Arithmetic Logic Unit (ALU) – Introduction to Computer (計算機概論)

Winston H. Hsu (徐宏民)
National Taiwan University, Taipei

November 7, 2022

Office: R512, CSIE Building
Communication and Multimedia Lab (通訊與多媒體實驗室)
http://winstonhsu.info

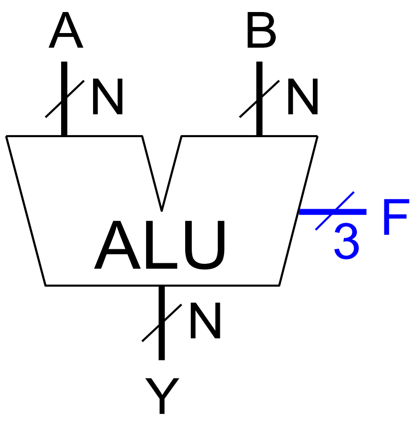
The majority of the slides are from the textbook

Administrative Matters

- Today's lecture section
  - Section 5.1 – 5.3 in [Harris] except prefix adder, 5.2.7

Arithmetic Logic Unit (ALU)

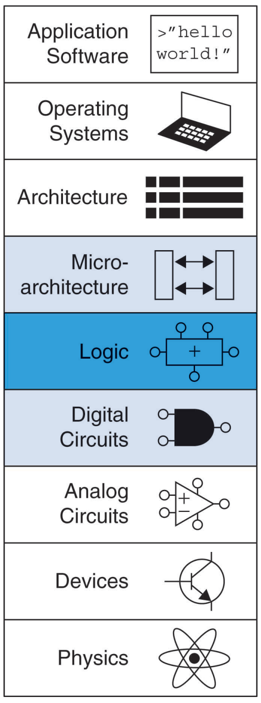
ALU combines a variety of mathematical and logical operations into a single unit and forms the heart of most computer systems.



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

Topics

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

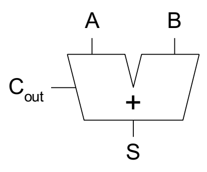


Introduction

- 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

1-Bit Adders

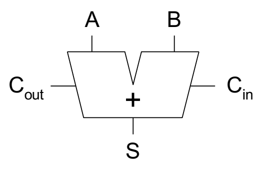
Half Adder



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

S =
C_{out} =

Full Adder

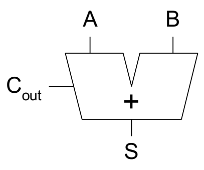


C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =
C_{out} =

1-Bit Adders

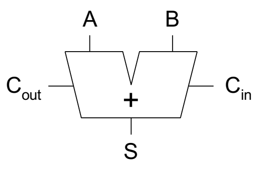
Half Adder



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

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

Full Adder



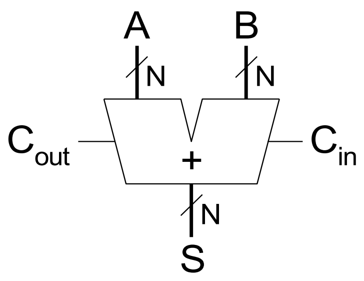
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

Multibit Adders (CPAs)

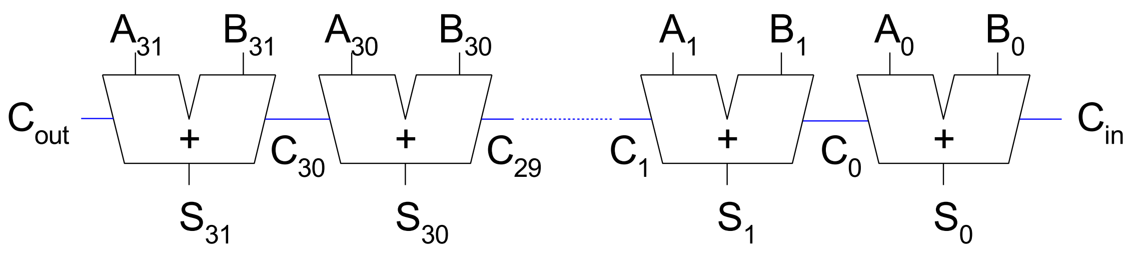
- Types of carry propagate adders (CPAs):
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

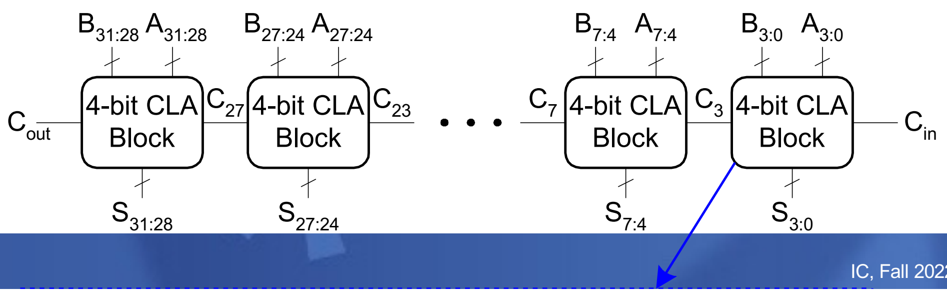
- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



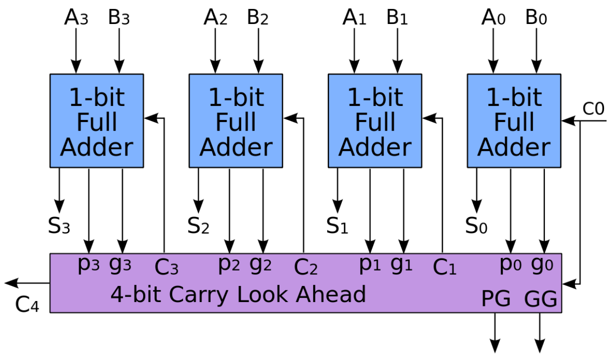
Ripple-Carry Adder Delay

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

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.

$$P_i = A_i + B_i$$

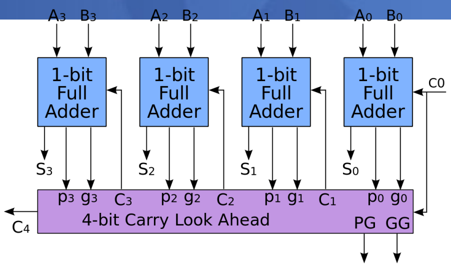
    - The carry out of column *i* (C_i) is:

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

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 $P_{3:0}$) :

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

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

$$C_3 = G_{3:0} + P_{3:0} C_{in}$$

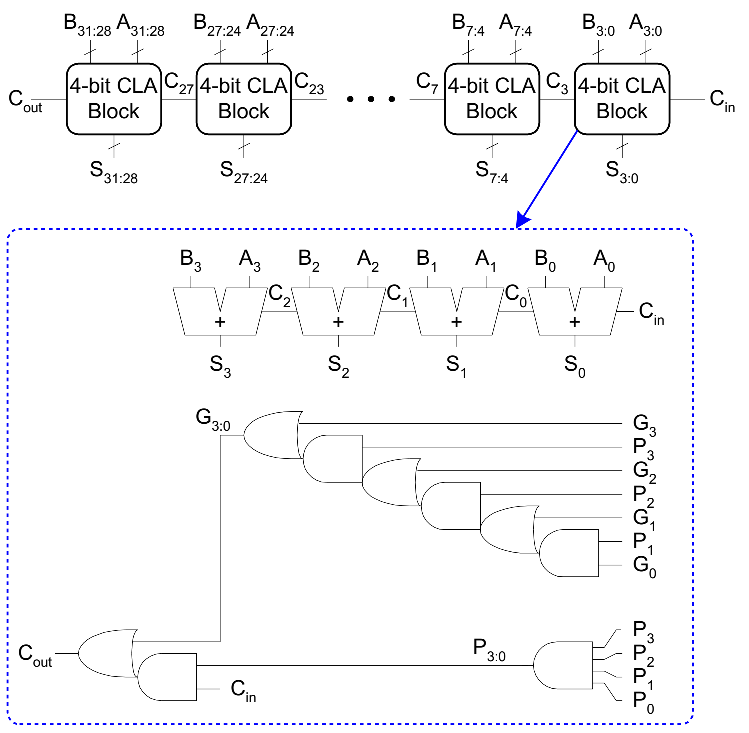
- Generally,

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

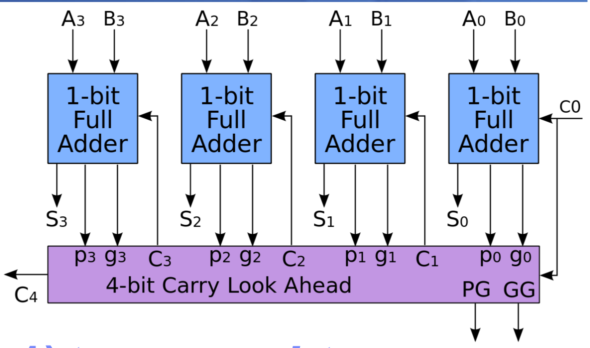
$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

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

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 (in parallel)

  – t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$ (in parallel)

  – 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 $N > 16$

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

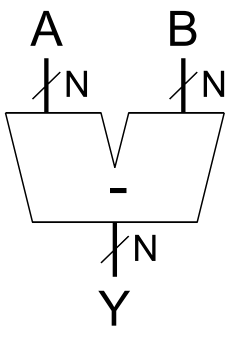
$$\begin{aligned}
t_{ripple} &= Nt_{FA} = 32(300 \text{ ps}) \\
&= 9.6 \text{ ns} \\
t_{CLA} &= t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA} \\
&= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\
&= 3.3 \text{ ns} \\
t_{PA} &= t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR} \\
&= [100 + \log_2 32(200) + 100] \text{ ps} \\
&= 1.2 \text{ ns}
\end{aligned}$$

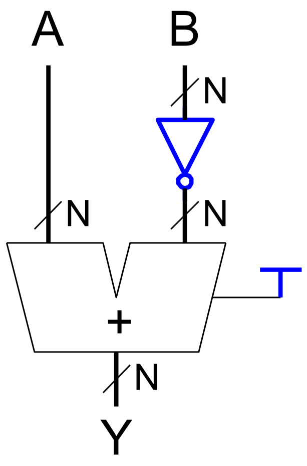
Subtracter

$Y = A - B = A + B' + 1$

(+1 can be with $C_{in} = 1$)

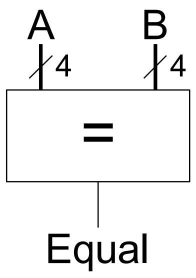
Symbol **Implementation**

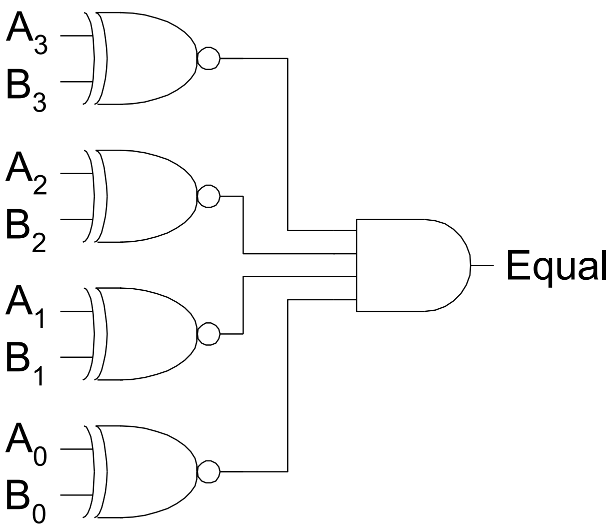




Comparator: Equality

Symbol **Implementation**

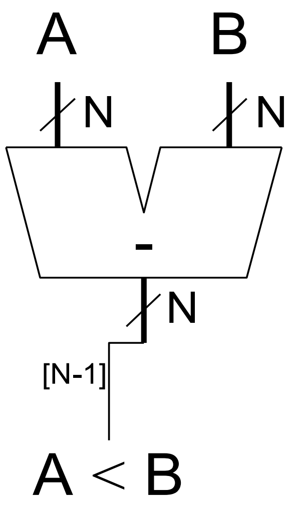




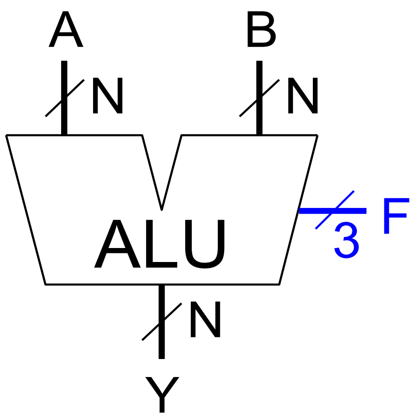
XNOR gate

Comparator: Less Than

Computing A-B and looking at the sign of the results. If the result is negative, the sign bit is 1.



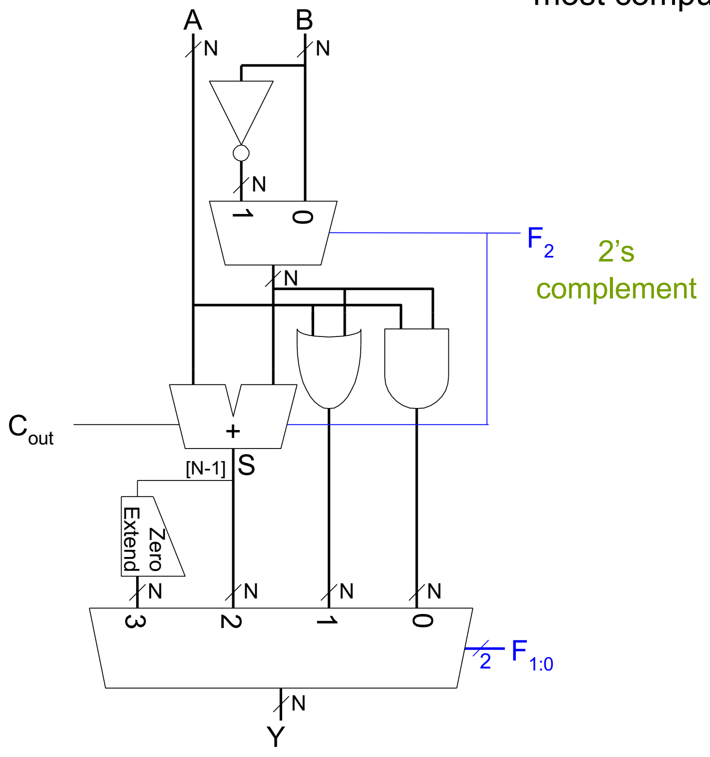
Arithmetic Logic Unit (ALU)



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

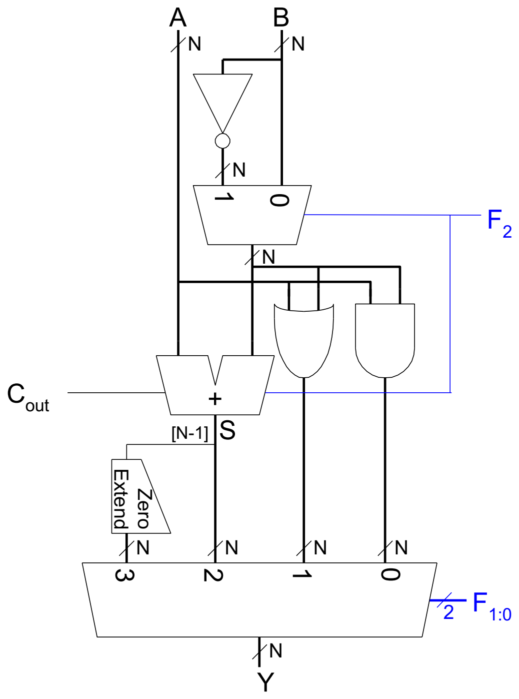
ALU Design

ALU combines a variety of mathematical and logical operations into a single unit and forms the heart of most computer systems.



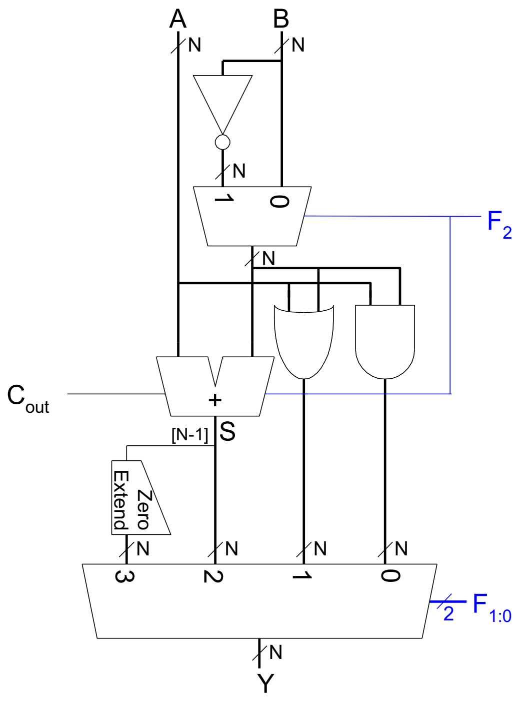
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: *A* = 25 and *B* = 32

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.

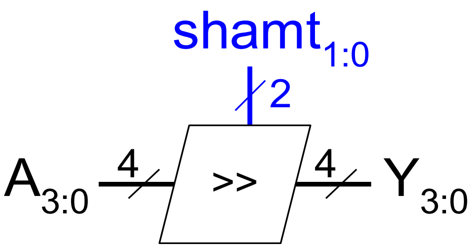
Shifters

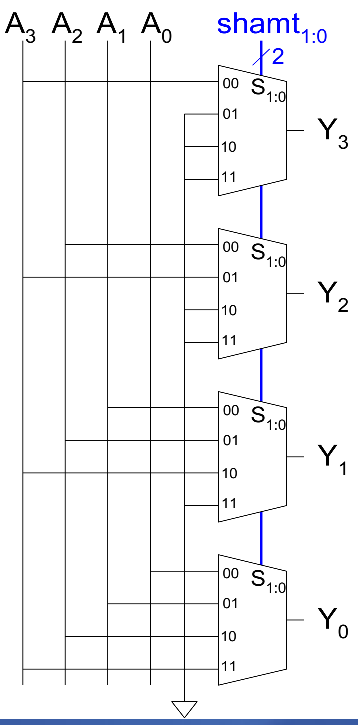
- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb) (for multiplying and dividing signed numbers).
  - 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





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

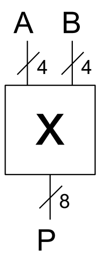
- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand (realized by AND)
- **Shifted** partial products **summed** to form result

Decimal		Binary	
230	multiplicand	0101	
x 42	multiplier	x 0111	
460	partial	0101	
+ 920	products	0101	(shift left)
9660		0101	
		+ 0000	
	result	0100011	

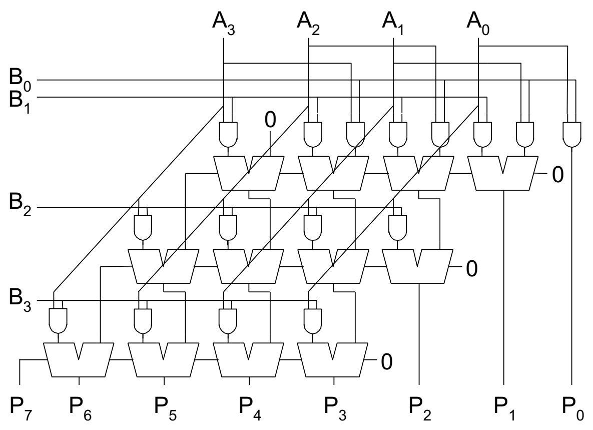
230 x 42 = 9660

5 x 7 = 35

4 x 4 Multiplier



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



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.

Fixed-Point Number Example

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

01111000

Fixed-Point Number Example

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

Fixed-Point Number Example

- 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:

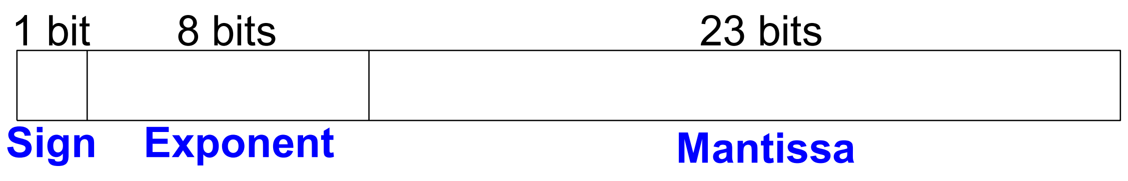
$$273 = 2.73 \times 10^2$$

- 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



- 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 (1/3)

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 (2/3)

- First bit of the mantissa is always 1:
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- 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 (3/3)

- *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)

Floating-Point Example

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

1. Convert decimal to binary:

   $58.25_{10} = 111010.01_2$

2. Write in binary scientific notation:

   1.1101001×2^5

3. Fill in fields:

   Sign bit: 1 (negative)
   8 exponent bits: (127 + 5) = 132 = 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

127 – 128 = -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

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

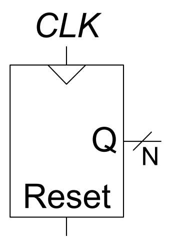
1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

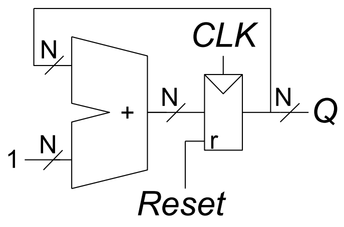
in hexadecimal: 0x40980000

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**

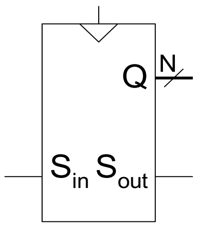




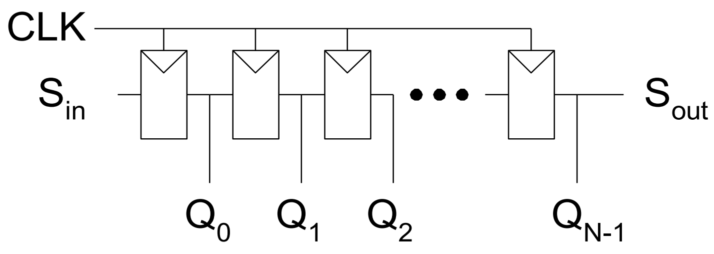
Shift Registers

- 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:

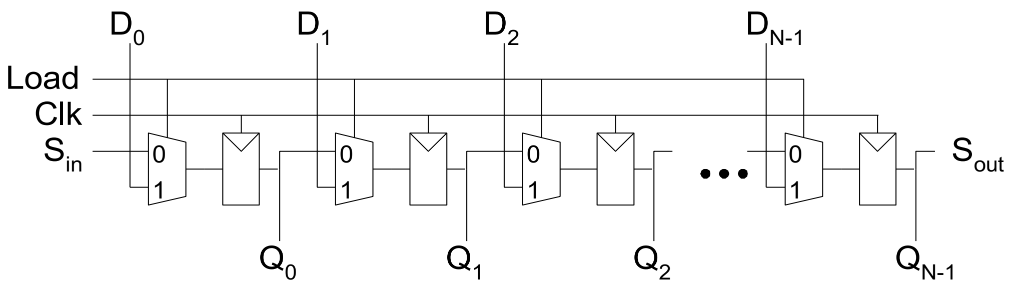


Implementation:



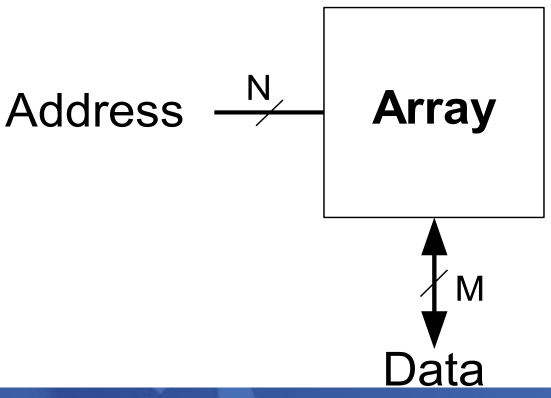
Shift Register with Parallel Load

- 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})



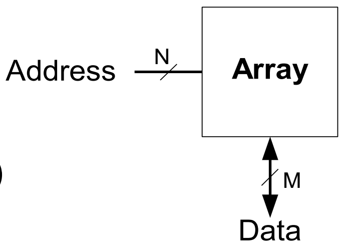
Memory Arrays

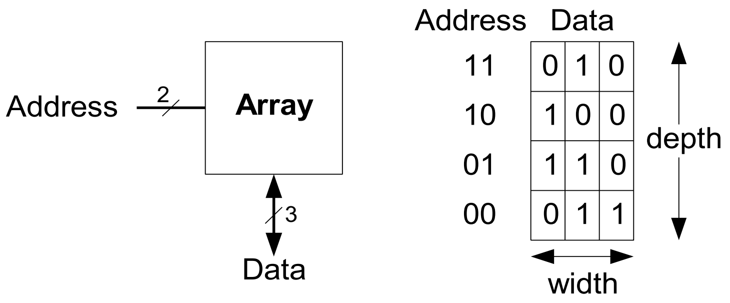
- Efficiently store large amounts of data
- 3 common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- *M*-bit data value read/ written at each unique *N*-bit address



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 = $2^N \times M$

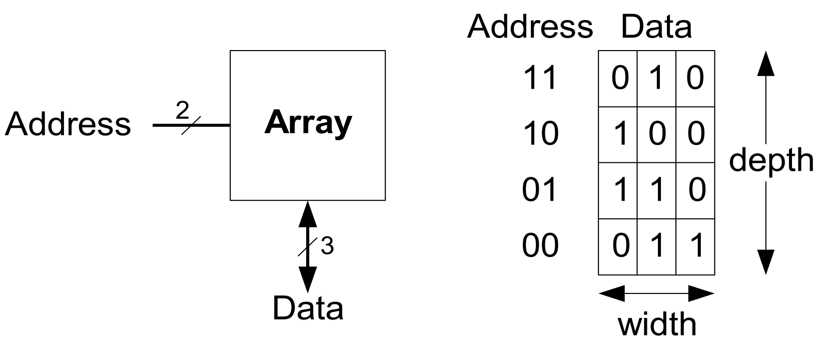




Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

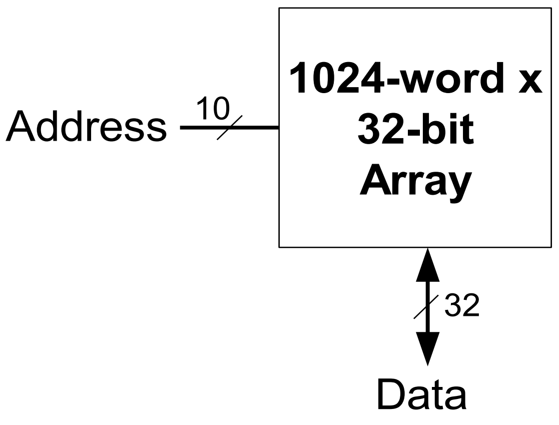
Memory Array Example

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

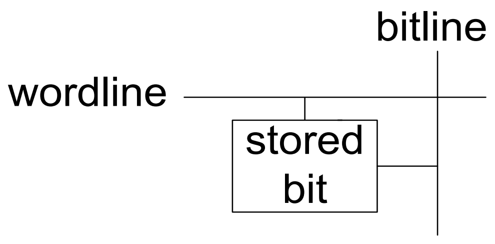


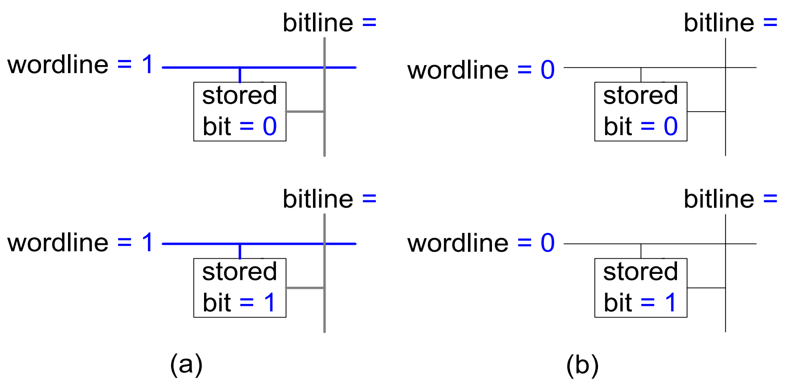
Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

Memory Arrays

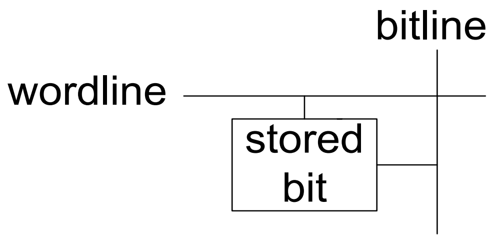


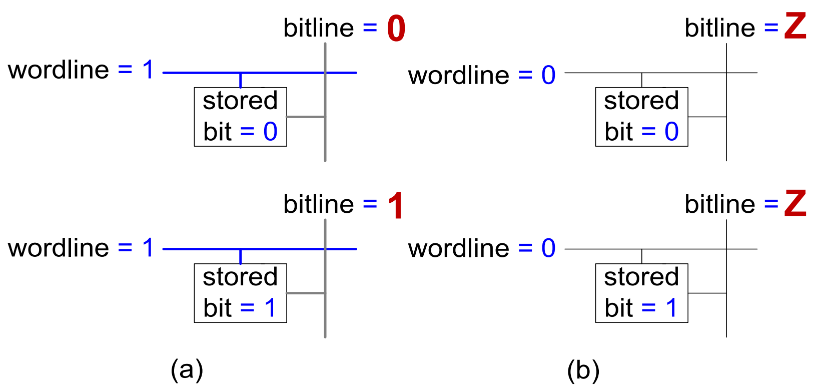
Memory Array Bit Cells





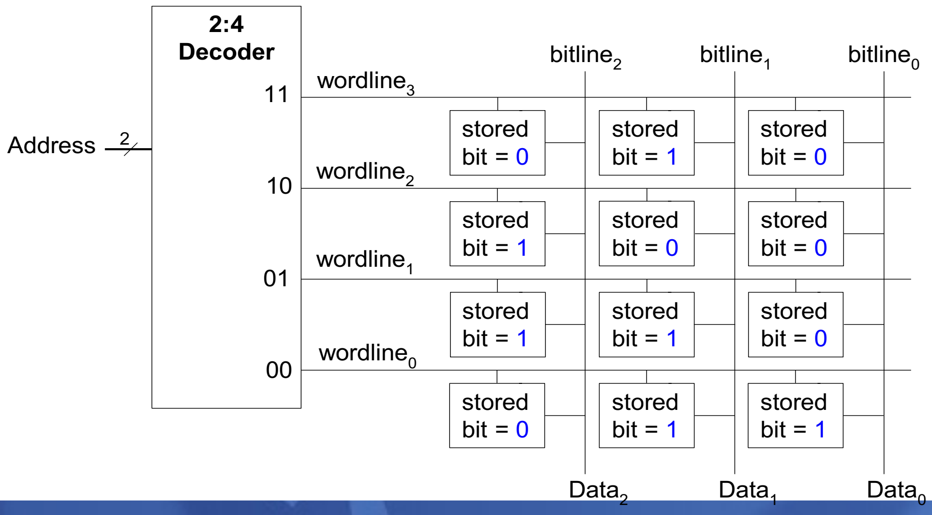
Memory Array Bit Cells





Memory Array

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



Type 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
  - SRAM和DRAM的差異在於，DRAM得隨時充電，而SRAM儲存記憶不必作自動充電的動作，會出現充電動作的唯一時刻是有寫入動作時。如果沒有寫入的指令，在SRAM裏不會有任何東西被更動，這也是它為什麼被稱為靜態的原因。SRAM的優點是它比DRAM快得多。缺點則是它比DRAM貴許多，通常被採用來作為快取記憶體（Cache Memory）。