Computer Architecture

Chapter 5: Digital Building Blocks

Largely based on slides by David and Sarah Harris

Chapter 5 Topics

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

Introduction

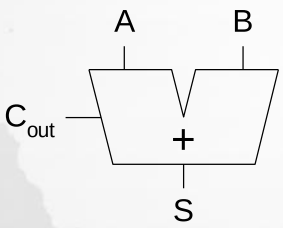
- Digital building blocks:
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- Building blocks demonstrate hierarchy, modularity, and regularity:
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure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

Adders

- 1-bit Adder
- Multibit Adder
- Ripple-Carry Adder
- Carry-Lookahead Adder
- Prefix Adder

1-bit Adders

Half Adder

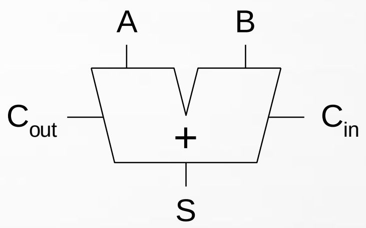


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



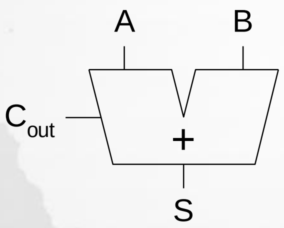
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

1-bit Adders

Half Adder

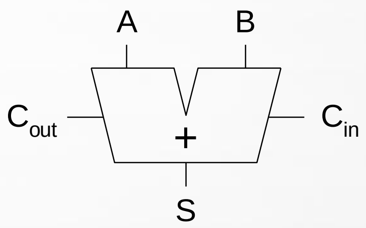


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S =$

$C_{out} =$

Full Adder



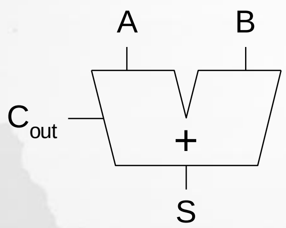
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S =$

$C_{out} =$

1-bit Adders

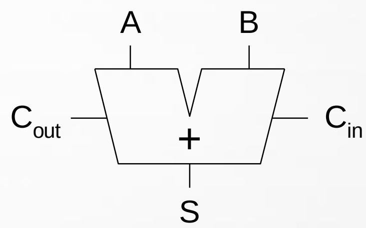
Half Adder



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

$$S = A \oplus B$$
$$C_{out} = AB$$

Full Adder

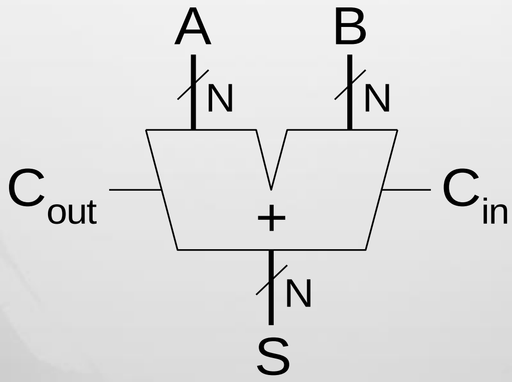


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

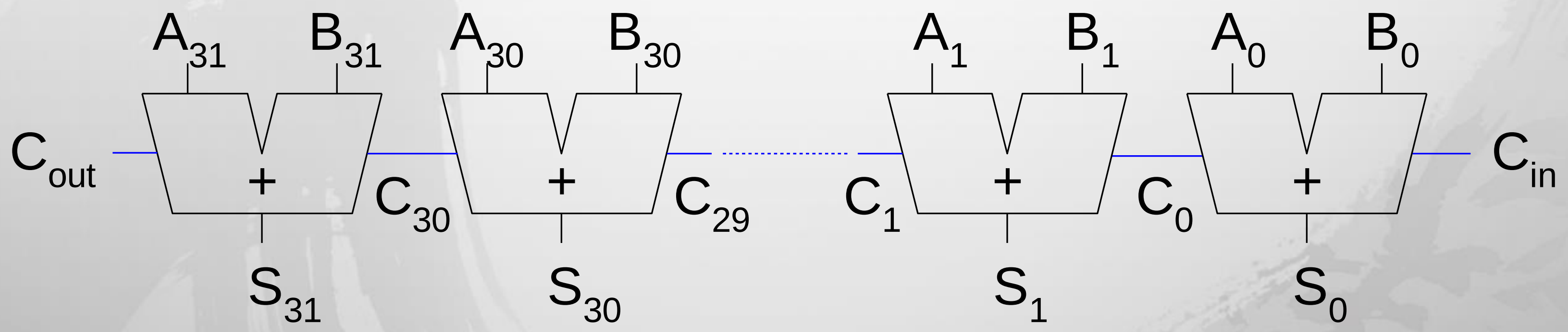
Multibit Adders (CPAs)

- 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



Ripple-Carry Adder

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



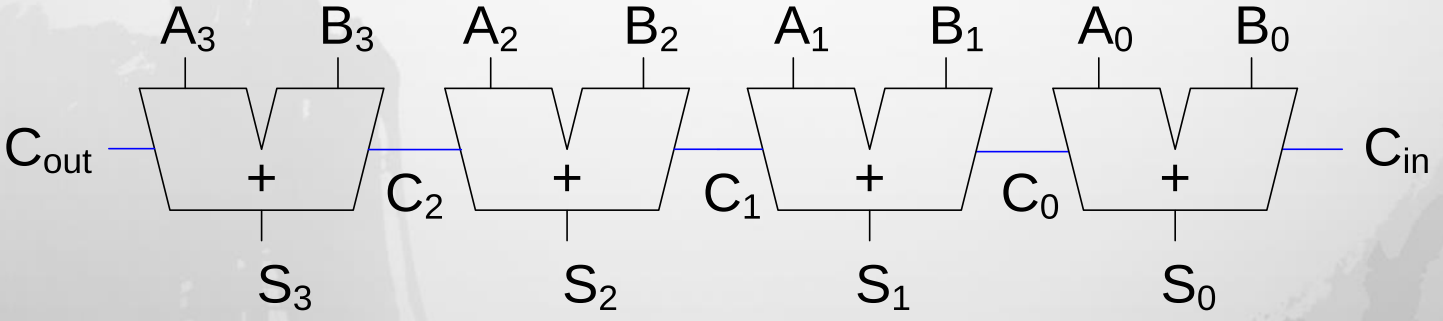
Ripple-Carry Adder Delay

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

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

Exercise

- What are S and C when the following operations are performed respectively by a 4-bit ripple-carry adder? Assume C_{in}=0
  - 1011 + 0010
  - 0101 + 1100



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.

$$G_i = A_i B_i$$

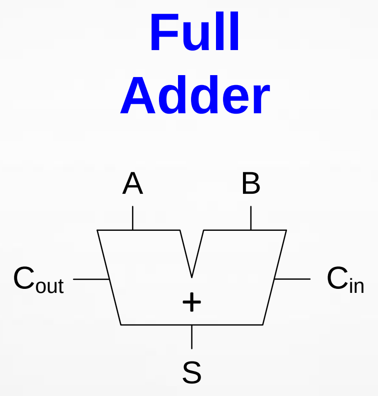
    - Column *i* will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

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

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

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

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 $P_{3:0}$) :

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

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

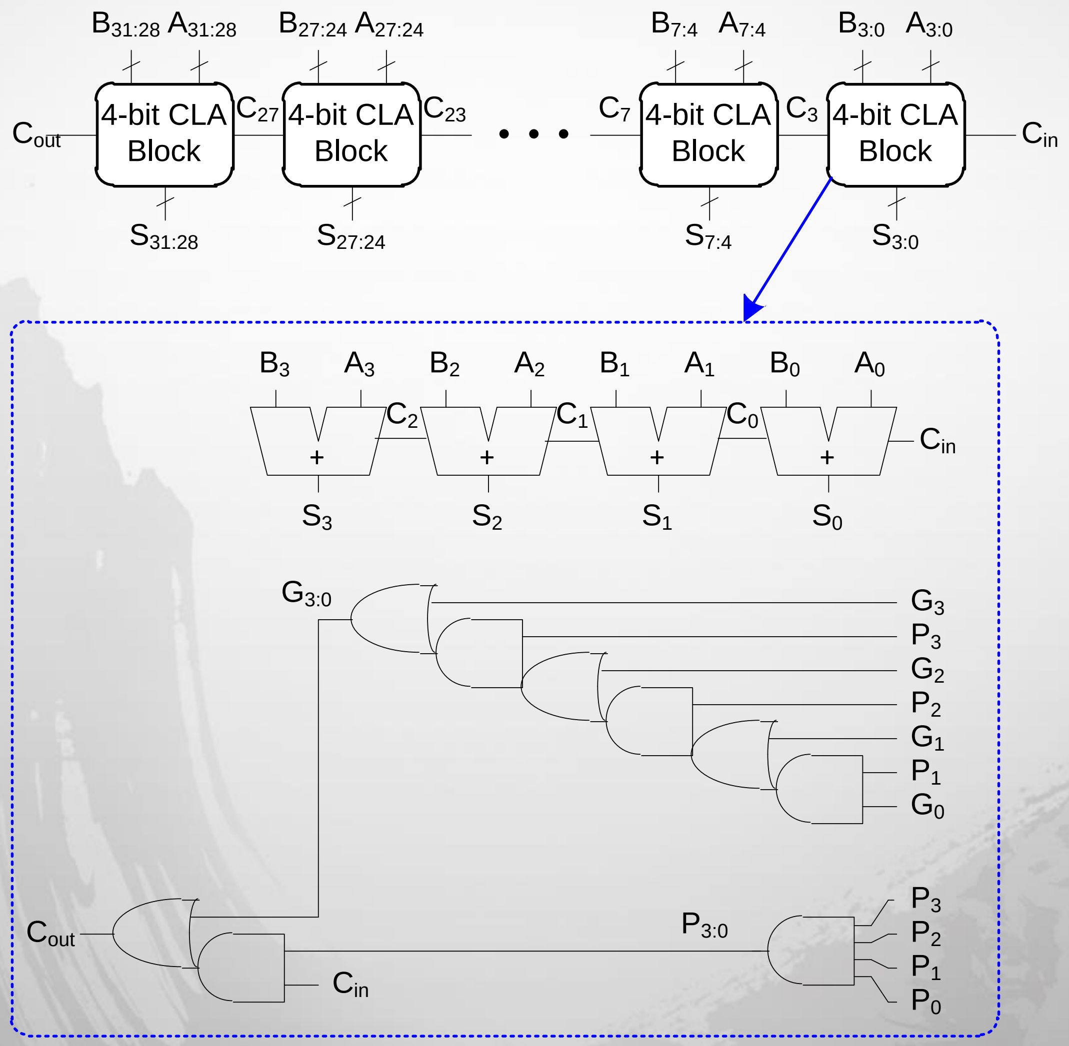
- **Generally,**

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

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

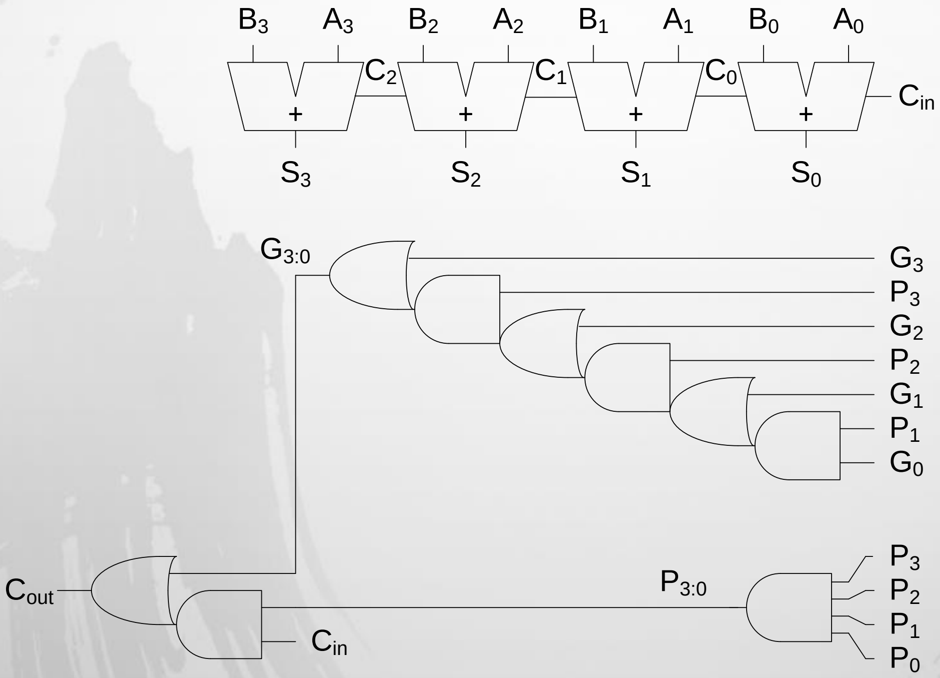
$$C_i \;= G_{i:j} \;+ P_{i:j}\,C_{i\text{-}1}$$

32-bit CLA with 4-bit Blocks

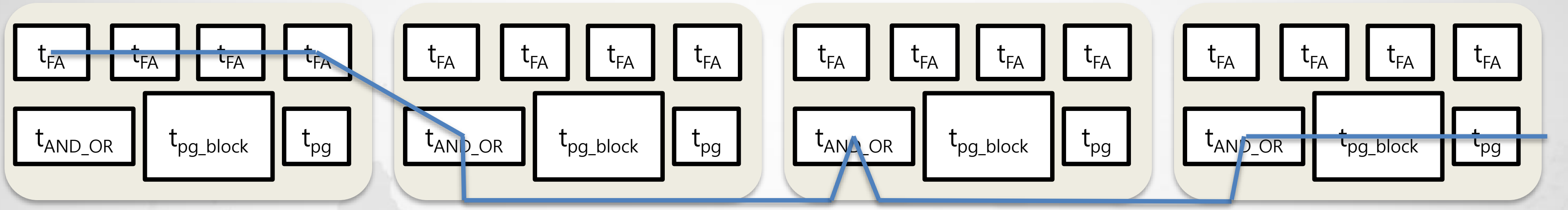


Exercise

- 0101+1100



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

Prefix Adder

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

$$\boldsymbol{S_i = (A_i \oplus B_i) \oplus C_i}$$

- 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

- 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

- 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 $i-1$:

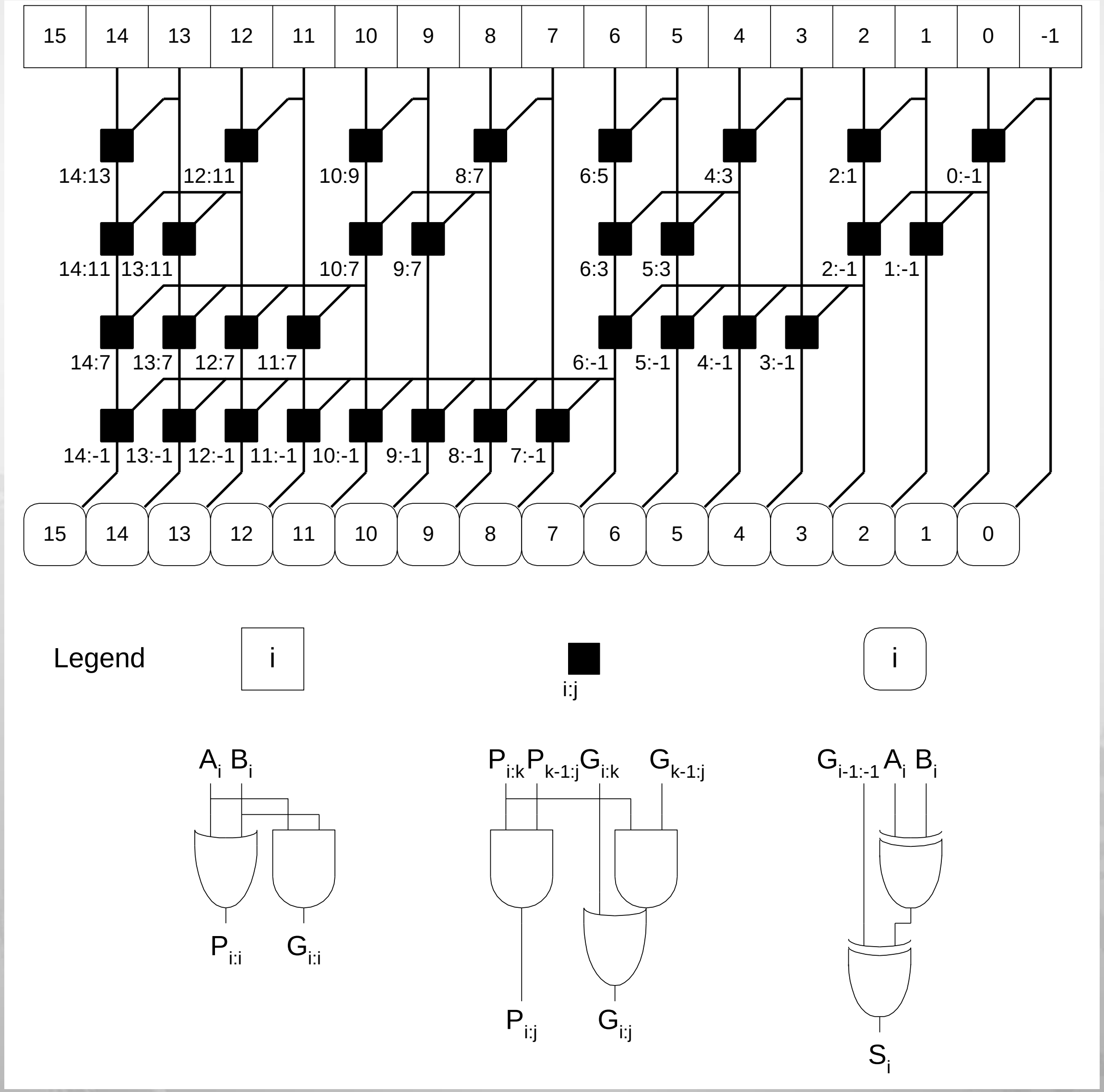
  $$C_{i-1} = G_{i-1:-1}$$

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

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

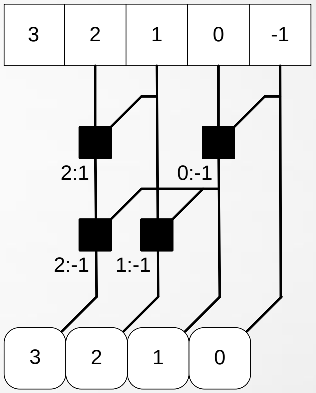
- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

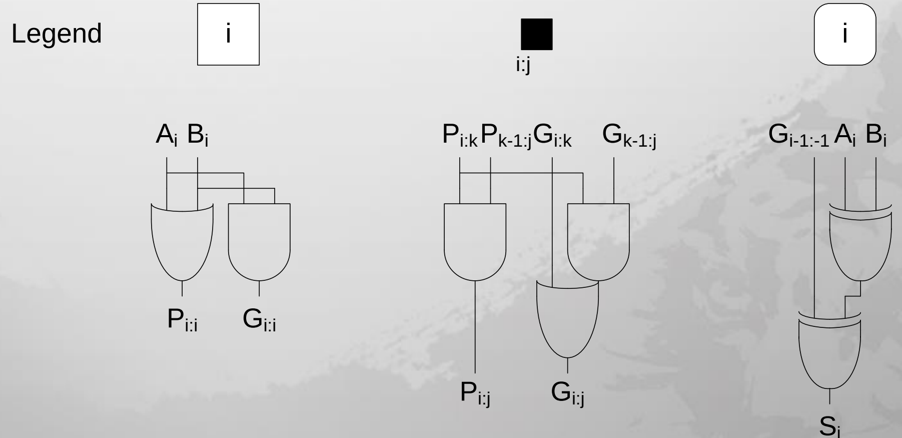
Prefix Adder Schematic



Exercise

- 0101 + 1100





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

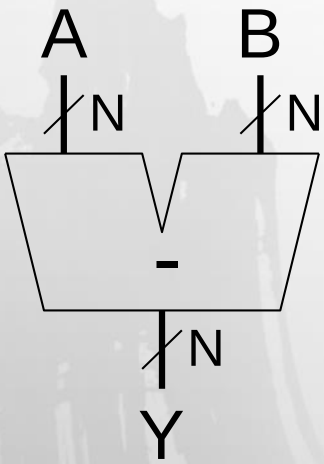
$$t_{\text{ripple}} = Nt_{FA} = 32(300 \text{ ps})$$
$$= \mathbf{9.6\ ns}$$

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$
$$= [100 + 600 + (7)200 + 4(300)] \text{ ps}$$
$$= \mathbf{3.3\ ns}$$

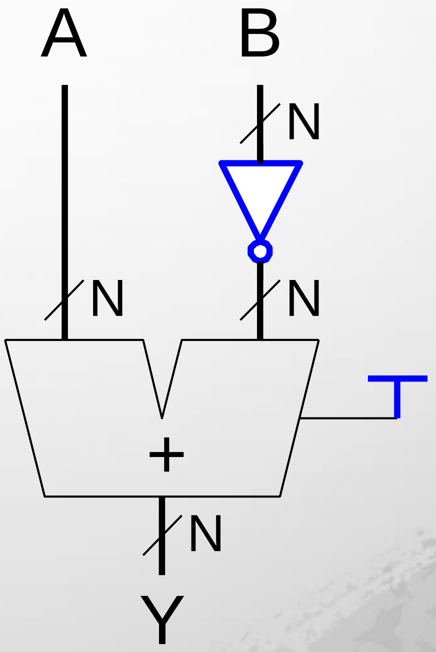
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$
$$= [100 + \log_2 32(200) + 100] \text{ ps}$$
$$= \mathbf{1.2\ ns}$$

Subtracter

Symbol

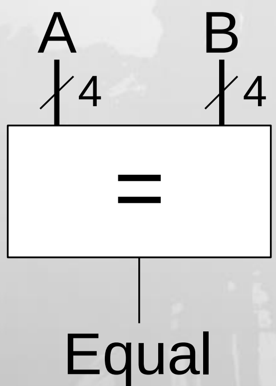


Implementation

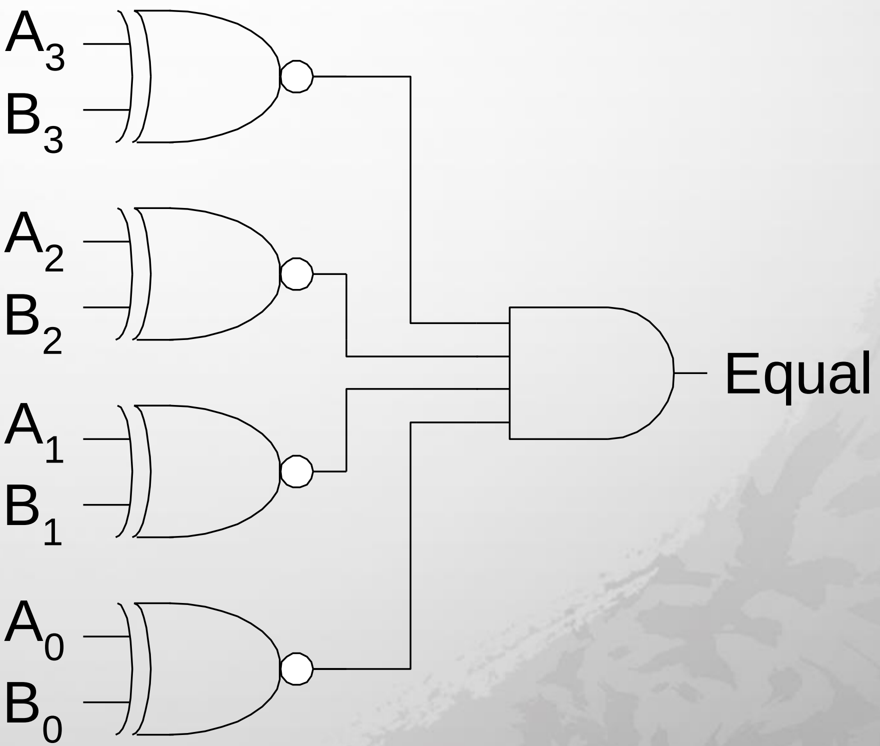


Comparator: Equality

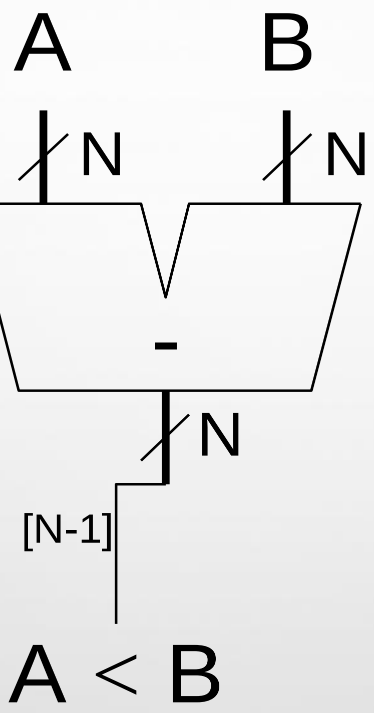
Symbol



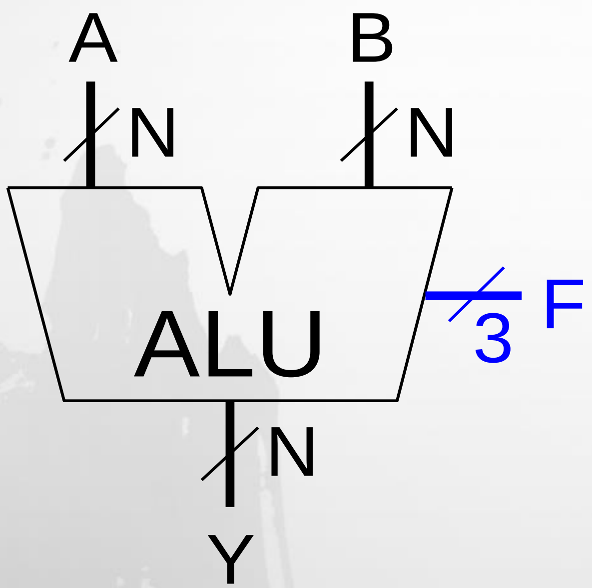
Implementation



Comparator: Less Than

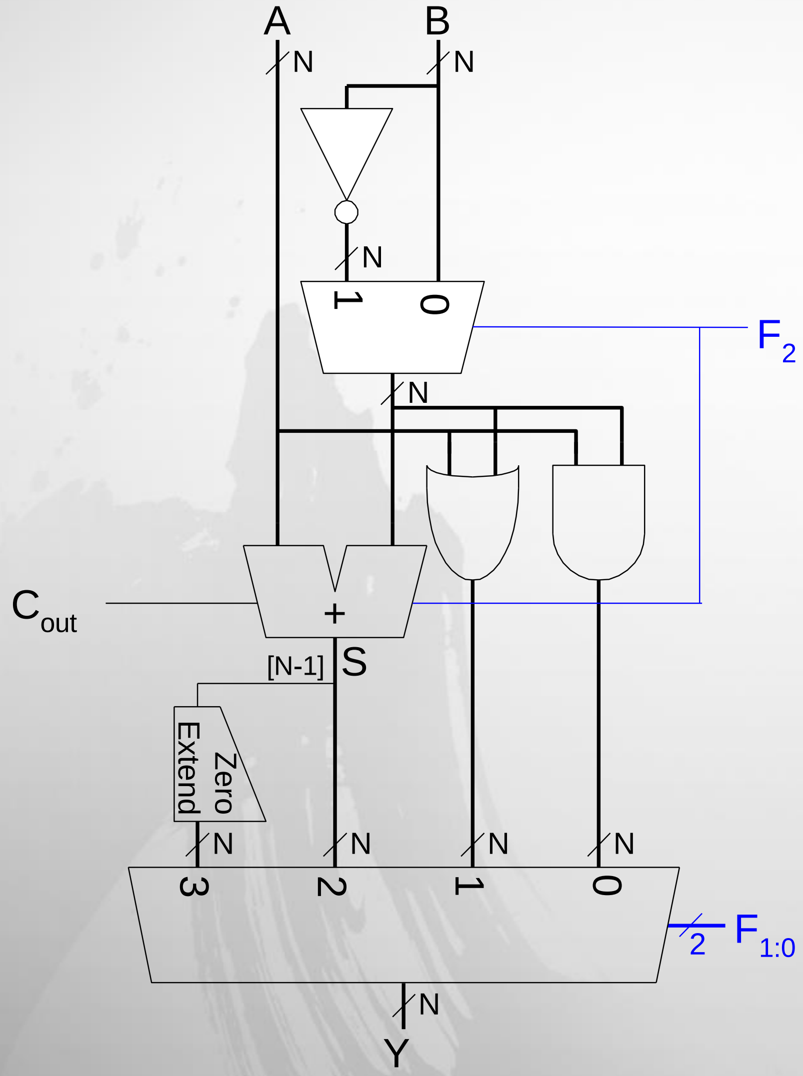


Arithmetic Logic Unit (ALU)



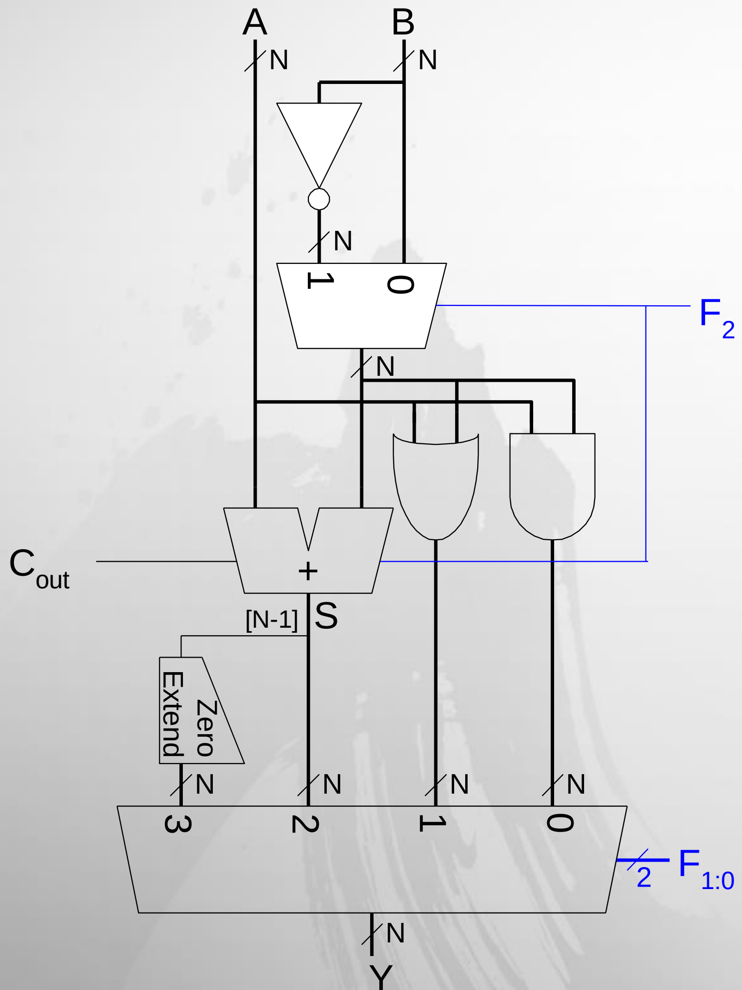
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Design



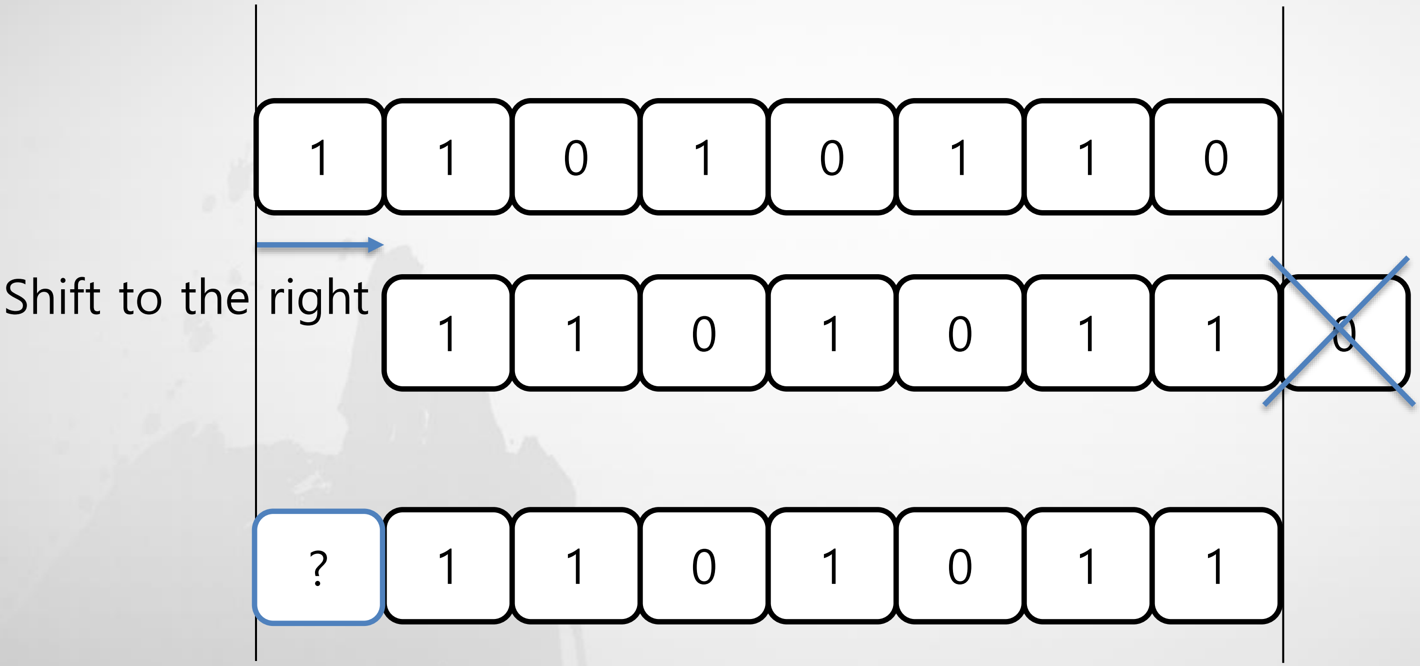
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: A = 25 and B = 32
  - A < B, so Y should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0}$ = **111**
    - F_2 = **1** (adder acts as subtracter), so 25 - 32 = -7
    - -7 has 1 in the most significant bit (S_{31} = 1)
    - $F_{1:0}$ = **11** multiplexer selects Y = S_{31} (zero extended) = 0x00000001

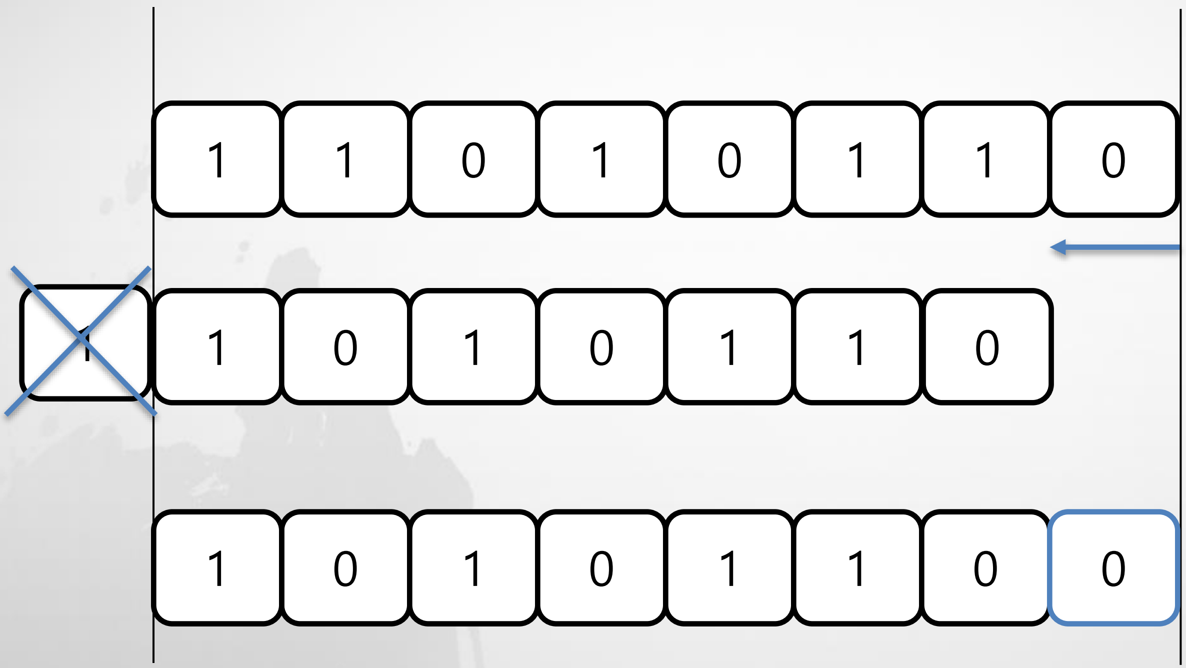
Shifters



Pad zero to the empty space (logical shift)

Extend the sign bit to the empty space (arithmetic shift)

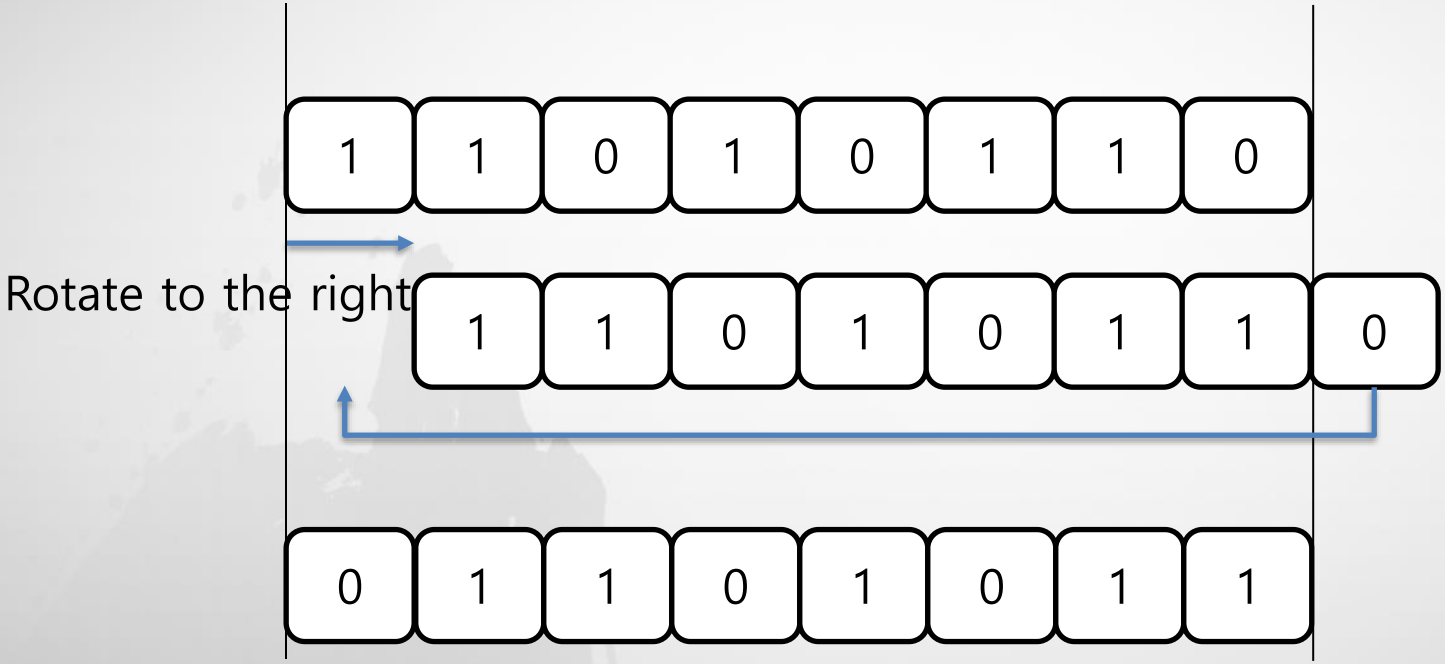
Shifters



Shift to the left

Pad zero to the empty space (logical/arithmetic shift

Rotator



Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =

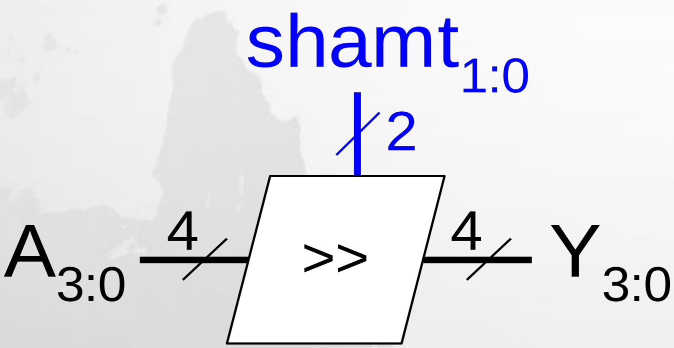
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =

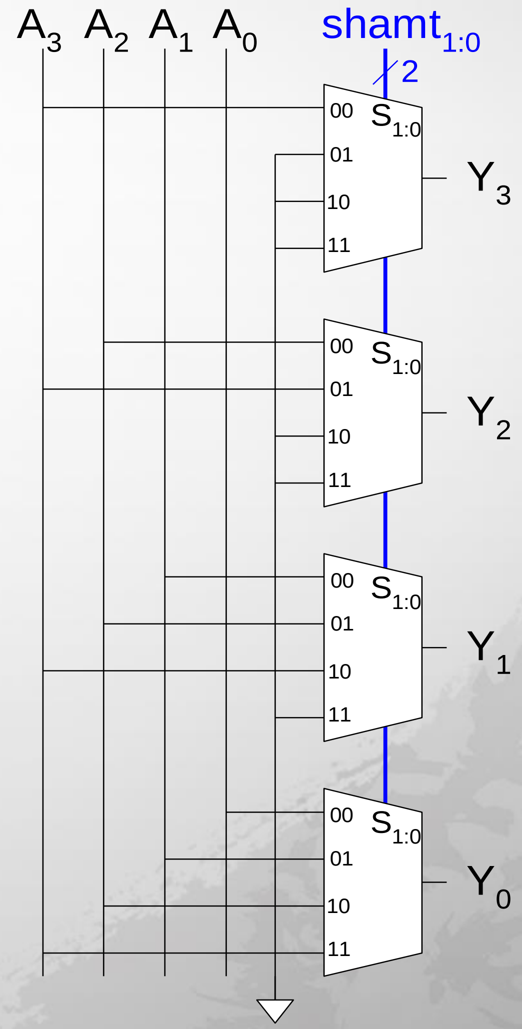
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

Shifters

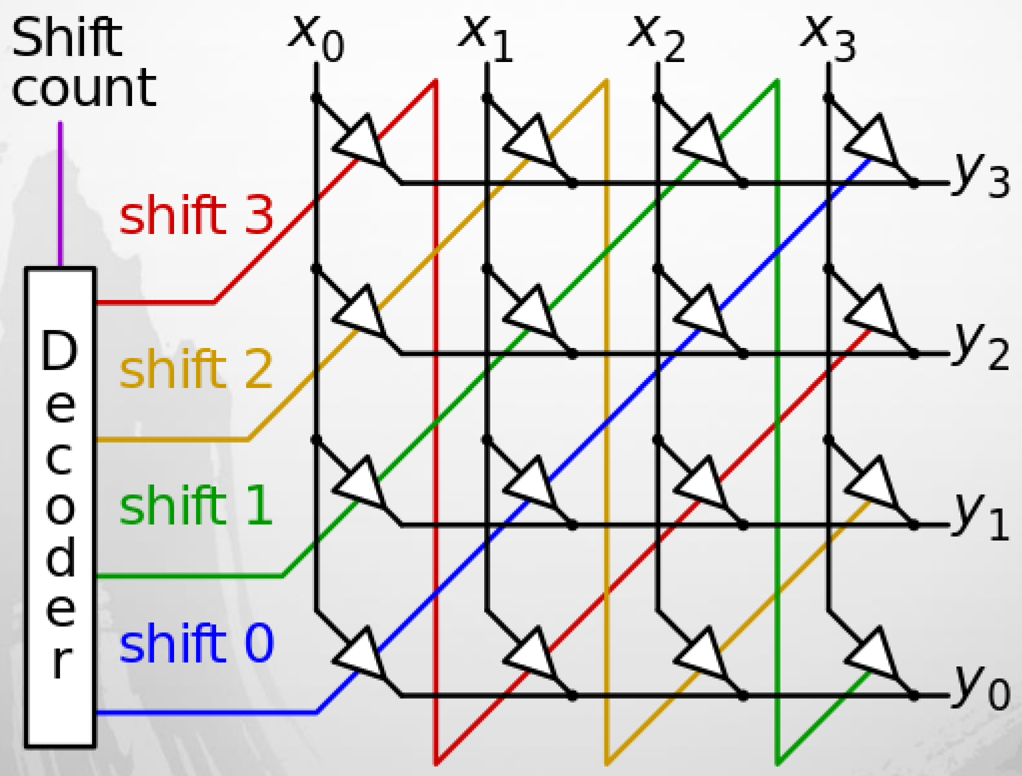
- **Logical shifter:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
  - Ex: 11001 >>> 2 = **11**110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:**
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Shifter Design





Barrel Shifter



Shifters as Multipliers, Dividers

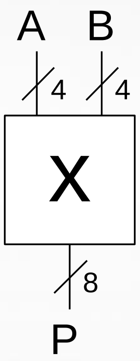
- $A << N = A \times 2^N$
  - **Example:** 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - **Example:** 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)
- $A >>> N = A \div 2^N$
  - **Example:** 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - **Example:** 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Multipliers

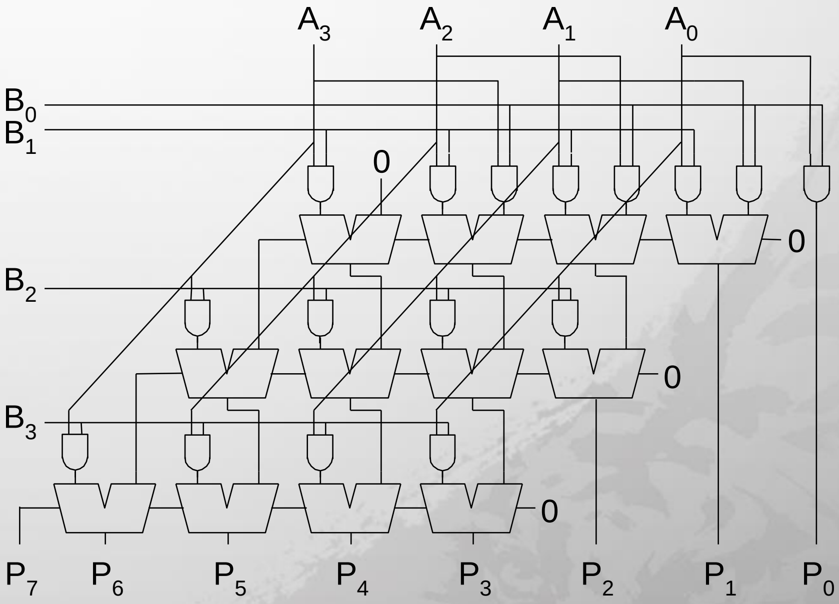
- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011
230 x 42 = 9660		5 x 7 = 35

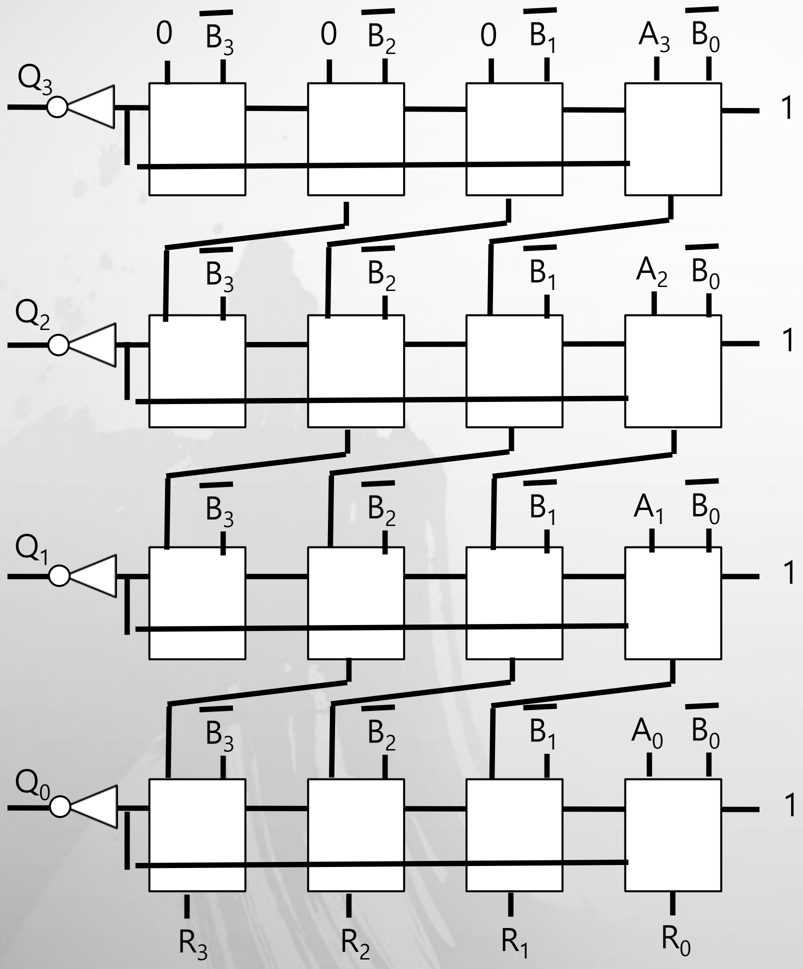
4 x 4 Multiplier

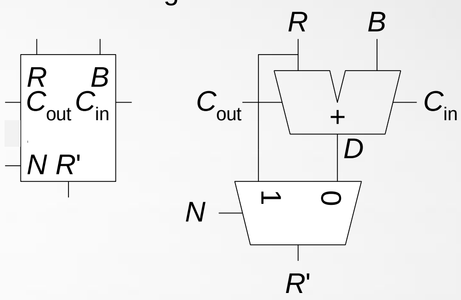


				A_3	A_2	A_1	A_0
			x	B_3	B_2	B_1	B_0
				A_3B_0	A_2B_0	A_1B_0	A_0B_0
			A_3B_1	A_2B_1	A_1B_1	A_0B_1	
		A_3B_2	A_2B_2	A_1B_2	A_0B_2		
+	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0



4 x 4 Divider





$A/B = QB + R$

Algorithm:

$R' = 0$

for i = N-1 to 0

$R = \{R' << 1.\ A_i\}$

$D = R - B$

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

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

$R = R'$

Exercise

- Compute 1010 / 11

```
     0011
  11|1010
     0
     10
     00
     101
      11
      100
       11
        1
```

$A/B = QB + R$

Algorithm:

$R' = 0$

for i = N-1 to 0

  $R = \{R' << 1.\ A_i\}$

  $D = R - B$

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

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

$R = R'$

i=3)

$R = (R' << 1).\ A_3 = 1$

$D = R\text{-}B = 1 - 11 < 0$

$Q_3 = 0,\ R' = R = 1$

i=2)

$R = (R' << 1).\ A_2 = 10$

$D = R\text{-}B = 10 - 11 < 0$

$Q_2 = 0,\ R' = R = 10$

i=1)

$R = (R' << 1).\ A_1 = 101$

$D = R\text{-}B = 101 - 11 > 0$

$Q_1 = 1,\ R' = D = 10$

i=0)

$R = (R' << 1).\ A_0 = 100$

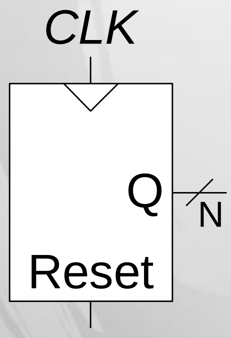
$D = R\text{-}B = 100 - 11 > 0$

$Q_0 = 1,\ R' = D = 1$

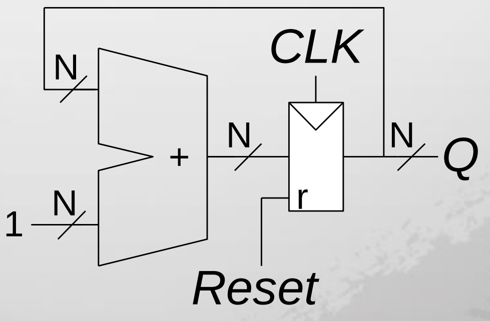
Counters

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

Symbol



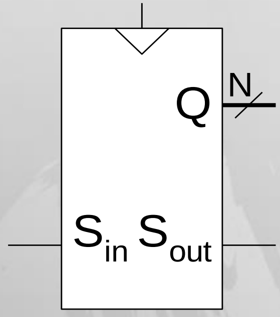
Implementation



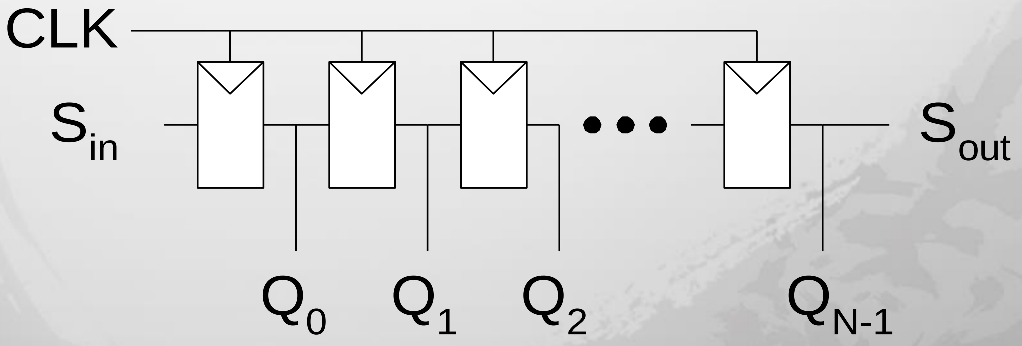
Shift Registers

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:

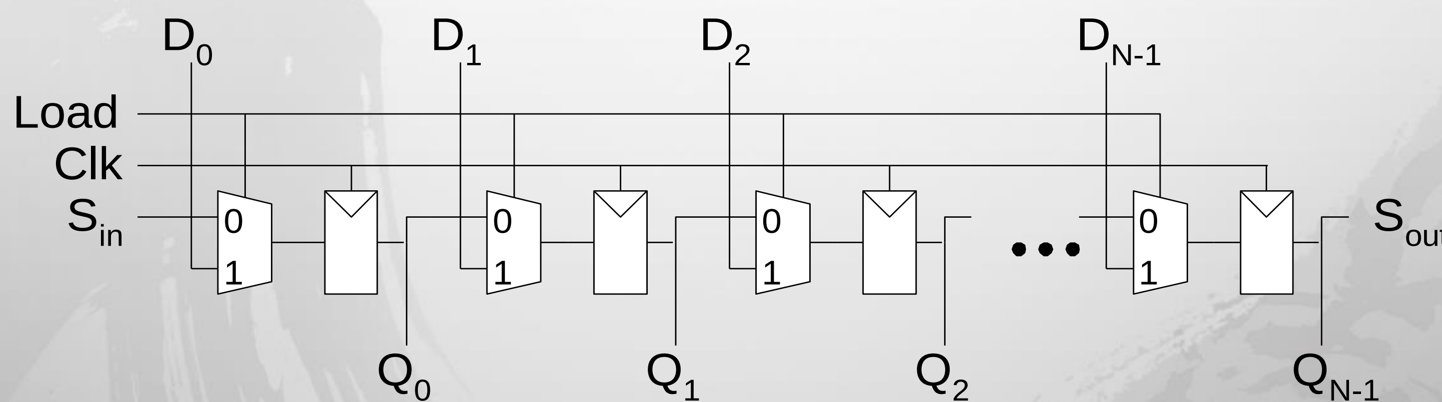


Implementation:



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})



Number Systems

- Numbers we can represent using binary representations
  - **Positive numbers**
    - Unsigned binary
  - **Negative numbers**
    - Two's complement
    - Sign/magnitude numbers

- What about **fractions**?

Numbers with Fractions

- Two common notations:
  - **Fixed-point:** binary point fixed
  - **Floating-point:** binary point floats to the right of the most significant 1

Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

  - Binary point is implied
  - The number of integer and fraction bits must be agreed upon beforehand

Fixed-Point Number Example

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

01111000

Signed Fixed-Point Numbers

- **Representations:**
  - Sign/magnitude
  - Two's complement
- **Example:** Represent -7.5_{10} using 4 integer and 4 fraction bits
  - **Sign/magnitude:**

    11111000
  - **Two's complement:**

    1. +7.5: 01111000
    2. Invert bits: 10000111
    3. Add 1 to lsb: + 1

    10001000

Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write 273_{10} in scientific notation:

$$273 = 2.73 \times 10^2$$

- In general, a number is written in scientific notation as:

$$\pm\ M \times B^E$$

  - **M** = mantissa
  - **B** = base
  - **E** = exponent
  - In the example, M = 2.73, B = 10, and E = 2

Floating-Point Numbers

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa

- **Example:** represent the value 228_{10} using a 32-bit floating point representation
  - We show three versions –final version is called the **IEEE 754 floating-point standard**

Floating-Point Representation 1

1. Convert decimal to binary (**don't reverse steps 1 & 2!**):

   $228_{10} = 11100100_2$

2. Write the number in "binary scientific notation":

   $11100100_2 = 1.11001_2 \times 2^7$

3. Fill in each field of the 32-bit floating point number:
   - The sign bit is positive (0)
   - The 8 exponent bits represent the value 7
   - The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

Floating-Point Representation 2

- First bit of the mantissa is always 1:
  - $228_{10} = 11100100_2 = \mathbf{1.11001} \times 2^7$
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Floating-Point Representation 3

- *Biased exponent*: bias = 127 (01111111_2)
  - Biased exponent = bias + exponent
  - Exponent of 7 is stored as:
  - $127 + 7 = 134 = 0x10000110_2$
- The **IEEE 754 32-bit floating-point representation** of 228_{10}

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**

in hexadecimal: **0x43640000**

Floating-Point Example

- Write -58.25_{10} in floating point (IEEE 754)

1. Convert decimal to binary:

   $58.25_{10} = \mathbf{111010.01_2}$

2. Write in binary scientific notation:

   $$\mathbf{1.1101001 \times 2^5}$$

3. Fill in fields:
   - **Sign bit:** 1 (negative)
   - **8 exponent bits:** (127 + 5) = 132 = $\mathbf{10000100_2}$
   - **23 fraction bits:** **110 1001 0000 0000 0000 0000**

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

in hexadecimal: **0xC2690000**

Floating-Point: Special Cases

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

Floating-Point Precision

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127

- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

Floating-Point: Rounding

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest
- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

Floating-Point Addition

1. Extract exponent and fraction bits
2. Prepend leading 1 to form mantissa
3. Compare exponents
4. Shift smaller mantissa if necessary
5. Add mantissas
6. Normalize mantissa and adjust exponent if necessary
7. Round result
8. Assemble exponent and fraction back into floating-point format

Floating-Point Addition Example

- Add the following floating-point numbers:

  0x3FC00000

  0x40500000

Floating-Point Addition Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

Floating-Point Addition Example

3. **Compare exponents**
   128 – 127 = 1, so shift N1 right by 1 bit

4. **Shift smaller mantissa if necessary**
   shift N1's mantissa: 1.1 >> 1 = 0.11 ($\times 2^1$)

5. **Add mantissas**

$$
\begin{array}{r}
0.11 \times 2^1 \\
+\ 1.101 \times 2^1 \\
\hline
10.011 \times 2^1
\end{array}
$$

Floating-Point Addition Example

6. **Normalize mantissa and adjust exponent if necessary**

   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. **Round result**

   No need (fits in 23 bits)

8. **Assemble exponent and fraction back into floating-point format**

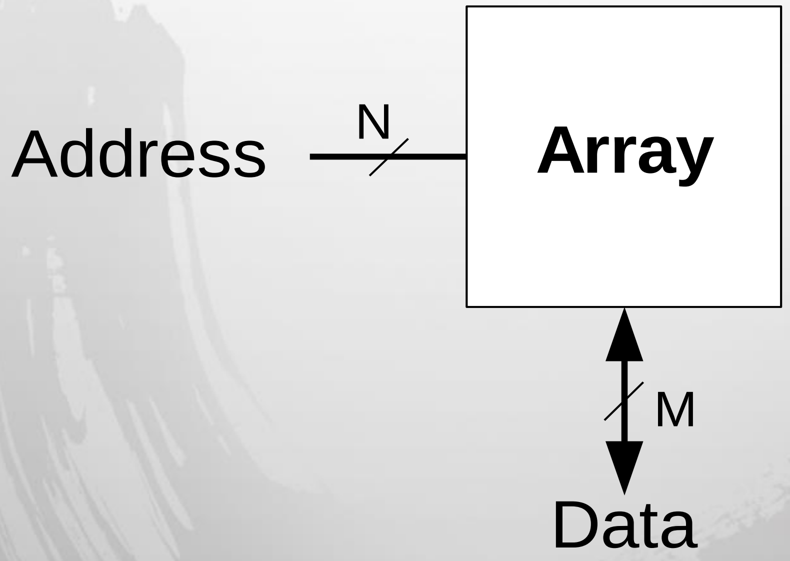
   $S = 0$, $E = 2 + 127 = 129 = 10000001_2$, $F = 001100..$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

in hexadecimal: **0x40980000**

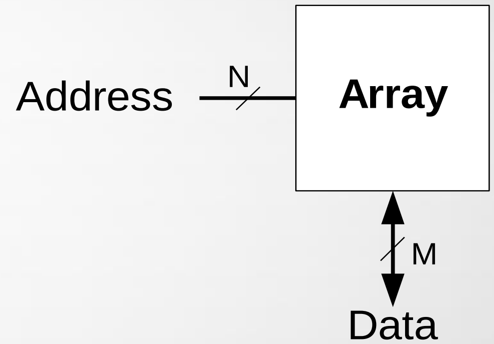
Memory Arrays

- 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



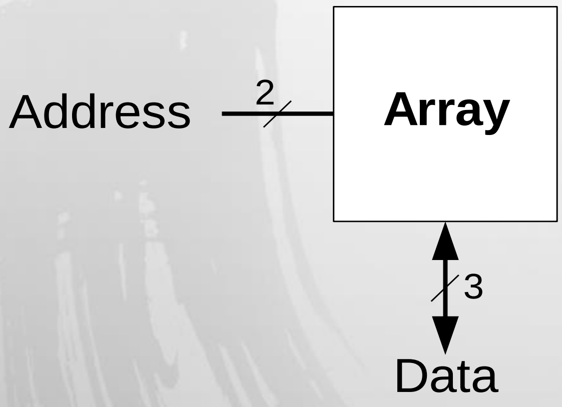
Memory Arrays

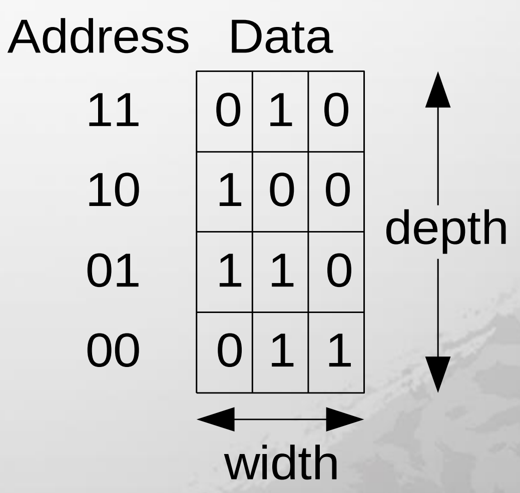
- 2-dimensional array of bit cells
- Each bit cell stores one bit
- N address bits and M data bits:
  - 2^N rows and M columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$



Memory Array Example

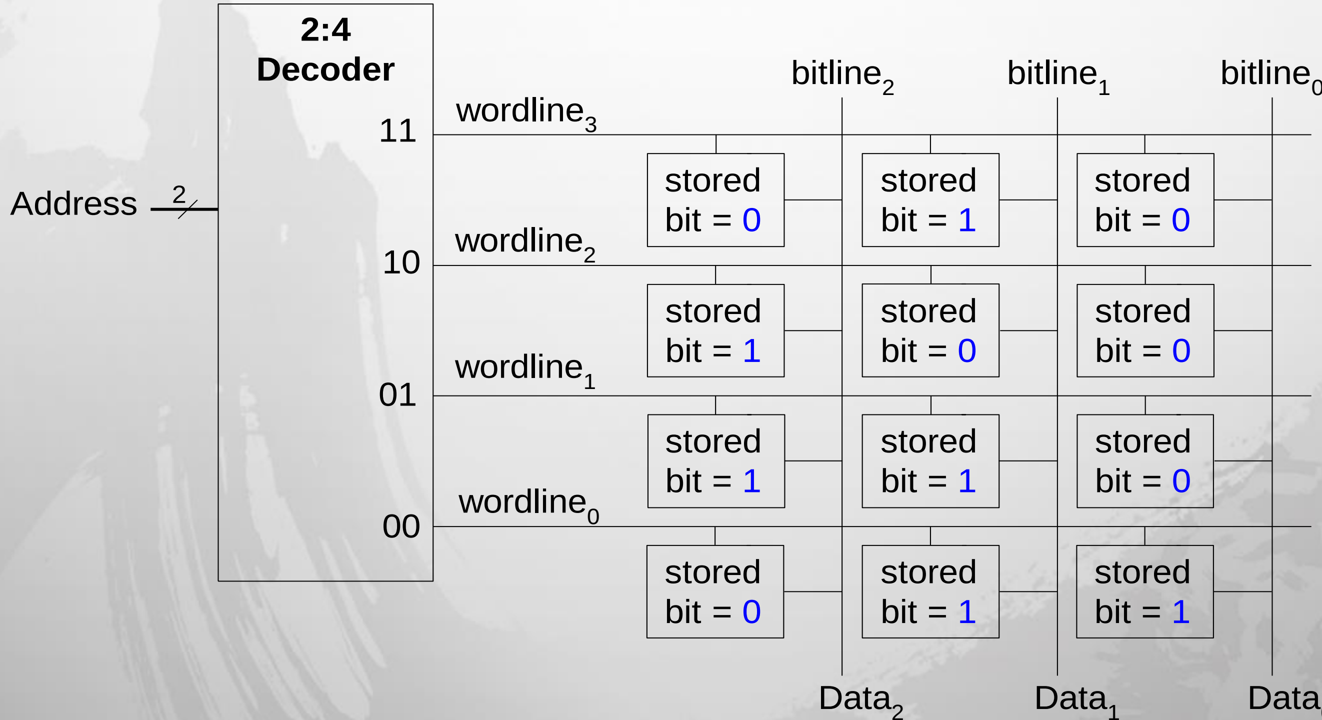
- $2^2 \times$ 3-bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100



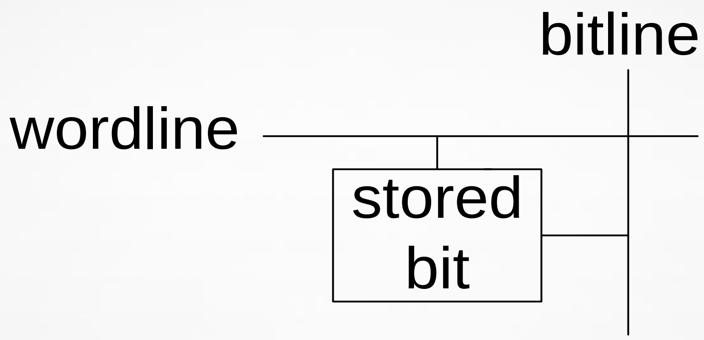


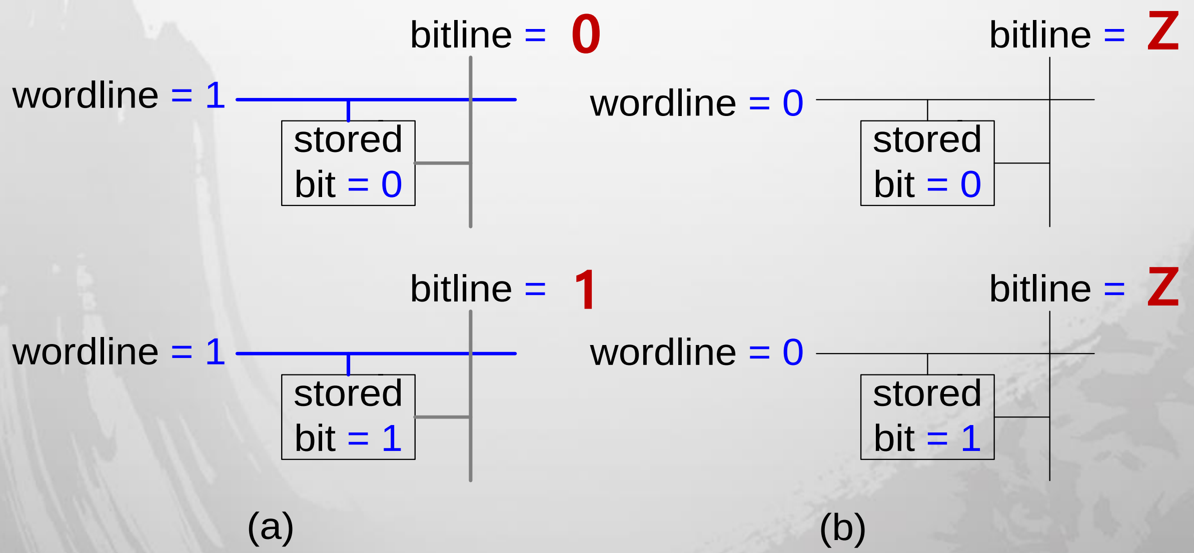
Memory Array

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



Memory Array Bit Cells





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 (nonvolatile memories)
  - 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

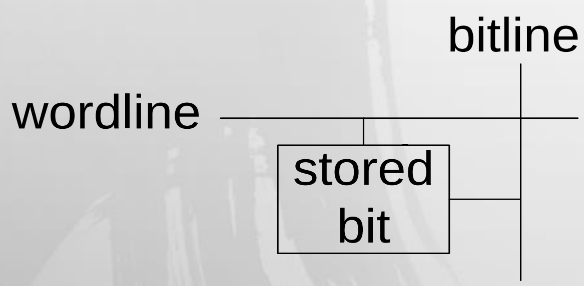
- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
  - DRAM uses a capacitor
  - SRAM uses cross-coupled inverters

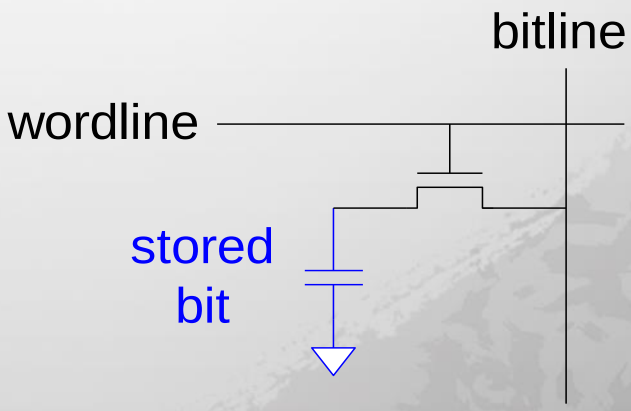
Robert Dennard, 1932 -

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

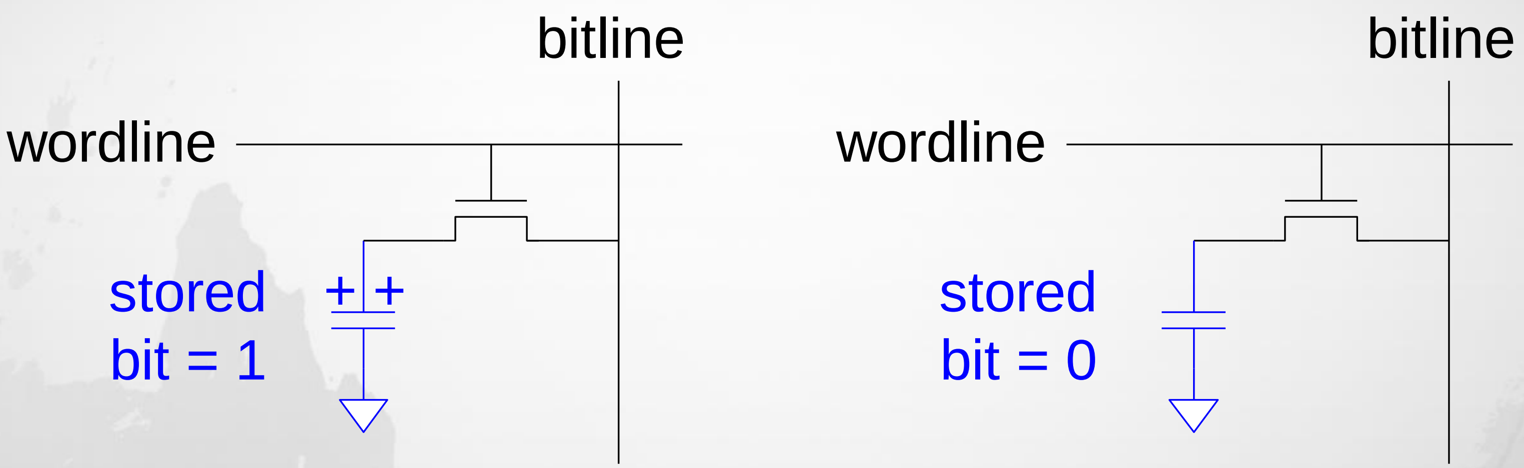
DRAM

- 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

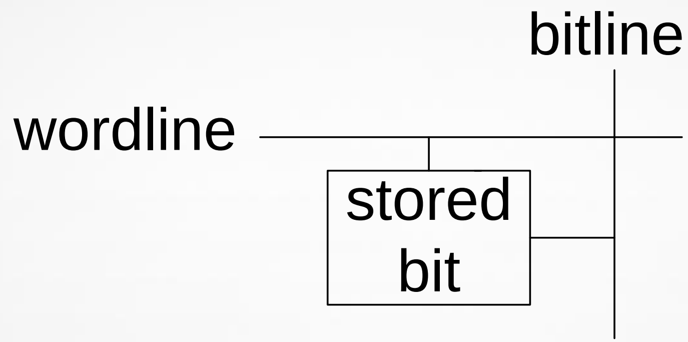


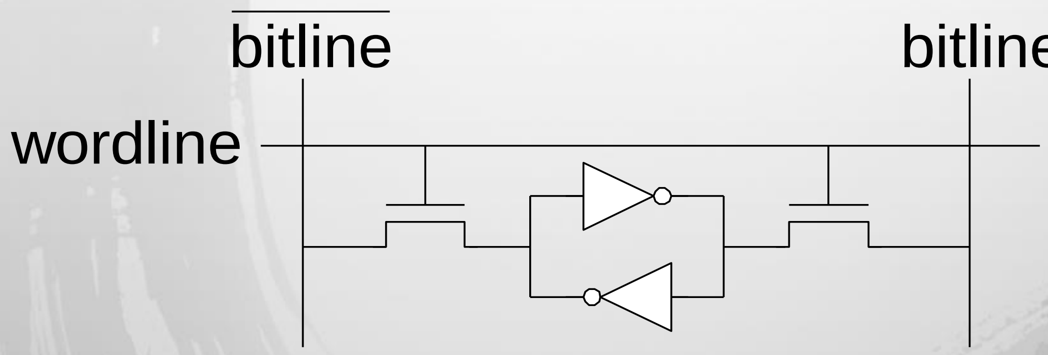


DRAM

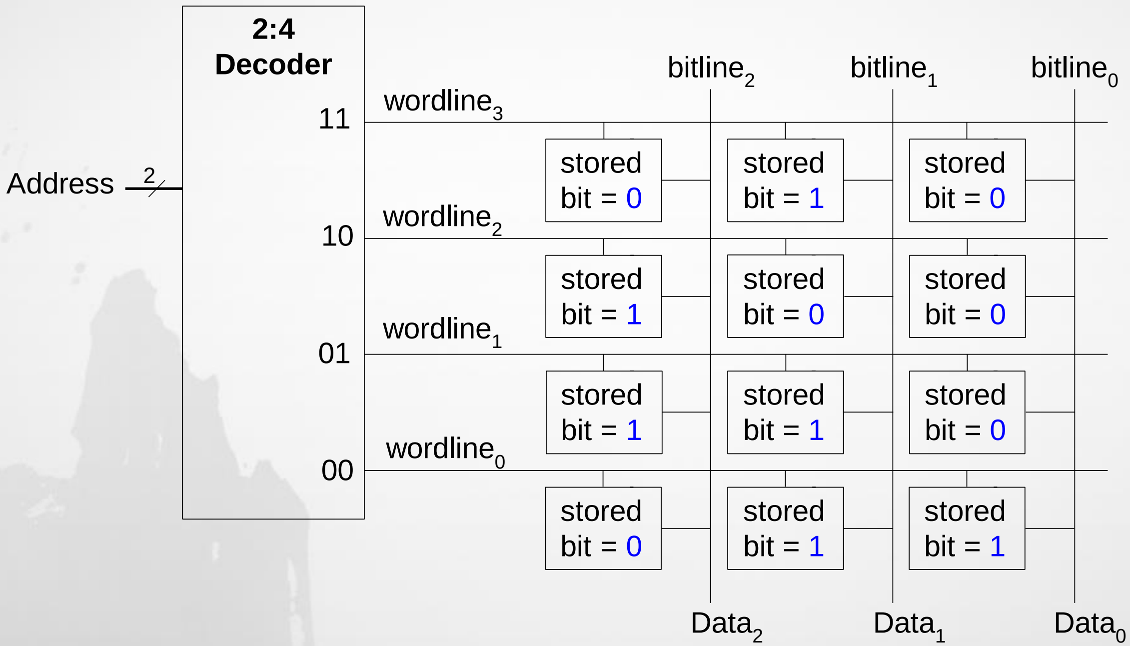

bitline
wordline
stored
bit = 1
+ +
bitline
wordline
stored
bit = 0

SRAM

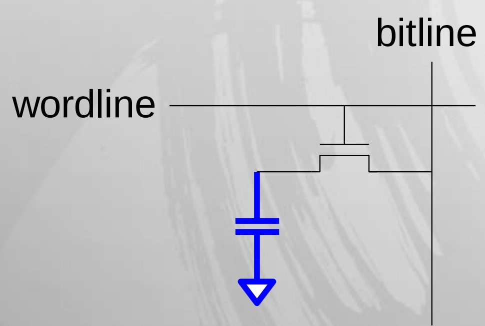




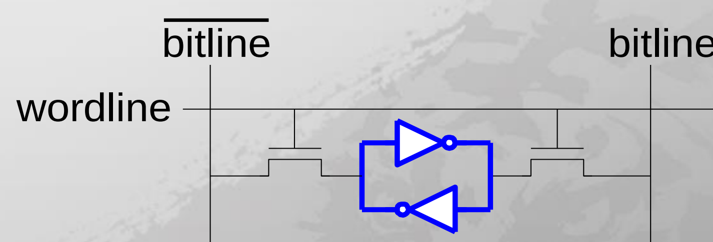
Memory Arrays Review



DRAM bit cell:



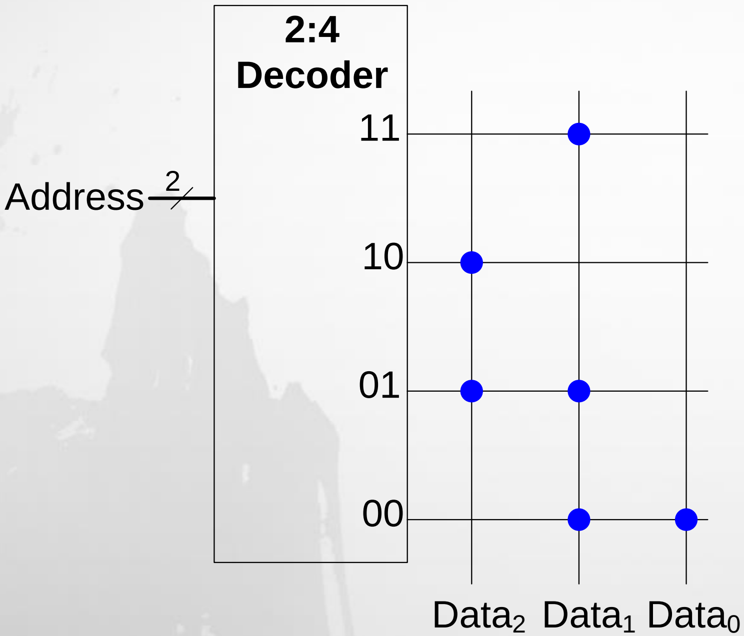
SRAM bit cell:

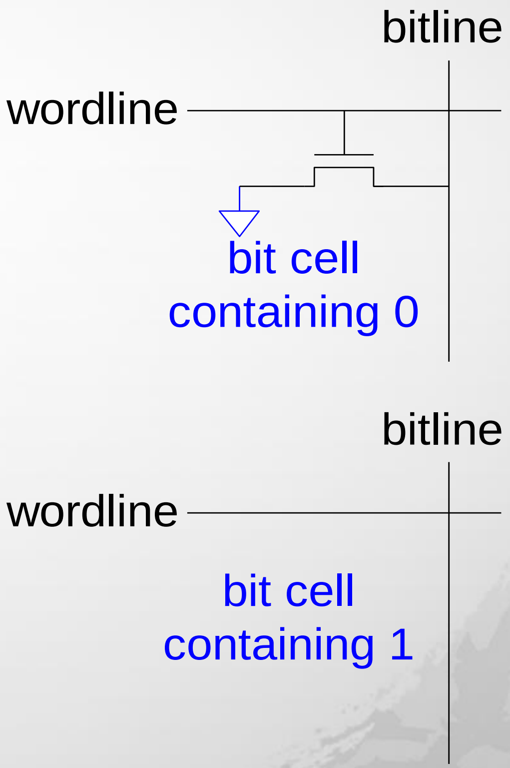


DRAM vs SRAM

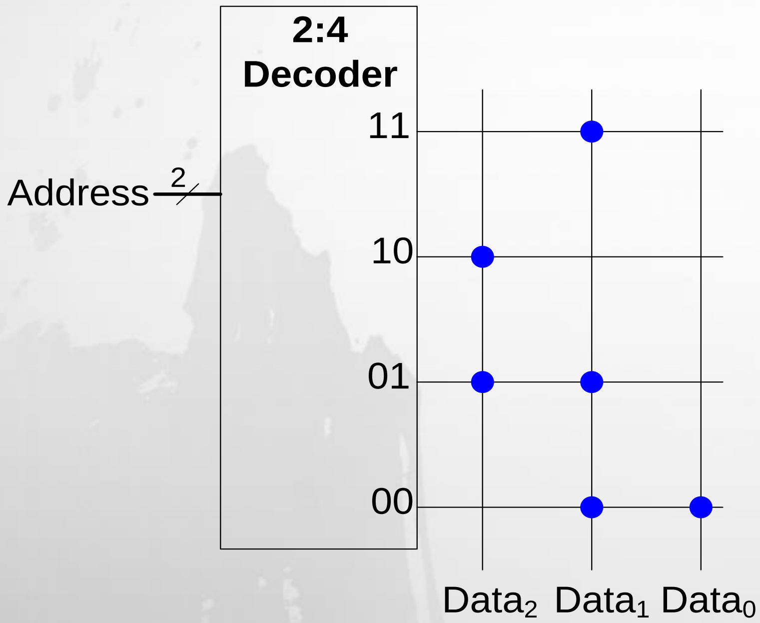
	DRAM	SRAM
Speed	Slower	Faster
Size	Smaller	Bigger
Cost	Less expensive	More expensive
Access	Harder	Easier
Construction	Simple	Difficult
Power consumption	More	Less

ROM: Dot Notation





ROM Storage

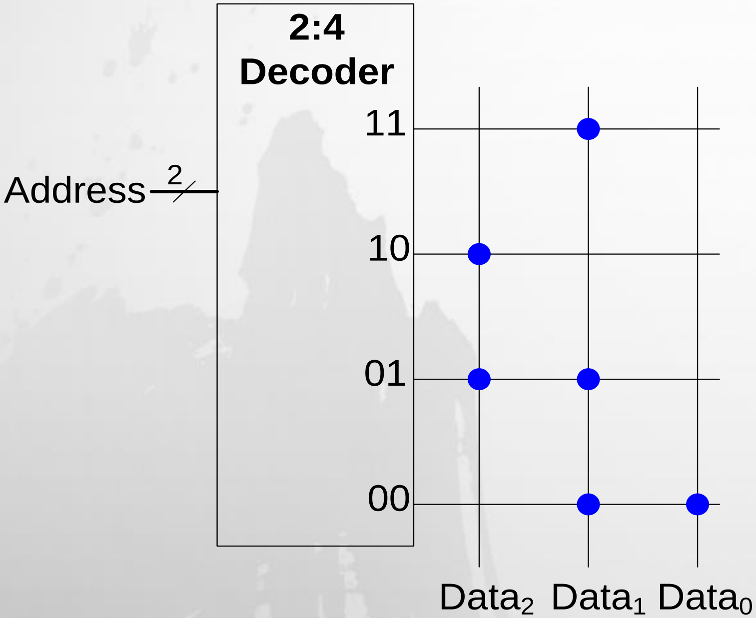


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

ROM Logic



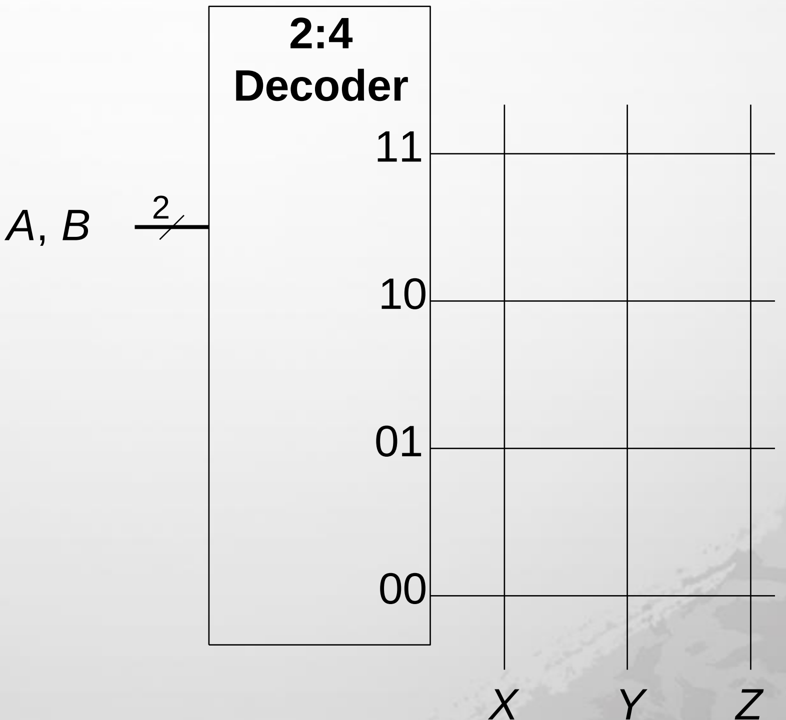
$$Data_2 = A_1 \oplus A_0$$

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

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

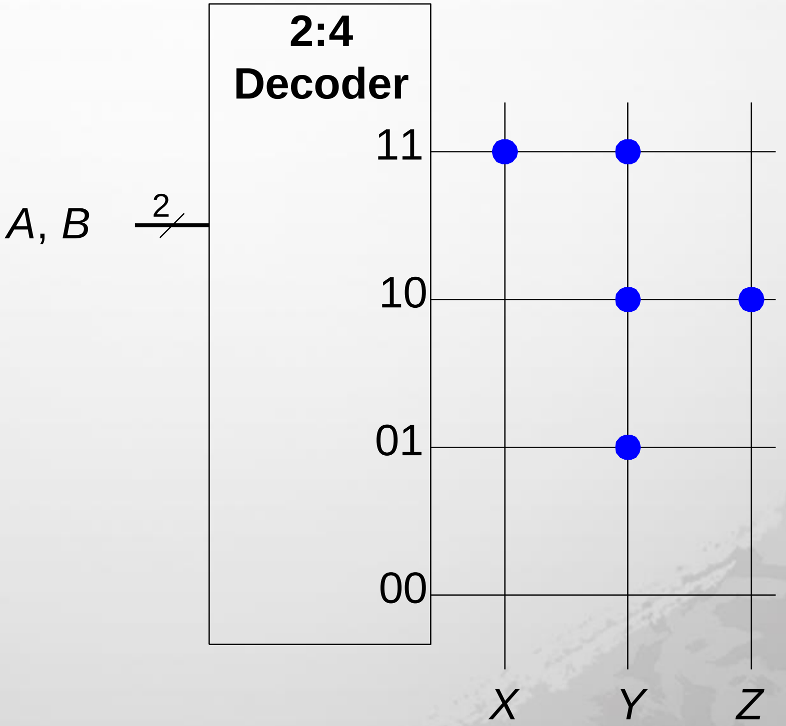
Example: Logic with ROMs

- Implement the following logic functions using a $2^2 \times$ 3-bit ROM:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$

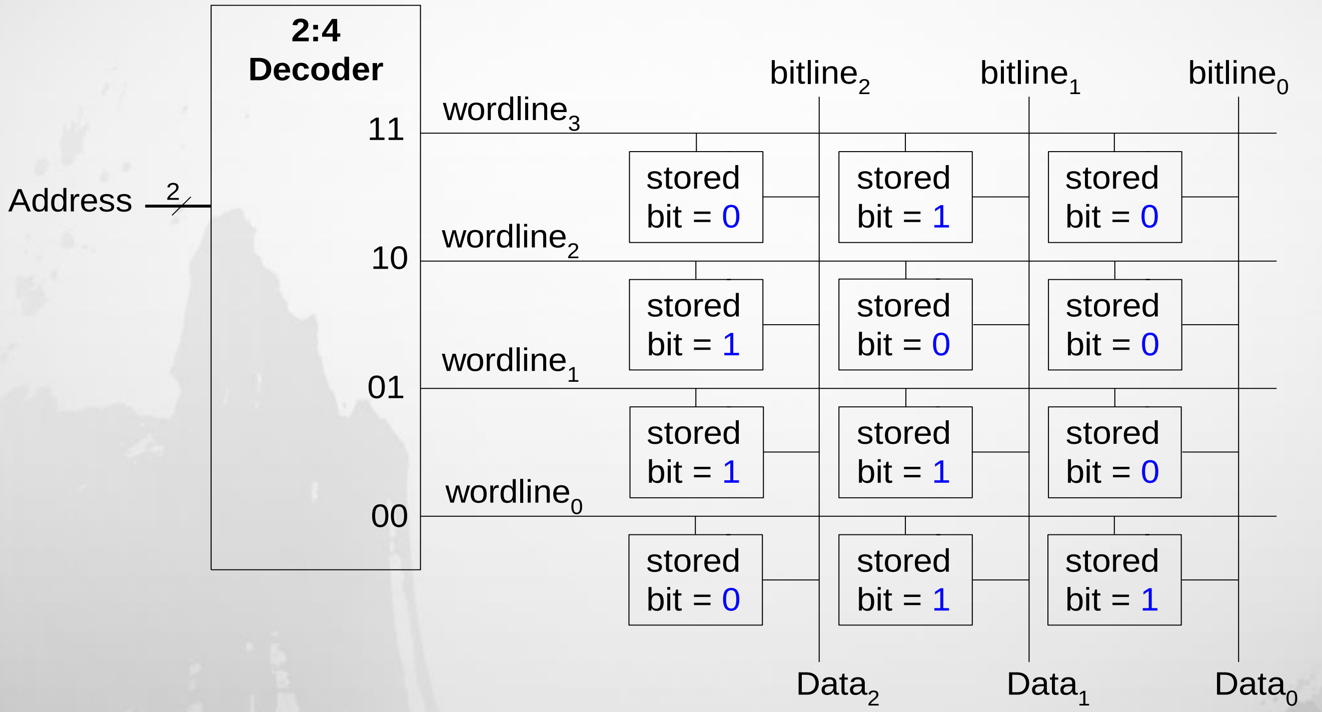


Example: Logic with ROMs

- Implement the following logic functions using a $2^2 \times$ 3-bit ROM:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



Logic with Any Memory Array



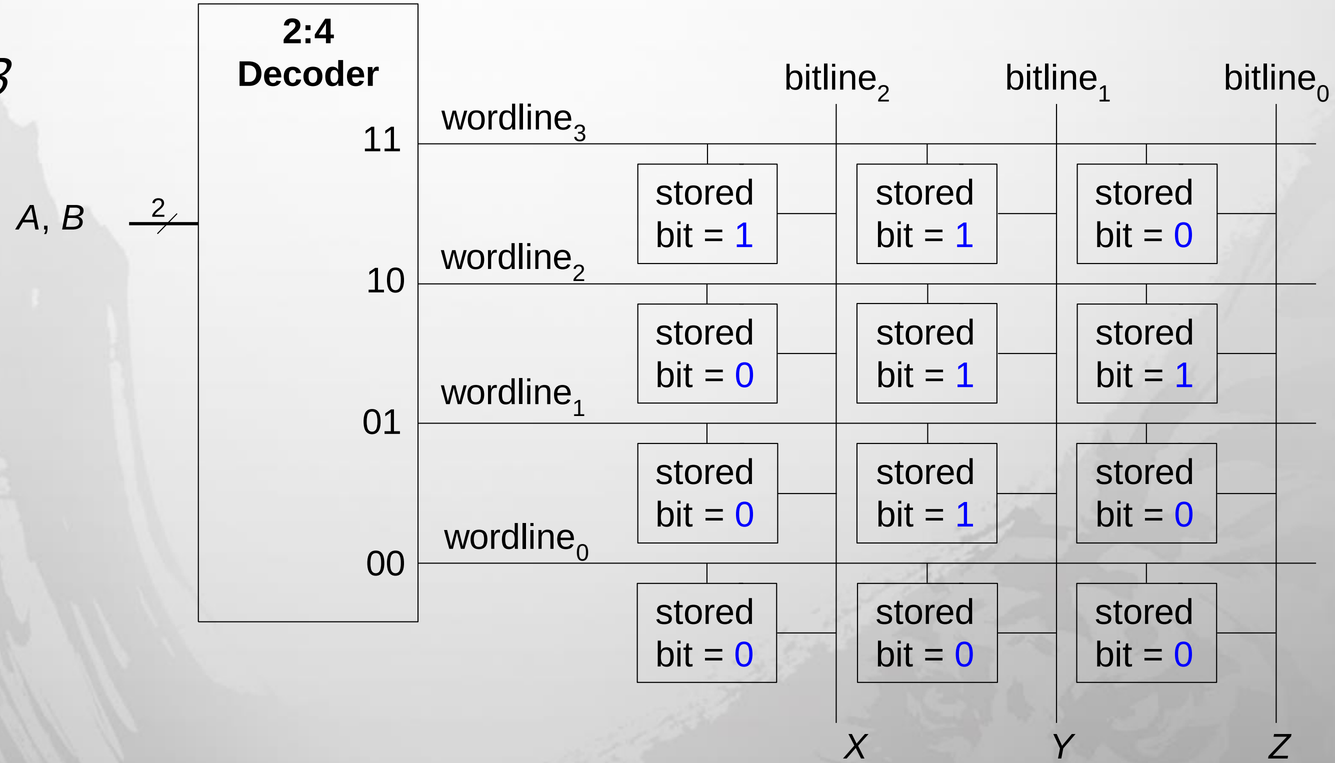
$$Data_2 = A_1 \oplus A_0,\ Data_1 = \overline{A_1} + A_0,\ Data_0 = \overline{A_1}\,\overline{A_0}$$

Logic with Memory Arrays

- Implement the following logic functions using a $2^2 \times$ 3-bit memory array:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$

Logic with Memory Arrays

- Implement the following logic functions using a $2^2 \times$ 3-bit memory array:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



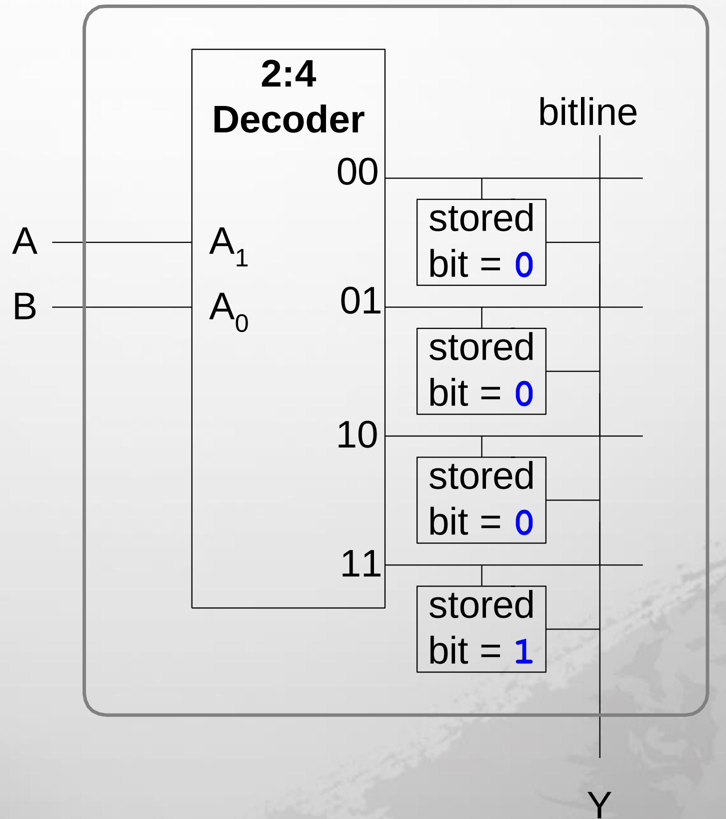
Logic with Memory Arrays

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

Truth Table

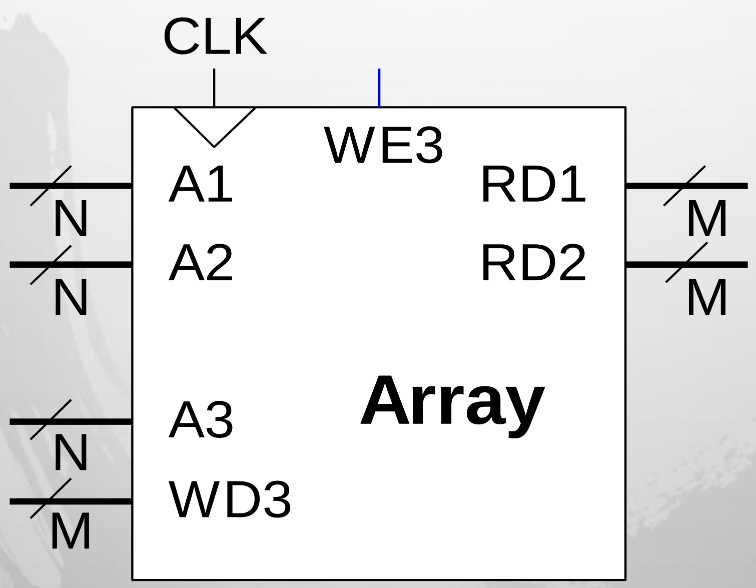
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

4-word x 1-bit Array



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

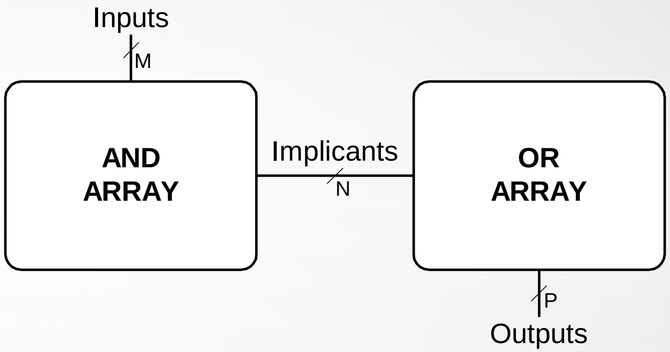


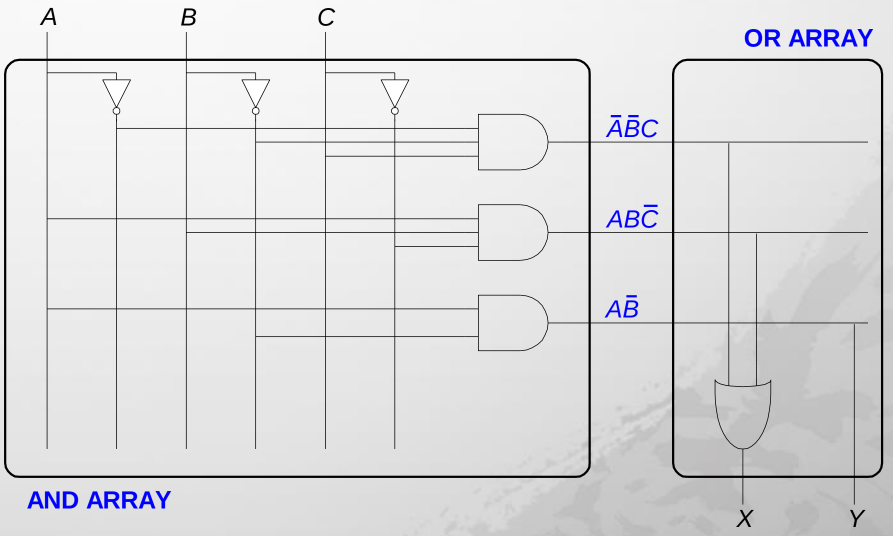
Logic Arrays

- **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

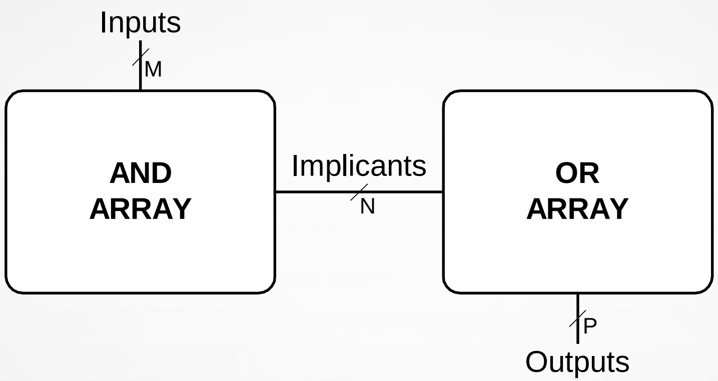
PLAs

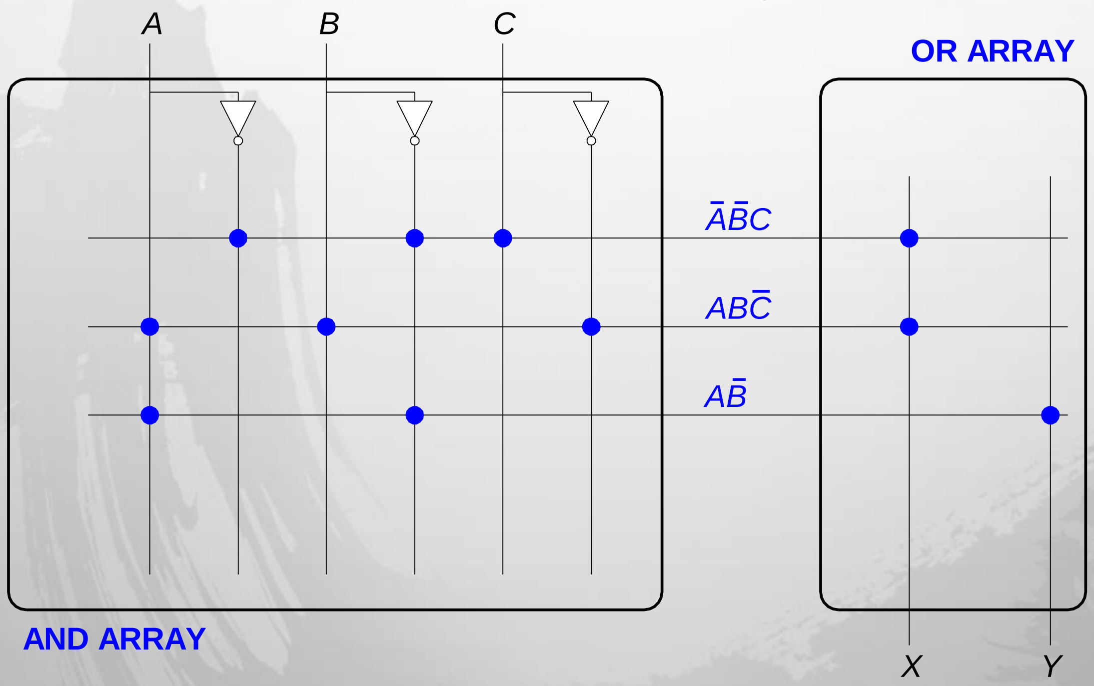
- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = A\bar{B}$





PLAs: Dot Notation





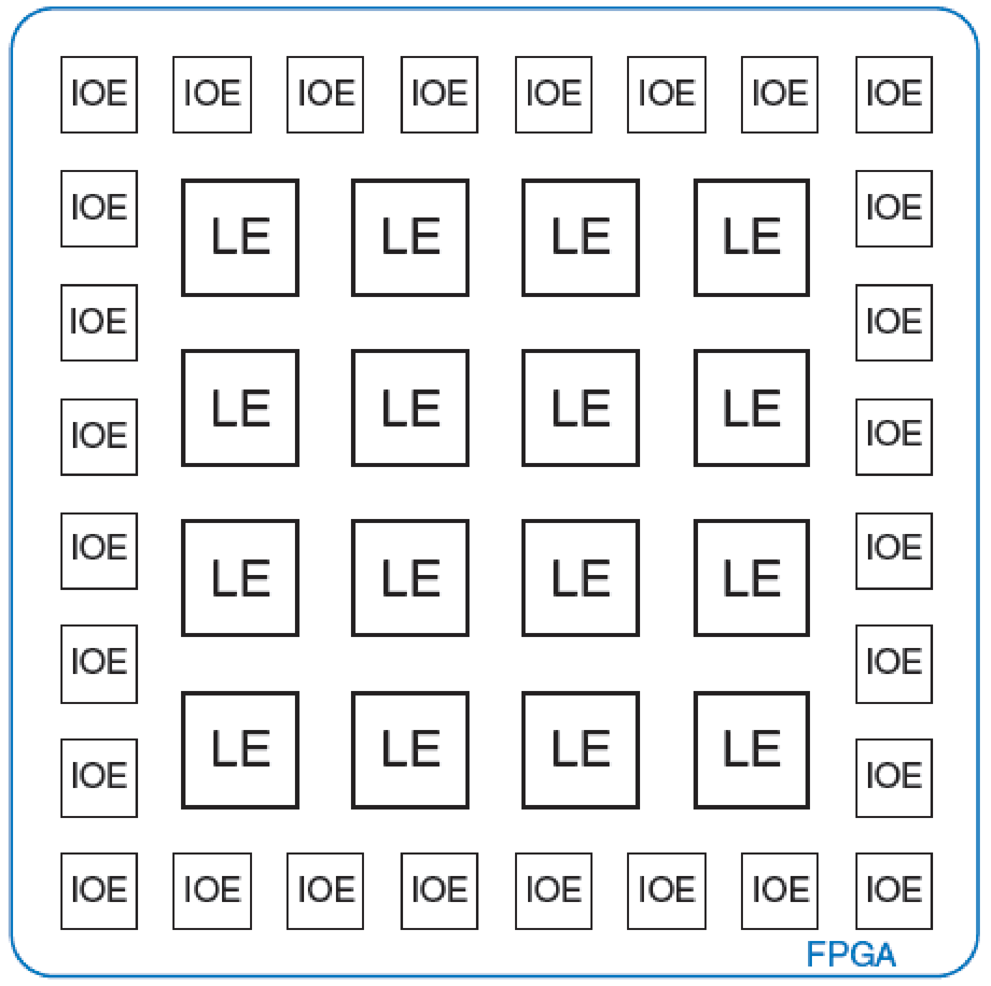
Exercise

- Implement the following logic with ROM and PLA, respectively. Represent your implementation with the DOT notation.
  - $X = A\overline{B}C + AB + \overline{B}C$
  - $Y = A\overline{B}C + AB$
  - $Z = AB + \overline{B}C$

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

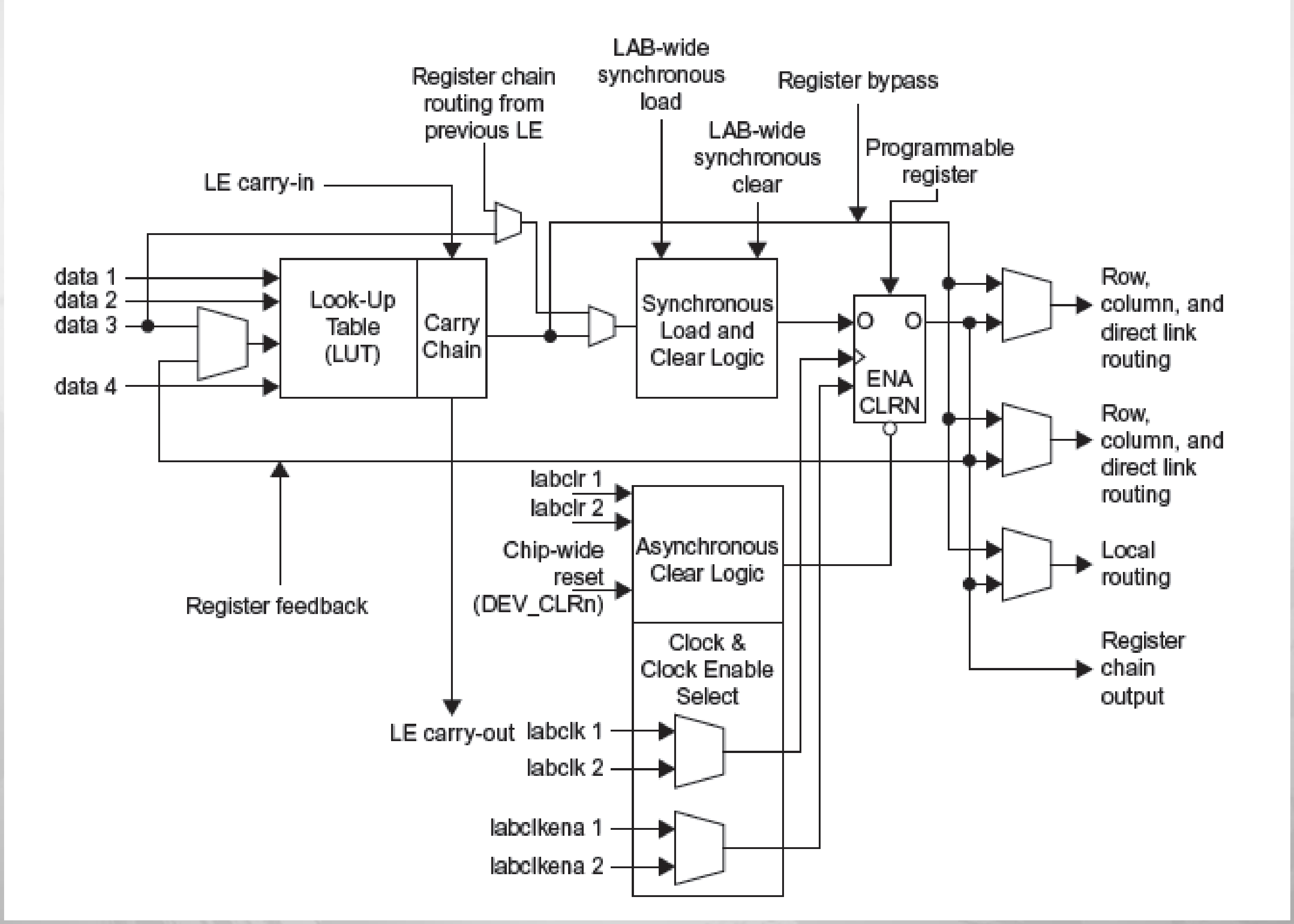
General FPGA Layout



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

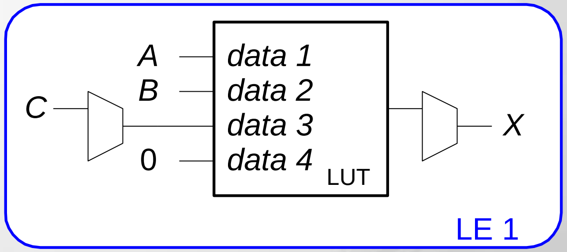
LE Configuration Example

- Show how to configure a Cyclone IV LE to perform the following functions:
  - $X = \overline{A}\,\overline{B}C + AB\overline{C}$
  - $Y = A\overline{B}$

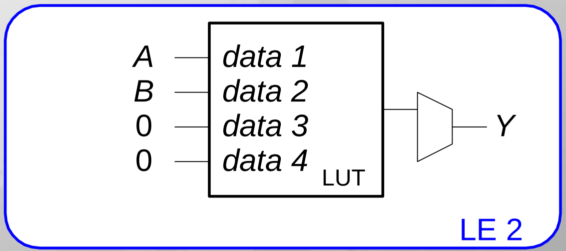
LE Configuration Example

- Show how to configure a Cyclone IV LE to perform the following functions:
  - $X = \overline{A}\overline{B}C + AB\overline{C}$
  - $Y = A\overline{B}$

(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