MODULE 1

INTRODUCTION TO ASIC's

INTRODUCTION TO ASICs

An ASIC ("a-sick") is an **application-specific integrated circuit**. It is an Integrated Circuit (IC) designed to perform a specific function for a specific application. As opposed to a standard, general purpose off-the-shelf part such as a commercial microprocessor or a 7400 series IC.

Gate equivalent - a unit of size measurement corresponding to a 4 transistor gate equivalent (e.g. a 2 input NOR gate).

History of integration:

Integrated circuit is a circuit in which all or some of the circuit elements are inseparably associated and electrically interconnected to form a complete functional device. Advances in IC technology, primarily smaller features and larger chips, have allowed the number of transistors in an integrated circuit to double every two years, a trend known as Moore's law. This increased capacity has been used to decrease cost and increase functionality. As a resultant the various integration levels emerged based on the Moore's Law.

Levels of integration	***Number of gates per chip***	***Year***
Small Scale Integration (SSI)	~10 gates per chip.	1960's
Medium Scale Integration (MSI)	~100–1000 gates per chip.	1970's
Large Scale Integration (LSI)	~1000–10,000 gates per chip.	1980's
Very Large Scale Integration (VLSI)	~10,000–100,000 gates per chip.	1990's
Ultra Large Scale Integration (ULSI)	~1M–10M gates per chip.	*2000 and above*

History of technology:

1. **Bipolar technology**
2. **Transistor–transistor logic** (**TTL**)
3. **Metal Oxide-Silicon** (**MOS**) technology because it was difficult to make metal-gate n-channel MOS (**nMOS** or **NMOS**)
4. **Complementary MOS** (**CMOS**) greatly reduced power.

The **feature size** is the smallest shape you can make on a chip and is measured in **λ** or **lambda.**

Origin of ASICs:

The **standard parts**, initially used to design **microelectronic systems**, were gradually replaced with a combination of **glue logic**, **custom ICs**, **dynamic random access memory** (**DRAM**) and **static RAM** (**SRAM**).

History of ASICs:

The *IEEE Custom Integrated Circuits Conference* (CICC) and *IEEE International ASIC Conference* document the development of ASICs

Application-specific standard products (**ASSPs**) are a cross between standard parts and ASICs.

Types of ASIC

ICs are made on a wafer. Circuits are built up with successive mask layers.

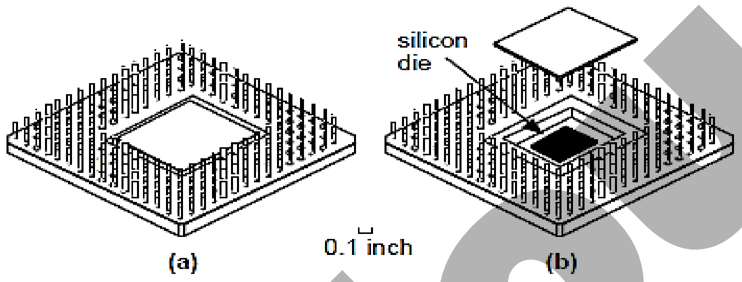


FIGURE 1.1 An integrated circuit (IC).

(a) A pin-grid array (PGA) package. (b) The silicon die or chip is under the package lid.

Note: In a PGA, the package is square or rectangular, and the pins are arranged in a regular array on the underside of the package.

The number of masks used to define the interconnect and other layers is different between various categories of ASICs.

1. Full custom ASIC
2. Standard cell based ASIC and Gate Array based ASIC.
3. Programmable ASIC: PLDs (FPGA, CPLD etc…)

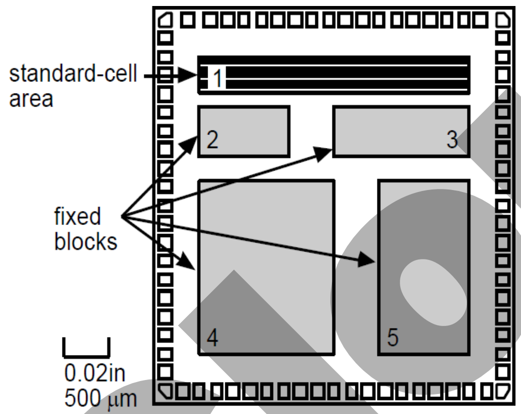
1. **Full Custom ASIC**:

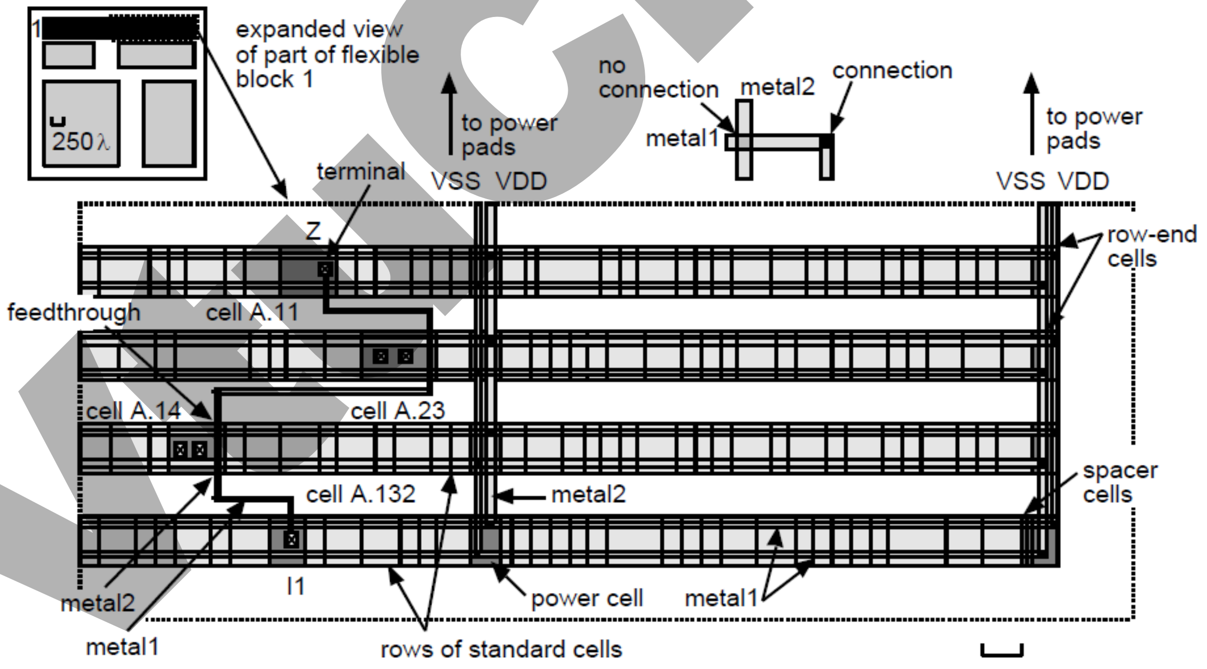
- All mask layers are customized in a **full-custom ASIC**.
- It only makes sense to design a full-custom IC if there are no libraries available.
- Full-custom offers the highest performance and lowest part cost (smallest die size) with the disadvantages of increased design time, complexity, design expense, and highest risk.
- Microprocessors were exclusively full-custom, but designers are increasingly turning to semicustom ASIC techniques in this area too.
- Other examples of full-custom ICs or ASICs are requirements for high-voltage (automobile), analog/digital (communications), or sensors and actuators.

2. **Standard Cell Based ASIC (CBIC):**

A cell-based ASIC (CBIC - “sea-bick”)

- Uses Standard cells possibly mega cells, mega functions, full custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs).
- All mask layers are customized transistors and interconnect.
- Custom blocks can be embedded.
- Manufacturing lead time is about eight weeks.



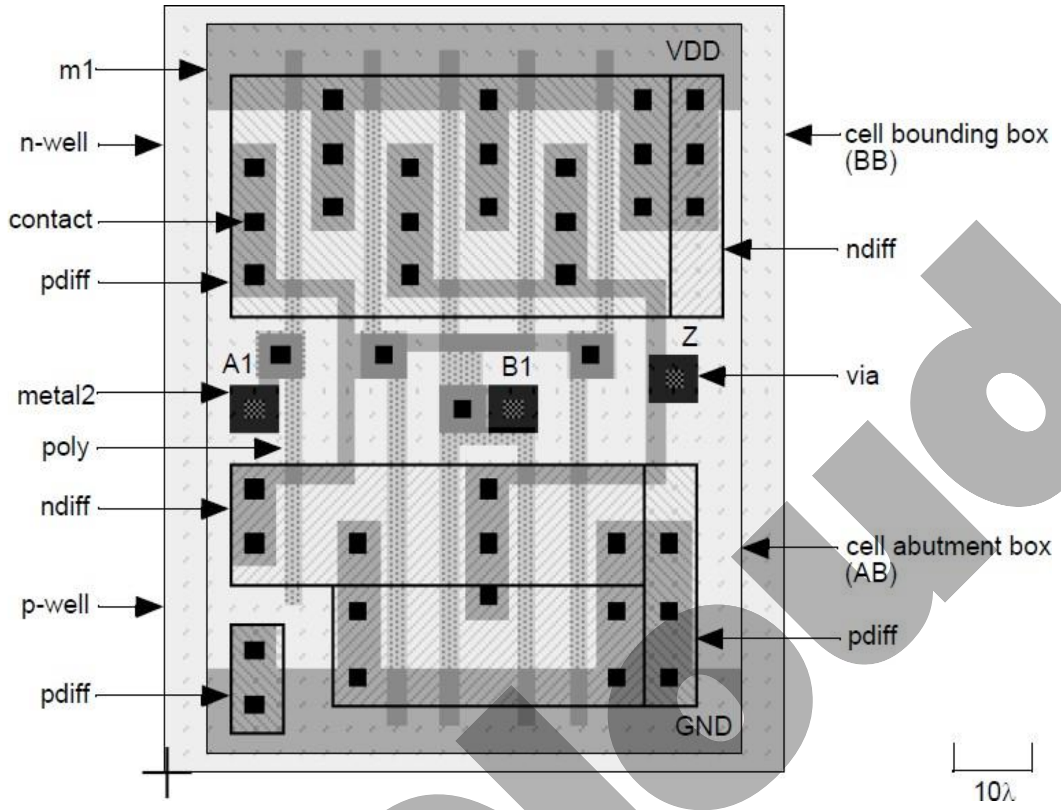


Routing a CBIC (cell-based IC)

• A “wall” of standard cells forms a flexible block

• metal2 may be used in a feed through cell to cross over cell rows that use metal1 for wiring

• Other wiring cells: spacer cells, row-end cells, and power cells.



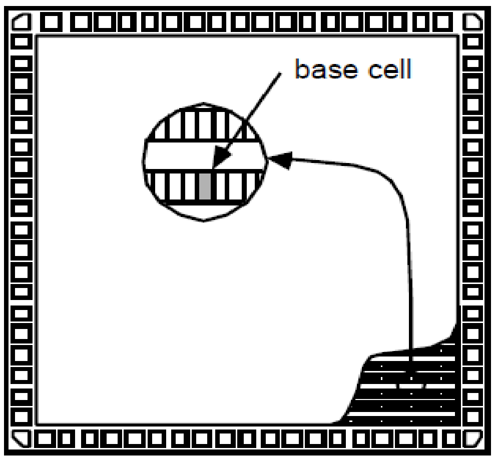
*Mask Layout to show a **standard cell** from a **standard-cell library.***

3. Gate-Array-Based ASICs

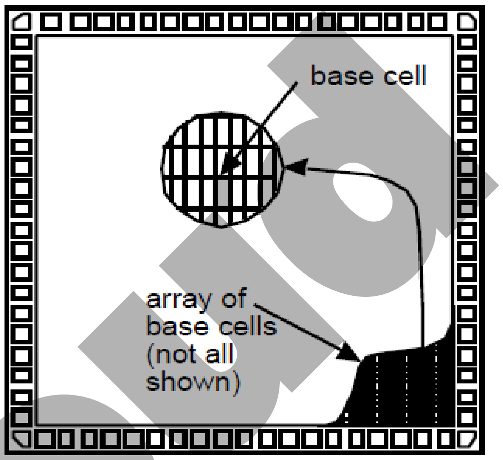
- In a gate-array-based ASIC, the transistors are predefined on the silicon wafer.
- The predefined pattern of transistors is called the *base array.*
- The smallest element that is replicated to make the base array is called the *base* or *primitive cell.*
- The top level interconnect between the transistors is defined by the designer in custom masks - *Masked Gate Array (MGA).*
- Design is performed by connecting predesigned and characterized logic cells from a library (macros).
- 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:
    1. Channeled Gate Array
    2. Channelless Gate Array
    3. Structured Gate Array

3.1 Channeled Gate Array

- 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.



Channeled Gate Array



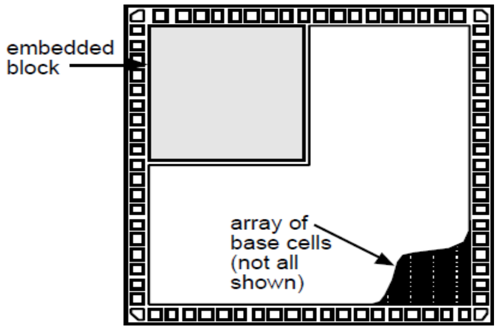
Channelless Gate Array

3.2 Channelless Gate Array

- There are no predefined areas set aside for routing - routing is over the top of the gate-array devices.
- Achievable logic density is higher than for channeled gate arrays.
- Manufacturing lead time is between two days and two weeks.

3.3 Structured Gate Array

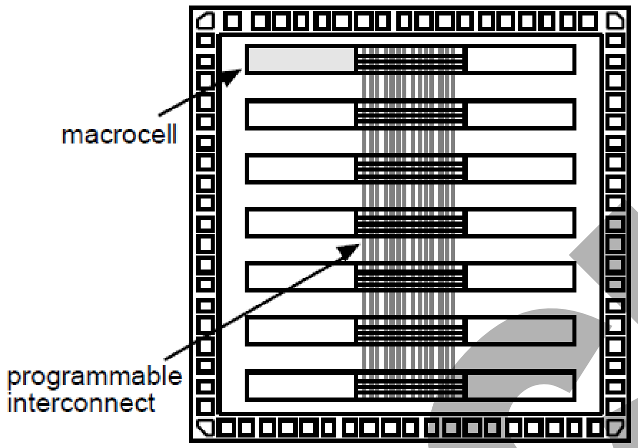
- Only the interconnect is customized
- Custom blocks (the same for each design) can be embedded
  1. These can be complete blocks such as a processor or memory array, or
  2. An array of different base cells better suited to implementing a specific function.
- Manufacturing lead time is between two days and two weeks.



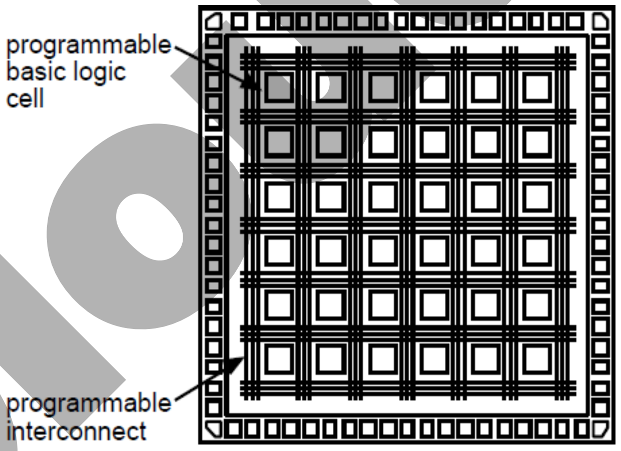
Structured Gate Array

4. Programmable Logic Devices

- No customized mask layers or logic cells.
- Fast design turnaround.
- A single large block of programmable interconnects.
  Ex: Erasable PLD (EPLD), Mask-programmed PLD.
- A matrix of logic macro cells that usually consist of programmable array logic followed by a flip-flop or latch.



Programmable Logic Device (PLD) die

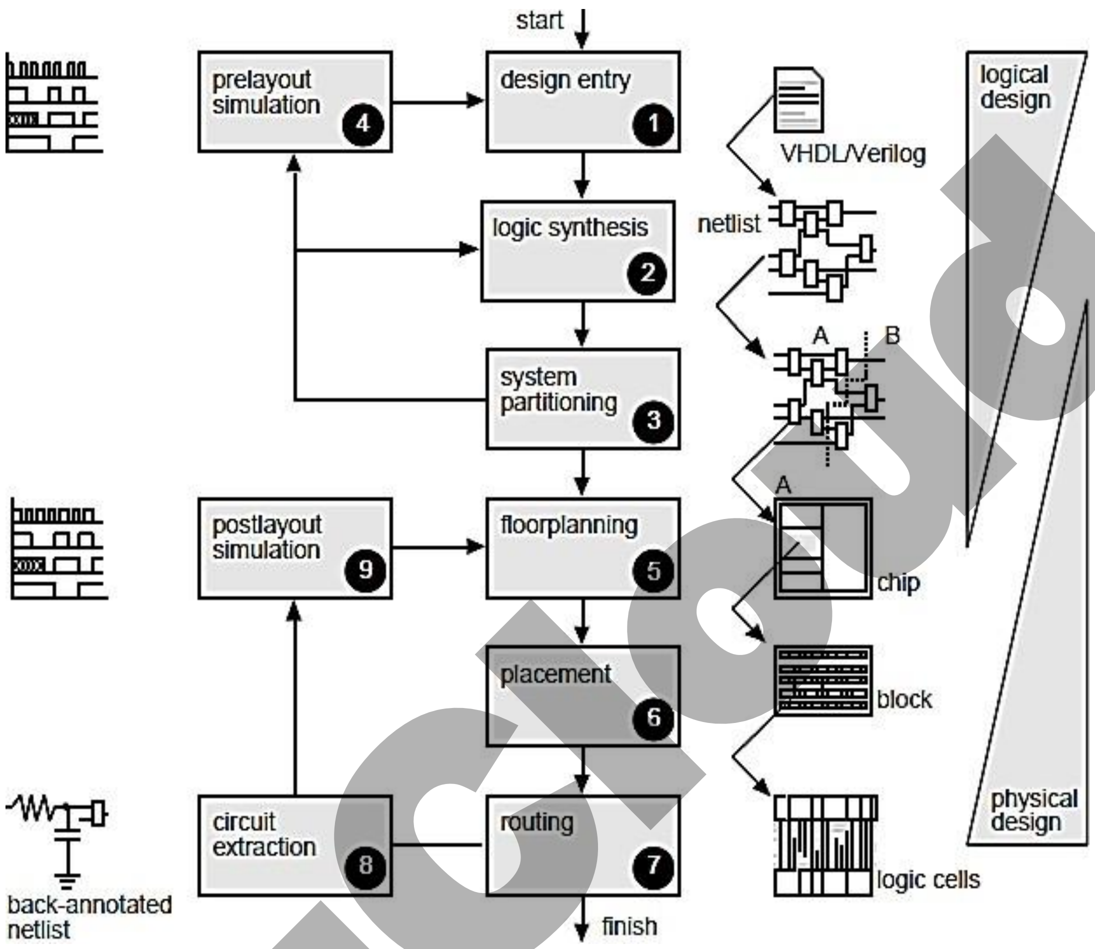


FPGA die

5. Field Programmable Gate Array

- 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

Design Flow:



ASIC design flow.

Steps 1–4 are **logical design**, and steps 5–9 are **physical design**

1. **Design entry** - Using a hardware description language (HDL) or schematic entry.
2. ***Logic synthesis*** - Produces a net list - 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. ***Floor planning*** - Arrange the blocks of the net list 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.

ASIC Cell Libraries:

A library of cells is used by the designer to design the logic function for an ASIC

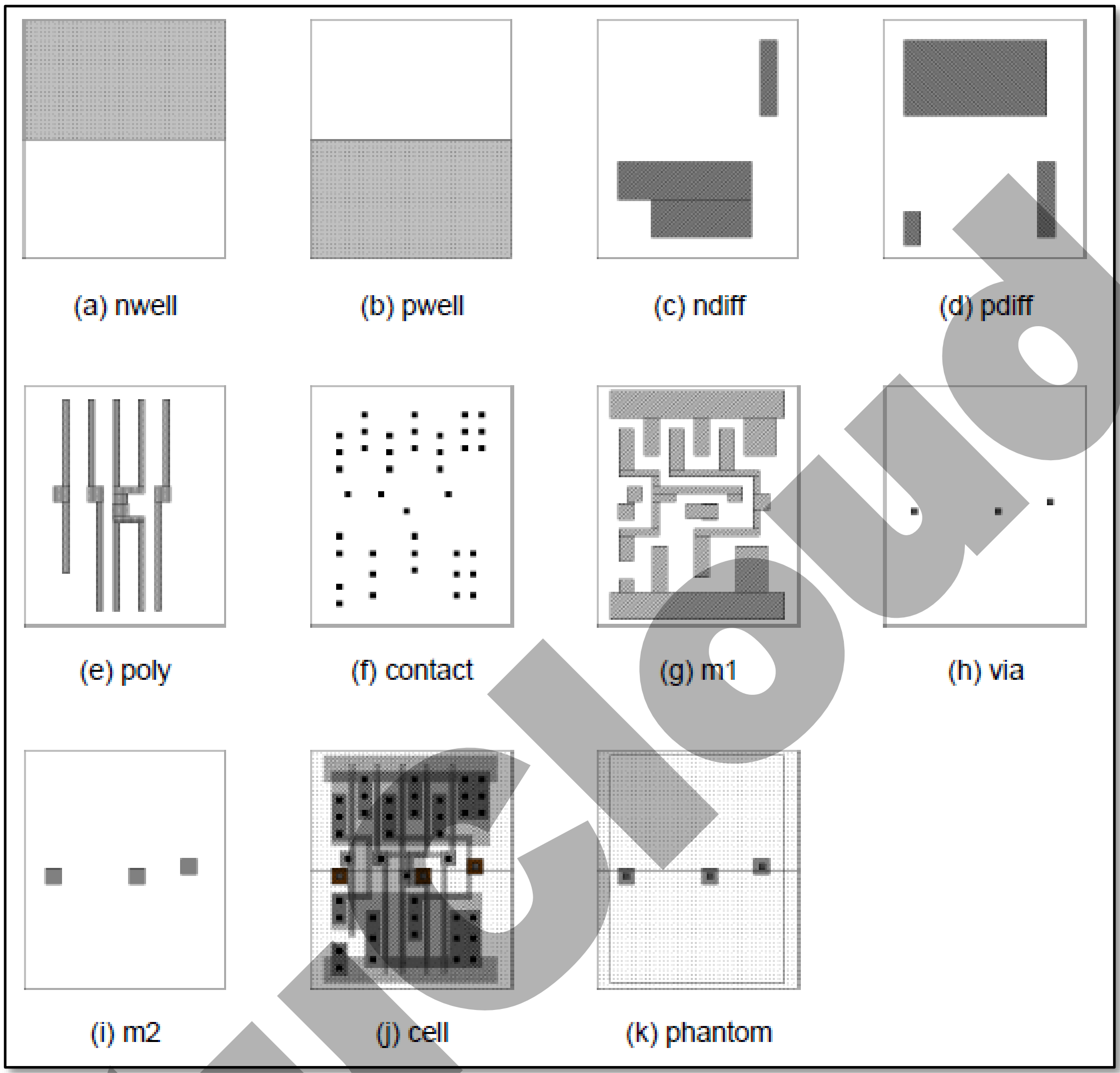
Options for cell library:

1. **Use a design kit from the ASIC vendor**
    - Usually requires the use of ASIC vendor approved tools
    - Cells are "phantoms" - empty boxes that get filled in by the vendor when you deliver, or 'hand off" the netlist
    - Vendor may provide more of a "guarantee" that design will work
2. **Buy an ASIC-vendor library from a library vendor**
    - Library vendor is different from fabricator (foundry)
    - Library may be approved by the foundry (qualified cell library)
    - Allows the designer to own the masks (tooling) for the part when finished
3. **You can build your own cell library**
    - 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 behavioral model
- 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

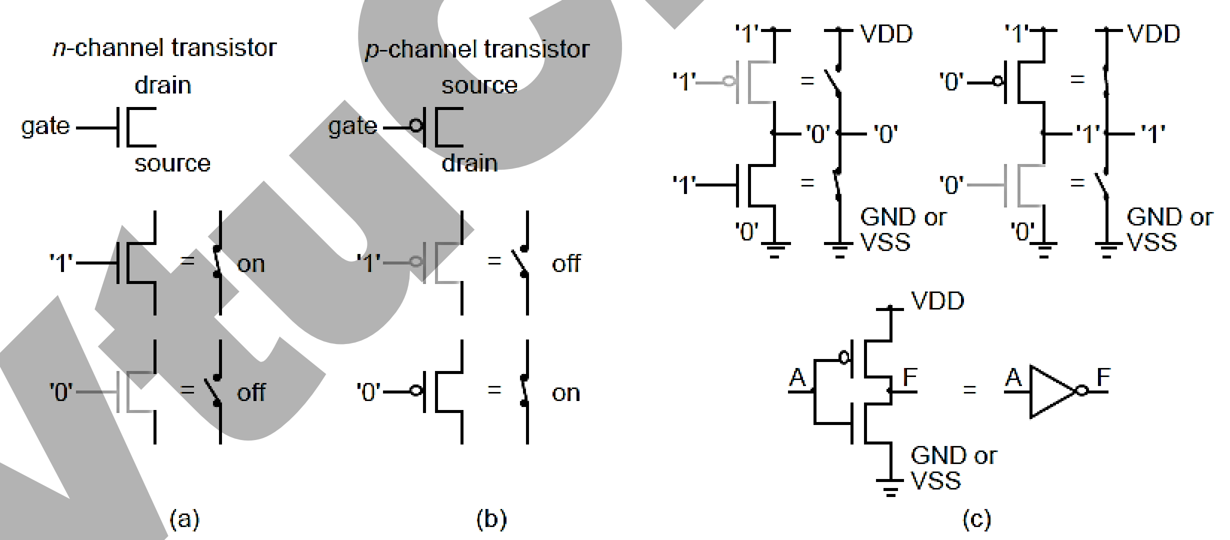


The mask layers of a standard cell

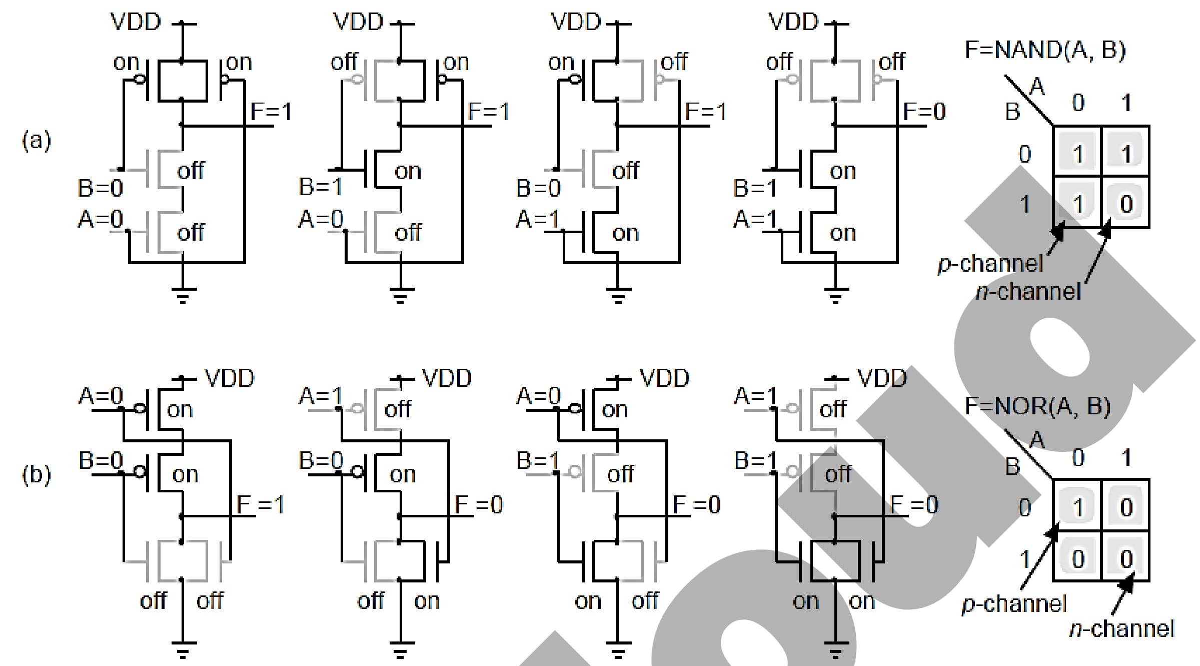
CMOS Logic:

A CMOS transistor (or device) has four terminals: gate, source, drain, and a fourth terminal that we shall ignore until the next section. A CMOS transistor is a switch. The switch must be conducting or on to allow current to flow between the source and drain terminals (using open and closed for switches is confusing for the same reason we say a tap is on and not that it is closed). The transistor source and drain terminals are equivalent as far as digital signals are concerned—we do not worry about labeling an electrical switch with two terminals.

We turn a transistor on or off using the gate terminal. There are two kinds of CMOS transistors: n -channel transistors and p-channel transistors. An n -channel transistor requires a logic '1' (from now on I'll just say a '1') on the gate to make the switch conducting (to turn the transistor on). A p -channel transistor requires a logic '0' (again from now on, I'll just say a '0') on the gate to make the switch non conducting (to turn the transistor off). The p -channel transistor symbol has a bubble on its gate to remind us that the gate has to be a '0' to turn the transistor on . All this is shown in (a) and (b). If we connect an n -channel transistor in series with a p -channel transistor, as shown in Figure(c), we form an inverter.

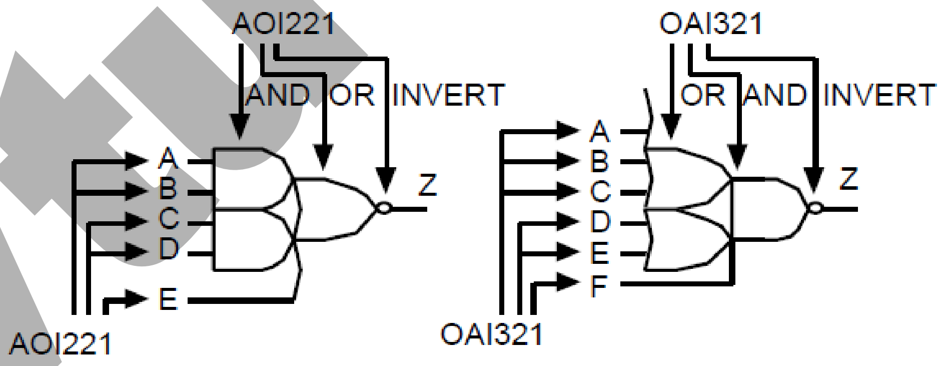


CMOS transistors viewed as switches • a CMOS inverter



CMOS logic.
(a) A two-input NAND logic cell. (b) A two-input NOR logic cell. The n -channel and p -channel transistor switches implement the '1's and '0's of a Karnaugh map.

Other Logics: The AND-OR-INVERT (AOI) and the OR-AND-INVERT (OAI) logic cells are particularly efficient in CMOS.



We can express the function of the AOI221 cell in Figure as:

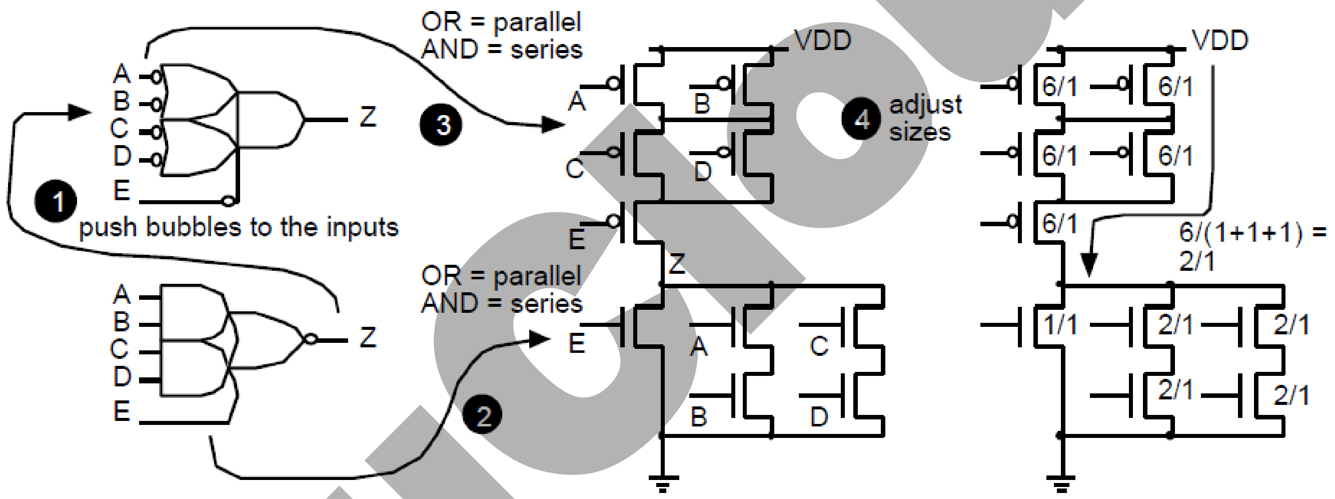
Z = (A · B + C · D + E)'

The AOI family of cells with three index numbers or less

Cell type[1]	Cells	Number of unique cells
Xa1	X21, X31	2
Xa11	X211, X311	2
Xab	X22, X33, X32	3
Xab1	X221, X331, X321	3
Xabc	X222, X333, X332, X322	4
Total		14

[1]Xabc: X={AOI, AO, OAI, OA}; a, b, c = {2, 3}; {} means "choose one."

CMOS Structure of AOI221 and its ratios (βn,βp values)



Data path Logic Cells:

Suppose we wish to build an n -bit adder (that adds two n -bit numbers) and to exploit the regularity of this function in the layout. We can do so using a data path structure.

Full adder (FA): The following two functions, SUM and COUT, implement the sum and carry out for a full adder (FA) with two data inputs (A, B) and a carry in, CIN:

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

$$COUT = A \cdot B + A \cdot CIN + B \cdot CIN = MAJ(A, B, CIN).$$

- Parity function ('1' for an odd numbers of '1's)
- Majority function ('1' if the majority of the inputs are '1')

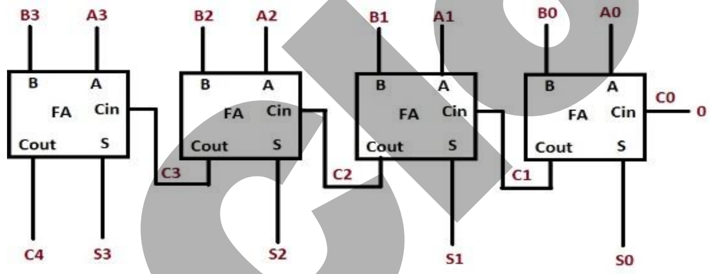
INPUTS			OUTPUTS	
A	B	CIN	SUM	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

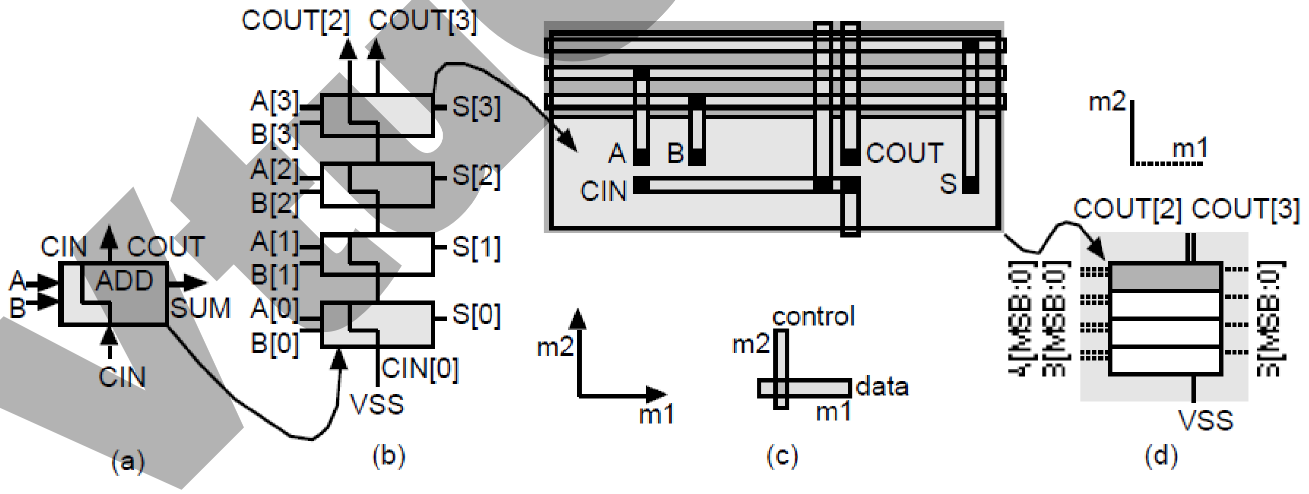
Data Path Adder:

Data path adder is a Ripple Carry adder.

Ripple Carry Adder:

A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the succeeding next most significant full adder. It is called a ripple carry adder because each carry bit gets rippled into the next stage.





A data path adder.

(a) A full-adder (FA) cell with inputs (A and B), a carry in, CIN, sum output, S, and carry out, COUT. (b) A 4-bit adder. (c) The layout, using two-level metal, with data in m1 and control in m2. In this example the wiring is completed outside the cell. It is also possible to design the datapath cells to contain the wiring. Using three levels of metal, it is possible to wire over the top of the datapath cells. (d) The datapath layout.

Data path Elements:

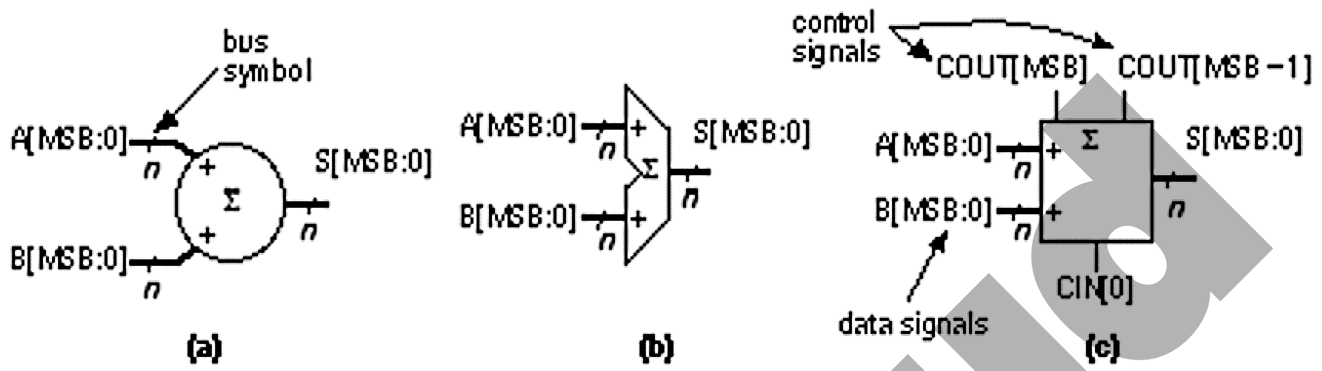


FIGURE above shows the Symbols for a data path adder.

(a) A data bus is shown by a heavy line (1.5 point) and a bus symbol. If the bus is n -bits wide then MSB = n – 1. (b) An alternative symbol for an adder. (c) Control signals are shown as lightweight (0.5 point) lines.

Figure above shows a typical datapath symbols for an adder (people rarely use the IEEE standards in ASIC datapath libraries). I use heavy lines (they are 1.5 point wide) with a stroke to denote a data bus (that flows in the horizontal direction in a datapath), and regular lines (0.5 point) to denote the control signals (that flow vertically in a datapath). At the risk of adding confusion where there is none, this stroke to indicate a data bus has nothing to do with mixed-logic conventions. 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 . Sometimes we shall use A[MSB] or A[LSB] to refer to these bits. Notice that if we have an n -bit bus and LSB = 0, then MSB = n – 1. Also, for example, A[4] is the fifth bit on the bus (from the LSB). We use a ' S ' or 'ADD' inside the symbol to denote an adder instead of '+', so we can attach '–' or '+/–' to the inputs for a subtracter or adder/subtracter.

Some schematic datapath symbols include only data signals and omit the control signals—but we must not forget them. In Figure (C), for example, we may need to explicitly tie CIN[0] to VSS and use COUT[MSB] and COUT[MSB – 1] to detect overflow.

Binary Arithmetic Operation:

Binary arithmetic

Operation	Binary Number Representation			
	Unsigned	Signed magnitude	Ones' complement	Two's complement
	no change	if positive then MSB=0 else MSB=1	if negative then flip bits	if negative then {flip bits; add 1}
3=	0011	0011	0011	0011
–3=	NA	1011	1100	1101
zero=	0000	0000 or 1000	1111 or 0000	0000
max. positive=	1111=15	0111=7	0111=7	0111=7
max. negative=	0000=0	1111=–7	1000=–7	1000=–8
addition= S= A+B =addend+augend SG(A)=sign of A	S=A+B	if SG(A)=SG(B) then S=A+B else {if B<A then S=A–B else S=B–A}	S= A+B+COUT[MSB] COUT is carry out	S=A+B
addition result: OV=overflow, OR=out of range	OR=COUT[MSB] COUT is carry out	if SG(A)=SG(B) then OV=COUT[MSB] else OV=0 (impossible)	OV= XOR(COUT[MSB], COUT[MSB–1])	OV= XOR(COUT[MSB], COUT[MSB–1])
SG(S)=sign of S S= A+B	NA	if SG(A)=SG(B) then SG(S)=SG(A) else {if B<A then SG(S)=SG(A) else SG(S)=SG(B)}	NA	NA
subtraction= D= A–B =minuend –subtrahend	D=A–B	SG(B)=NOT(SG(B)); D=A+B	Z=–B (negate); D=A+Z	Z=–B (negate); D=A+Z
subtraction result: OV=overflow, OR=out of range	OR=BOUT[MSB] BOUT is borrow out	as in addition	as in addition	as in addition
negation: Z=–A (negate)	NA	Z=A; SG(Z)=NOT(SG(A))	Z=NOT(A)	Z=NOT(A)+1

Adders:

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

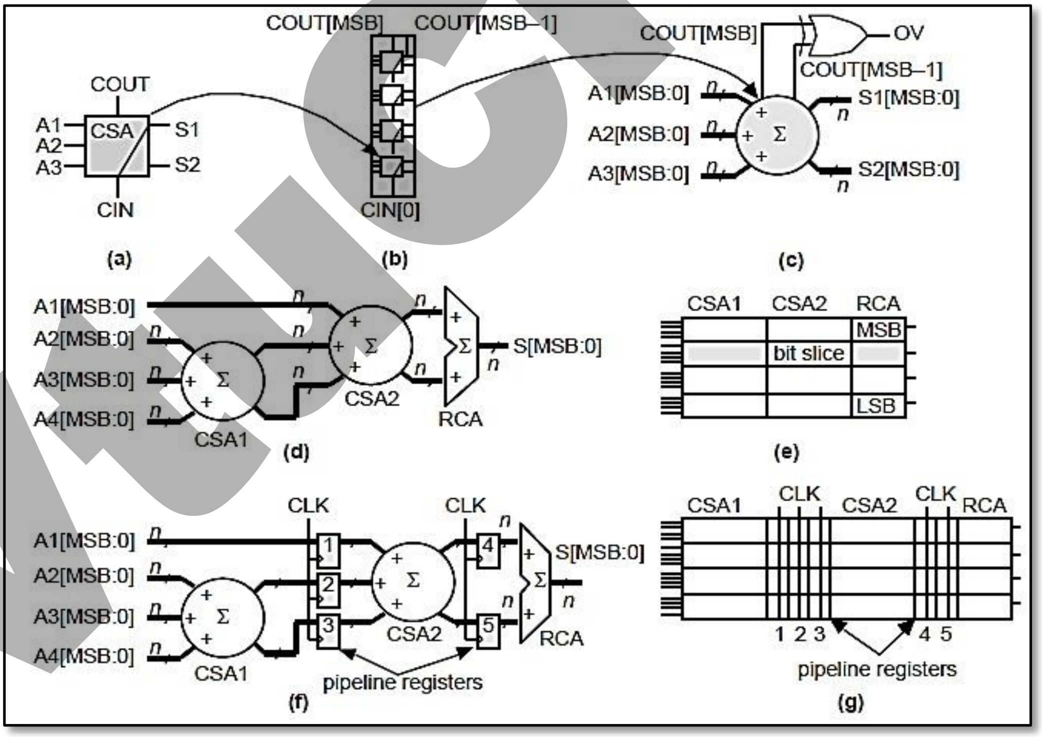
method 1	method 2
$G[i] = A[i] \cdot B[i]$	$G[i] = A[i] \cdot B[i]$
$P[i] = A[i] \oplus B[i$	$P[i] = A[i] + B[i]$
$C[i] = G[i] + P[i] \cdot C[i-1]$	$C[i] = G[i] + P[i] \cdot C[i-1]$
$S[i] = P[i] \oplus C[i-1]$	$S[i] = A[i] \oplus B[i] \oplus C[i-1]$

Where C[i] is the carry-out signal from stage i , equal to the carry in of stage (i + 1). Thus, C[i]= COUT[i] = CIN[i + 1]. We need to be careful because C[0] might represent either the carry in or the carry out of the LSB stage. For an adder we set the carry in to the first stage (stage zero), C[–1] or CIN[0], to '0'.

If we consider a conventional RCA. The delay of an n -bit RCA is proportional to n and is limited by the propagation of the carry signal through all of the stages. We can reduce delay by using pairs of “go-faster” bubbles to change AND and OR gates to fast two-input NAND gates as shown in Figure (a). Alternatively, we can write the equations for the carry signal in two different ways:

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

or $$C[i] = (A[i] + B[i]) \cdot (P[i]' + C[i-1]), \text{ where } P[i]'=\text{NOT}(P[i])$$



The carry-save adder (CSA). (a) A CSA cell. (b) A 4-bit CSA. (c) Symbol for a CSA. (d) A four-input CSA. (e) The datapath for a four-input, 4-bit adder using CSAs with a ripple-carry adder (RCA) as the final stage. (f) A pipelined adder. (g) The datapath for the pipelined version showing the pipeline registers as well as the clock control lines that use m2.

Carry Save adder:

We can register the CSA stages by adding vectors of flip-flops as shown in Figure (f). This reduces the adder delay to that of the slowest adder stage, usually the CPA. By using registers between stages of combinational logic we use pipelining to increase the speed and pay a price of increased area (for the registers) and introduce latency. It takes a few clock cycles (the latency, equal to n clock cycles for an n -stage pipeline) to fill the pipeline, but once it is filled, the answers emerge every clock cycle. Ferris wheels work much the same way. When the fair opens it takes a while (latency) to fill the wheel, but once it is full the people can get on and off every few seconds.

(We can also pipeline the RCA. We add i registers on the A and B inputs before ADD[i] and add (n – i) registers after the output S[i], with a single register before each C[i].)

Carry Bypass adders (CBA):

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. Thus, for example, to bypass the carries for bits 4–7 (stages 5–8) of an adder we can compute

BYPASS = P[4].P[5].P[6].P[7] and then use a MUX as follows:

$$C[7]=(G[7]+P[7]\cdot C[6])\cdot BYPASS'+C[3]\cdot BYPASS$$

Adders based on this principle are called carry-bypass adders (CBA). Large, custom adders employ Manchester-carry chains to compute the carries and the bypass operation using TGs or just pass transistors. These types of carry chains may be part of a predesigned ASIC adder cell, but are not used by ASIC designers.

Carry Skip Adder:

Instead of checking the propagate signals we can check the inputs. For example we can compute

SKIP = (A[i – 1] ⊕ B[i – 1]) + (A[i] ⊕ B[i]) and then use a 2:1 MUX to select C[i]. Thus,

$$CSKIP[i] = (G[i] + P[i] \cdot C[i-1]) \cdot SKIP' + C[i-2] \cdot SKIP$$

This is a carry-skip adder. Carry-bypass and carry-skip adders may include redundant logic (since the carry is computed in two different ways—we just take the first signal to arrive). We must be careful that the redundant logic is not optimized away during logic synthesis.

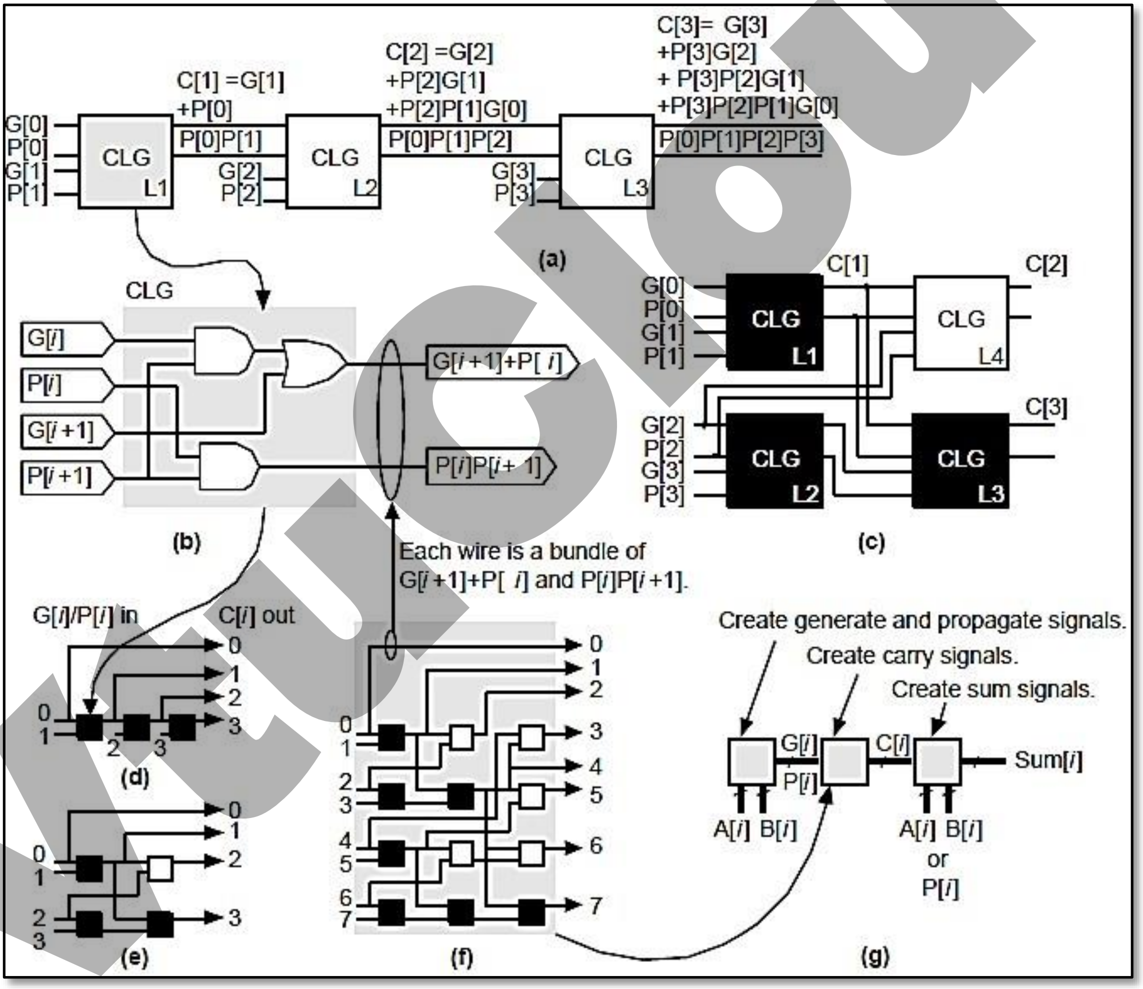
Carry Look Ahead Adder: Brent–Kung carry-lookahead adder

If we find the recursive carries to look ahead the possibilities of carry then it is easier for Computation.

The following equation represents the Carry look ahead adder for 4 bits. C[0]=Cin.

$$\begin{aligned} C[1] &= G[1] + P[1] \cdot C[0] \\ &= G[1] + P[1] \cdot (G[0] + P[1] \cdot C[-1]) \\ &= G[1] + P[1] \cdot G[0] \end{aligned}$$

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



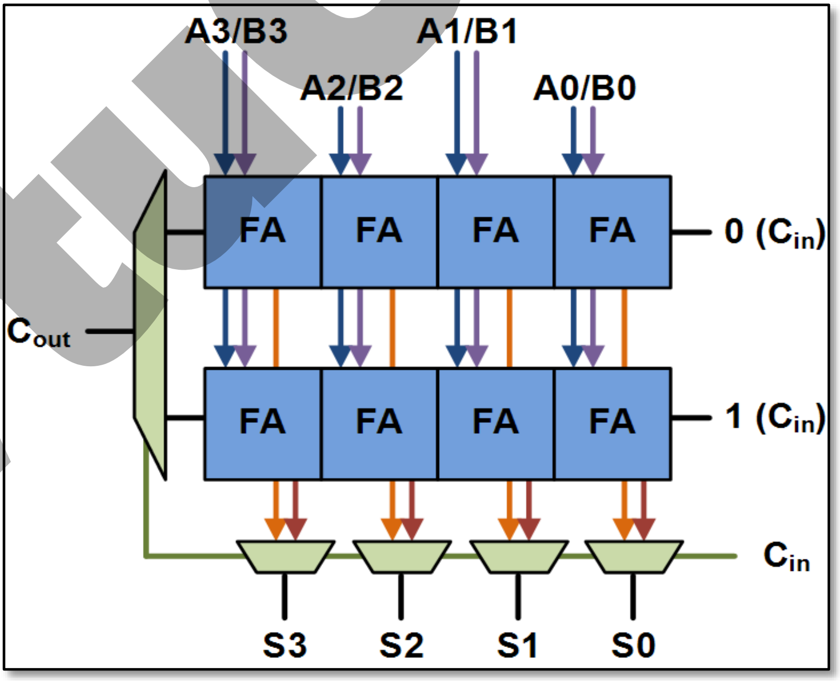
The Brent–Kung carry-lookahead adder (CLA). (a) Carry generation in a 4-bit CLA. (b) A cell to generate the lookahead terms, C[0]–C[3]. (c) Cells L1, L2, and L3 are rearranged into a tree that has less delay. Cell L4 is added to calculate C[2] that is lost in the translation. (d) and (e) Simplified representations of parts a and c. (f) The lookahead logic for an 8-bit adder. The inputs, 0–7, are the propagate and carry terms formed from the inputs to the adder. (g) An 8-bit Brent–Kung CLA.

The outputs of the look ahead logic are the carry bits that (together with the inputs) form the sum. One 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:

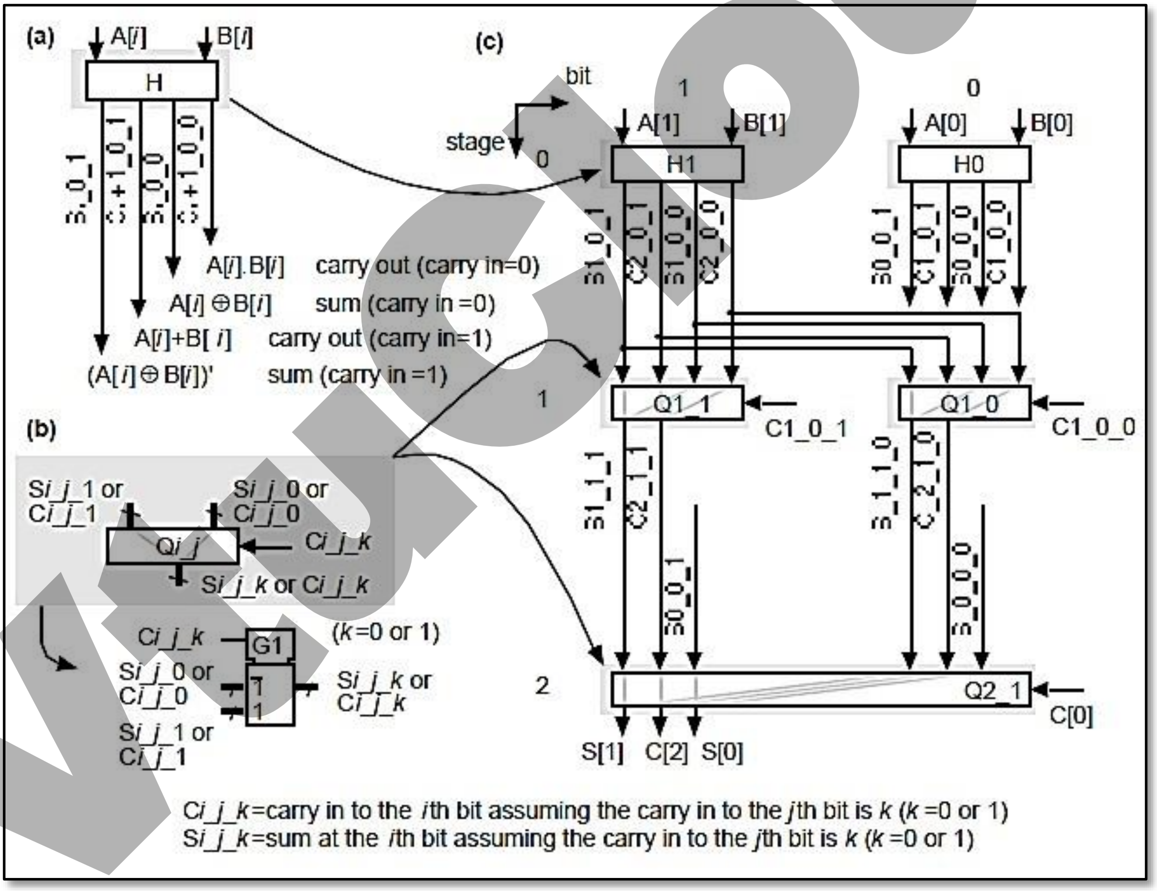
In a carry-select adder we duplicate two small adders (usually 4-bit or 8-bit adders—often CLAs) for the cases CIN = '0' and CIN = '1' and then use a MUX to select the case that we need—wasteful, but fast. A carry-select adder is often used as the fast adder in a datapath library because its layout is regular.

We can use the carry-select, carry-bypass, and carry-skip architectures to split a 12-bit adder, for example, into three blocks. The delay of the adder is then partly dependent on the delays of the MUX between each block. Suppose the delay due to 1-bit in an adder block (we shall call this a bit delay) is approximately equal to the MUX delay. In this case may be faster to make the blocks 3, 4, and 5-bits long instead of being equal in size. Now the delays into the final MUX are equal—3 bit-delays plus 2 MUX delays for the carry signal from bits 0–6 and 5 bit-delays for the carry from bits 7–11. Adjusting the block size reduces the delay of large adders (more than 16 bits).



Conditional sum Adder:

We can extend the idea behind a carry-select adder 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 an 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 (n– i + 1) bit conditional sums from the MSB adder using 2(n – i + 1) two-input MUXes. This is a conditional-sum adder (also often abbreviated to CSA). We can recursively apply this technique. For example, we can split a 16-bit adder using i = 8 and n = 8, then we can split one or both 8–bit adders again—and so on.



(a)
A[i]
B[i]
H
S_0_1
C_1_0_1
S_0_0
C_1_0_0
A[i].B[i] carry out (carry in=0)
A[i] ⊕ B[i] sum (carry in =0)
A[i]+B[i] carry out (carry in=1)
(A[i] ⊕ B[i])' sum (carry in =1)
(b)
Si_j_1 or Ci_j_1
Si_j_0 or Ci_j_0
Qi_j
Ci_j_k
Si_j_k or Ci_j_k
G1
(k=0 or 1)
(c)
bit
stage
1
0
A[1] B[1]
A[0] B[0]
H1
H0
S1_0_1
C2_0_1
S1_0_0
C2_0_0
S0_0_1
C1_0_1
S0_0_0
C1_0_0
Q1_1
Q1_0
C1_0_1
C1_0_0
S1_1_1
C2_1_1
S1_1_0
C2_1_0
S0_0_1
S0_0_0
2
Q2_1
C[0]
S[1] C[2] S[0]
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)

Figure above shows the simplest form of an n -bit conditional-sum adder that uses n single-bit conditional adders, H (each with four outputs: two conditional sums, true carry, and complement carry), together with a tree of 2:1 MUXes (Qi_j). The conditional-sum adder is usually the fastest of all the adders we have discussed.

Example for 8 bit Conditional sum adder:

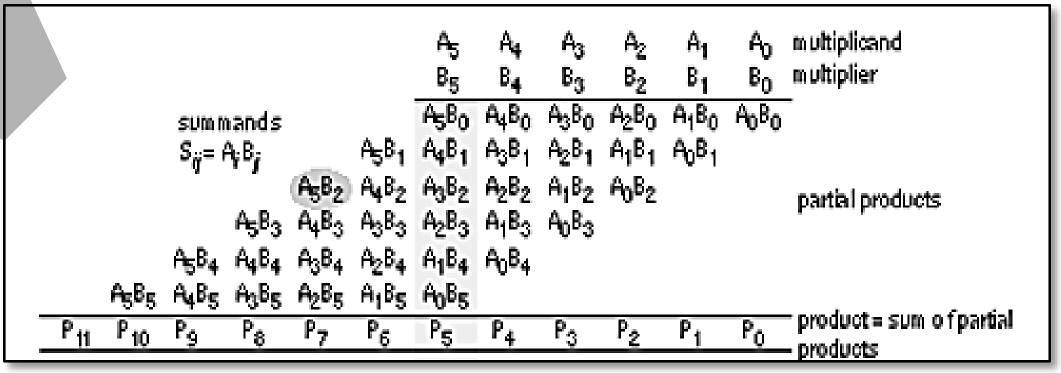
```
module m8bitCSum (C0, a, b, s, C8);                    // Verilog Conditional sum adder for an FPGA    //1
input [7:0] C0, a, b; output [7:0] s; output C8;                                                         //2
wire A7,A6,A5,A4,A3,A2,A1,A0,B7,B6,B5,B4,B3,B2,B1,B0,S8,S7,S6,S5,S4,S3,S2,S1,S0;                        //3
wire C0, C2, C4_2_0, C4_2_1, S5_4_0, S5_4_1, C6, C6_4_0, C6_4_1,C8;                                      //4
assign {A7,A6,A5,A4,A3,A2,A1,A0} = a; assign{B7,B6,B5,B4,B3,B2,B1,B0} = b;                               //5
assign s = { S7,S6,S5,S4,S3,S2,S1,S0 };                                                                  //6
assign S0 = A0^B0^C0 ;                        //start of level 1: & = AND, ^ = XOR, | =OR, ! = NOT      //7
assign S1 = A1^B1^(A0&B0|(A0|B0)&C0) ;                                                                   //8
assign C2 = A1&B1|(A1|B1)&(A0&B0|(A0|B0)&C0) ;                                                           //9
assign C4_2_0 = A3&B3|(A3|B3)&(A2&B2) ; assign C4_2_1 =A3&B3|(A3|B3)&(A2|B2) ;                           //10
assign S5_4_0 = A5^B5^(A4&B4) ; assign S5_4_1 = A5^B5^(A4|B4) ;                                          //11
assign C6_4_0 = A5&B5|(A5|B5)&(A4&B4) ; assign C6_4_1 =A5&B5|(A5|B5)&(A4|B4);                            //12
assign S2 = A2^B2^C2 ;                                                     // start of level 2          //13
assign S3 = A3^B3^(A2&B2|(A2|B2)&C2) ;                                                                   //14
assign S4 = A4^B4^(C4_2_0|C4_2_1&C2) ;                                                                   //15
assign S5 = S5_4_0&!(C4_2_0|C4_2_1&C2)|S5_4_1&(C4_2_0|C4_2_1&C2) ;                                       //16
assign C6 = C6_4_0|C6_4_1&(C4_2_0|C4_2_1&C2) ;                                                           //17
assign S6 = A6^B6^C6 ; // start of level 3                                                               //18
assign S7 = A7^B7^(A6&B6|(A6|B6)&C6) ;                                                                   //19
assign C8 = A7&B7|(A7|B7s)&(A6&B6|(A6|B6)&C6) ;                                                          //20
endmodule
```

<u>Multipliers:</u>

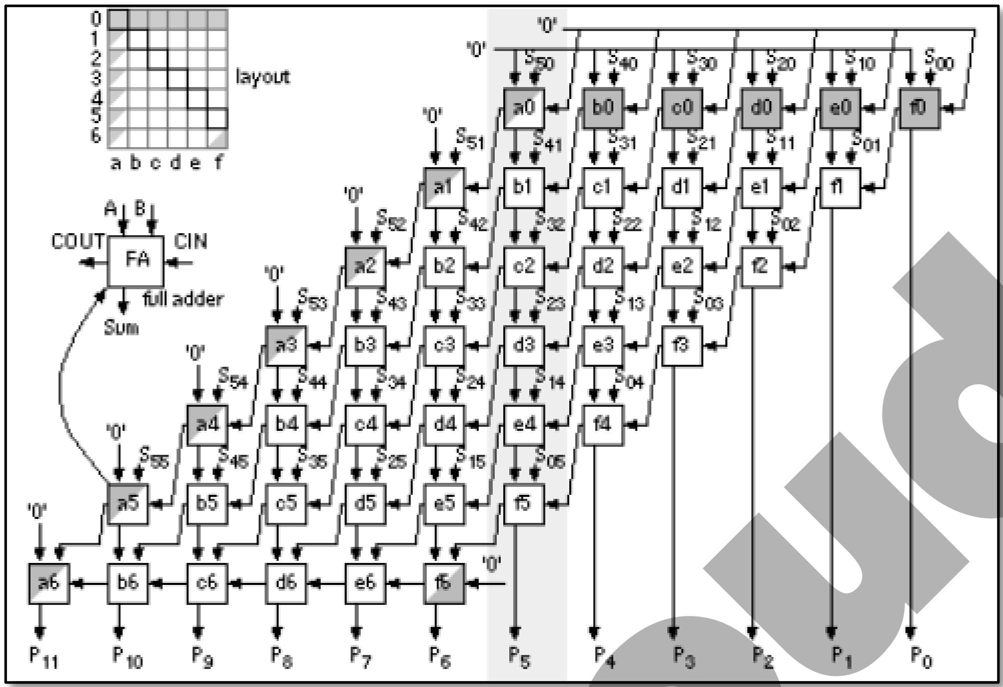
Figure below shows a symmetric 6-bit array multiplier (an n -bit multiplier multiplies two n -bit numbers; we shall use n -bit by m -bit multiplier if the lengths are different). Adders a0–f0 may be eliminated, which then eliminates adders a1–a6, leaving an asymmetric CSA array of 30 (5 × 6) adders (including one half adder). An n -bit array multiplier has a delay proportional to n plus the delay of the CPA.

There are two items we can attack to improve the performance of a multiplier:

1. The number of partial products and
2. The addition of the partial products.



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
$S_{ij} = A_iB_j$
A_5B_0 A_4B_0 A_3B_0 A_2B_0 A_1B_0 A_0B_0
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
partial products
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 partial products



Suppose we wish to multiply 15 (the multiplicand) by 19 (the multiplier) mentally. It is easier to calculate 15 × 20 and subtract 15. In effect we complete the multiplication as 15 ×(20 – 1) and we could write this as 15 × 2 1 , with the overbar representing a minus sign. Now suppose we wish to multiply an 8-bit binary number, A, by B = 00010111 (decimal 16 + 4 + 2 + 1 = 23). It is easier to multiply A by the canonical signed-digit vector (CSD vector) D = 0010 1 001 (decimal 32 – 8 + 1 = 23) since this requires only three add or subtract operations (and a subtraction is as easy as an addition). We say B has a weight of 4 and D has a weight of 3. By using D instead of B we have reduced the number of partial products by 1 (= 4 – 3).

We can recode (or encode) any binary number, B, as a CSD vector, D, as follows (canonical means there is only one CSD vector for any number):

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

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: $D_0 = B_0 + C_0 - 2C_1 = 1 + 0 - 2 = \bar{1}$,

$D_1 = B_1 + C_1 - 2C_2 = 1 + 1 - 2 = 0$,

$D_2 = B_2 + C_2 - 2C_3 = 0 + 1 - 0 = 1$,

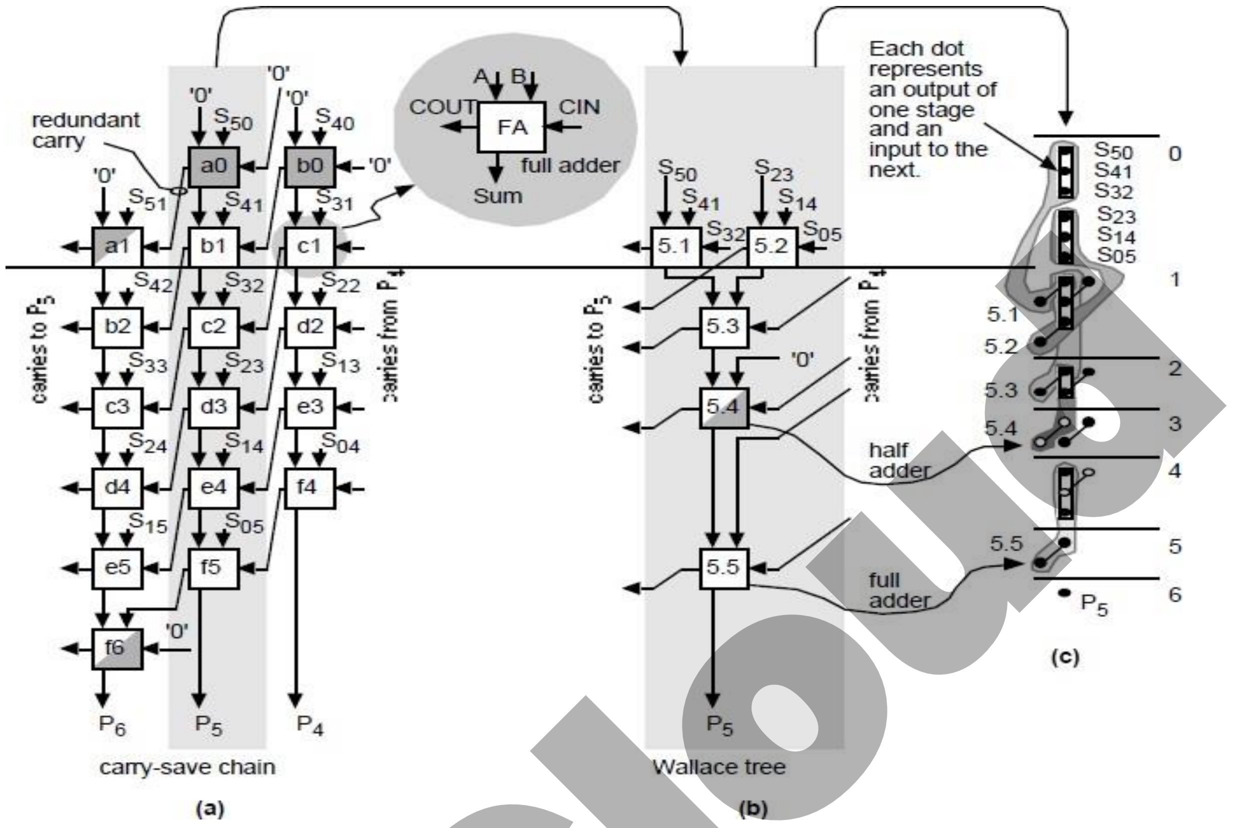
so that D= $10\bar{1}$ (decimal 4–1=3).

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

B=101001 (decimal 9–32=–23) ⇒ E=$\bar{1}\,\bar{2}1$ (decimal –16–8+1=–23)

B=01011 (eleven) ⇒ E=$1\bar{1}\,\bar{1}$ (16–4–1)

B=101 ⇒ E=$\bar{1}1$

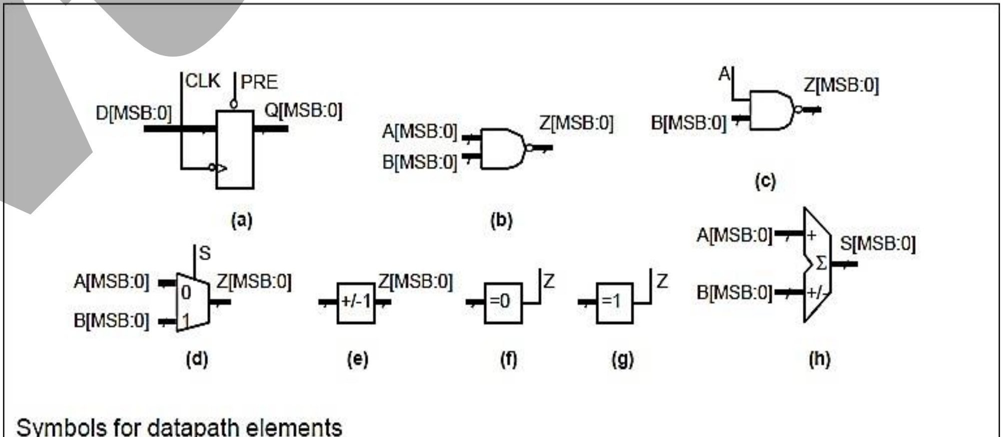


Tree-based multiplication. (a) The portion of above Figure that calculates the sum bit, P_5 , using a chain of adders (cells a0–f5). (b) We can collapse this chain to a Wallace tree (cells 5.1–5.5). (c) The stages of multiplication.

Other Datapath Operators:

Full subtracter

$$\begin{aligned} \mathrm{DIFF} &= A \oplus \mathrm{NOT}(B) \oplus \mathrm{NOT}(BIN) \\ &= \mathrm{SUM}(A, \mathrm{NOT}(B), \mathrm{NOT}(BIN)) \\ \mathrm{NOT}(BOUT) &= A \cdot \mathrm{NOT}(B) + A \cdot \mathrm{NOT}(BIN) + \mathrm{NOT}(B) \cdot \mathrm{NOT}(BIN) \\ &= \mathrm{MAJ}(\mathrm{NOT}(A), B, \mathrm{NOT}(BIN)) \end{aligned}$$

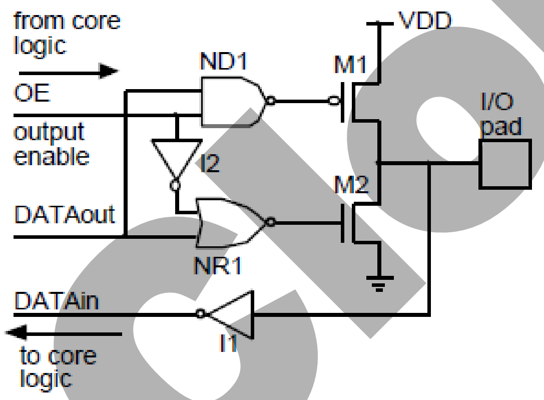


Symbols for datapath elements

I/O Cells:

A three-state bidirectional output buffer (Tri-State ® is a registered trademark of National Semiconductor). 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 , M1 and M2, are disconnected. This allows multiple drivers to be connected on a bus. It is up to the designer to make sure that a bus never has two drivers—a problem known as contention.

In order to prevent the problem opposite to contention—a bus floating to an intermediate voltage when there are no bus drivers—we can use a bus keeper or bus-hold cell (TI calls this Bus-Friendly logic). A bus keeper normally acts like two weak (low drive-strength) cross-coupled inverters that act as a latch to retain the last logic state on the bus, but the latch is weak enough that it may be driven easily to the opposite state. Even though bus keepers act like latches, and will simulate like latches, they should not be used as latches, since their drive strength is weak.



The three-state buffer allows us to employ the same pad for input and output— bidirectional I/O . When we want to use the pad as an input, we set OE low and take the data from DATA in. Of course, it is not necessary to have all these features on every pad: We can build output-only or input-only pads.

Cell Compiler:

The process of hand crafting circuits and layout for a full-custom IC is a tedious, time-consuming, and error-prone task.

There are two types of automated layout assembly tools, often known as a silicon compilers.

1. The first type produces a specific kind of circuit, a RAM compiler or multiplier compiler etc….
2. The second type of compiler is more flexible, usually providing a programming language that assembles or tiles layout from an input command file, but this is full-custom IC design.

We can build a register file from latches or flip-flops, but, at 4.5–6.5 gates (18–26 transistors) per bit, this is an expensive way to build memory. Dynamic RAM (DRAM) can use a cell with only one transistor, storing charge on a capacitor that has to be periodically refreshed as the charge leaks away. ASIC RAM is invariably static (SRAM), so we do not need to refresh the bits. When we refer to RAM in an ASIC environment we almost always mean SRAM. Most ASIC RAMs use a six-transistor cell (four transistors to form two cross-coupled inverters that form the storage loop, and two more transistors to allow us to read from and write to the cell). RAM compilers are available that produce single-port RAM (a single shared bus for read and write) as well as dual-port RAMs , and multiport RAMs . In a multi-port RAM the compiler may or may not handle the problem of address contention (attempts to read and write to the same RAM address simultaneously). RAM can be asynchronous (the read and write cycles are triggered by control and/or address transitions asynchronous to a clock) or synchronous (using the system clock).

In addition to producing layout we also need a model compiler so that we can verify the circuit at the behavioral level, and we need a netlist from a netlist compiler so that we can simulate the circuit and verify that it works correctly at the structural level. Silicon compilers are thus complex pieces of software. We assume that a silicon compiler will produce working silicon even if every configuration has not been tested. This is still ASIC design, but now we are relying on the fact that the tool works correctly and therefore the compiled blocks are correct by construction.