



ECE-332:437

DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 6

Arithmetic Circuits




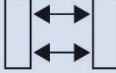
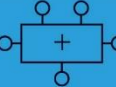
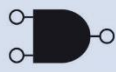
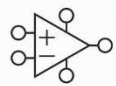


Nagi Naganathan
September 29, 2016

Topics to cover today – September 29, 2016

- Lecture 6 – Chapter 5 – Digital Building Blocks
 - Adders
 - Memory – SRAM ...

Chapter 5 :: Topics

- Introduction
- Arithmetic Circuits
- Number Systems
- Sequential Building Blocks
- Memory Arrays
- Logic Arrays

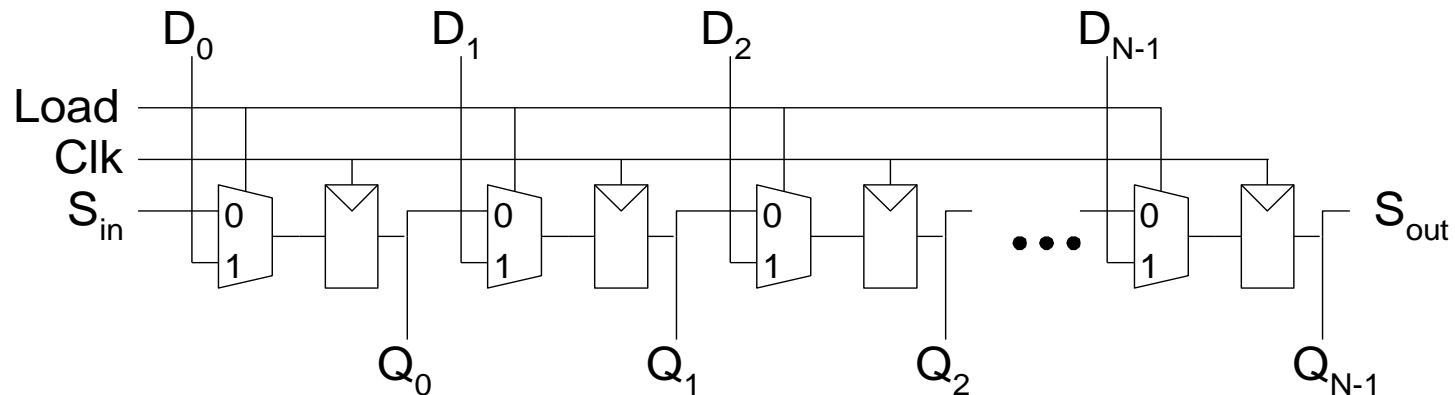
| | |
|----------------------|---|
| Application Software |  |
| Operating Systems |  |
| Architecture |  |
| Micro-architecture |  |
| Logic |  |
| Digital Circuits |  |
| Analog Circuits |  |
| Devices |  |
| Physics |  |

Introduction

- **Digital building blocks:**
 - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
 - Hierarchy of simpler components
 - Well-defined interfaces and functions
 - Regular structure easily extends to different sizes
- **Will use these building blocks in Chapter 7 to build microprocessor**

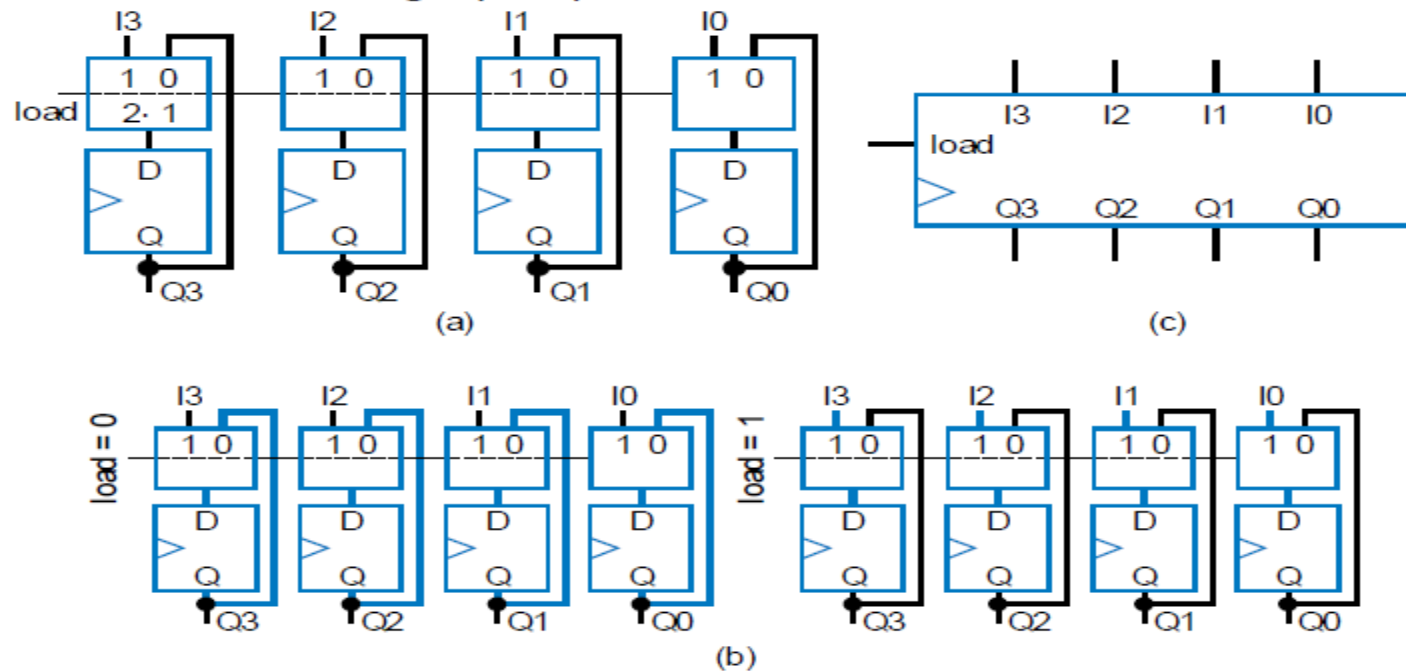
Shift Register with Parallel Load

- When $Load = 1$, acts as a normal N -bit register
- When $Load = 0$, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})



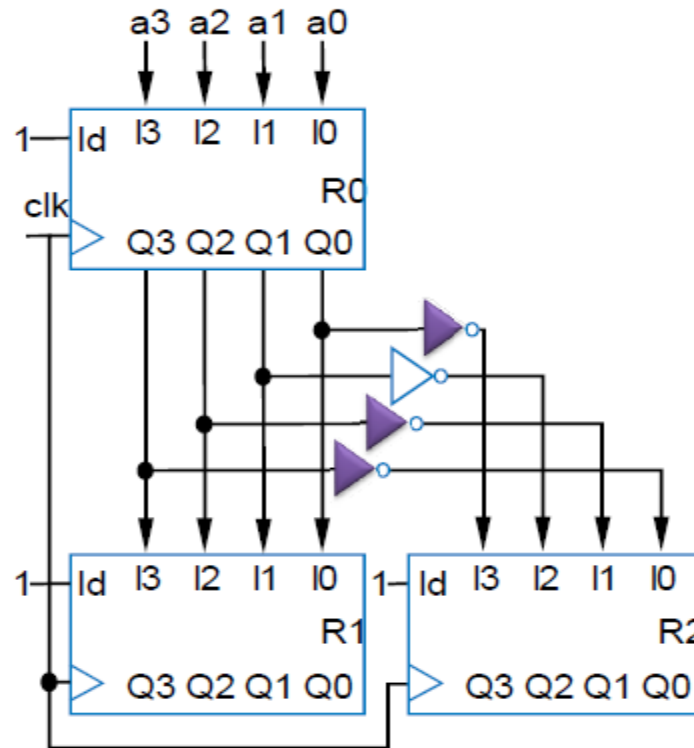
Register with Parallel Load

- Add 2x1 mux to front of each flip-flop
- Register's load input selects mux input to pass
 - Either existing flip-flop value, or new value to load



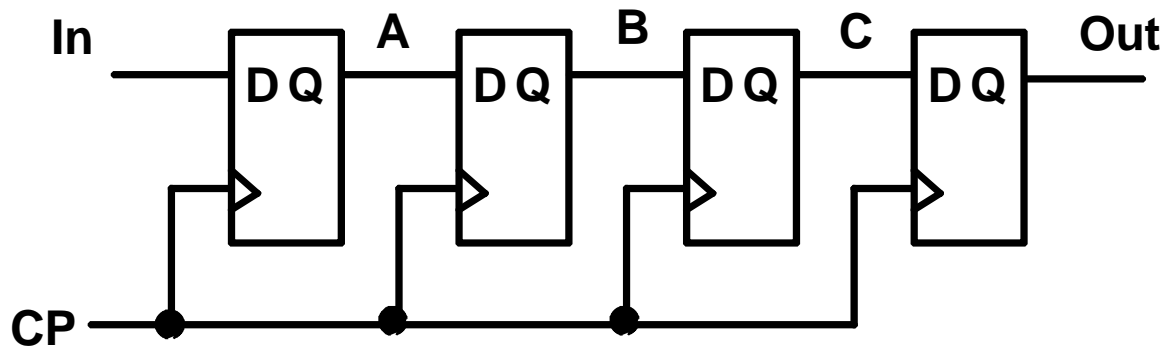
Register with Parallel Load

- This example will show how registers load simultaneously on clock cycles
 - Notice that all load inputs set to 1 in this example -- just for demonstration purposes



Shift Registers

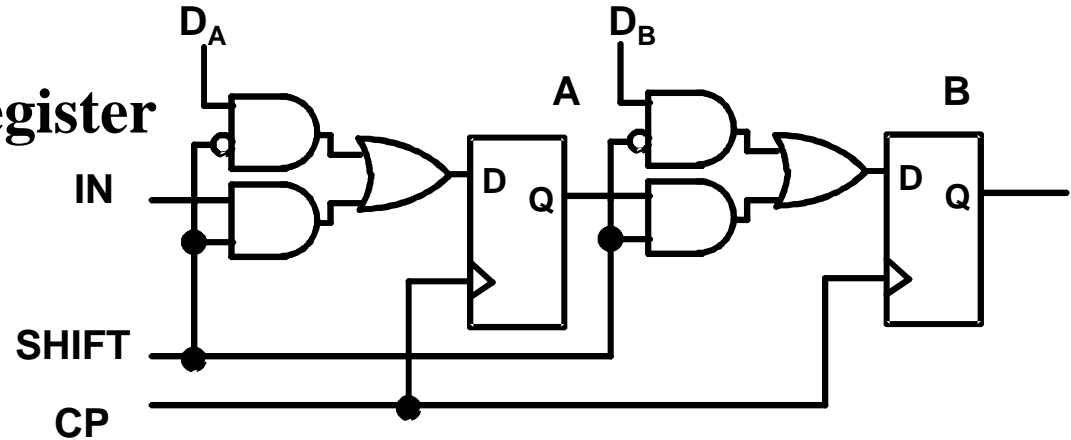
- Shift Registers move data laterally within the register toward its MSB or LSB position
- In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:



- Data input, In, is called a *serial input* or the *shift right input*.
- Data output, Out, is often called the *serial output*.
- The vector (A, B, C, Out) is called the *parallel output*.

Parallel Load Shift Registers

- By adding a mux between each shift register stage, data can be shifted or loaded
- If SHIFT is low, A and B are replaced by the data on D_A and D_B lines, else data shifts right on each clock.
- By adding more bits, we can make n -bit parallel load shift registers.
- A parallel load shift register with an added “hold” operation that stores data unchanged is given in Figure 7-10 of the text.

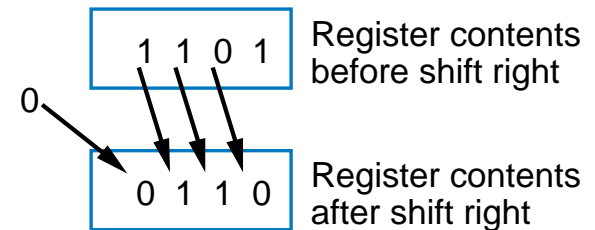


Shift Registers with Additional Functions

- By placing a 4-input multiplexer in front of each D flip-flop in a shift register, we can implement a circuit with shifts right, shifts left, parallel load, hold.
- Shift registers can also be designed to shift more than a single bit position right or left
- Shift registers can be designed to shift a variable number of bit positions specified by a variable called a *shift amount*.

Shift Register

- Shift right
 - Move each bit one position right
 - Rightmost bit is “dropped”
 - Assume 0 shifted into leftmost bit



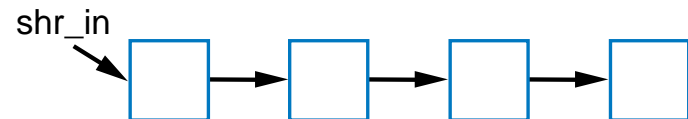
Q: Do four right shifts on 1001, showing value after each shift

A:

| | |
|------|------------|
| 1001 | (original) |
| 0100 | |
| 0010 | |
| 0001 | |
| 0000 | |

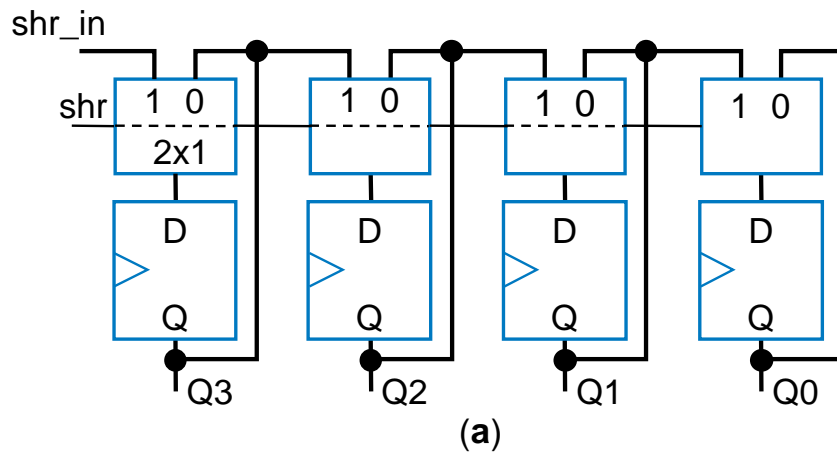
A dashed arrow points from the original value 1001 down to 0000, indicating the sequence of shifts.

- Implementation: Connect flip-flop output to next flip-flop's input

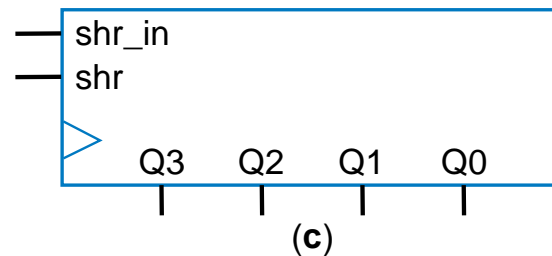
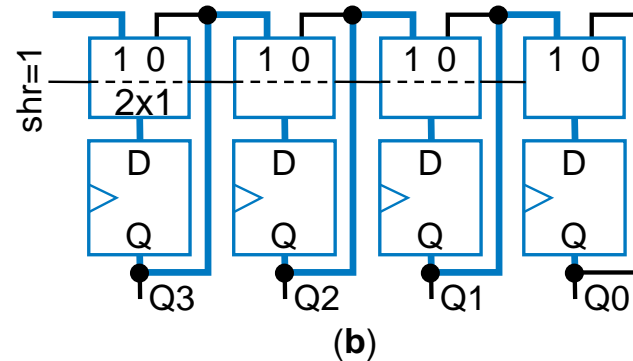


Shift Register

- To allow register to either shift or retain, use 2x1 muxes
 - shr: “0” means retain, “1” shift
 - shr_in: value to shift in
 - May be 0, or 1

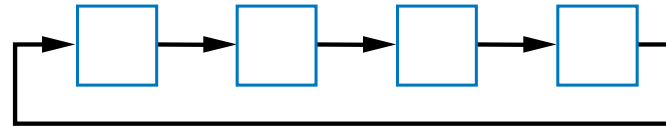
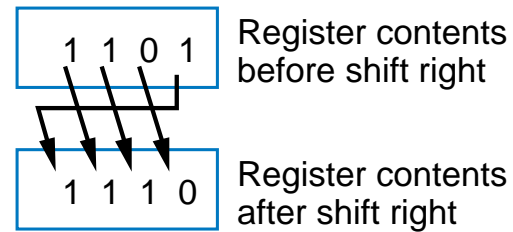


Left-shift register also easy to design



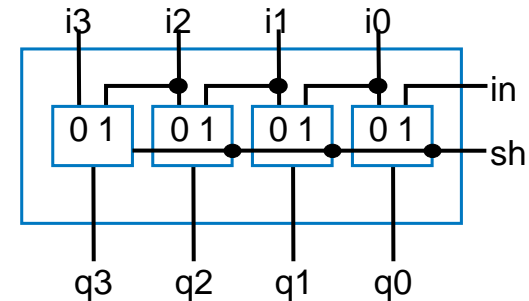
Rotate Register

- Rotate right: Like shift right, but leftmost bit comes from rightmost bit



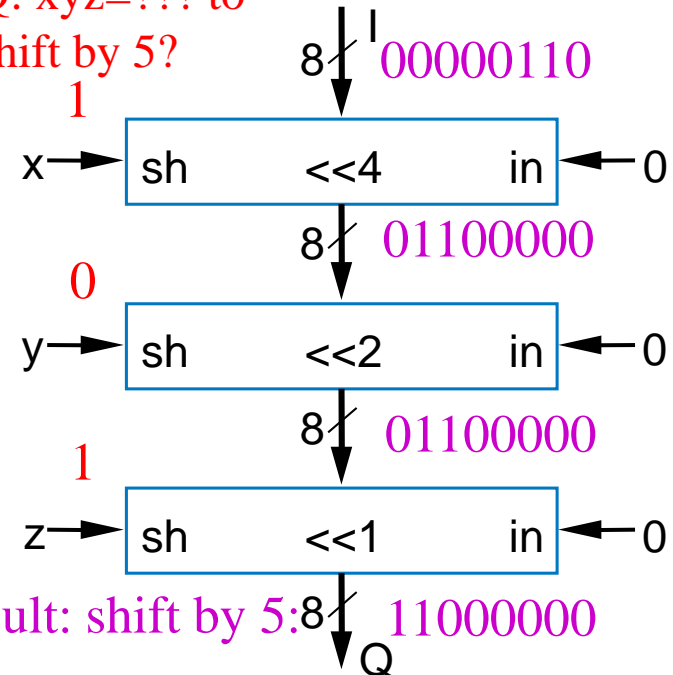
Barrel Shifter

- A shifter that can shift by any amount
 - 4-bit barrel left shifter can shift left by 0, 1, 2, or 3 positions
 - 8-bit barrel left shifter can shift left by 0, 1, 2, 3, 4, 5, 6, or 7 positions
 - (Shifting an 8-bit number by 8 positions is pointless -- you just lose all the bits)
- Could design using 8x1 muxes
 - Too many wires
- More elegant design
 - Chain three shifters: 4, 2, and 1
 - Can achieve any shift of 0..7 by enabling the correct combination of those three shifters, i.e., shifts should sum to desired amount



Shift by 1 shifter uses 2x1 muxes. 8x1 mux solution for 8-bit barrel shifter: too many wires.

Q: xyz=??? to shift by 5?



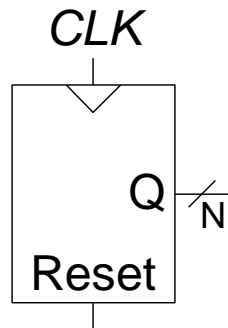
Net result: shift by 5: 11000000



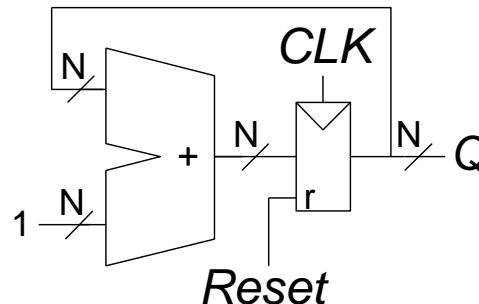
Counters

- Increments on each clock edge.
- Used to cycle through numbers. For example,
 - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
 - Counts sequence such as A,B,C,D,
- Example uses:
 - Digital clock displays
 - Program counter: keeps track of current instruction executing

Symbol



Implementation

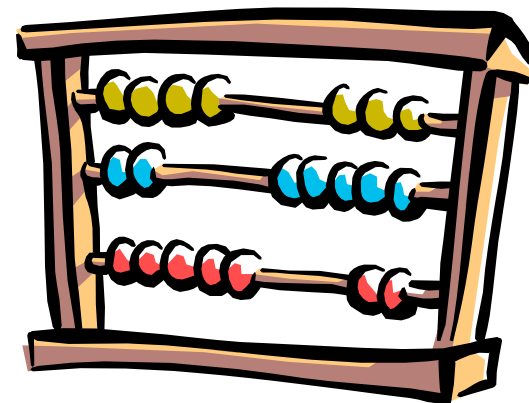


Counters

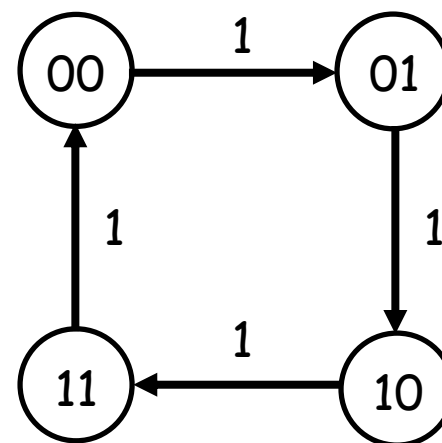
- **Counters** are sequential circuits which "count" through a specific state sequence. They can count up, count down, or count through other fixed sequences. Two distinct types are in common usage:
- **Ripple Counters**
 - Clock connected to the flip-flop clock input on the LSB bit flip-flop
 - For all other bits, a flip-flop output is connected to the clock input, thus circuit is not truly synchronous!
 - Output change is delayed more for each bit toward the MSB.
 - Resurgent because of low power consumption
- **Synchronous Counters**
 - Clock is directly connected to the flip-flop clock inputs
 - Logic is used to implement the desired state sequencing

Counters (adapted from various sources)

- Counters are a specific type of sequential circuit.
- Like registers, the state, or the flip-flop values themselves, serves as the “output.”
- The output value increases by one on each clock cycle.
- After the largest value, the output “wraps around” back to 0.
- Using two bits, we’d get something like this:



| Present State | | Next State | |
|---------------|---|------------|---|
| A | B | A | B |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |



Counter

- Modulus is the number of states in the cycle
- Modulo-k Counter
 - Modulo-4 counter – 0,1,2,3, 0,1,2,3
 - Modulo-5 counter – 0, 1,2,3,4, 0,1,2,3,4
- M-to-N Counter
 - 3-to-8 Counter – 3,4,5,6,7,8, 3,4 ...
- Clocked Sequential Circuit whose state diagram contains a single cycle

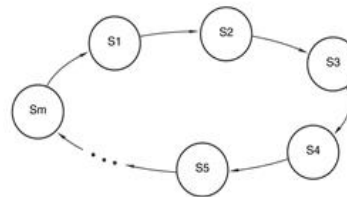


Figure 8-23

General structure of a counter's state diagram—a single cycle.

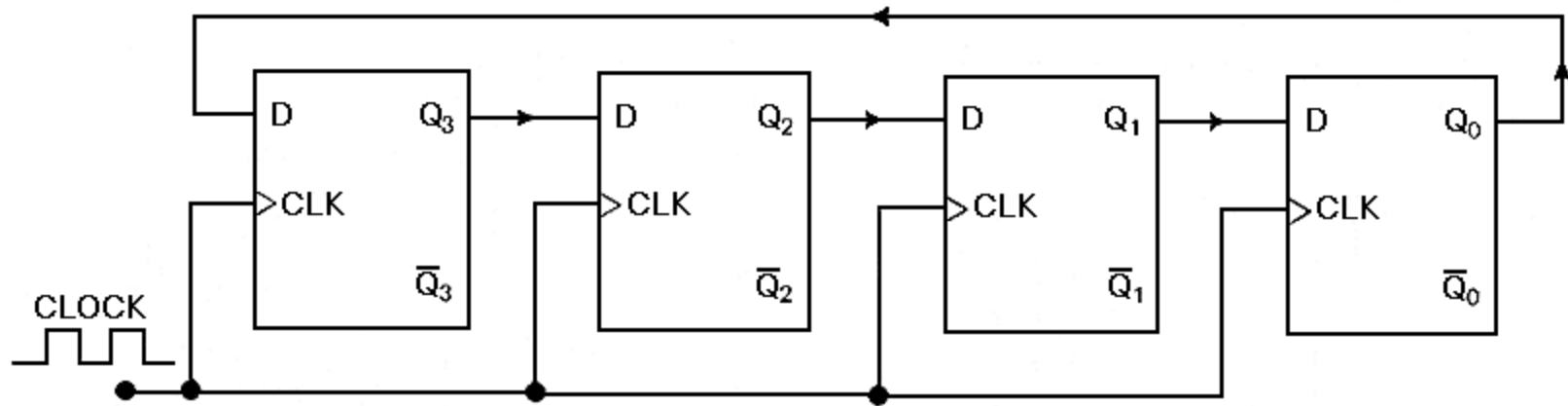
Counters

- Special sequential circuits (FSMs) that sequence through a set of outputs.
- Examples:
 - binary counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, 001, ...
 - gray code counter:
000, 010, 110, 100, 101, 111, 011, 001, 000, 010, 110, ...
 - one-hot counter: 0001, 0010, 0100, 1000, 0001, 0010, ...
 - BCD counter: 0000, 0001, 0010, ..., 1001, 0000, 0001
 - pseudo-random sequence generators: 10, 01, 00, 11, 10, 01, 00, ...

Benefits of counters (adapted from various sources)

- Counters can act as simple clocks to keep track of “time.”
- You may need to record how many times something has happened.
 - How many bits have been sent or received?
 - How many steps have been performed in some computation?
- All processors contain a **program counter**, or **PC**.
 - Programs consist of a list of instructions that are to be executed one after another (for the most part).
 - The PC keeps track of the instruction currently being executed.
 - The PC increments once on each clock cycle, and the next program instruction is then executed.

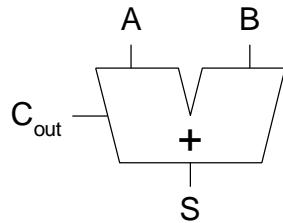
❖ Ring Counter



Ring counters are implemented using shift registers. It is essentially a circulating shift register connected so that the last flip-flop shifts its value into the first flip-flop. There is usually only a single 1 circulating in the register, as long as clock pulses are applied. (Starts 1000- \rightarrow 0100- \rightarrow 0010- \rightarrow 0001 repeat)

1-Bit Adders

Half Adder

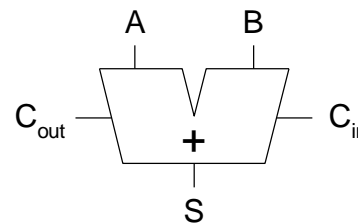


| A | B | C_{out} | S |
|---|---|-----------|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

$$S =$$

$$C_{out} =$$

Full Adder



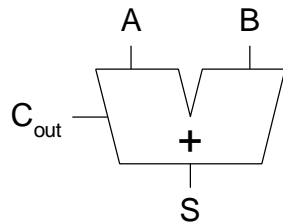
| C_{in} | A | B | C_{out} | S |
|----------|---|---|-----------|---|
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

$$S =$$

$$C_{out} =$$

1-Bit Adders

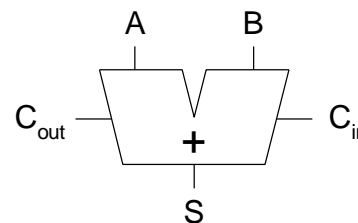
Half Adder



| A | B | C_{out} | S |
|---|---|-----------|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$\begin{matrix} S \\ C_{out} \end{matrix} =$$

Full Adder

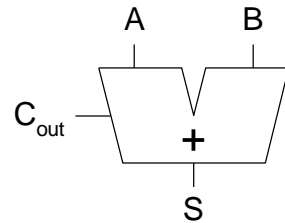


| C_{in} | A | B | C_{out} | S |
|----------|---|---|-----------|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$\begin{matrix} S \\ C_{out} \end{matrix} =$$

1-Bit Adders

Half Adder

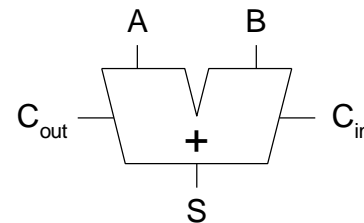


| A | B | C_{out} | S |
|---|---|-----------|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$S = A \oplus B$$

$$C_{out} = AB$$

Full Adder



| C_{in} | A | B | C_{out} | 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 |

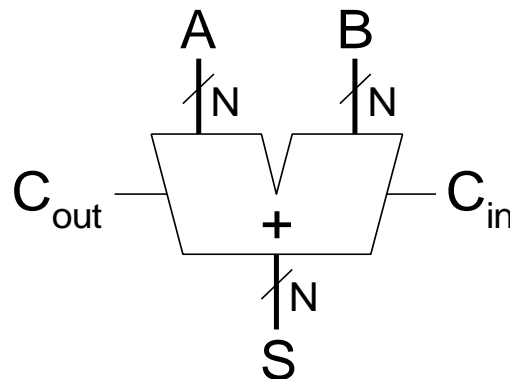
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Multibit Adders (CPAs)

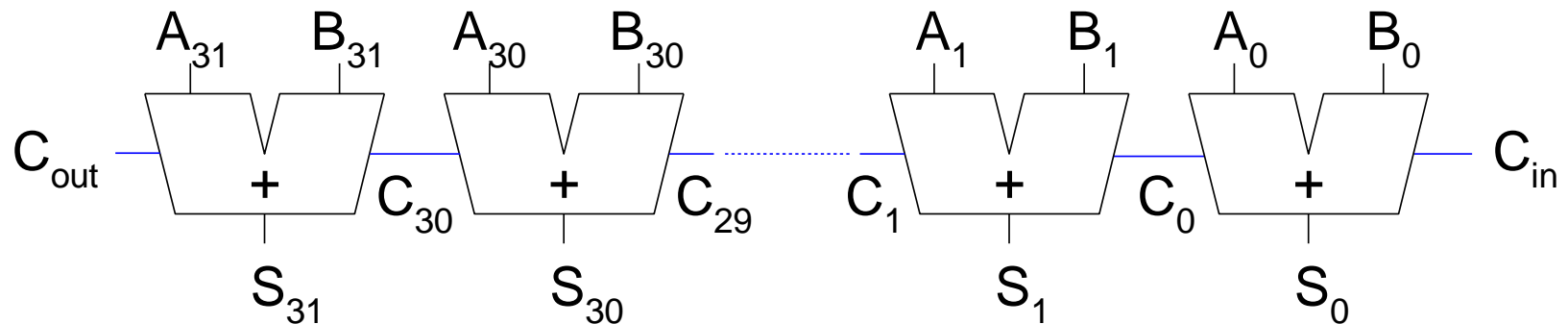
- Types of carry propagate adders (CPAs):
 - Ripple-carry (slow)
 - Carry-lookahead (fast)
 - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



Ripple-Carry Adder Delay

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

Carry-Lookahead Adder

- Compute carry out (C_{out}) for k -bit blocks using *generate* and *propagate* signals
- **Some definitions:**
 - Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out
 - Generate (G_i) and propagate (P_i) signals for each column:
 - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

- Column i will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

- The carry out of column i (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$



Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k -bit blocks
- **Step 3:** C_{in} propagates through each k -bit propagate/generate block

Carry-Lookahead Adder

- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

$$P_{3:0} = P_3 P_2 P_1 P_0$$

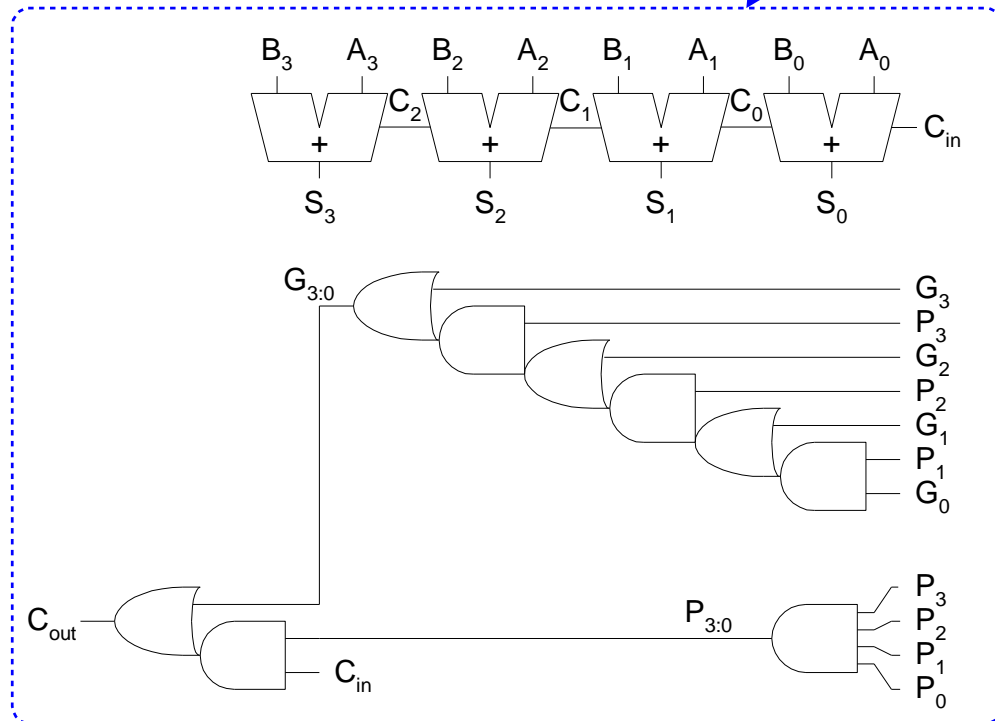
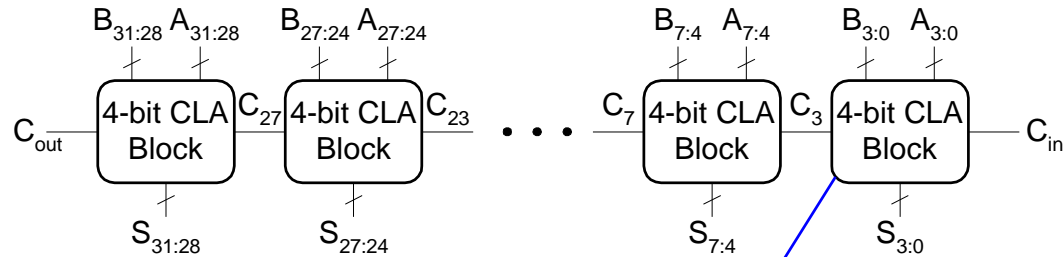
- **Generally,**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$

32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

For N -bit CLA with k -bit blocks:

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}, G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k -bit CLA block

An N -bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$



Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
- $\log_2 N$ stages

Prefix Adder

- Carry in either *generated* in a column or *propagated* from a previous column.

- Column -1 holds C_{in} , so

$$G_{-1} = C_{in}, P_{-1} = 0$$

- Carry in to column i = carry out of column $i-1$:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$: generate signal spanning columns $i-1$ to -1

- Sum equation:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, ... (called *prefixes*)

Prefix Adder

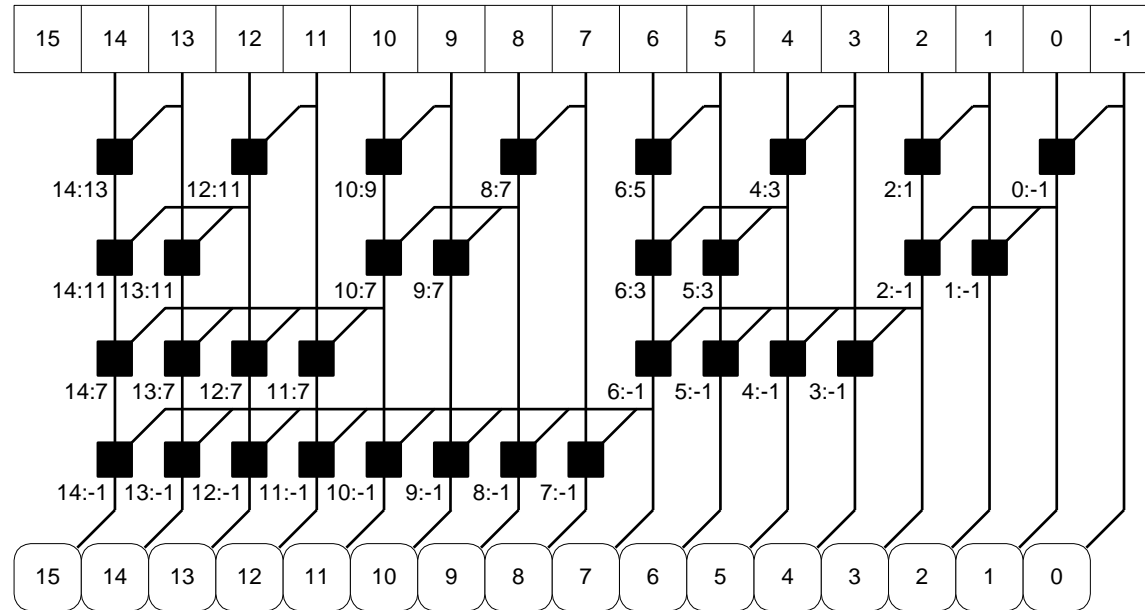
- Generate and propagate signals for a block spanning bits $i:j$:

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

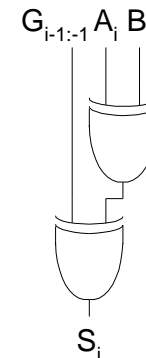
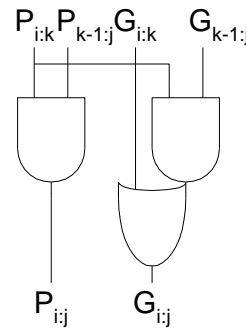
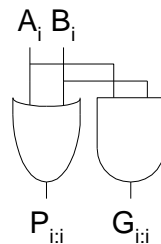
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
 - **Generate:** block $i:j$ will generate a carry if:
 - upper part ($i:k$) generates a carry or
 - upper part propagates a carry generated in lower part ($k-1:j$)
 - **Propagate:** block $i:j$ will propagate a carry if *both* the upper and lower parts propagate the carry

Prefix Adder Schematic



Legend



Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR}$$

- t_{pg} : delay to produce $P_i G_i$ (AND or OR gate)
- t_{pg_prefix} : delay of black prefix cell (AND-OR gate)

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

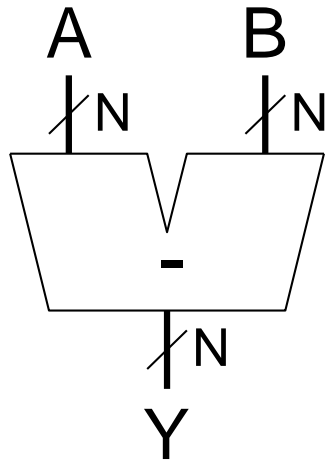
$$\begin{aligned}t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}}\end{aligned}$$

$$\begin{aligned}t_{CLA} &= t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}}\end{aligned}$$

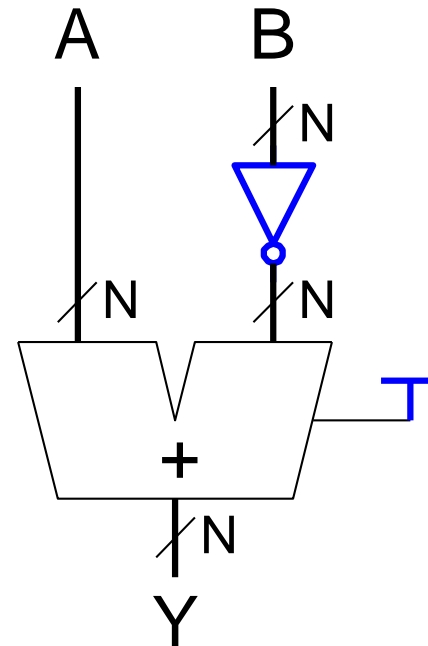
$$\begin{aligned}t_{PA} &= t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR} \\ &= [100 + \log_2 32(200) + 100] \text{ ps} \\ &= \mathbf{1.2 \text{ ns}}\end{aligned}$$

Subtractor

Symbol

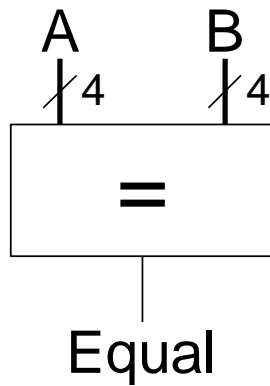


Implementation

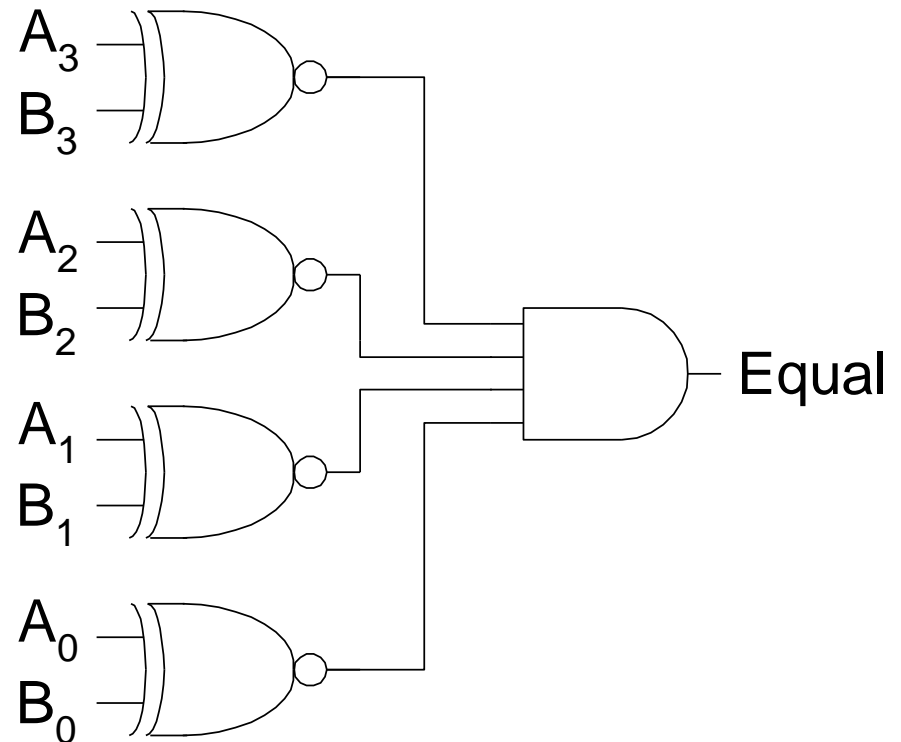


Comparator: Equality

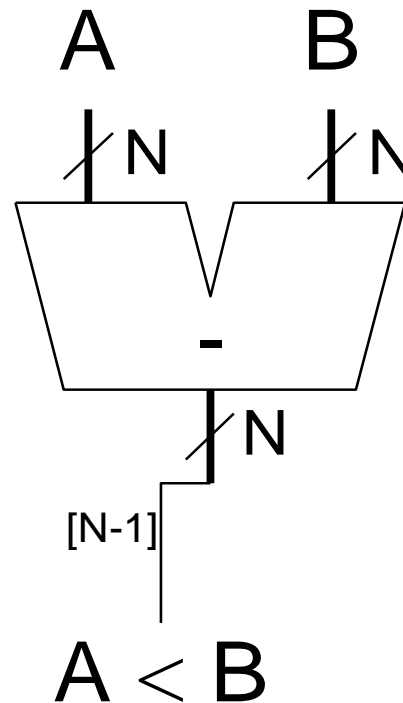
Symbol



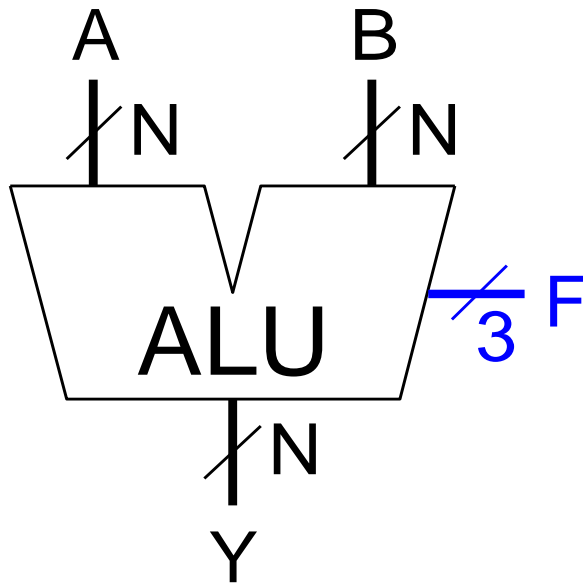
Implementation



Comparator: Less Than

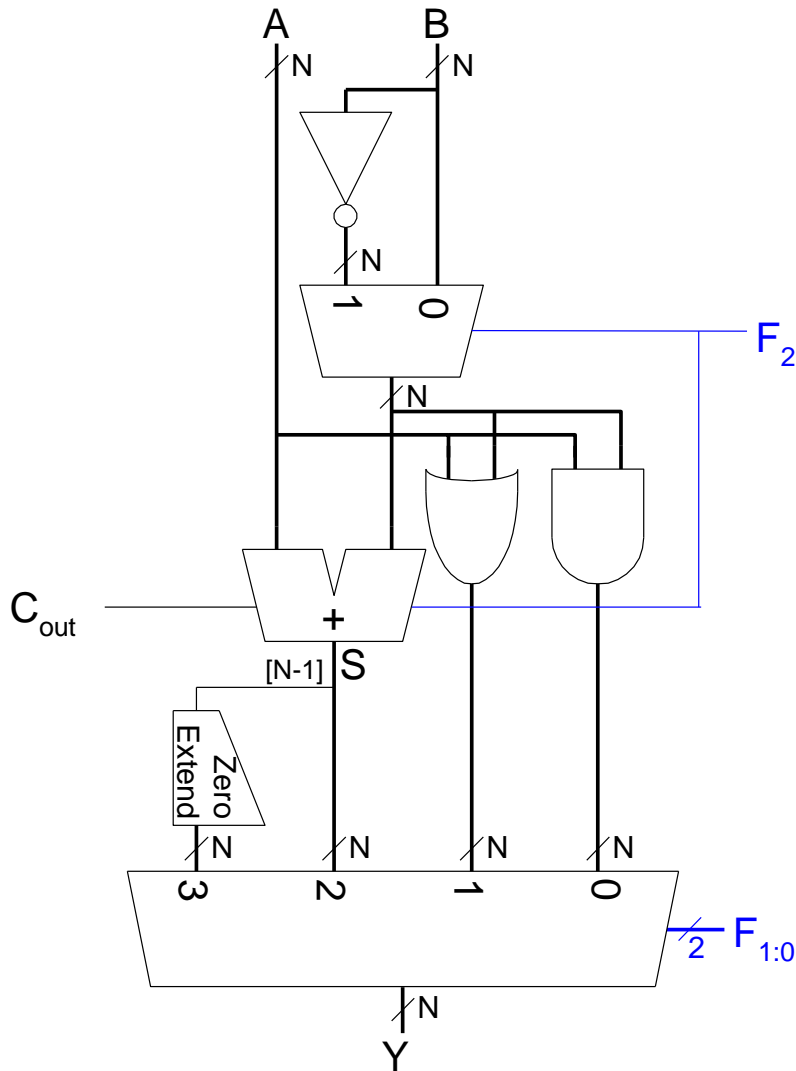


Arithmetic Logic Unit (ALU)



| $F_{2:0}$ | Function |
|-----------|---------------|
| 000 | $A \& B$ |
| 001 | $A B$ |
| 010 | $A + B$ |
| 011 | not used |
| 100 | $A \& \sim B$ |
| 101 | $A \sim B$ |
| 110 | $A - B$ |
| 111 | SLT |

ALU Design



| $F_{2:0}$ | Function |
|-----------|---------------|
| 000 | $A \& B$ |
| 001 | $A B$ |
| 010 | $A + B$ |
| 011 | not used |
| 100 | $A \& \sim B$ |
| 101 | $A \sim B$ |
| 110 | $A - B$ |
| 111 | SLT |

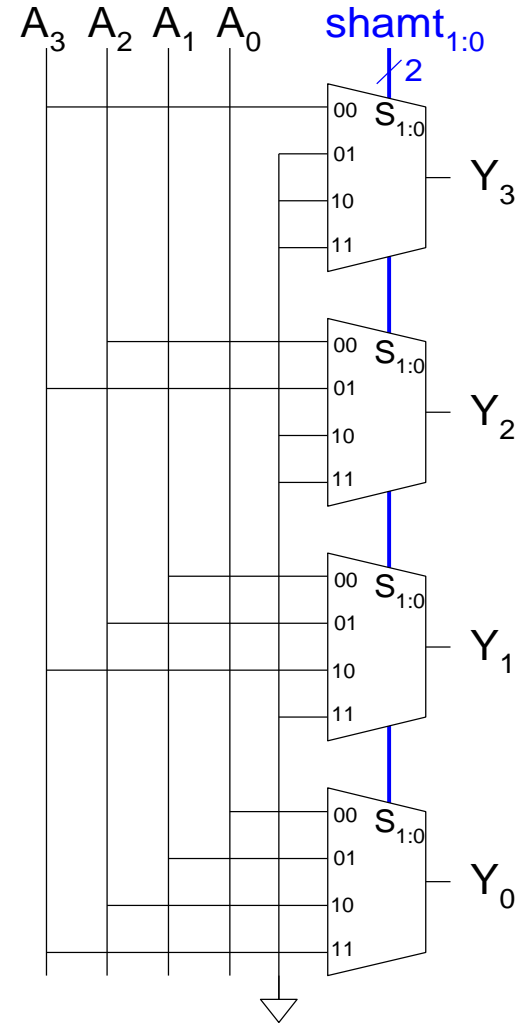
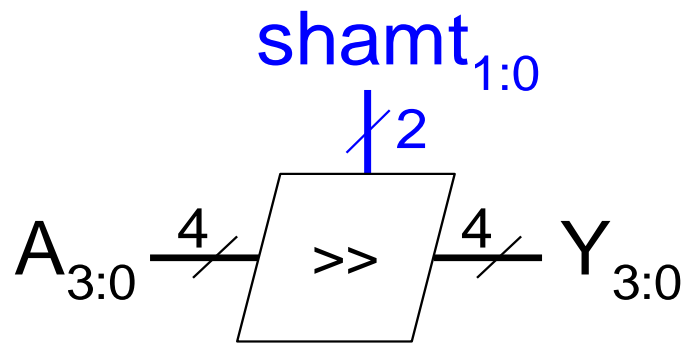
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
 - Ex: $11001 \gg 2 =$
 - Ex: $11001 \ll 2 =$
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
 - Ex: $11001 \ggg 2 =$
 - Ex: $11001 \lll 2 =$
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
 - Ex: $11001 \text{ ROR } 2 =$
 - Ex: $11001 \text{ ROL } 2 =$

Shifters

- **Logical shifter:**
 - Ex: $11001 \gg 2 = 00110$
 - Ex: $11001 \ll 2 = 00100$
- **Arithmetic shifter:**
 - Ex: $11001 \ggg 2 = 11110$
 - Ex: $11001 \lll 2 = 00100$
- **Rotator:**
 - Ex: $11001 \text{ ROR } 2 = 01110$
 - Ex: $11001 \text{ ROL } 2 = 00111$

Shifter Design



Shifters as Multipliers, Dividers

- $A \ll N = A \times 2^N$
 - **Example:** $00001 \ll 2 = 00100$ ($1 \times 2^2 = 4$)
 - **Example:** $11101 \ll 2 = 10100$ ($-3 \times 2^2 = -12$)
- $A \gg N = A \div 2^N$
 - **Example:** $01000 \gg 2 = 00010$ ($8 \div 2^2 = 2$)
 - **Example:** $10000 \gg 2 = 11100$ ($-16 \div 2^2 = -4$)

Multipliers

- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

Decimal

$$\begin{array}{r} 230 \\ \times 42 \\ \hline 460 \\ + 920 \\ \hline 9660 \end{array}$$

$$230 \times 42 = 9660$$

Binary

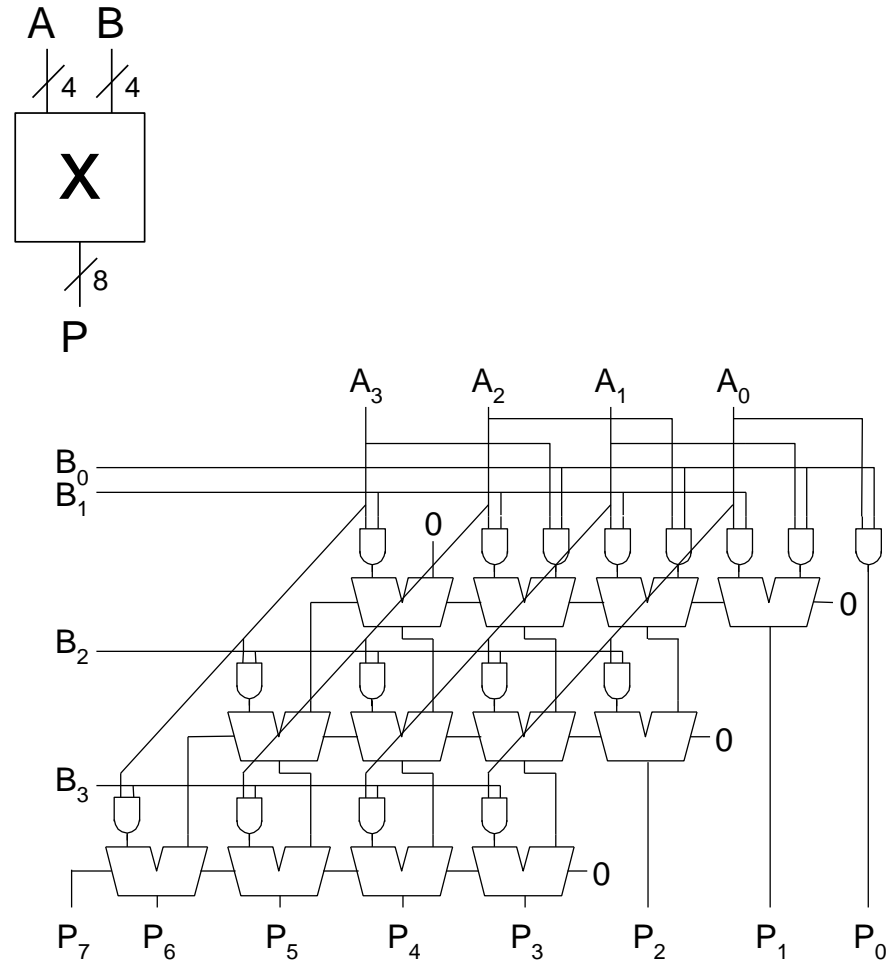
| | |
|--------------|---------|
| multiplicand | 0101 |
| multiplier | x 0111 |
| | <hr/> |
| partial | 0101 |
| products | 0101 |
| | 0101 |
| | + 0000 |
| | <hr/> |
| result | 0100011 |

$$5 \times 7 = 35$$

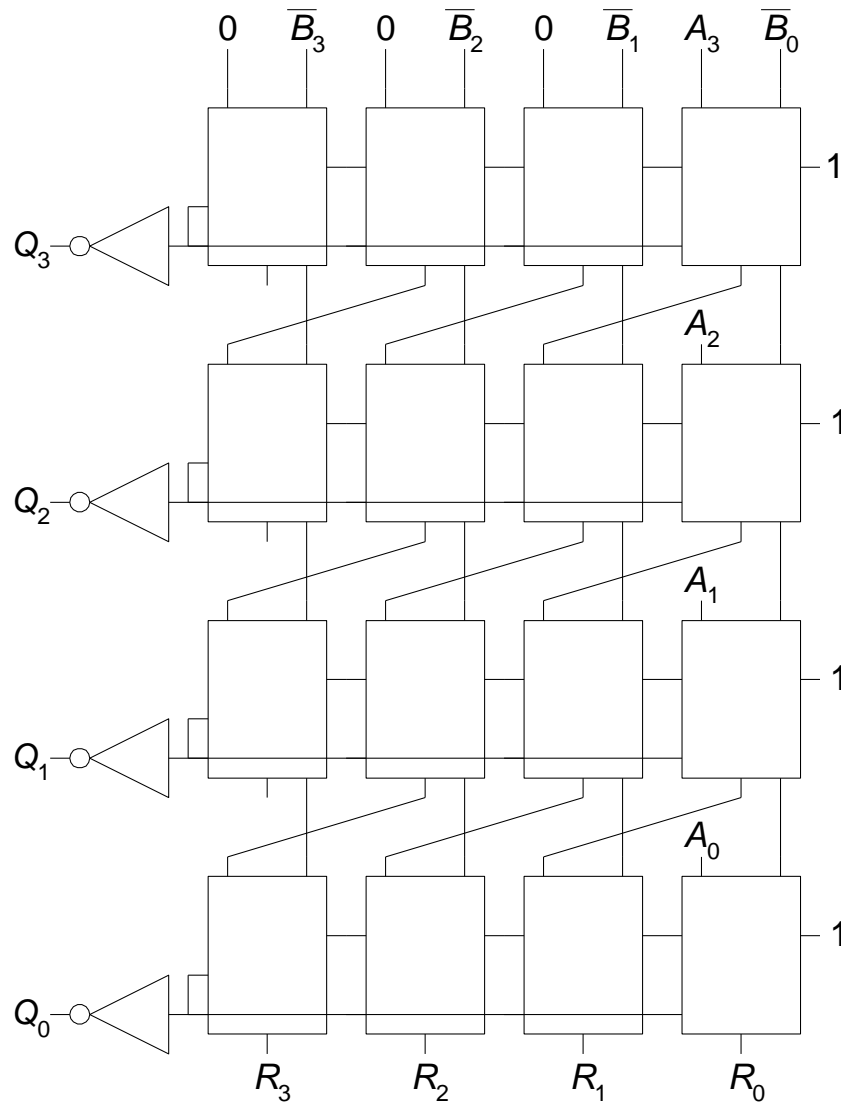


4 x 4 Multiplier

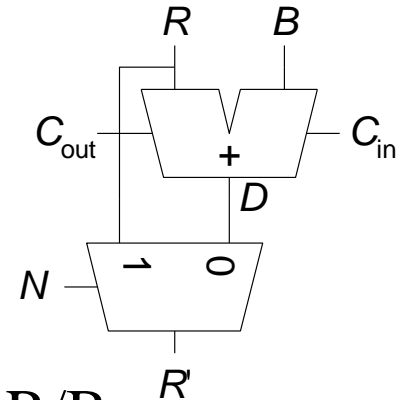
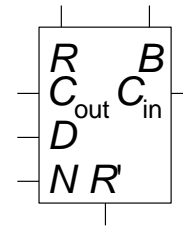
$$\begin{array}{r}
 \begin{array}{cccc}
 & A_3 & A_2 & A_1 & A_0 \\
 \times & B_3 & B_2 & B_1 & B_0 \\
 \hline
 & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & \\
 + A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & \\
 \hline
 P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
 \end{array}
 \end{array}$$



4 x 4 Divider



Legend



$$A/B = Q + R/B$$

Algorithm:

$$R' = 0$$

for $i = N-1$ to 0

$$R = \{R' \ll 1, A_i\}$$

$$D = R - B$$

if $D < 0$, $Q_i = 0$, $R' = R$

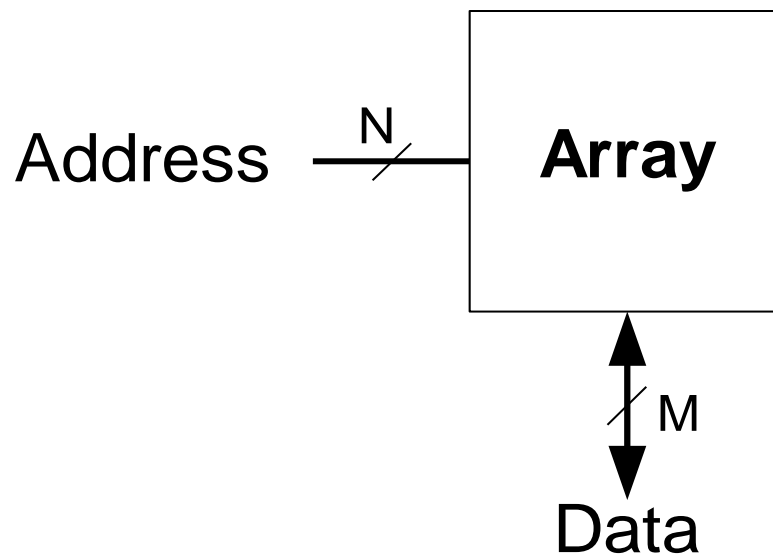
else $Q_i = 1$, $R' = D$

$$R' = R$$



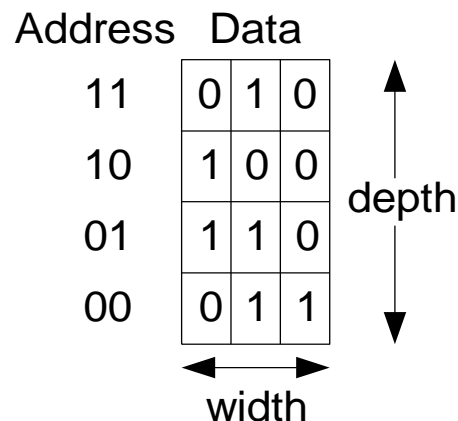
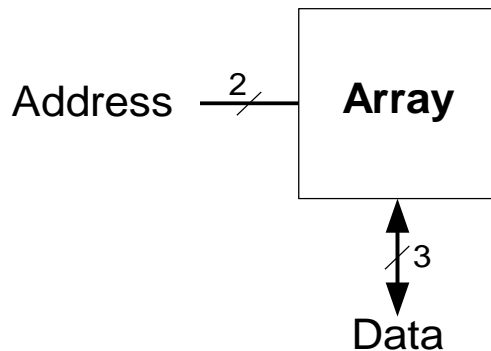
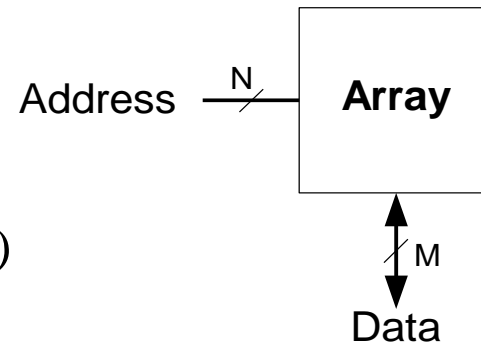
Memory Arrays

- Efficiently store large amounts of data
- 3 common types:
 - Dynamic random access memory (DRAM)
 - Static random access memory (SRAM)
 - Read only memory (ROM)
- M -bit data value read/ written at each unique N -bit address



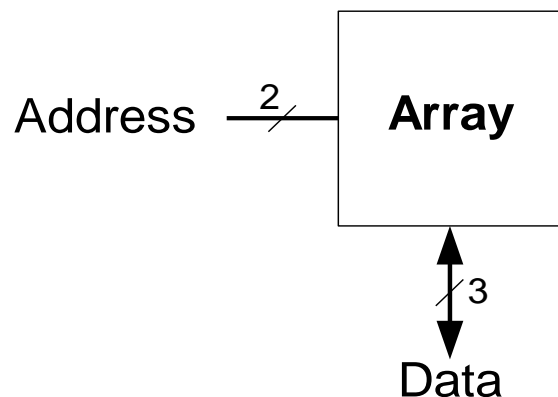
Memory Arrays

- 2-dimensional array of bit cells
- Each bit cell stores one bit
- N address bits and M data bits:
 - 2^N rows and M columns
 - **Depth:** number of rows (number of words)
 - **Width:** number of columns (size of word)
 - **Array size:** depth \times width = $2^N \times M$



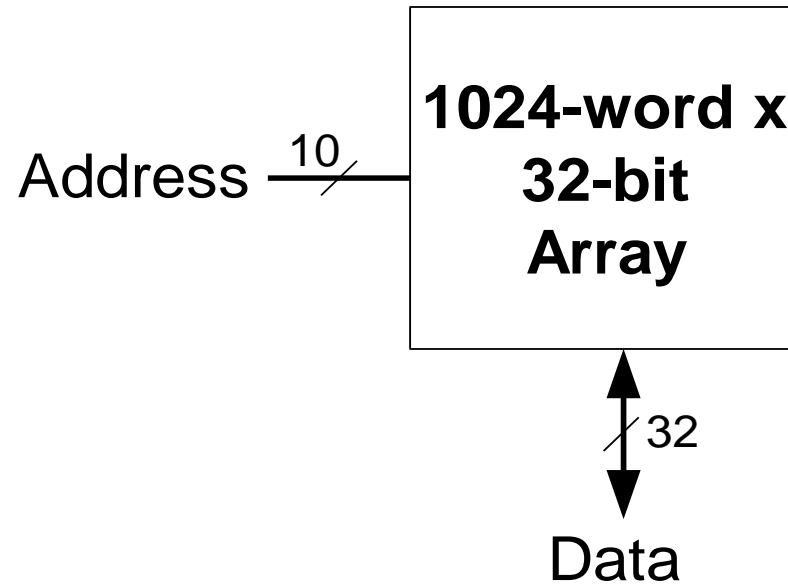
Memory Array Example

- $2^2 \times 3$ -bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

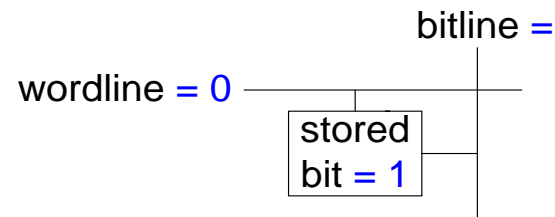
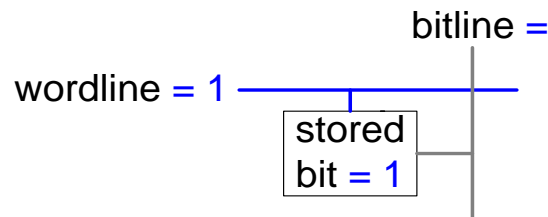
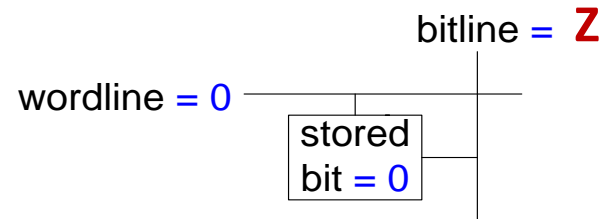
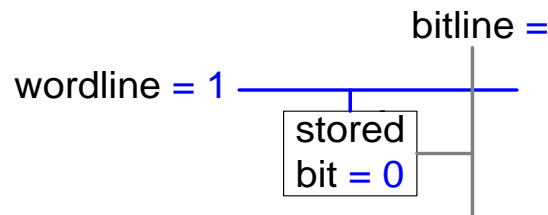
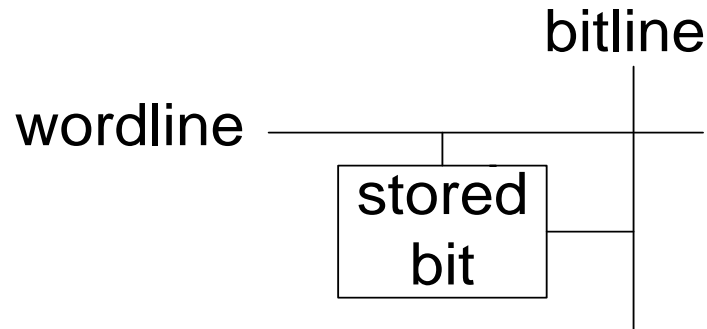


| Address | Data | | | |
|---------|-------------|---|---|-----------------|
| 11 | 0 | 1 | 0 | depth ↑ ↓ |
| 10 | 1 | 0 | 0 | |
| 01 | 1 | 1 | 0 | |
| 00 | 0 | 1 | 1 | |
| | width ←→ | | | |

Memory Arrays



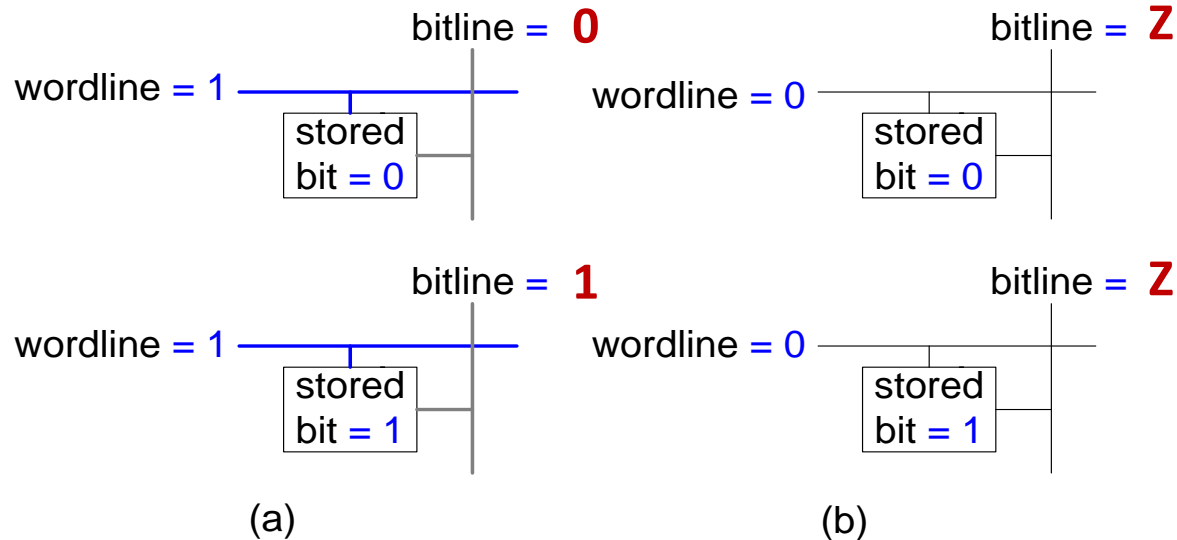
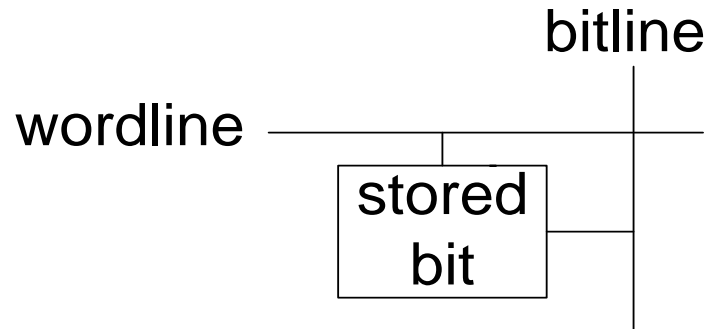
Memory Array Bit Cells



(a)

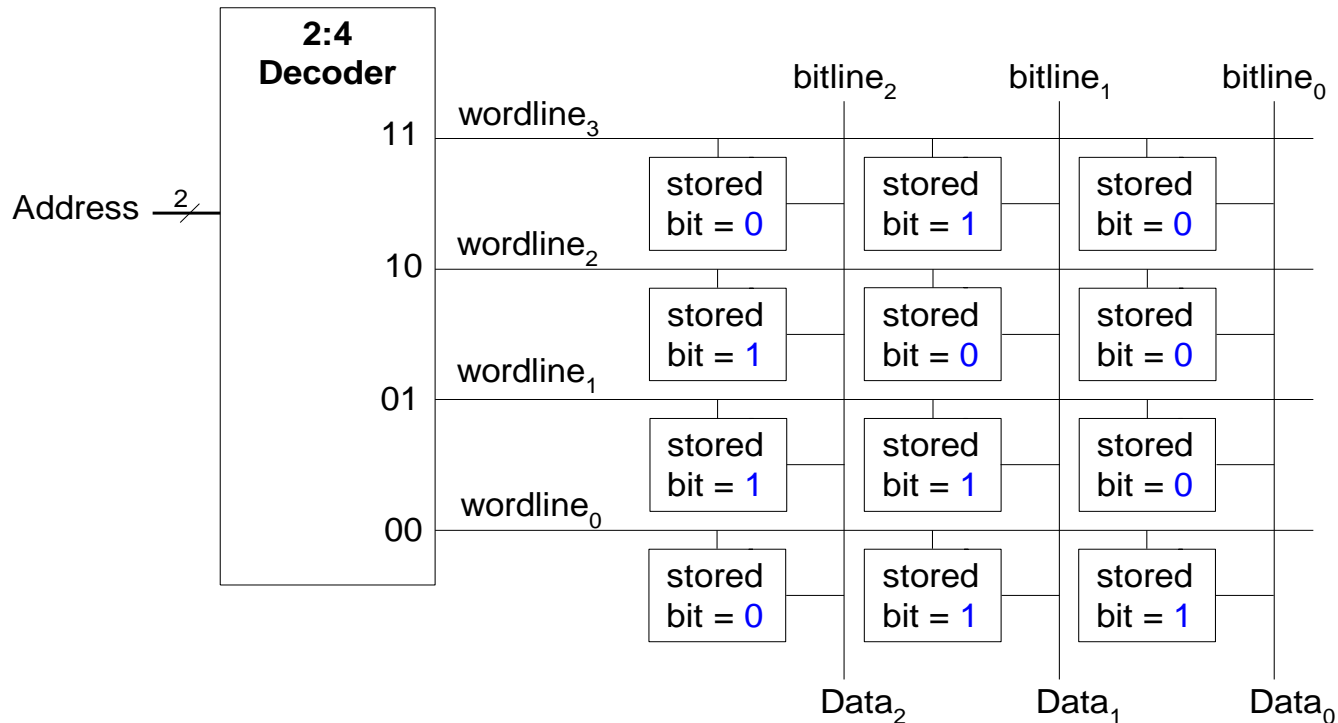
(b)

Memory Array Bit Cells



Memory Array

- **Wordline:**
 - like an enable
 - single row in memory array read/written
 - corresponds to unique address
 - only one wordline HIGH at once



Types of Memory

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

RAM: Random Access Memory

- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)

ROM: Read Only Memory

- **Nonvolatile:** retains data when power off
- Read quickly, but writing is impossible or slow
- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.



Types of RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
 - DRAM uses a capacitor
 - SRAM uses cross-coupled inverters

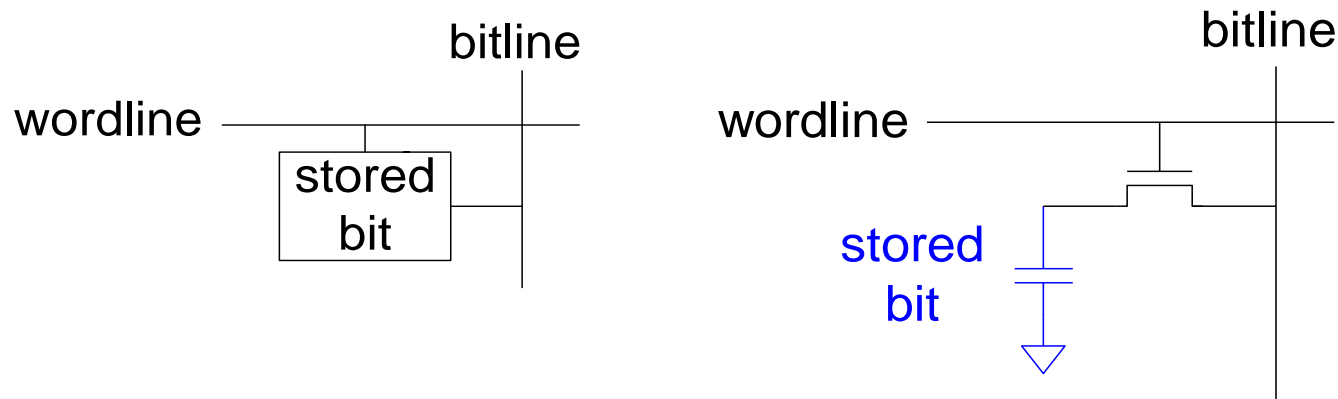
Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers

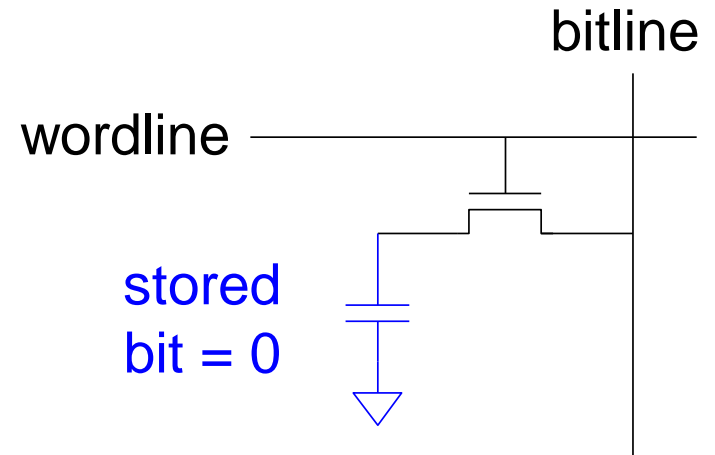
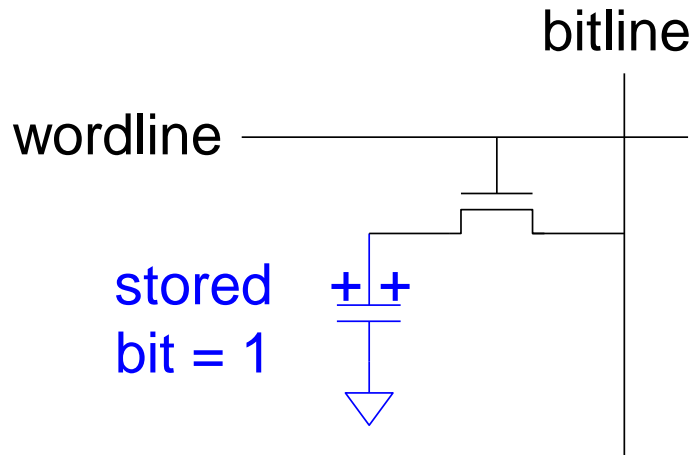


DRAM

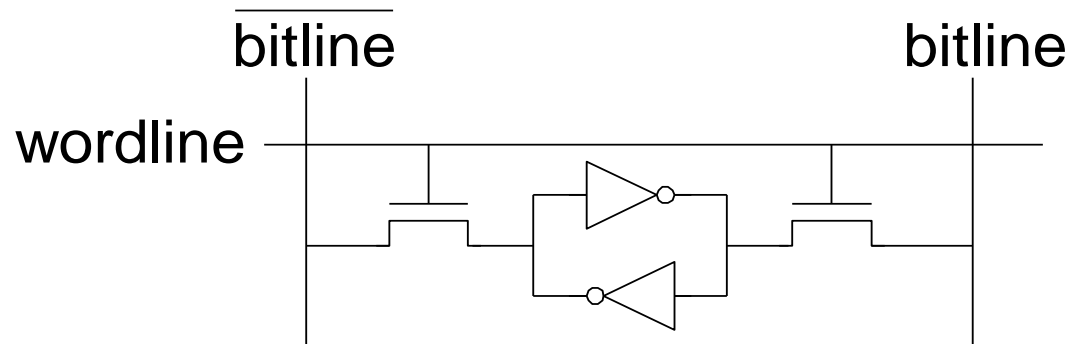
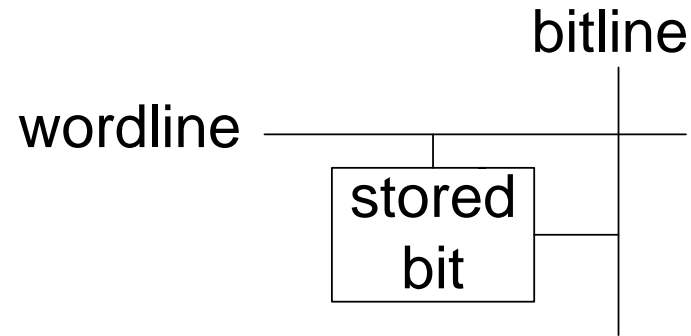
- Data bits stored on capacitor
- *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
 - Charge leakage from the capacitor degrades the value
 - Reading destroys the stored value



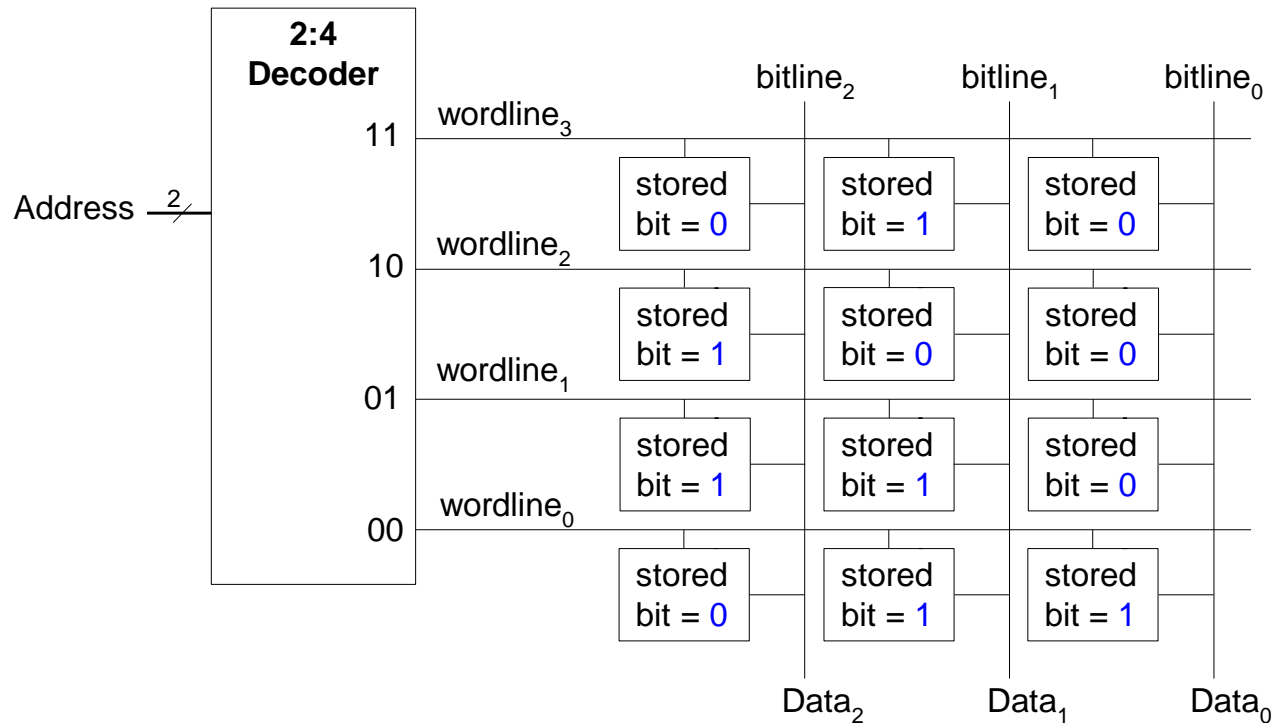
DRAM



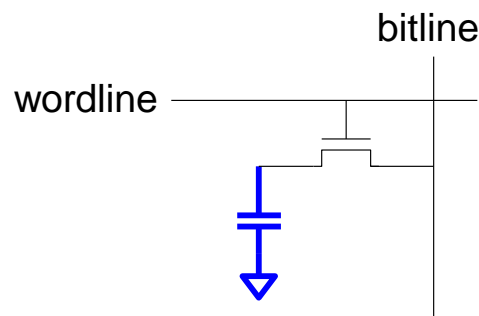
SRAM



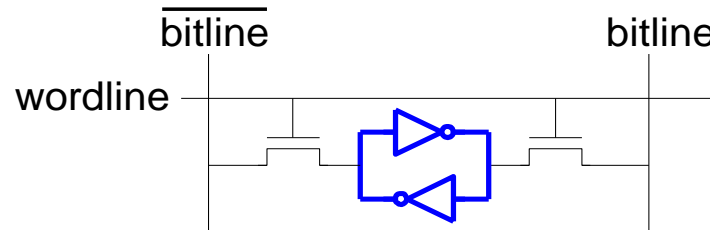
Memory Arrays Review



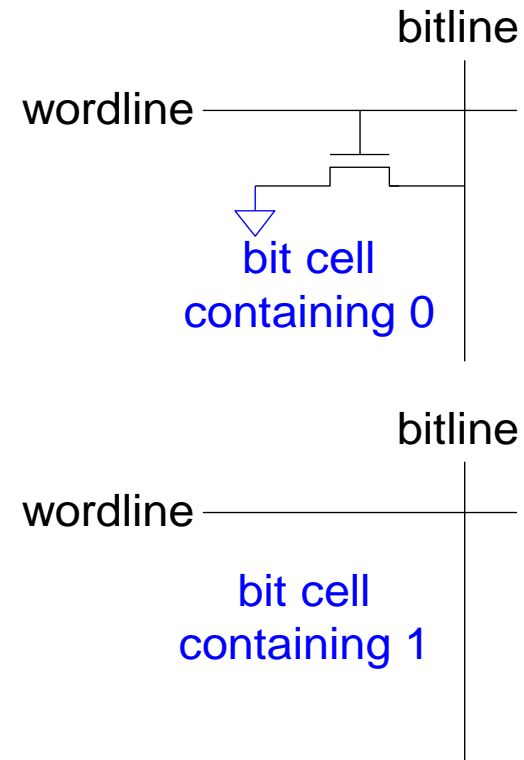
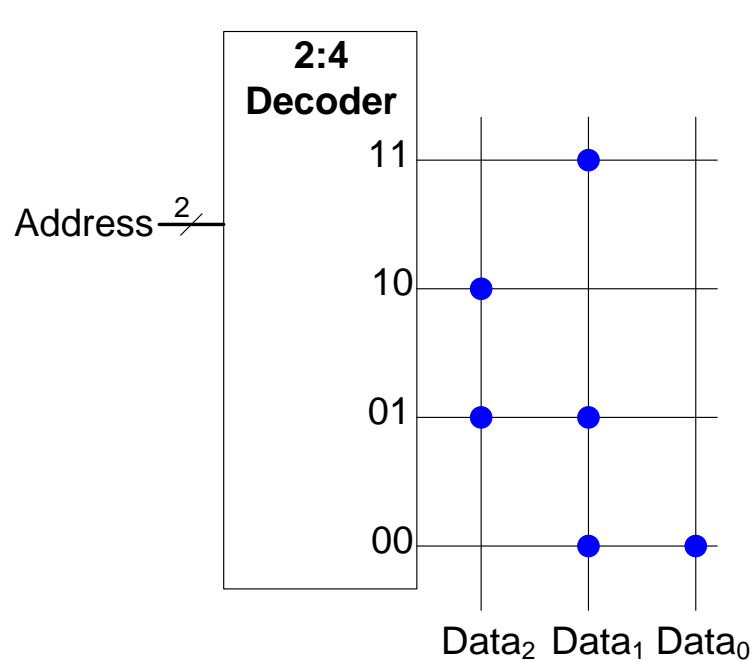
DRAM bit cell:



SRAM bit cell:

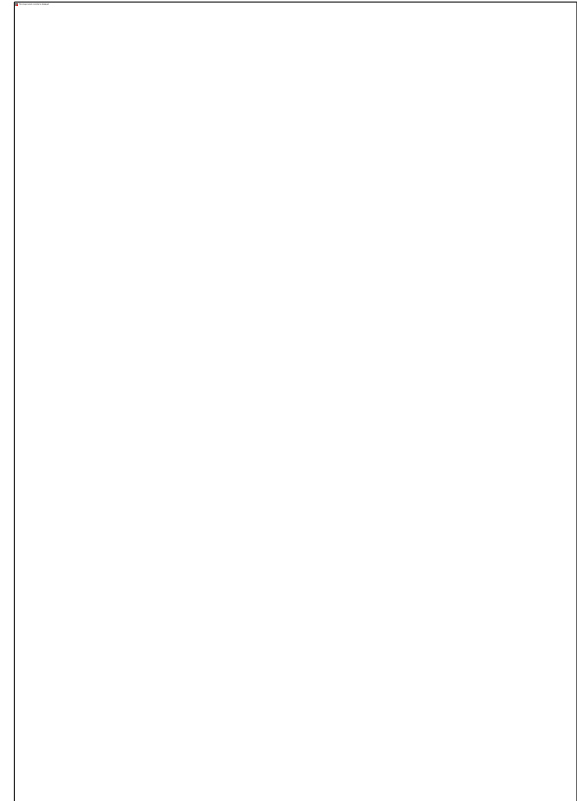


ROM: Dot Notation

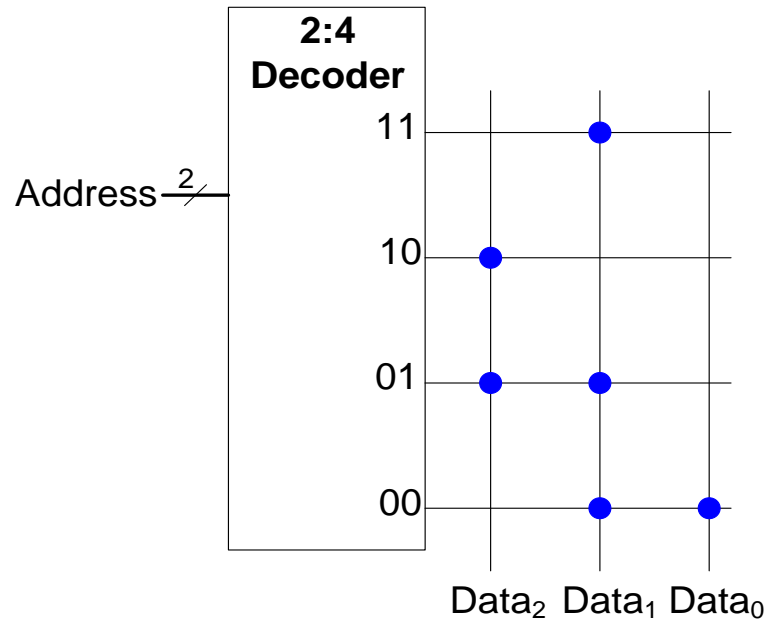


Fujio Masuoka, 1944 -

- Developed memories and high speed circuits at Toshiba, 1971-1994
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a \$25 billion per year market

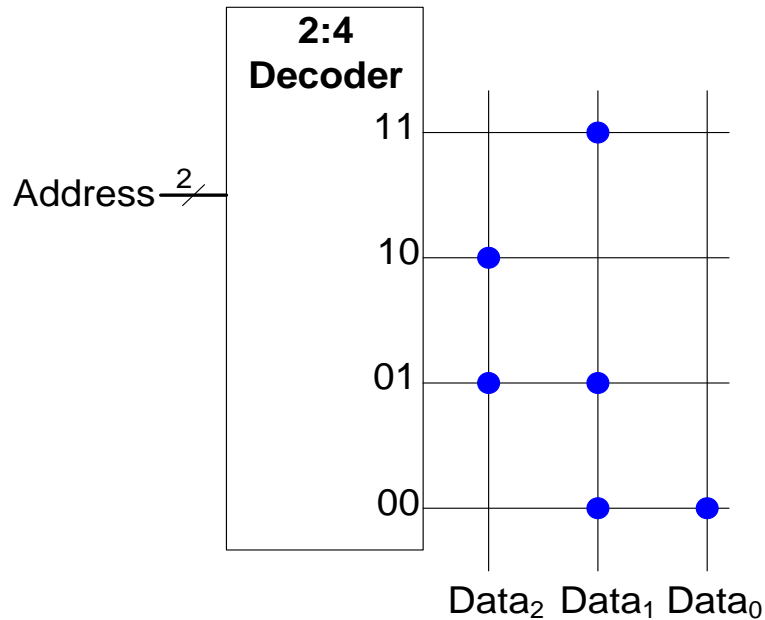


ROM Storage



| Address | Data | | | depth ↑ ↓ |
|---------|------|---|-------------|-----------------|
| 11 | 0 | 1 | 0 | |
| 10 | 1 | 0 | 0 | |
| 01 | 1 | 1 | 0 | |
| 00 | 0 | 1 | 1 | |
| | | | width ←→ | |

ROM Logic



$$Data_2 = A_1 \oplus A_0$$

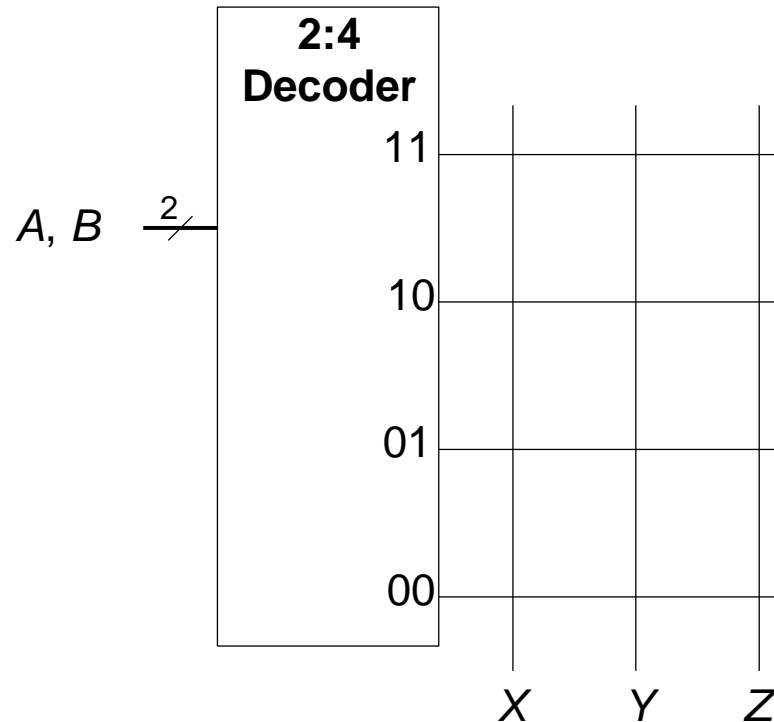
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1} \overline{A_0}$$

Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$ -bit ROM:

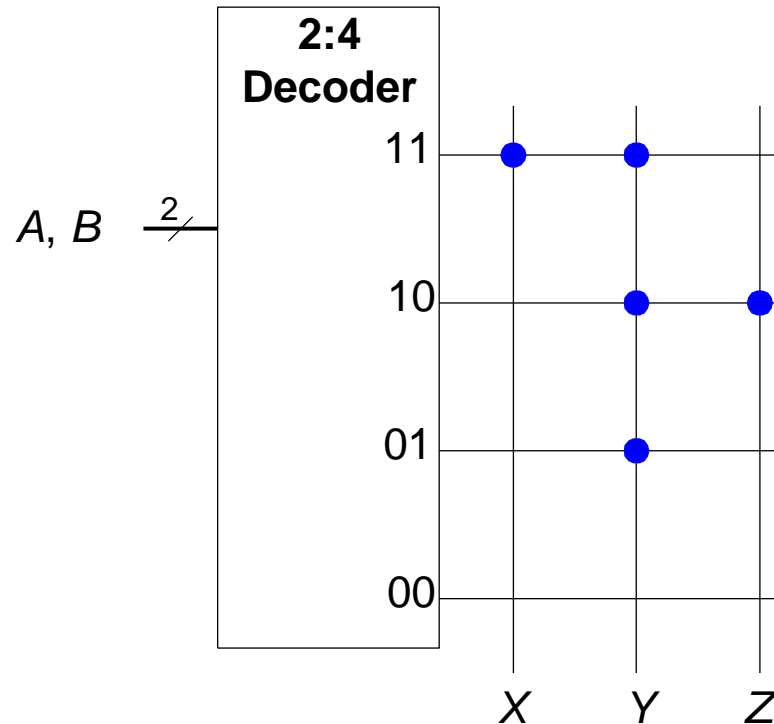
- $X = AB$
- $Y \equiv A + B$
- $Z = A \oplus B$



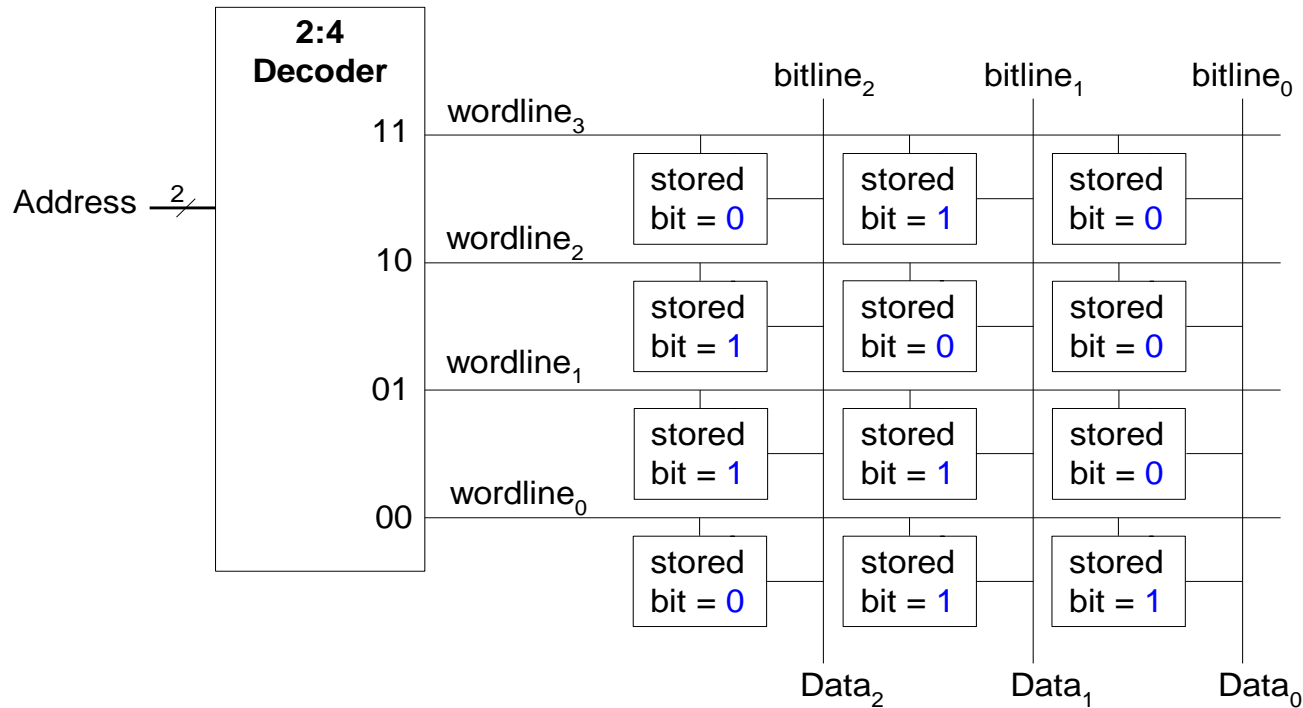
Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$ -bit ROM:

- $X = AB$
- $Y \equiv A + B$
- $Z = A \oplus B$



Logic with Any Memory Array



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \bar{A}_1 + A_0$$

$$Data_0 = \bar{A}_1 \bar{A}_0$$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$ -bit memory array:

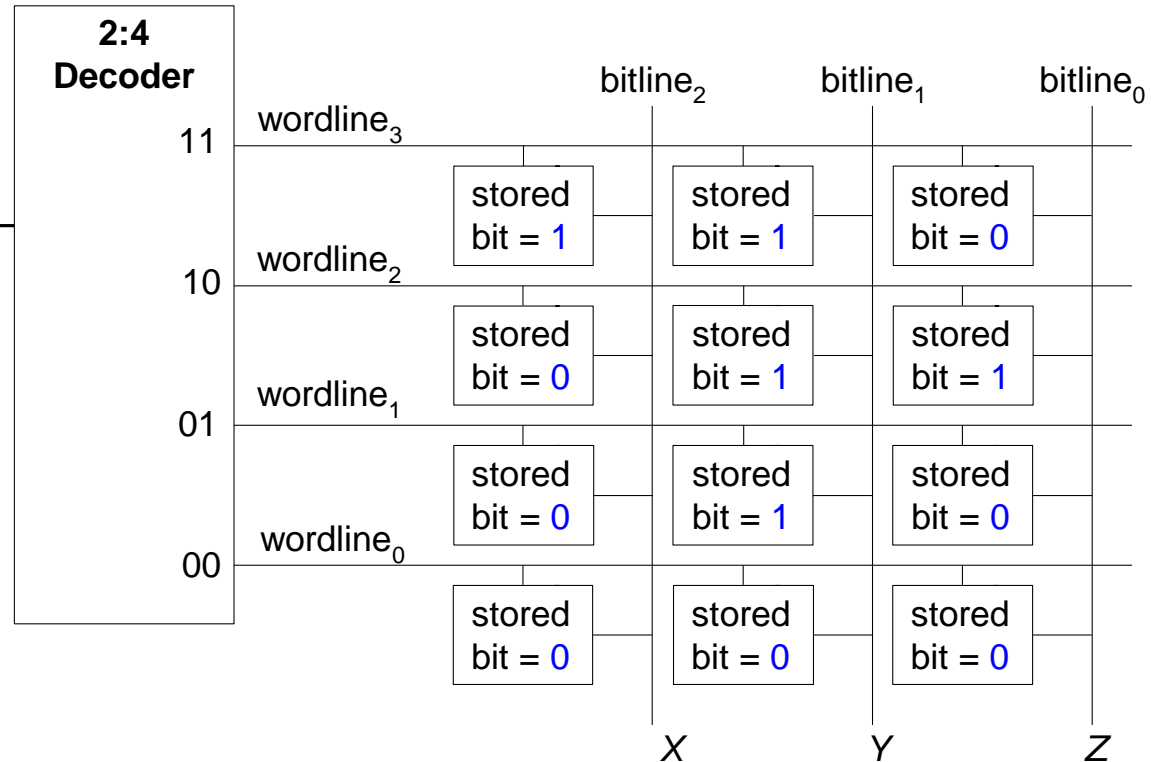
- $X = AB$
- $Y = A + B$
- $Z = \overline{A} B$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$ -bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = \overline{A} B$

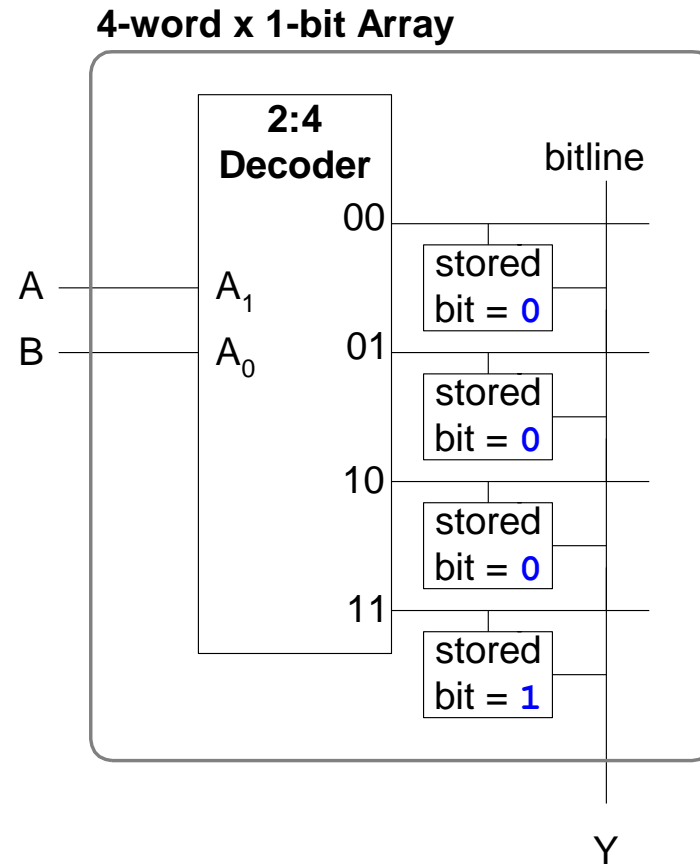
A, B $\xrightarrow{2/}$



Logic with Memory Arrays

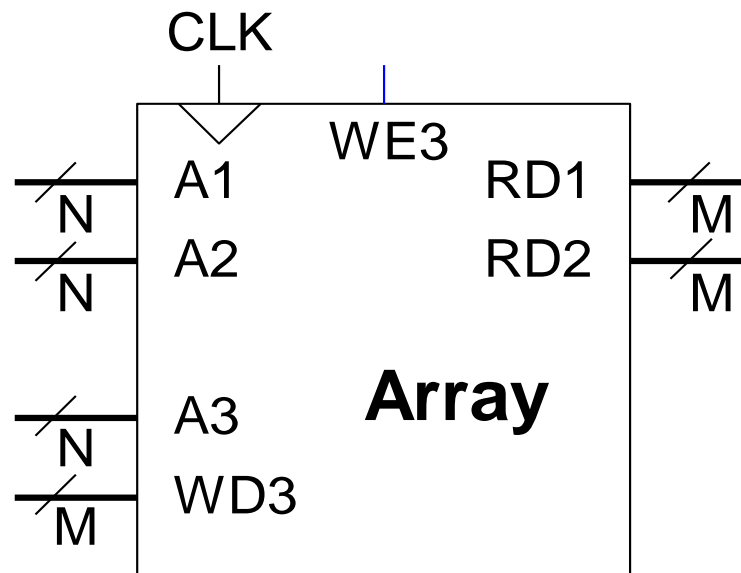
Called *lookup tables* (LUTs): look up output at each input combination (address)

| Truth Table | | |
|-------------|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
 - 2 read ports (A1/RD1, A2/RD2)
 - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory



SystemVerilog Memory Arrays

```
// 256 x 3 memory module with one read/write port
module dmem( input  logic      clk, we,
              input  logic[7:0]  a
              input  logic [2:0] wd,
              output logic [2:0] rd);

    logic [2:0] RAM[255:0];

    assign rd = RAM[a];

    always @(posedge clk)
        if (we)
            RAM[a] <= wd;
endmodule
```

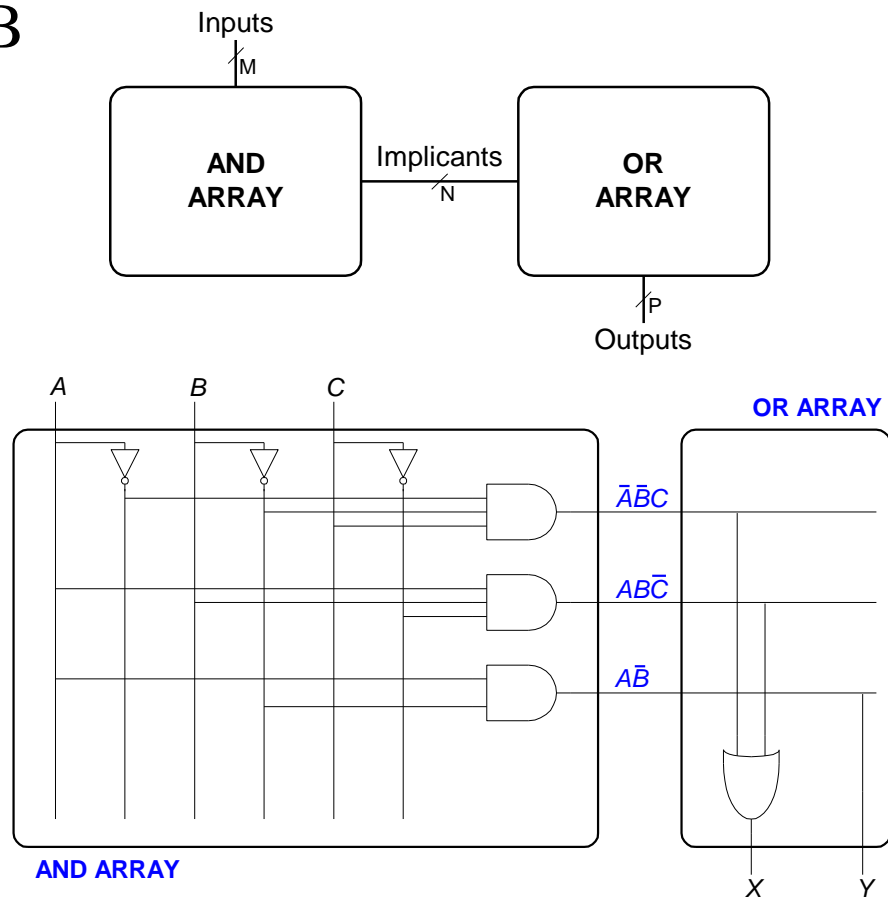
Logic Arrays

- **PLAs** (Programmable logic arrays)
 - AND array followed by OR array
 - Combinational logic only
 - Fixed internal connections
- **FPGAs** (Field programmable gate arrays)
 - Array of Logic Elements (LEs)
 - Combinational and sequential logic
 - Programmable internal connections

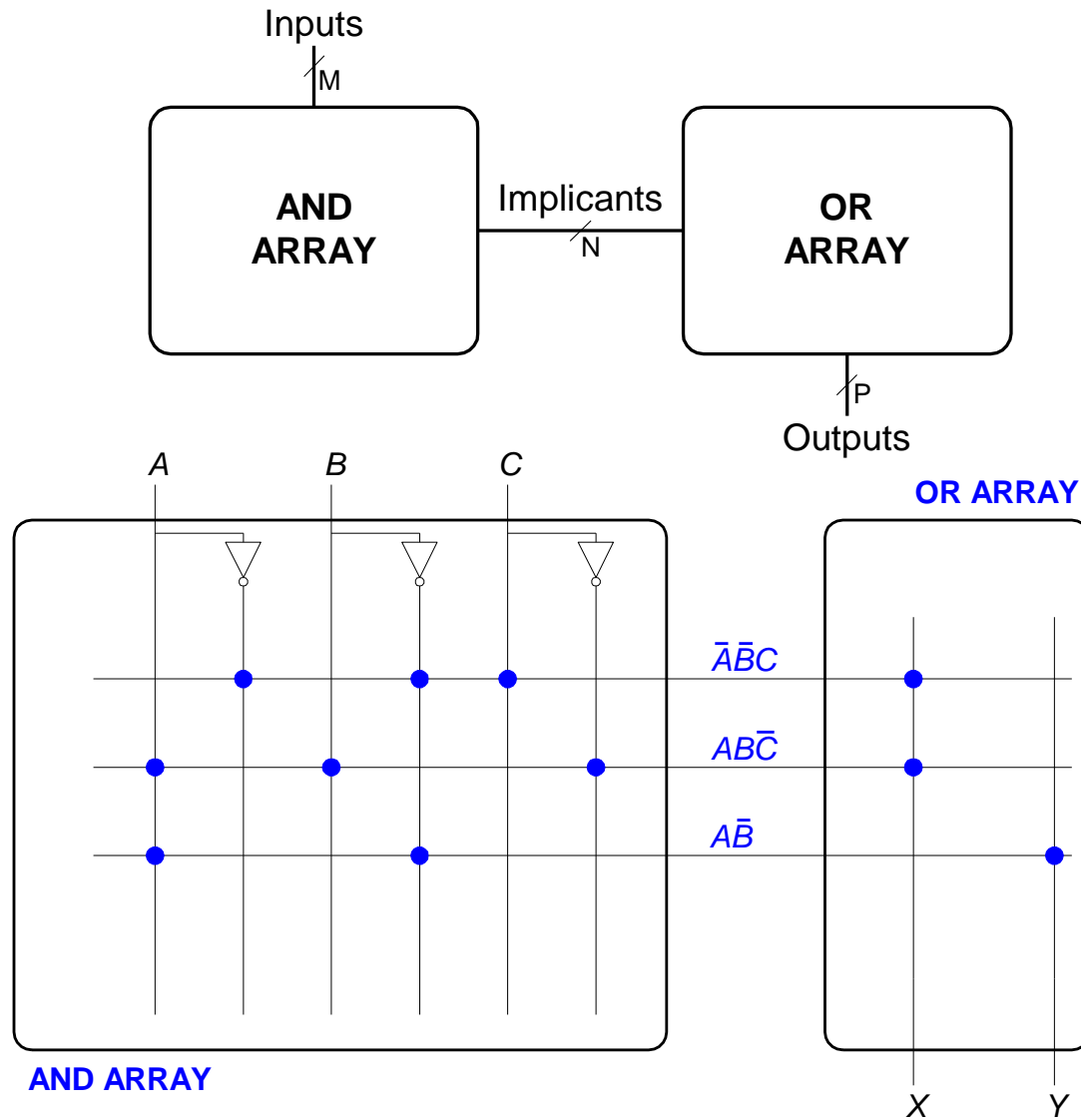
PLAs

PLAs

- $X = \bar{A}\bar{B}C + ABC\bar{C}$
- $Y = A\bar{B}$



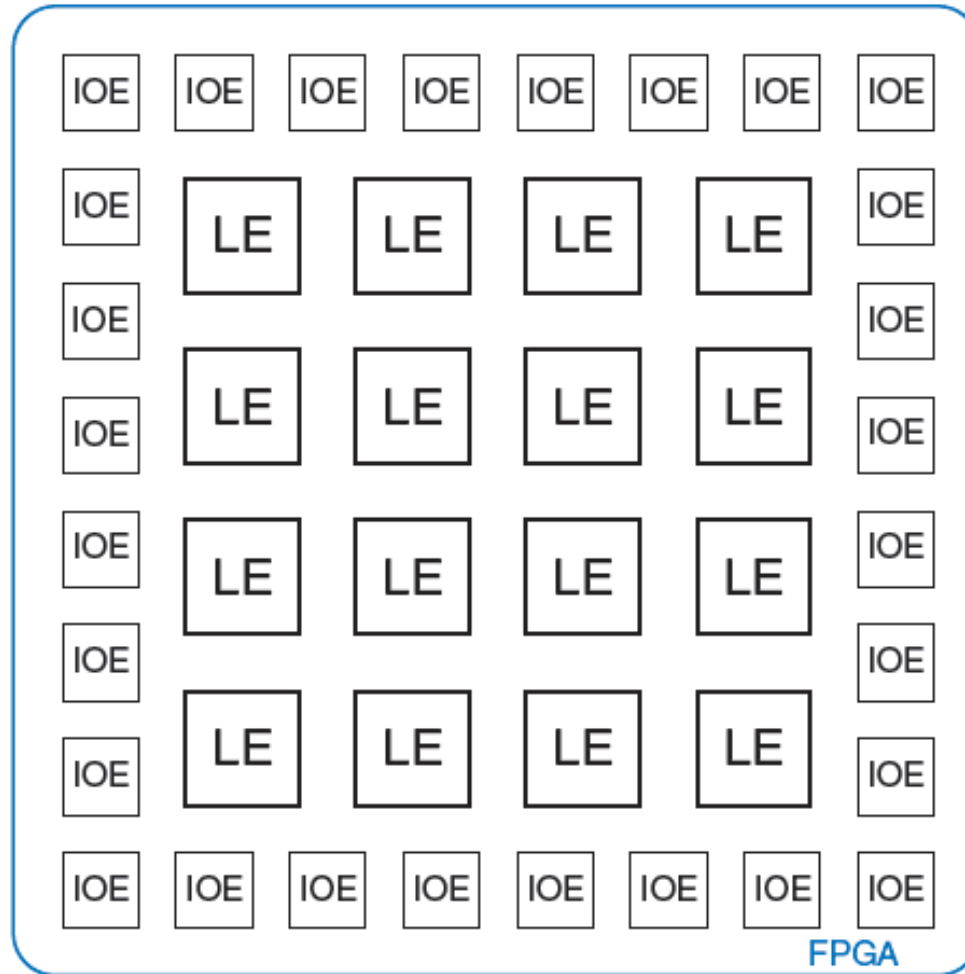
PLAs: Dot Notation



FPGA: Field Programmable Gate Array

- Composed of:
 - **LEs** (Logic elements): perform logic
 - **IOEs** (Input/output elements): interface with outside world
 - **Programmable interconnection:** connect LEs and IOEs
 - Some FPGAs include other building blocks such as multipliers and RAMs

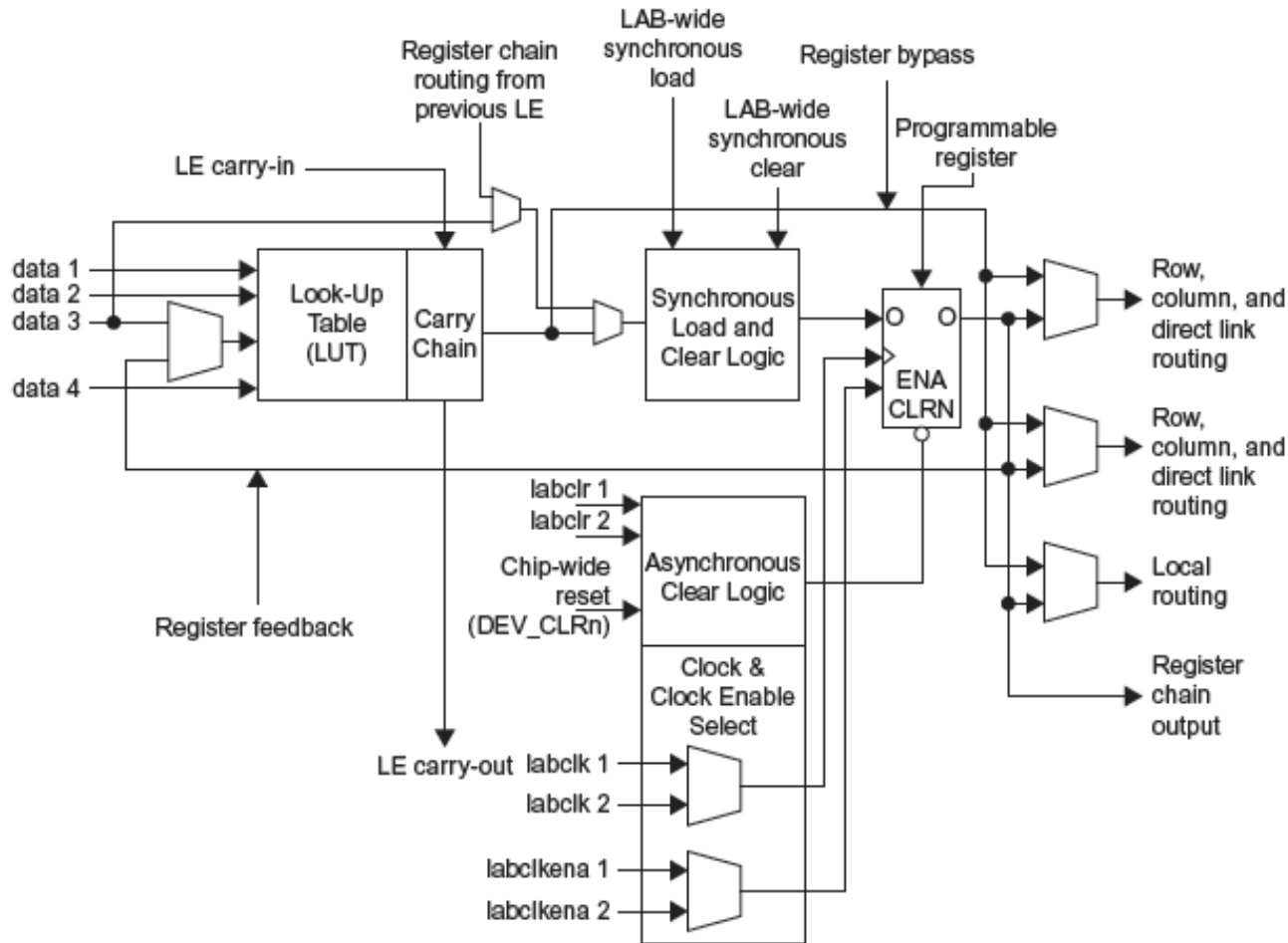
General FPGA Layout



LE: Logic Element

- Composed of:
 - **LUTs** (lookup tables): perform combinational logic
 - **Flip-flops**: perform sequential logic
 - **Multiplexers**: connect LUTs and flip-flops

Altera Cyclone IV LE



Altera Cyclone IV LE

- The Spartan CLB has:
 - 1 four-input LUT
 - 1 registered output
 - 1 combinational output

LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

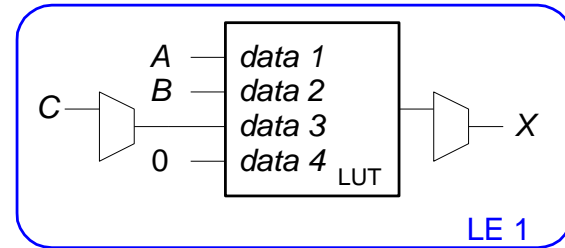
- $X = \overline{A}\overline{B}C + A\overline{B}C$
- $Y = A\overline{B}$

LE Configuration Example

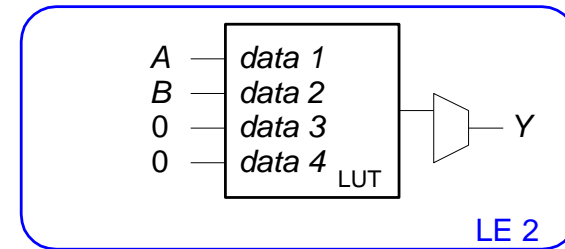
Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{A}\overline{B}C + A\overline{B}C$
- $Y = A\overline{B}$

| (A) data 1 | (B) data 2 | (C) data 3 | data 4 | (X) LUT output |
|---------------|---------------|---------------|--------|-------------------|
| 0 | 0 | 0 | X | 0 |
| 0 | 0 | 1 | X | 1 |
| 0 | 1 | 0 | X | 0 |
| 0 | 1 | 1 | X | 0 |
| 1 | 0 | 0 | X | 0 |
| 1 | 0 | 1 | X | 0 |
| 1 | 1 | 0 | X | 1 |
| 1 | 1 | 1 | X | 0 |



| (A) data 1 | (B) data 2 | data 3 | data 4 | (Y) LUT output |
|---------------|---------------|--------|--------|-------------------|
| 0 | 0 | X | X | 0 |
| 0 | 1 | X | X | 0 |
| 1 | 0 | X | X | 1 |
| 1 | 1 | X | X | 0 |



FPGA Design Flow

Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design