- This ASIC uses two separate layers of metal interconnect (metal1 and metal2) running at right angles to each other (like traces on a printed-circuit board).
- Interconnections between logic cells uses spaces (called channels) between the rows of cells.
- ASICs may have three (or more) layers of metal allowing the cell rows to touch with the interconnect running over the top of the cells.


Copper
Dielectric
Interconnect Levels


Cu | Co | TaN
(Conductor | Liner | Barrier)
Dielectric
Cu
Co
TaN
Metal 2
Metal 1
Via
MOL
1
2
3
4
Devices (Transistors)

Gate-Array-Based ASICs

- In a gate-array-based ASIC, the transistors are predefined on the silicon wafer
- The predefined pattern of transistors is called the *base array*
- The smallest element that is replicated to make the base array (tiles on the floor) is called the *base* or *primitive cell.*
- The top few layers of metal, which define the interconnect between the transistors are defined by the designer in custom masks **- *Masked Gate Array (MGA)***
- Design is performed by connecting predesigned and characterized logic cells (macros) from a library
- After validation, automatic placement and routing are typically used to convert the macro-based design into a layout on the ASIC using primitive cells
- Types of MGAs:
    - Channeled Gate Array
    - Channelless Gate Array
    - Structured Gate Array

Channeled Gate Array

We leave space between the rows of transistors for Routing.

- Only the interconnect is customized
- The interconnect uses predefined spaces between rows of base cells
- Manufacturing lead time is between two days and two weeks

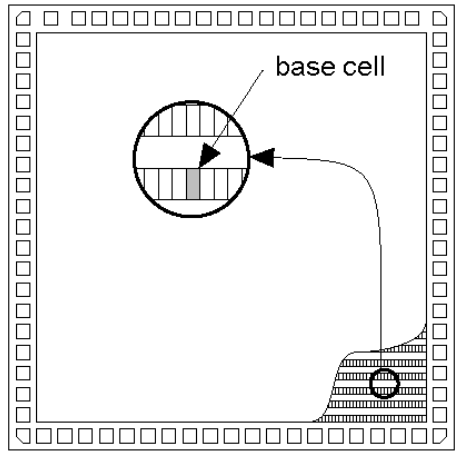


Figure 1.5 Channel gate-array die

Channelless Gate Array

There are no predefined areas set aside for routing - routing is over the top of the gate-array devices

- Only top few layers are customized the interconnect.
- Achievable logic density is higher than for channeled gate arrays
- Manufacturing lead time is between two days and two weeks

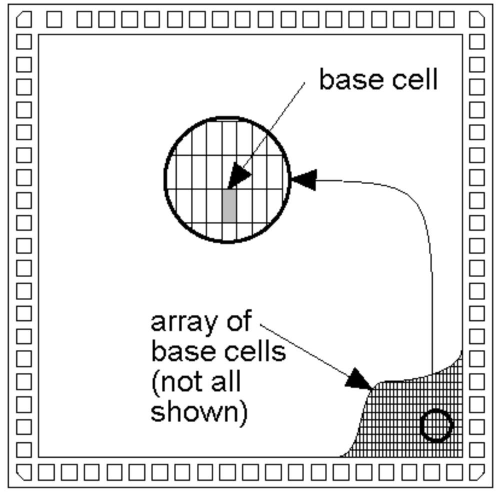


Figure 1.6 Sea-Of-Gates (SOG) array die

Structured Gate Array

Also known as embedded gate array.

- Combines the features of CBICs and MGA.
- MGA is fixed gate array base cell – implementation of memory is difficult and inefficient.
- Features:
  - Only the interconnect is customized
  - Custom blocks (the same for each design) can be embedded
  - Manufacturing lead time is between two days and two weeks.
- Advantages:
  - Improved area efficiency & increased performance from CBICs
  - Lower cost and faster turnaround from MGA

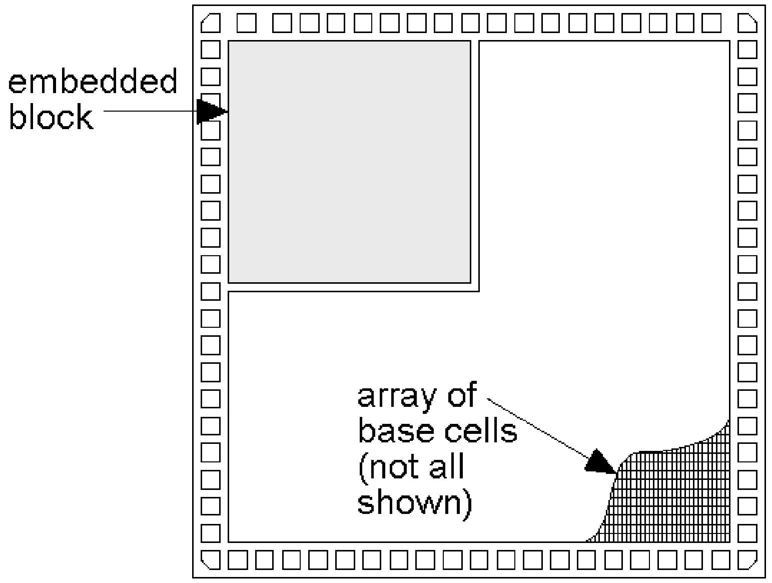


Figure 1.7 Gate array die with embedded block

Programmable Logic Devices

- Std. ICs available in std. configurations.
- sold in very high volume to many different customers
- Use different technologies to allow programming of the device.
  - Erasable PLD (EPLD)
  - Mask-programmed PLD

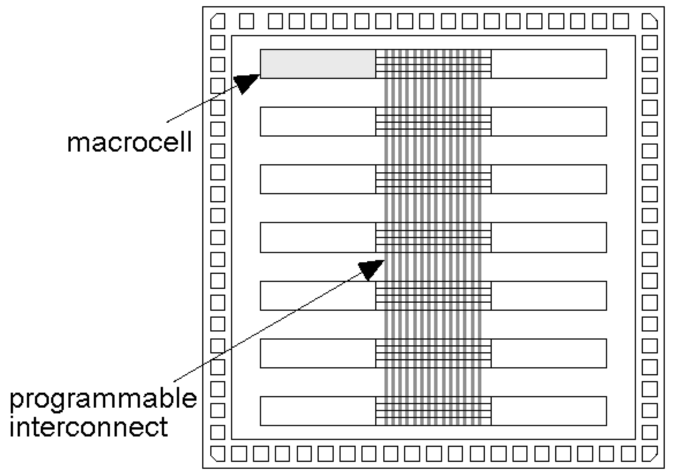


Figure 1.8 Programmable Logic Device (PLD) die

Features:

- No customized mask layers or logic cells
- Fast design turnaround
- A single large block of programmable interconnect
- A matrix of logic macrocells that usually consist of programmable array logic followed by a flip-flop or latch

- The macrocells typically consist of programmable array logic (PAL) followed by a flip-flop or latch.
- Connected using a large programmable interconnect block.

Field Programmable Gate Array

- Larger and more complex than PLDs
- Newest member of ASIC family replacing TTL in microelectronics
- Features:

- None of the mask layers are customized
- A method for programming the basic logic cells and the interconnect
- The core is a regular array of programmable basic logic cells that can implement combinational as well as sequential logic (flip-flops)
- A matrix of programmable interconnect surrounds the basic logic cells
- Programmable I/O cells surround the core
- Design turnaround is a few hours

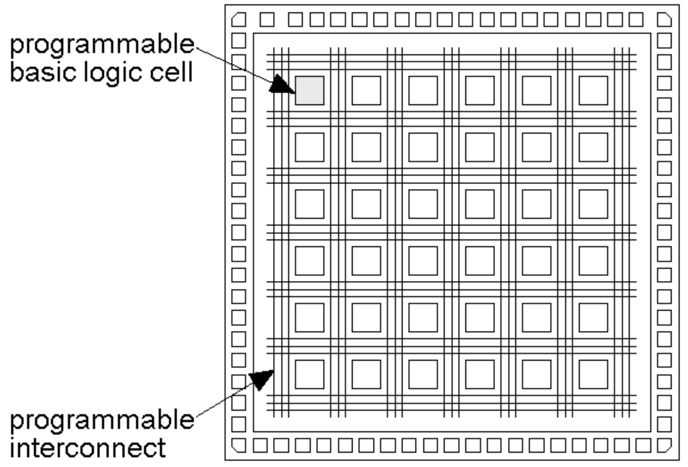


Figure 1.9 Field-Programmable Gate Array (FPGA) die

Design Flow

1. **Design entry** - Using a hardware description language (HDL) or schematic entry
2. ***Logic synthesis*** - Produces a netlist - logic cells and their connections
3. ***System partitioning*** - Divide a large system into ASIC-sized pieces
4. **Prelayout simulation** - Check to see if the design functions correctly
5. ***Floorplanning*** - Arrange the blocks of the netlist on the chip
6. ***Placement*** - Decide the locations of cells in a block
7. ***Routing*** - Make the connections between cells and blocks
8. ***Extraction*** - Determine the resistance and capacitance of the interconnect
9. **Post layout simulation** - Check to see the design still works with the added loads of the interconnect

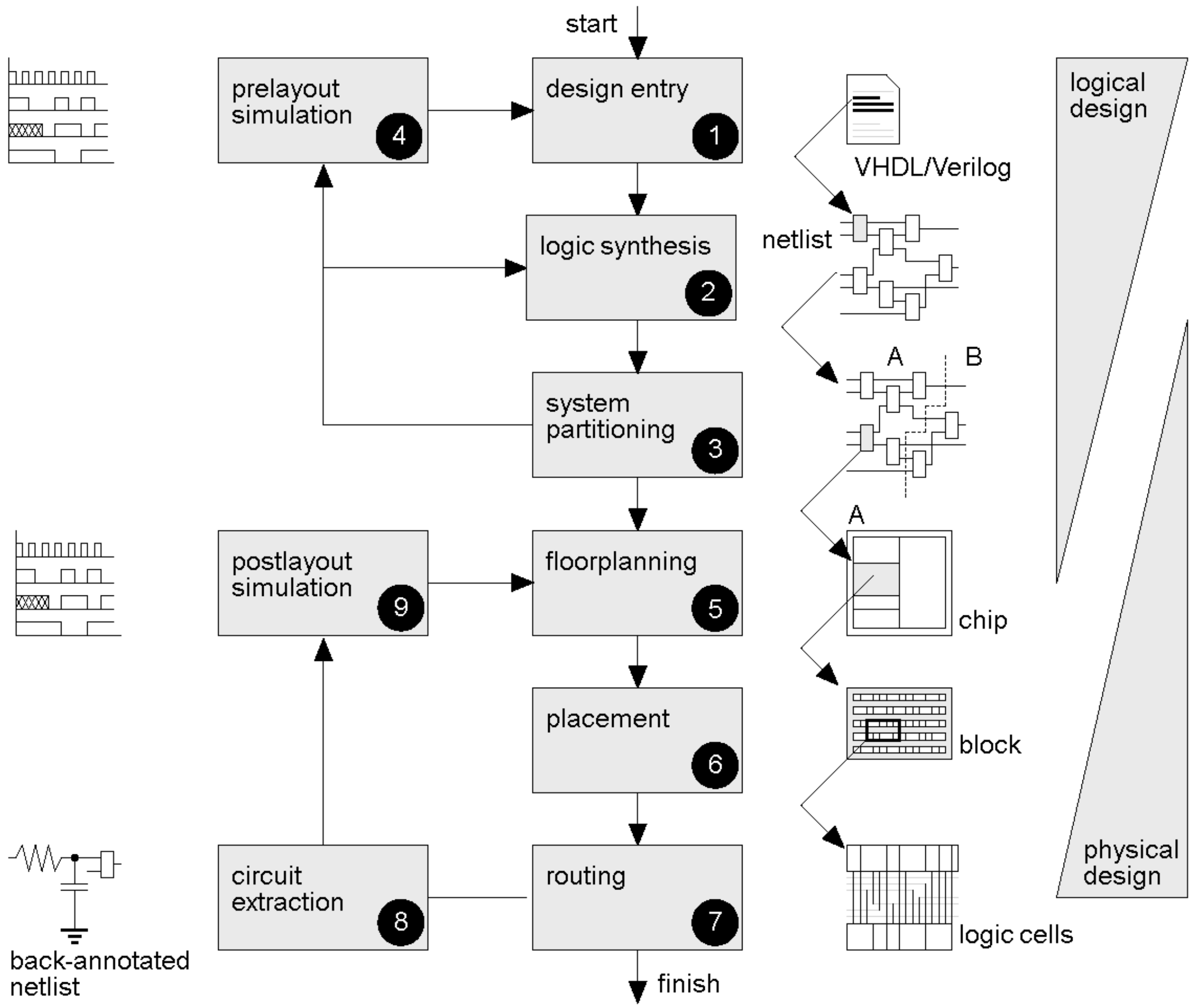


Figure 1.10 ASIC design flow

ASIC Cell Libraries

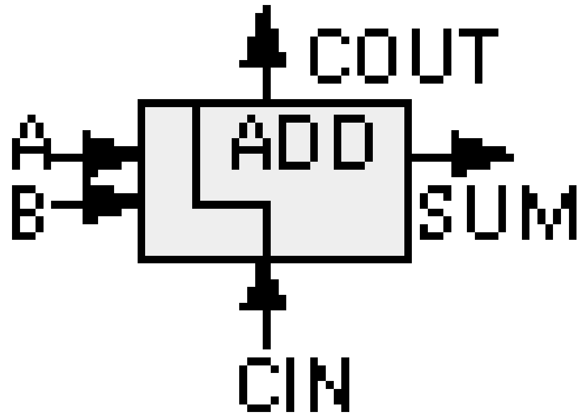
- Collection of low-level electronic logic functions such as AND, OR,INVERT, FFs,…
- A library of cells is used by the designer **to design the logic function for an ASIC**
- Options for cell library:
  - (1) Use a design kit from the ASIC vendor
    - Usually requires the use of ASIC vendor approved tools
    - Vendor may provide more of a "guarantee" that design will work
  - (2) Buy an cell library from third party
    - Library vendor is different from fabricator (foundry)
    - Called as Customer owned Tooling (COT)
    - Library may be approved by the foundry and they provides manufacturing details with no design help
  - (3) You can build your own cell library
    - Develop a cell library in-house.
    - Difficult and costly

ASIC Library Development

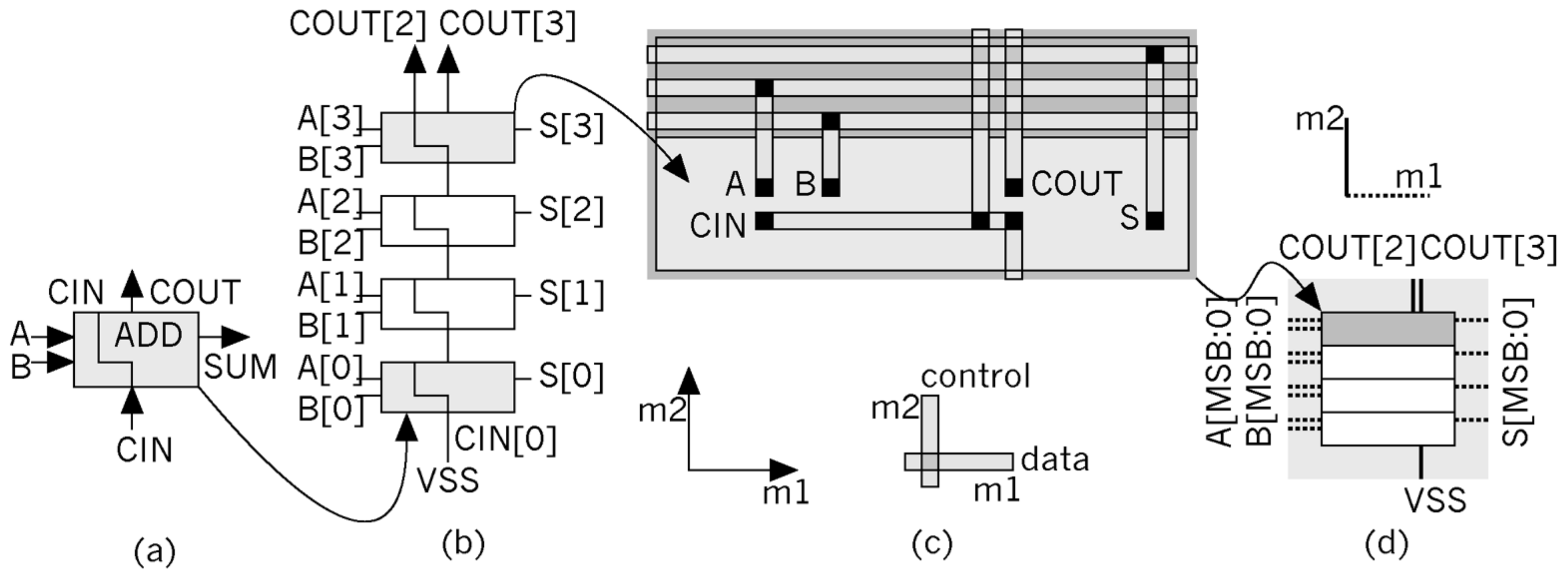
- A complete ASIC library (suitable for commercial use) must include the following for each cell and macro:
  - A physical layout
  - A VHDL or Verilog model
  - A detailed timing model
  - A test strategy
  - A circuit schematic
  - A cell icon (symbol)
  - A wire-load model
  - A routing model

Data path Logic Cells

- The logic that operates on data signals is called a datapath .
- The module ADD is a datapath cell or datapath element .



- To build an n -bit adder (that adds two n -bit numbers) : two functions, SUM and COUT with two data inputs (A, B) and a carry in, CIN.



A datapath adder. (a) A full adder (FA) cell. (b) A 4-bit adder. (c) Wiring layout using 2 level metal. (d) The datapath layout

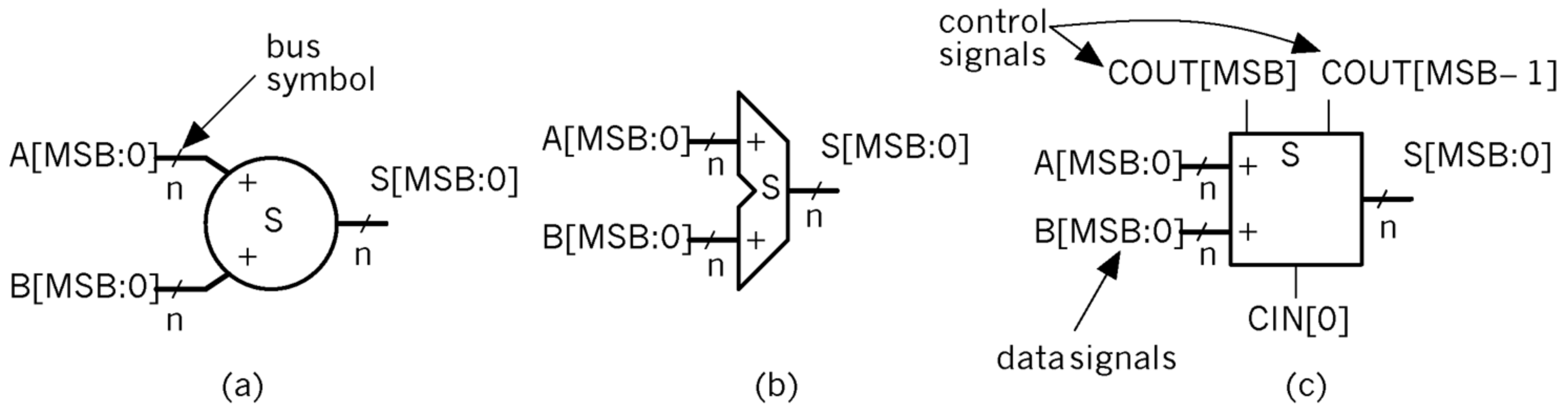
Advantages

- **Datapath layout automatically takes care of most of the interconnect between the cells** with the following advantages:
    - Regular layout produces ***predictable and equal delay*** for each bit.
    - ***Interconnect between cells*** can be built into each cell.

Disadvantages

- ***The overhead*** (ex: buffering and routing the control signals) ***can make a narrow*** (small number of bits) datapath larger and slower than a standard-cell (or even gate-array) implementation.
- Datapath ***cells have to be predesigned*** (otherwise we are using full-custom design) for use in a wide range of datapath sizes.
- Datapath cell design can be ***harder than designing gate-array macros*** or standard cells.
- ***Software to assemble*** a datapath ***is more complex*** and not as widely used as software for assembling standard cells or gate arrays.

Datapath Elements



Symbols for a datapath adder. (a) A generic symbol. (b) An alternate symbol. (c) A symbol with control lines. A data bus is shown by a heavy line (1.5 point) and
Control signals are shown as lightweight (0.5 point) lines.

- For a bus, A[31:0] denotes a 32-bit bus with A[31] as the leftmost or most-significant bit or MSB , and A[0] as the least-significant bit or LSB .
- Some schematic datapath symbols include only data signals and omit the control signals—but we must not forget them.

Adders

A	B	Cin	Sum	Cout
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

- To build an n -bit adder (that adds two n -bit numbers)
- The two functions, SUM and COUT with two data inputs (A, B) and a carry in, CIN.

$$\mathrm{SUM} = A \oplus B \oplus \mathrm{CIN} = \mathrm{SUM}(A, B, \mathrm{CIN}) = \mathrm{PARITY}(A, B, \mathrm{CIN})$$

$$\mathrm{COUT} = A \cdot B + A \cdot \mathrm{CIN} + B \cdot \mathrm{CIN} = \mathrm{MAJ}(A, B, \mathrm{CIN})$$

The sum uses the **parity function** ('1' if there are an odd numbers of '1's in the inputs).

The carry out, COUT, uses the 2-of-3 **majority function** ('1' if the majority of the inputs are '1').

- Different Adders:

Ripple carry adder	**Carry-lookahead adder**
Carry Save Adder	**Carry Select Adder**

A + B generates if and only if both A and B are 1. If we write G(A,B) to represent the binary predicate that is true if and only if A + B generates, we have:

$$\mathbf{G(A,B) = A \cdot B}$$

A + B propagates if and only if at least one of A or B is 1. If we write P(A,B) to represent the binary predicate that is true if and only if A + B propagates, we have:

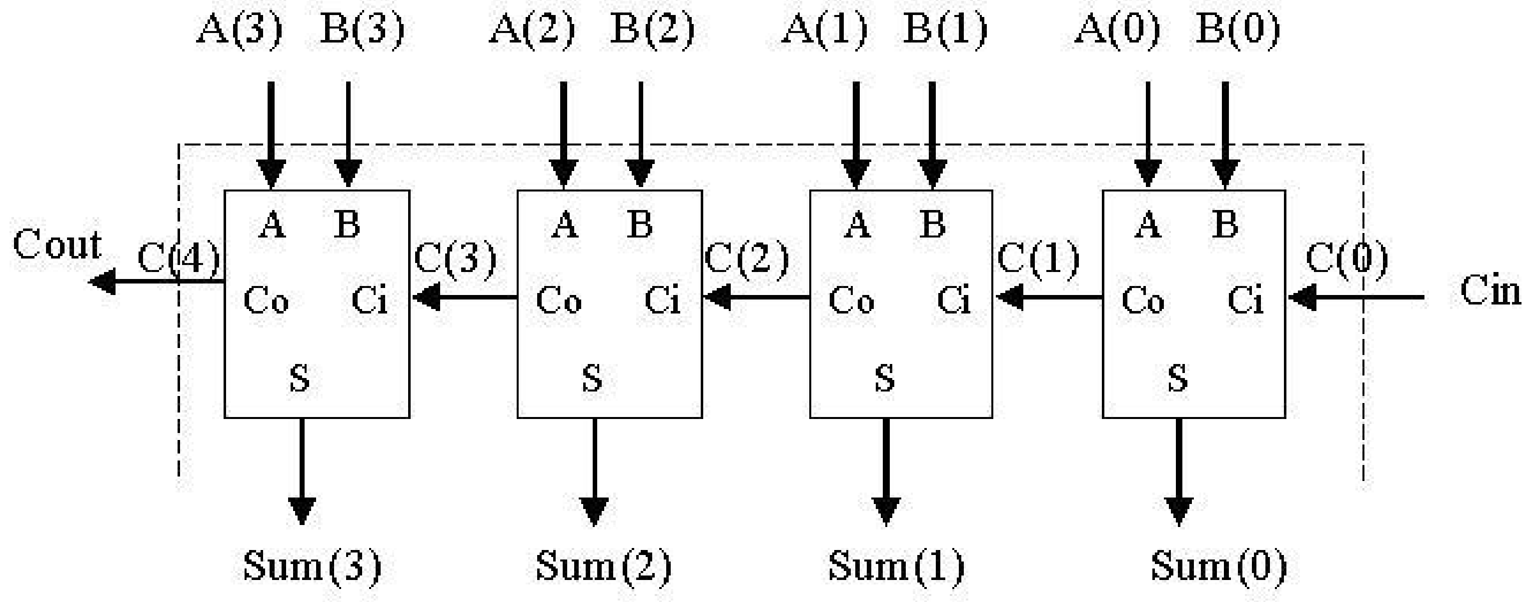
$$\mathbf{P(A,B) = A + B}$$

Some times, By considering input carry signal: $\mathbf{P[\,A,B\,] = A \oplus B}$

We can view addition in terms of generate , G[i], and propagate , P[i], signals.

method 1	**method 2**
$G[i] = A[i] \cdot B[i]$	$G[i] = A[i] \cdot B[i]$
$P[i] = A[i] \oplus B[i]$	$P[i] = A[i] + B[i]$
$C[i] = G[i] + P[i] \cdot C[i-1]$	$C[i] = G[i] + P[i] \cdot C[i-1]$
$S[i] = P[i] \oplus C[i-1]$	$S[i] = A[i] \oplus B[i] \oplus C[i-1]$

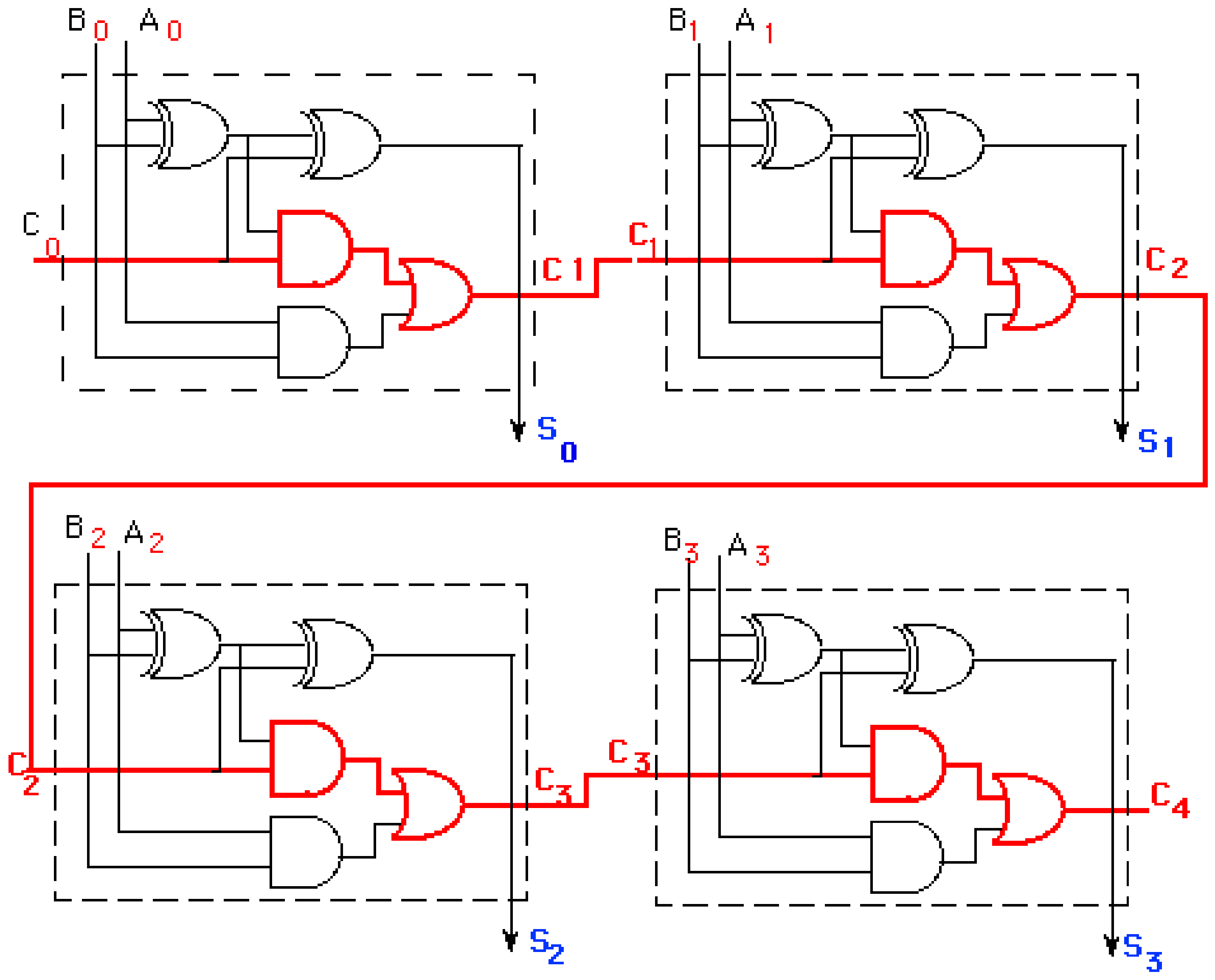
Ripple Carry Adder



4-bit **adder** with logic gates.

- It is possible to create a logical circuit using multiple full **adders** to add N-bit numbers.
- Each full **adder** inputs a C_{in}, which is the C_{out} of the previous **adder**. This kind of **adder** is called a **ripple-carry adder**
- since each **carry** bit "**ripples**" to the next full **adder**.

carry propagation through the circuit



Assume you want to add two operands A and B where

A= A3 A2 A1 A0

B=B3 B2 B1 B0

For example:

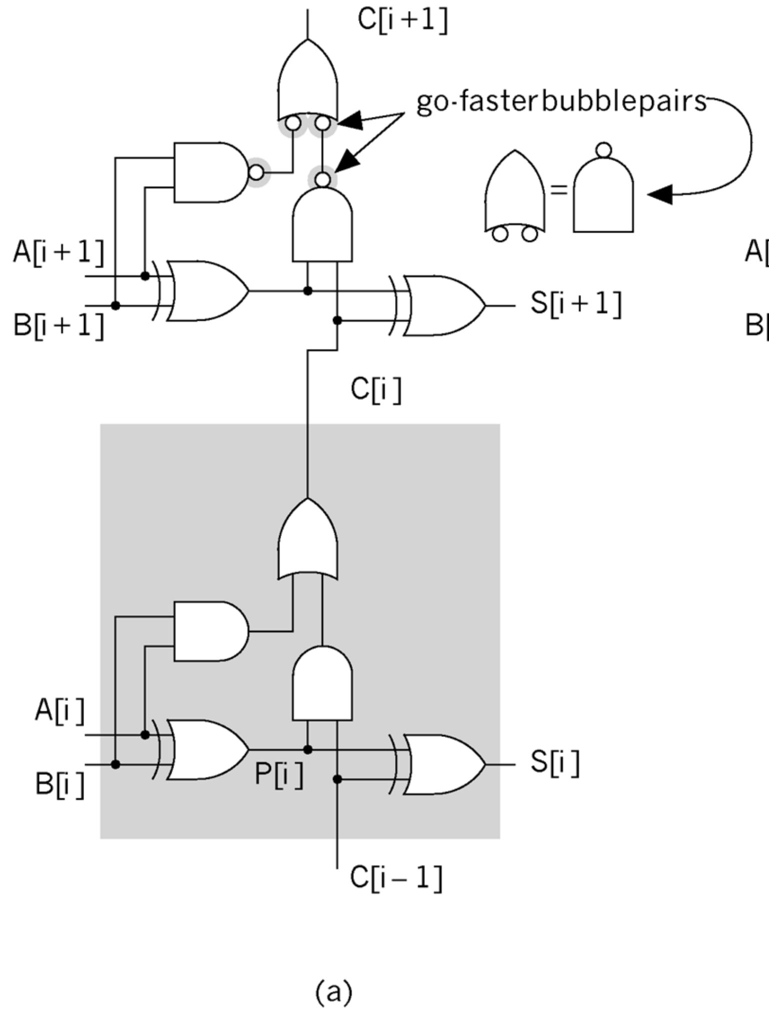
A= 1 0 1 1 +

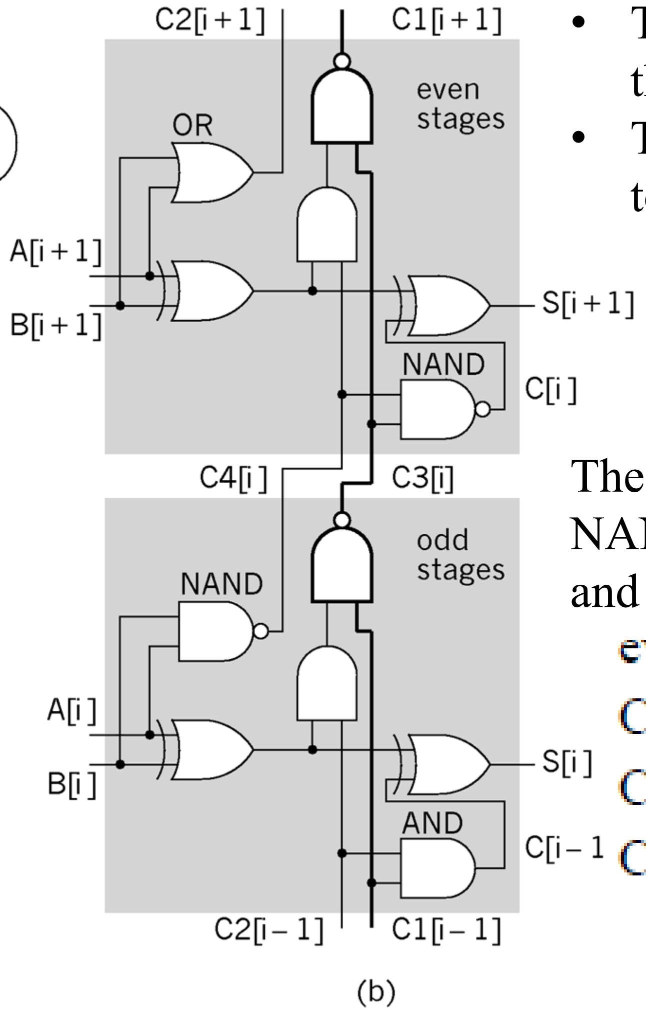
B= 1 1 0 1

A+B= 11 0 0 0

= C out S_3 S_2 S_1 S_0

Ripple Carry Adder





The ripple carry adder (RCA).
(a) A conventional RCA. (b) An implementation using alternate cells for even and odd bits.

- The delay of an n-bit RCA is limited by the propagation of the carry signal.
- This delay can reduced by using pairs of go-faster bubbles to change AND and OR gates to NAND gate.

$$C[i] = A[i]\cdot B[i] + P[i]\cdot C[i-1]$$
$$C[i] = (A[i] + B[i])\cdot(P[i]' + C[i-1])$$

These equations allow us to build the carry chain from 2-input NAND gates one per each cell, using different logic in even and odd stages.

even stages

$$C1[i]' = P[i]\cdot C3[i-1]\cdot C4[i-1]$$
$$C2[i] = A[i] + B[i]$$
$$C[i] = C1[i]\cdot C2[i]$$

odd stages

$$C3[i]' = P[i]\cdot C1[i-1]\cdot C2[i-1]$$
$$C4[i]' = A[i]\cdot B[i]$$
$$C[i] = C3[i]' + C4[i]'$$

Carry-Save Adder

Instead of propagating the carries through each stage of an RCA, we can use different approach known as CSA.

The CSA has 4 inputs as,

$$A1[\,i\,],\ A2[\,i\,],\ A3[\,i\,],\ CIN$$

and 3 outputs as,

$$S1[\,i\,],\ S2[\,i\,],\ COUT$$

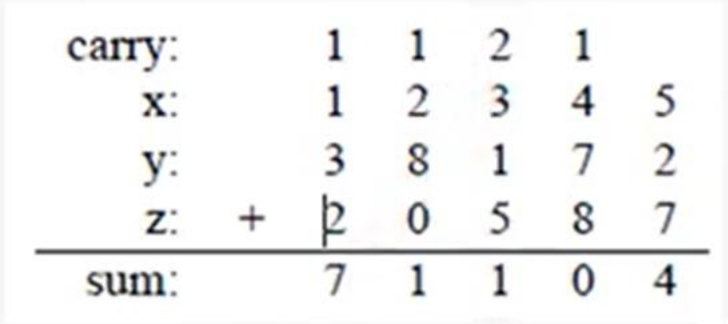
Can calculated by,

$$S1[\,i\,] = CIN\ ,$$

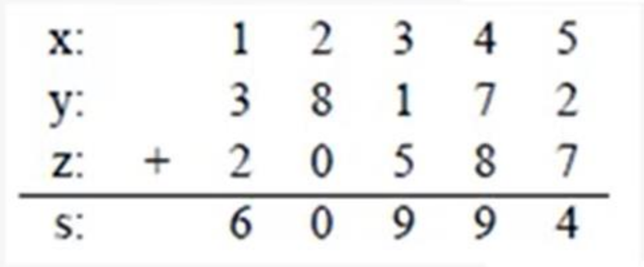
$$S2[\,i\,] = A1[\,i\,] \bullet A2[\,i\,] \bullet A3[\,i\,] = PARITY(A1[\,i\,],\ A2[\,i\,],\ A3[\,i\,])$$

$$COUT = A1[\,i\,] \cdot A2[\,i\,] + [(A1[\,i\,] + A2[\,i\,]) \cdot A3[\,i\,]] = MAJ(A1[\,i\,],\ A2[\,i\,],\ A3[\,i\,])$$

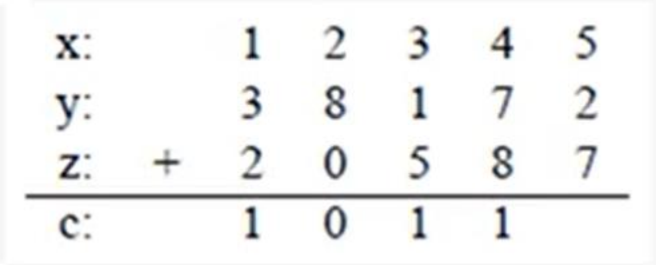
CARRY SAVE ADDER: MULTIPLE OPERANDS' ADDITION



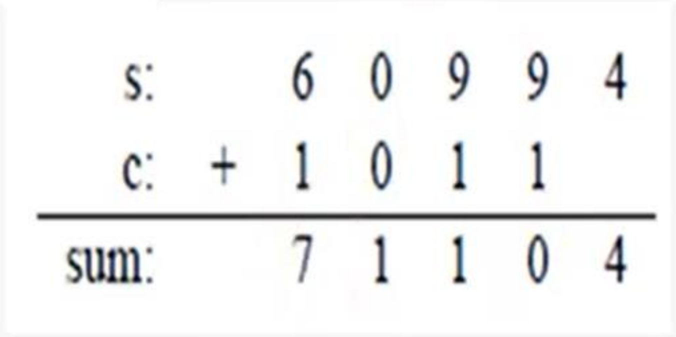
carry:		1	1	2	1	
x:		1	2	3	4	5
y:		3	8	1	7	2
z:	+	2	0	5	8	7
sum:		7	1	1	0	4



x:		1	2	3	4	5
y:		3	8	1	7	2
z:	+	2	0	5	8	7
s:		6	0	9	9	4

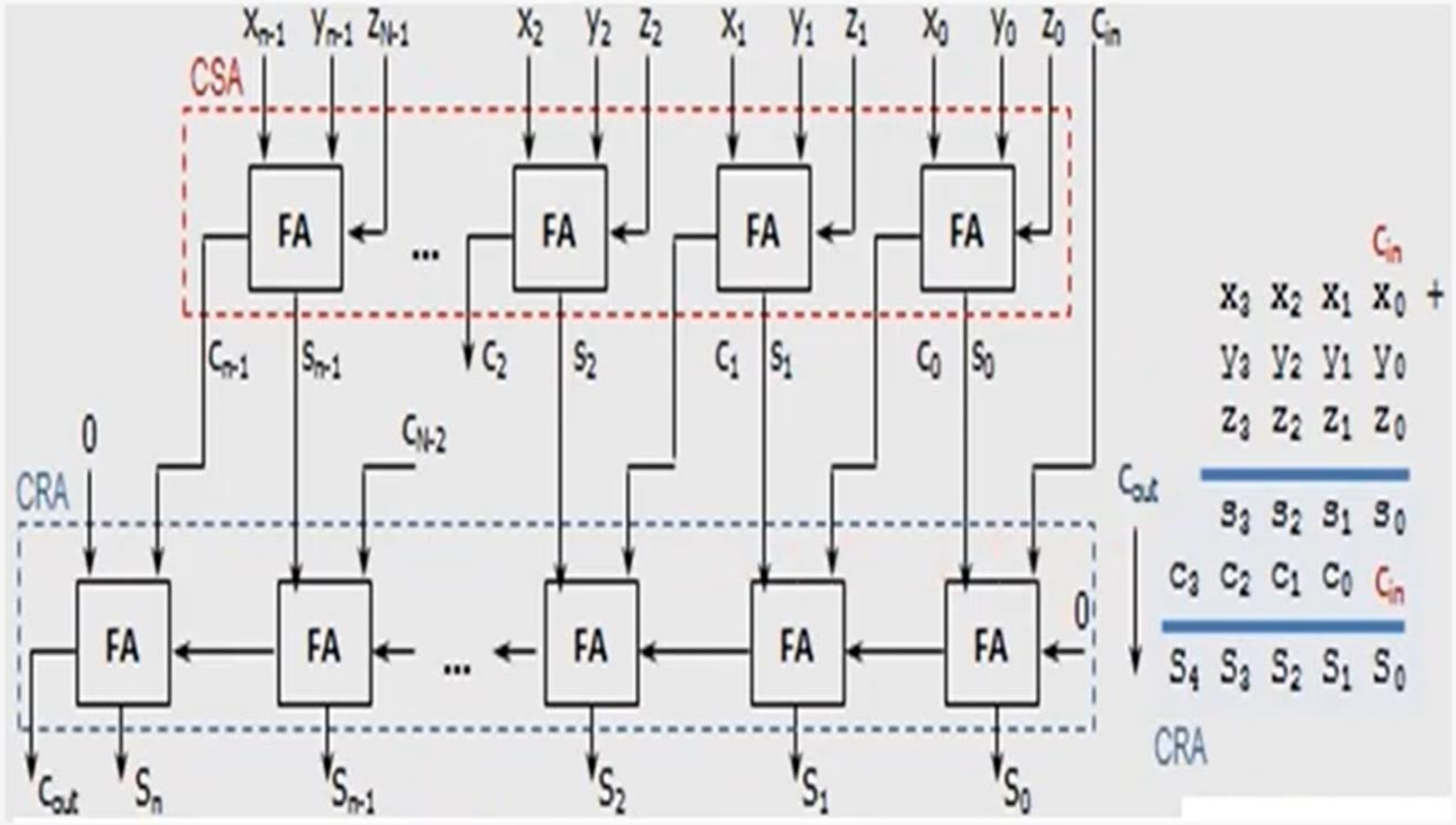


x:		1	2	3	4	5
y:		3	8	1	7	2
z:	+	2	0	5	8	7
c:		1	0	1	1	



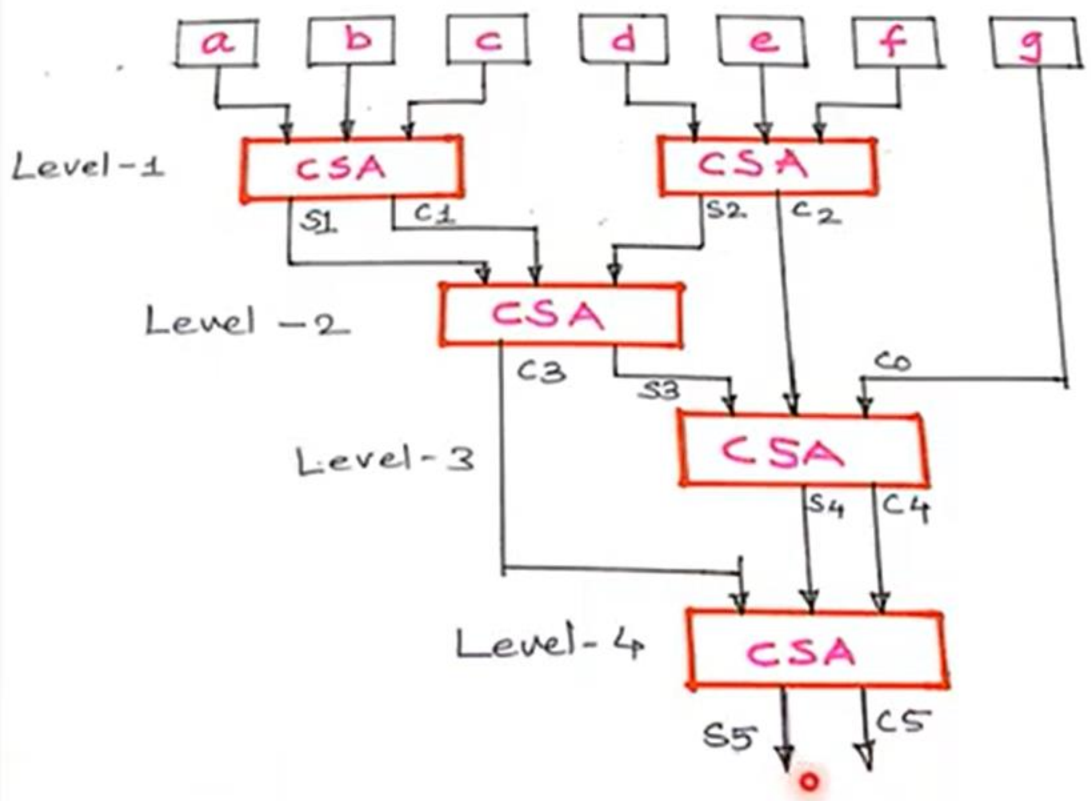
s:			6	0	9	9	4
c:	+	1	0	1	1	
sum:		7	1	1	0	4

N-BIT CSA ADDER WITH CARRY IN



X_{n-1} Y_{n-1} Z_{N-1}
X_2 Y_2 Z_2
X_1 Y_1 Z_1
X_0 Y_0 Z_0 C_{in}
CSA
FA
C_{n-1} S_{n-1} C_2 S_2 C_1 S_1 C_0 S_0
0
C_{N-2}
CRA
0
C_{out} S_n S_{n-1} S_2 S_1 S_0
C_{in}
x_3 x_2 x_1 x_0 +
y_3 y_2 y_1 y_0
z_3 z_2 z_1 z_0
C_{out}
s_3 s_2 s_1 s_0
c_3 c_2 c_1 c_0 C_{in}
S_4 S_3 S_2 S_1 S_0
CRA

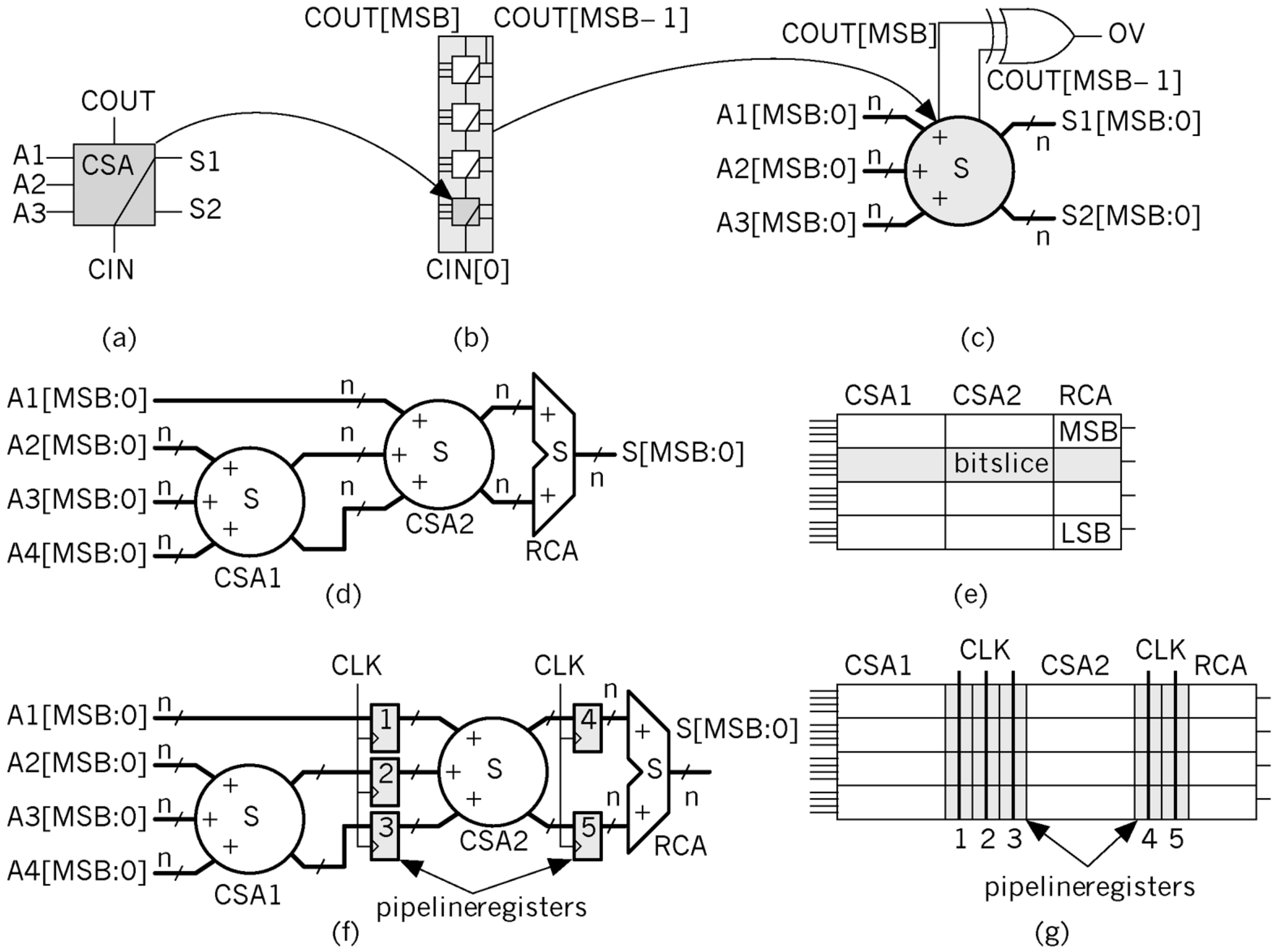
7 N-BIT WORDS TO 2 N-BIT WORD REDUCTION USING 5 CSAs



7 n-bit binary nos.

a	$a_{n-1}\ a_{n-2} \cdots a_2\, a_1\, a_0$
b	$b_{n-1}\ b_{n-2} \cdots b_2\, b_1\, b_0$
c	$c_{n-1}\ c_{n-2} \cdots c_2\, c_1\, c_0$
d	$d_{n-1}\ d_{n-2} \cdots d_2\, d_1\, d_0$
e	$e_{n-1}\ e_{n-2} \cdots e_2\, e_1\, e_0$
f	$f_{n-1}\ f_{n-2} \cdots f_2\, f_1\, f_0$
g	$g_{n-1}\ g_{n-2} \cdots g_2\, g_1\, g_0$

Carry-Save Adder



The carry-save adder (CSA).

(a)A CSA cell.

(b)A 4-bit CSA.

(c)Symbol for a CSA.

(d)A 4-input CSA.

(e)The datapath for a 4-bit adder using CSAs.

(f) A pipelined adder.

(g)The datapath for the pipelined version.

- The problem with an RCA is that every stage has to wait to make its carry decision, C[i], until the previous stage has calculated C[i -1].
- If we examine the propagate signals we can bypass this critical path.
- Example, to bypass the carries for bits 4 to 7(stages 5 to 8) of an adder

  BYPASS = P[4].P[5].P[6].P[7]
- Then use a MUX as : **C[7] = (G[7] + P[7] · C[6]) · BYPASS' + C[3] · BYPASS**
- Adders based on this principle are called **Carry-bypass Adders (CBA)** [Sato et al.,1992].

- Instead of checking the propagate signals we can check the inputs.
- Example we can compute

  SKIP = (A[i -1] • B[i -1]) + (A[i] • B[i])
- Then use a 2:1 MUX to select C[i].

  CSKIP[i] = (G[i] + P[i] · C[i -1]) · SKIP' + C[i -2] · SKIP
- Called as **Carry-skip Adder** [Keutzer, Malik, and Saldanha, 1991; Lehman, 1961].

Carry-bypass and carry-skip adders may include redundant logic is not optimized away during logic synthesis.

- Carry-bypass, Carry-skip architectures are **used to split the adders into different blocks.** (12 bit adder into 3 or 4 blocks).
- **Mux is available between each blocks** to select the sum or carry bit.
- **Delay of these adders are depends on the delay of the MUX** between each block.
- **Adjusting the block size reduces the delay** of large adders.

Look ahead Carry Adder

A **carry-lookahead adder** improves speed by reducing the amount of time required to determine **carry** bits.

$G(A,B) = A . B$
$P(A,B) = A + B$

$COUT = A \cdot B + A \cdot CIN + B \cdot CIN$

$\mathbf{C[\,i\,] = G[\,i\,] + P[\,i\,] \cdot C[\,i-1]}$

If we evaluate the above equation recursively for i = 1,2,3…, we get the following:

$C[1]=G[1] + P[1] \cdot C[0] =G[1] + P[1] \cdot G[0]$ {where, $C[0]=G[0]$}

$C[2]=G[2] + P[2] \cdot G[1] + P[2] \cdot P[1] \cdot G[0]$

$C[3]=G[3] + P[3] \cdot G[2] + P[3] \cdot P[2] \cdot G[1] + P[3] \cdot P[2] \cdot P[1] \cdot G[0]$.

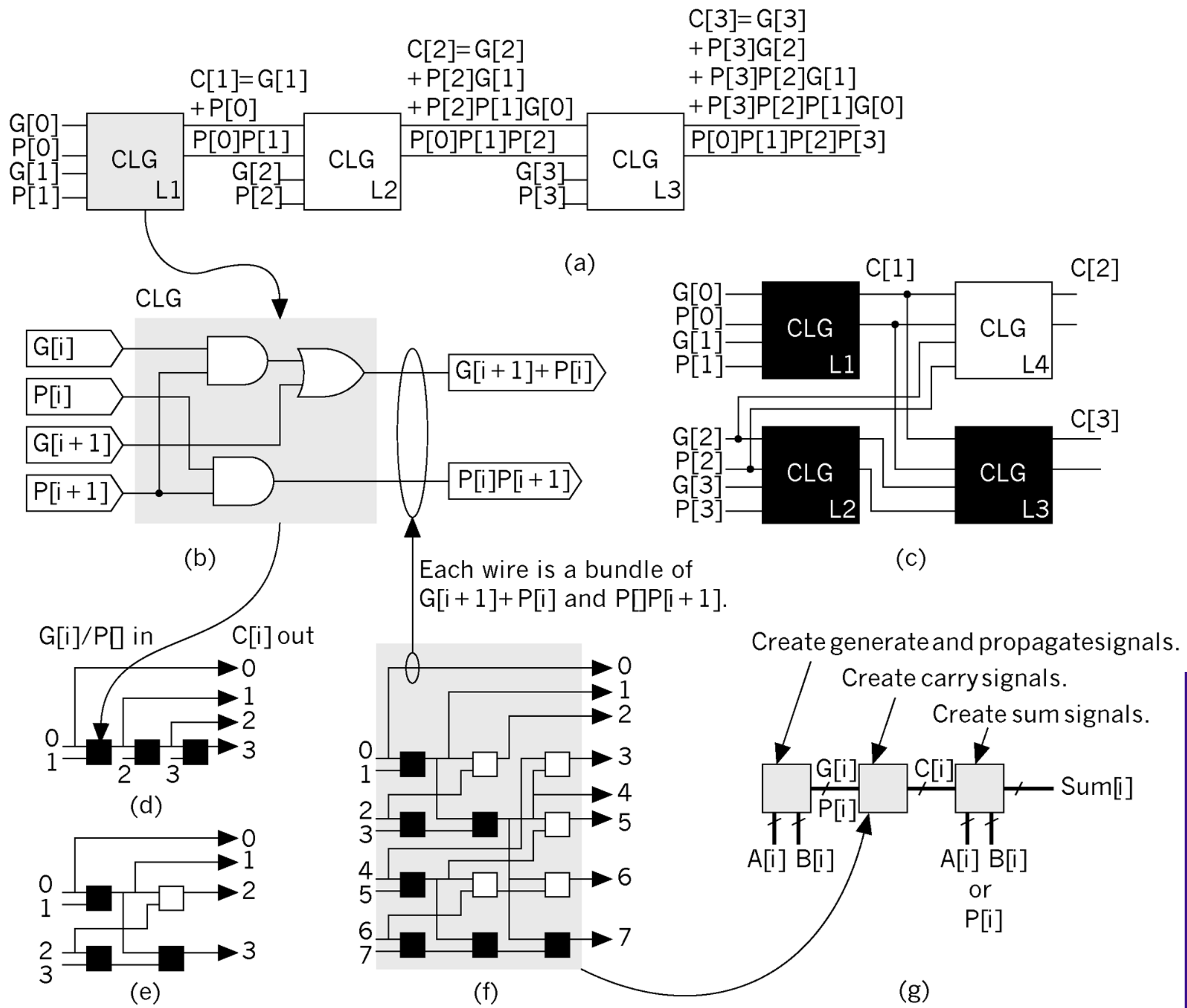
.

.

.

further equations become **more complex** but take **longer to calculate**, and the **logic becomes less regular** when implemented using cells with a limited number of inputs. **But a regular logical structure simplifies design.**

Look ahead Carry Adder



The carry-look ahead adder.

(a) Carry generation.

(b) Cell to generate look-ahead terms.

(c) Arrangement of cells.

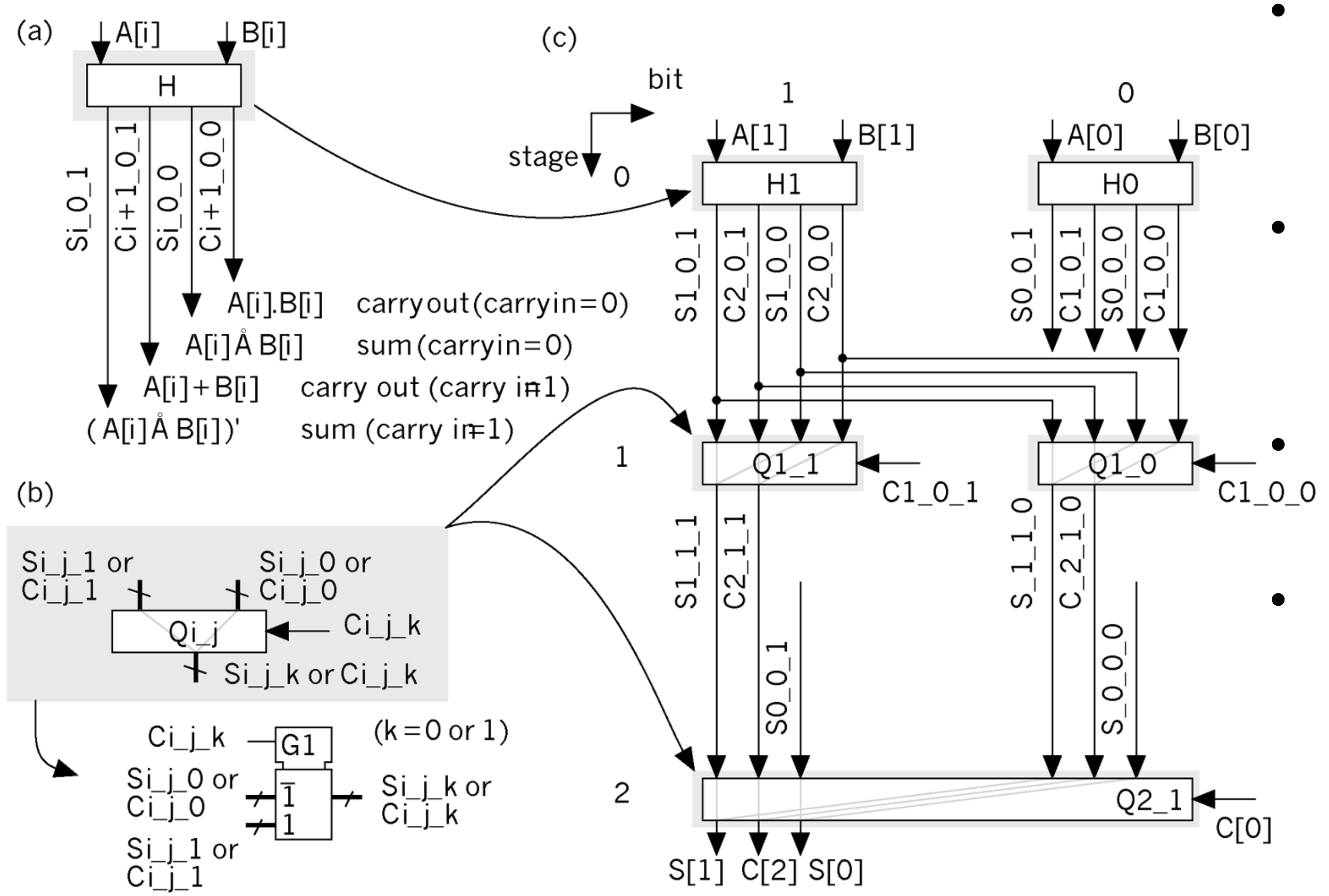
(d) and (e) Simplified representations of parts *a* and *c*.

(f) The look ahead logic for an 8 bit adder.

(g) An 8 bit CLA.

- **Advantage** of this adder is that **delays from the inputs to the outputs are more nearly equal than in other adders.**
- This tends **to reduce the number of unwanted and unnecessary switching events** and thus reduces power dissipation.

Carry Select Adder



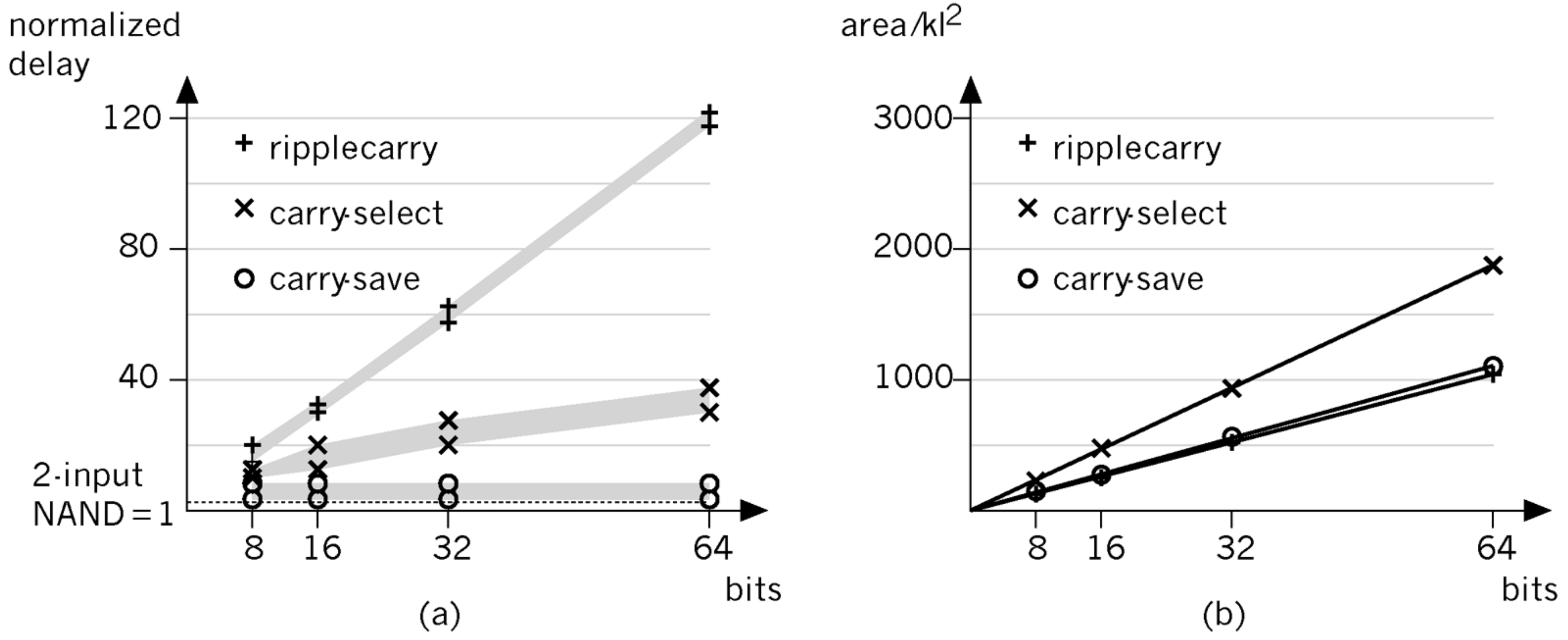
Ci_j_k = carry in to the ith bit assuming the carry in to the jth bit is k (k = 0 or 1)
Si_j_k = sum at the ith bit assuming the carry in to the jth bit is k (k = 0 or 1)

The conditional-sum adder
(a) A 1-bit conditional adder. (b) The multiplexer to select sums and carries.
(c) A 4-bit conditional-sum adder

Idea behind a carry-select adder (CSA) as follows:

- Suppose we have an n-bit adder that **generates two sums:** One sum assumes a **carry-in condition of '0'**, the other sum assumes **a carry-in condition of '1'.**
- We can **split this n -bit adder into two** as i –bit adder for the i LSBs and an (n-i)-bit adder for the n -i MSBs.
- Both of the smaller adders generate two conditional sums as well as true and complement carry signals.
- The two (true and complement) **carry signals from the LSB adder are used to select between the two conditional sums from the MSB adder using two-input MUXes**. This is a **conditional-sum adder** (also often abbreviated to CSA)[Sklansky, 1960].
- We can recursively apply this technique. Ex: **split a16-bit adder using i = 8 and n = 8; then we can split one or both 8 bit adders again and so on.**

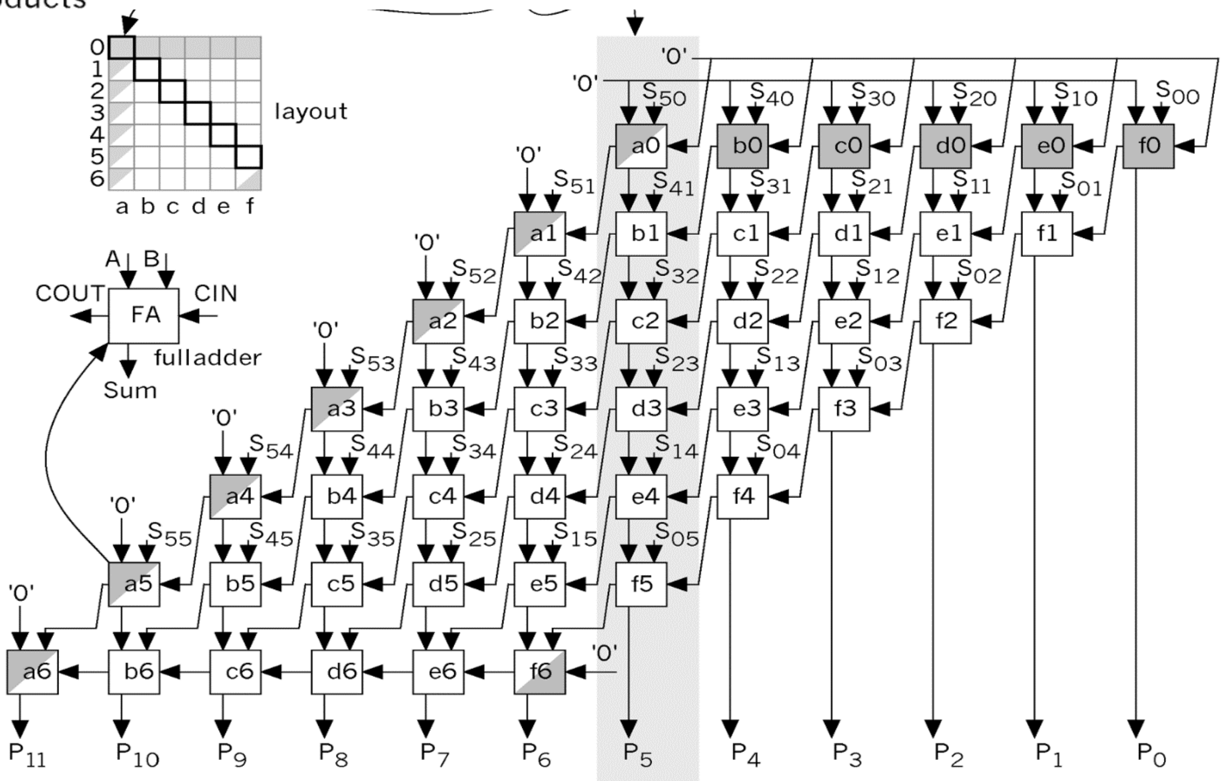
Comparison of Adder Implementations



Delay and area comparison for datapath adders. (a) Delay normalized to a two-input NAND logic cell delay. (b) Adder area.

Array Multiplier

						A_5	A_4	A_3	A_2	A_1	A_0	multiplicand
						B_5	B_4	B_3	B_2	B_1	B_0	multiplier
		summands				A_5B_0	A_4B_0	A_3B_0	A_2B_0	A_1B_0	A_0B_0	
		$S_{ij}=A_iB_j$			A_5B_1	A_4B_1	A_3B_1	A_2B_1	A_1B_1	A_0B_1		
				A_5B_2	A_4B_2	A_3B_2	A_2B_2	A_1B_2	A_0B_2			partialproducts
			A_5B_3	A_4B_3	A_3B_3	A_2B_3	A_1B_3	A_0B_3				
		A_5B_4	A_4B_4	A_3B_4	A_2B_4	A_1B_4	A_0B_4					
	A_5B_5	A_4B_5	A_3B_5	A_2B_5	A_1B_5	A_0B_5						
P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	product= sum of partia products



layout
0 1 2 3 4 5 6
a b c d e f
A B
COUT
CIN
FA
fulladder
Sum
'0'
S_{50} S_{40} S_{30} S_{20} S_{10} S_{00}
S_{51} S_{41} S_{31} S_{21} S_{11} S_{01}
S_{52} S_{42} S_{32} S_{22} S_{12} S_{02}
S_{53} S_{43} S_{33} S_{23} S_{13} S_{03}
S_{54} S_{44} S_{34} S_{24} S_{14} S_{04}
S_{55} S_{45} S_{35} S_{25} S_{15} S_{05}
a0 b0 c0 d0 e0 f0
a1 b1 c1 d1 e1 f1
a2 b2 c2 d2 e2 f2
a3 b3 c3 d3 e3 f3
a4 b4 c4 d4 e4 f4
a5 b5 c5 d5 e5 f5
a6 b6 c6 d6 e6 f6
P_{11} P_{10} P_9 P_8 P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0

A 6-bit array multiplier using a final carry-propagate adder.

To multiply 8bit binary number A by B=11111 (31), it is easier to multiply by **Canonical Signed Digit vector (CSD vector).**

The Canonic Signed Digit representation requires that no adjacent digits are nonzero. That is, the combinations 1 1; -1 1; 1 -1; or -1 -1 do not occur.

Example:

binary value of decimal 31 (B) = 1 1 1 1 1

CSD vector of decimal 31 (D) = 1 0 0 0 0 -1

B has a weight of 5, but D has a weight of 2 — **and saves hardware**

The reduction of non-zero digits in this scheme allows replacing full multipliers with ones using shifts and adds, resulting in more efficient implementations

We can recode (or encode) any binary number, B, as a CSD vector, D,

$$\boxed{D_i = B_i + C_i - 2C_{i+1}}$$

where C_{i+1} is the carry from the sum of $B_{i+1} + B_i + C_i$ (we start with $C_0 = 0$).

If B=011 (B_2=0, B_1=1, B_0=1; decimal 3), then:

$$D_0 = B_0 + C_0 - 2C_1 = 1 + 0 - 2 = \overline{1},$$
$$D_1 = B_1 + C_1 - 2C_2 = 1 + 1 - 2 = 0,$$
$$D_2 = B_2 + C_2 - 2C_3 = 0 + 1 - 0 = 1,$$

so that D= $10\overline{1}$ (decimal 4–1=3).

We can use a **radix** other than 2, for example **Booth encoding** (radix-4):

B=101001 (decimal =–23) $\Rightarrow$ E=$\overline{1}\ \overline{2}1$ (decimal –16–8+1=–23)

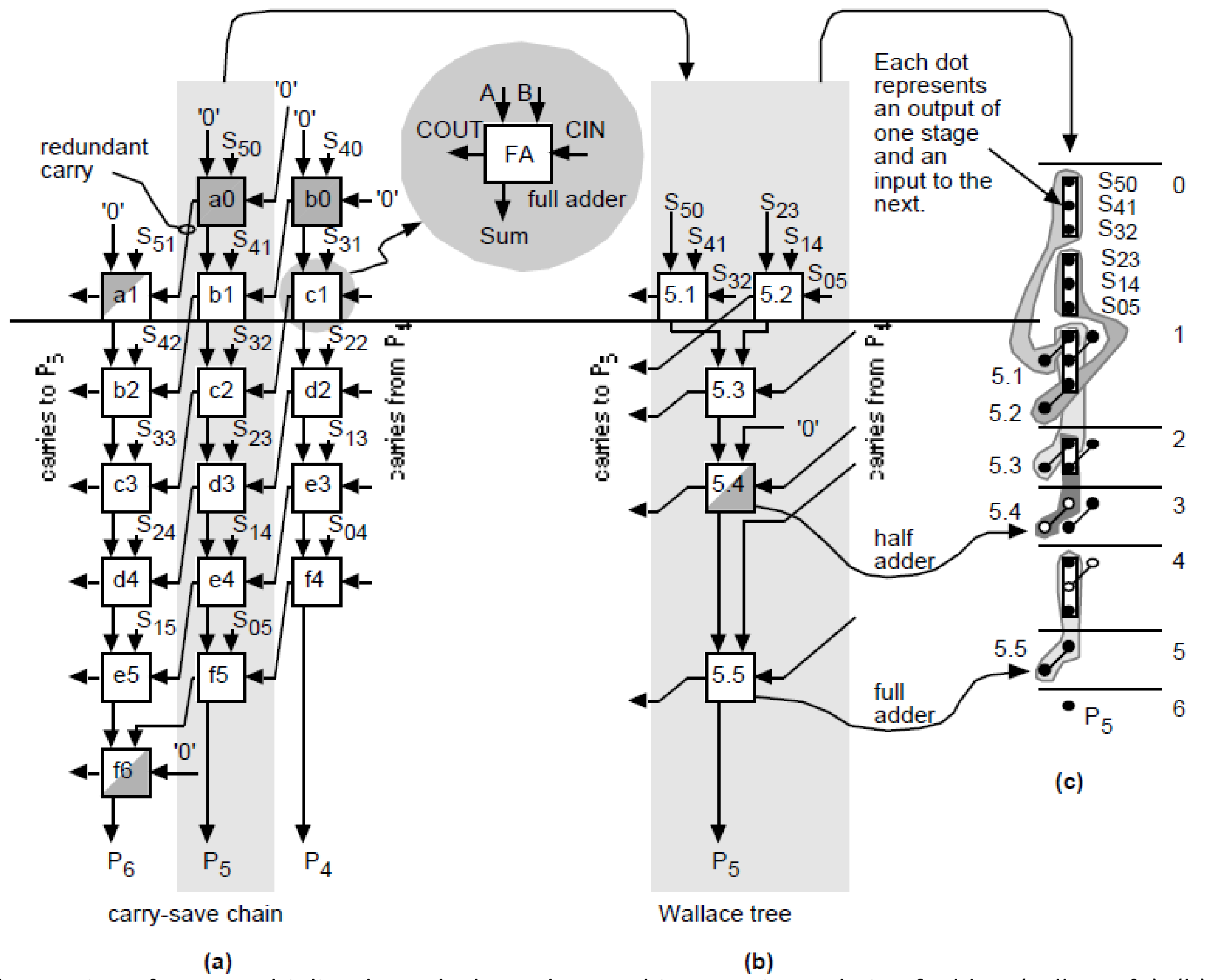
B=01011 (eleven) $\Rightarrow$ E=$1\overline{1}\ \overline{1}$ (16–4–1)

B=101 $\Rightarrow$ E=$\overline{1}1$

Next we turn our attention to improving the speed of addition in the CSA array.

Wallace-tree multiplier

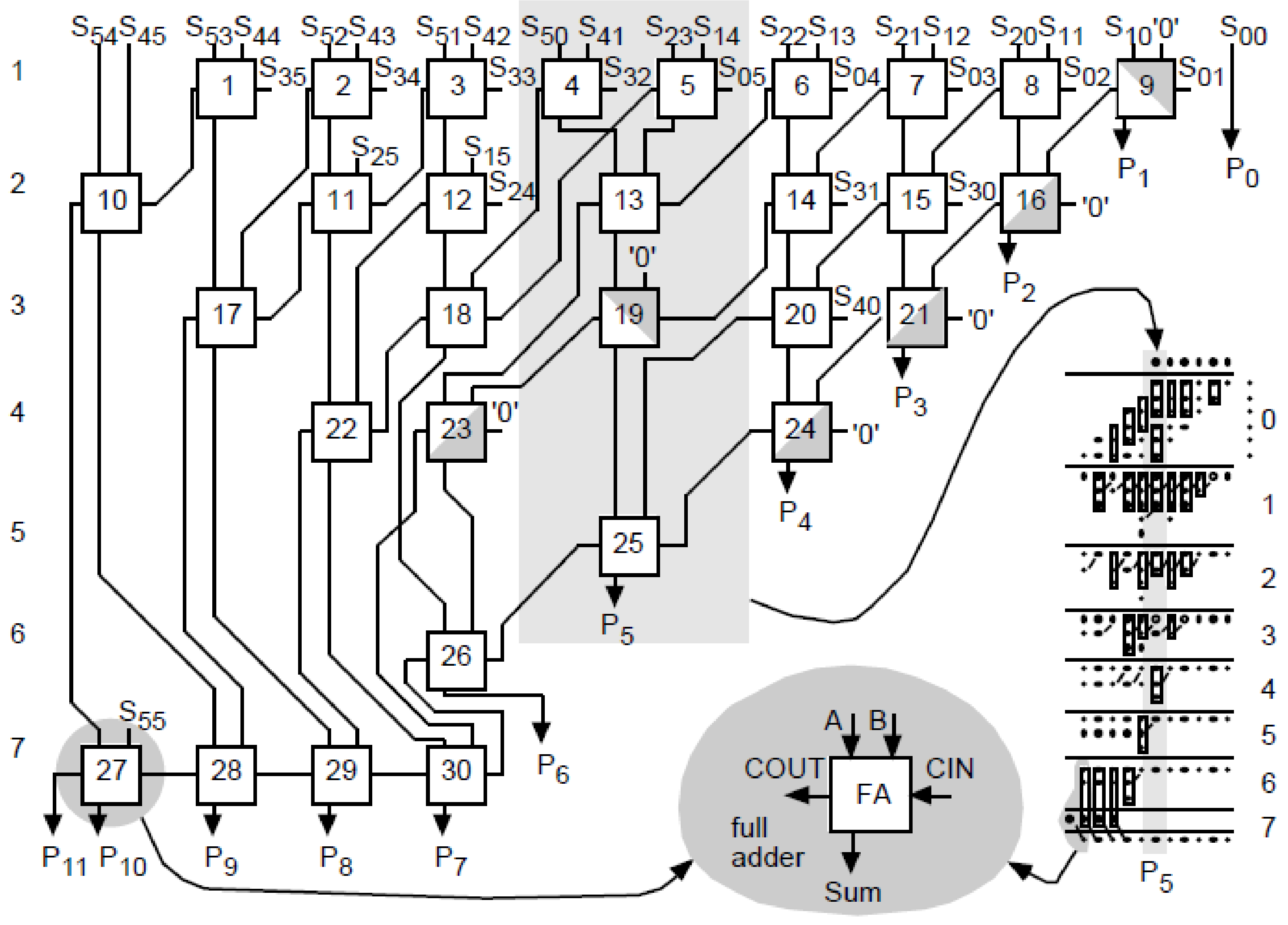
In a Wallace-tree multiplier we work forward from the multiplier inputs, compressing the number of signals to be added at each stage



(a)The portion of array multiplier that calculates the sum bit, P5, using a chain of adders (cells a0-f5). (b)We can collapse this chain to a Wallace tree (cells 5.1-5.5). (c)The stages of multiplication.

- The holes or dots are the outputs of one stage (and the inputs of the next).
- At each stage we have the following three choices:
  1. sum three outputs using a full adder (denoted by a box enclosing three dots)
  2. sum two outputs using a half adder (a box with two dots)
  3. pass the outputs directly to the next stage. The two outputs of an adder are joined by a diagonal line (full adders use black dots, half adders white dots).
- The object of the game is to choose (1), (2), or (3) at each stage to maximize the performance of the multiplier.

In a Wallace-tree multiplier we work forward from the multiplier inputs, compressing the number of signals to be added at each stage


S54 S45 S53 S44 S52 S43 S51 S42 S50 S41 S23 S14 S22 S13 S21 S12 S20 S11 S10 '0' S00
S35 S34 S33 S32 S05 S04 S03 S02 S01
S25 S15 S24 S31 S30 S40 S55
'0'
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11
1 2 3 4 5 6 7
COUT
CIN
A B
FA
full adder
Sum

A Wallace-tree multiplier works forward from the multiplier inputs

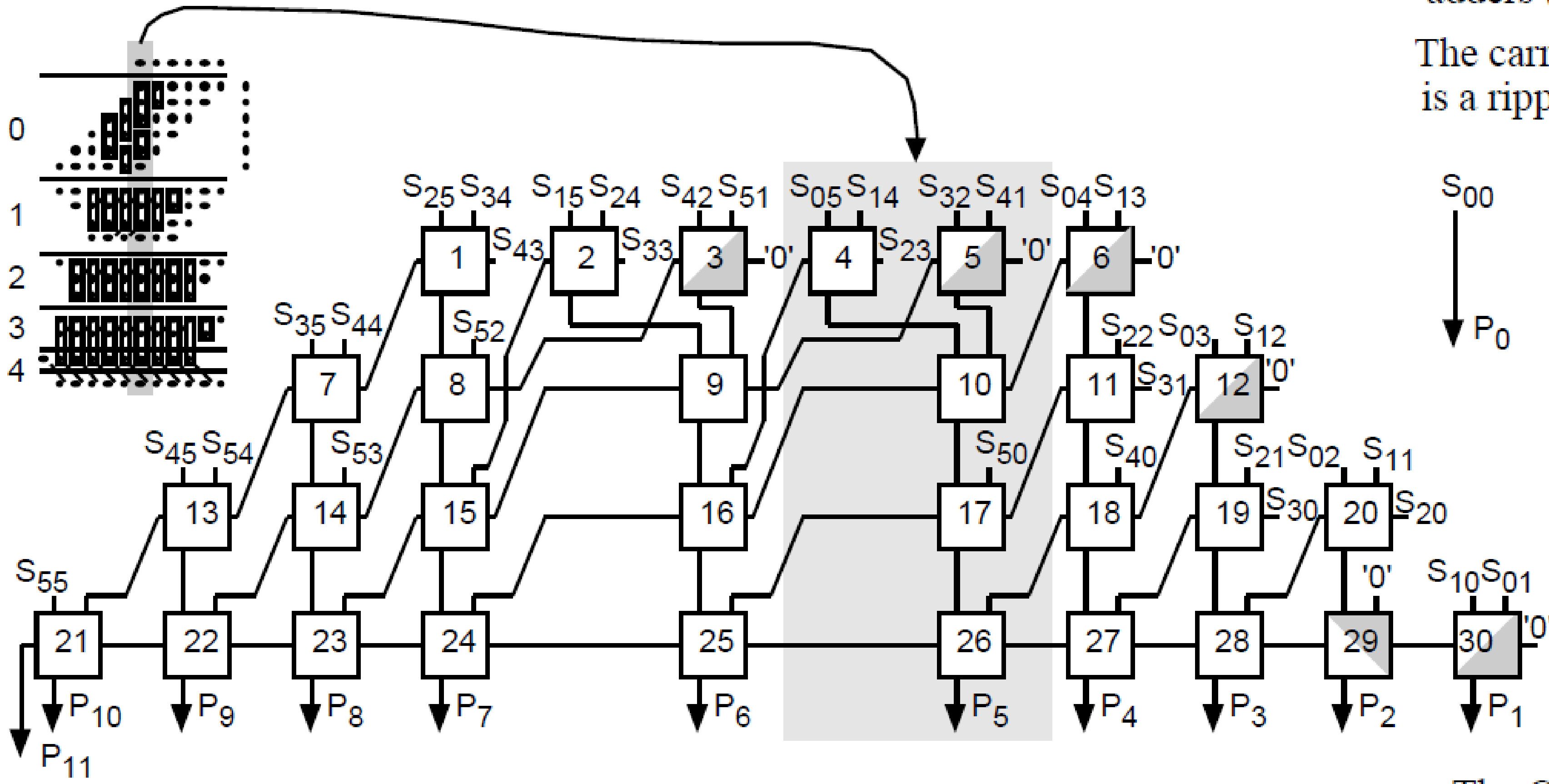
- Full adder is a **3:2 compressor** or **(3, 2) counter**
- Half adder is a **(2, 2) counter**

A 6-bit Wallace-tree multiplier. The carry-save adder (CSA) requires 26 adders (cells 1 26, six are half adders). The final carry-propagate adder (CPA) consists of 4 adder cells (27 30). The delay of the CSA is 6 adders. The delay of the CPA is 4 adders.

Dadda multiplier

The carry-save adder (CSA) requires 20 adders (cells 1 20, four are half adders).

The carry-propagate adder (CPA, cells 21 30) is a ripple-carry adder (RCA).



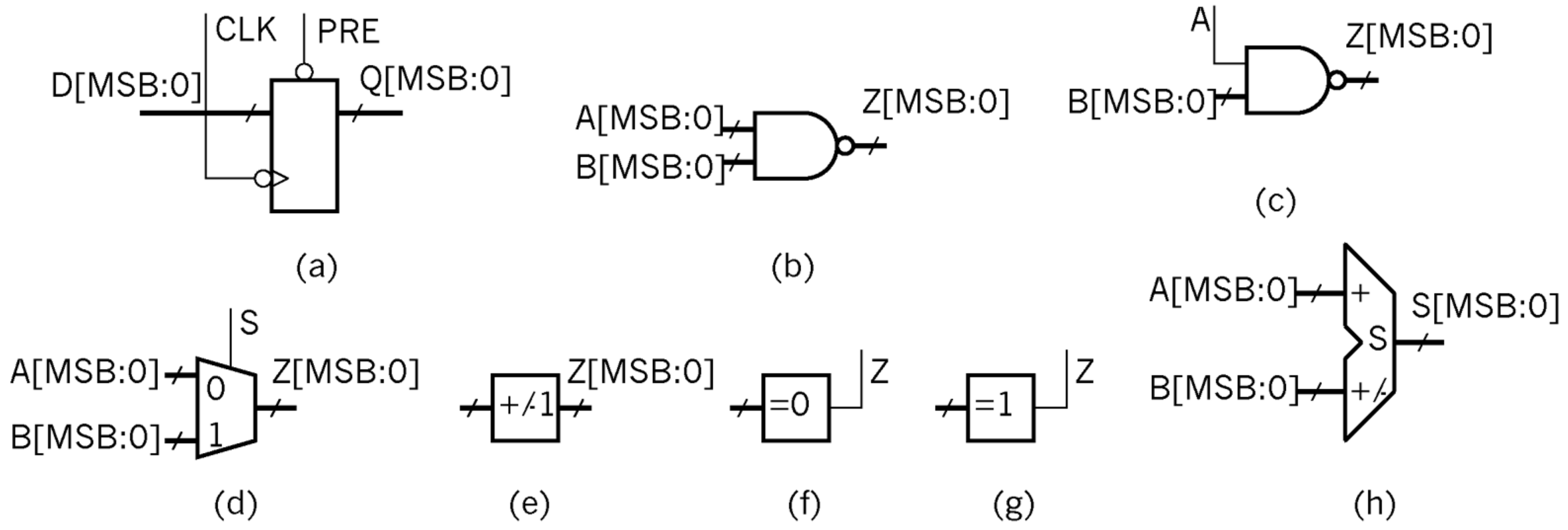
0
1
2
3
4
S25 S34 S15 S24 S42 S51 S05 S14 S32 S41 S04 S13 S00
S43 S33 '0' S23 '0' '0'
S52 S35 S44 S22 S03 S12 S31 '0'
S45 S54 S53 S50 S40 S21 S02 S11 S30 S20
S55 '0' S10 S01 '0'
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11

The CSA is smaller (20 versus 26 adders), faster (3 adder delays versus 6 adder delays),

The Dadda multiplier works backward from the final product

- Each stage has a maximum of 2, 3, 4, 6, 9, 13, 19, ...outputs (each successive stage is 3/2 times larger—rounded down to an integer

Other Datapath Elements



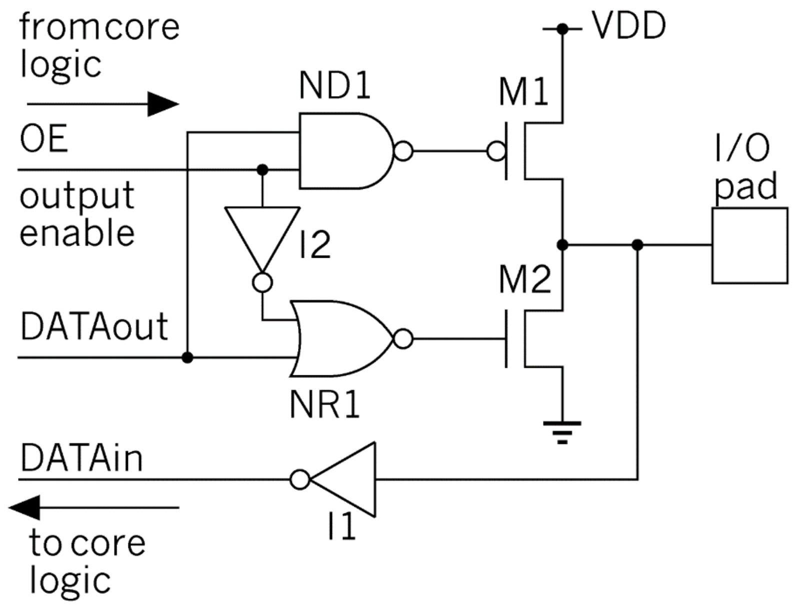
Symbols for datapath elements.

(a)An array or vector of flip-flops (a register).
(b)A two-input NAND cell with data bus inputs.
(c)A two-input NAND cell with a control input.
(d) A bus wide MUX.
(e)An incrementer/decrementer.
(f)An all-zeros detector.
(g)An all-ones detector.
(h)An adder/subtracter.

I/O Cells

- I/O pads are specialized to connect to the actual pins of the device
- Different types of I/O pads are provided to perform different functions
    - Digital input
    - Digital Output
    - Digital Bi-directional
    - Analog In/Output

When the output enable (OE)signal is high, the circuit functions as a noninverting buffer driving the value of DATAin onto the I/O pad. When OE is low, the output transistors or drivers , M1and M2, are disconnected.



A three-state bidirectional output buffer.

Cell Compilers

- layout for a full-custom IC is a tedious, time-consuming, and error-prone task.
- There are two types of automated layout assembly tools (**Silicon compilers**)
    - First type produces a specific kind of circuit

        a RAM compiler or multiplier compiler

    - Second type of compiler is more flexible

        usually providing a programming language that assembles or tiles layout from an input command file

- **Model compiler** - verify the circuit at the behavioral level
- **Netlist compiler** - simulate the circuit and verify that it works correctly at the structural level.