Chapter 5 :: Digital Building Blocks

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Chapter 5 :: Topics

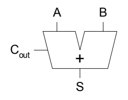
- **Introduction**
- **Arithmetic Circuits**
- **Number Systems**
- **Sequential Building Blocks**
- **Memory Arrays**
- **Logic Arrays**

Introduction

- Digital building blocks include:
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- Building blocks are important in their own right and they demonstrate hierarchy, modularity, and regularity:
  - They are built from a hierarchy of simpler components.
  - They have well-defined interfaces and functions.
  - Their regular structure is easily extended to different sizes.
- We'll use many of these building blocks to build a microprocessor in Chapter 7

1-Bit Adders

Half Adder

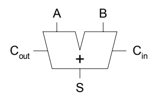


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

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		

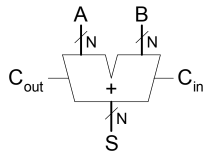
S =

C_{out} =

Multibit Adder, also called CPA

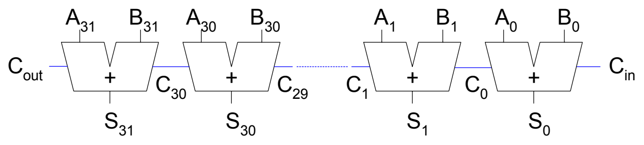
- Several types of carry propagate adders (CPAs) are:
  - Ripple-carry adders (slow)
  - Carry-lookahead adders (fast)
  - Prefix adders (faster)
- Carry-lookahead and prefix adders are faster for large adders but require more hardware.

Symbol



Ripple-Carry Adder

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



Ripple-Carry Adder Delay

- The delay of an *N*-bit ripple-carry adder is:

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

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

Carry-Lookahead Adder

- Computes the carry out (C_{out}) for *N*-bit blocks first, so the carry doesn't have to ripple through the entire chain.
- Does this by computing *generate* (*G*) and *propagate* (*P*) signals for columns and then *N*-bit blocks.
- A column (bit *i*) can produce a carry out by either *generating* a carry out or *propagating* a carry in to the carry out.
- We define generate (G_i) and propagate (P_i) signals for each column:
  - A column will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

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

$$P_i = A_i + B_i$$

- We compute the carry out of a column (C_i) as:

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

Carry-Lookahead Adder

- Now we compute generate and propagate signals for *N*-bit blocks.
- For example, we can calculate generate and propagate signals for a 4-bit block ($G_{3:0}$ and $P_{3:0}$) :
  - A 4-bit block will generate a carry out if column 3 generates a carry (G_3) or if column 3 propagates a carry (P_3) that was generated or propagated in a previous column as described by the following equation:

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

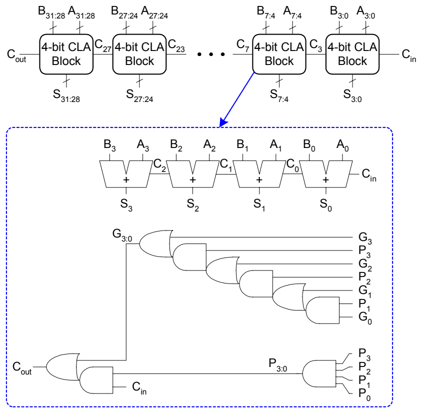
  - A 4-bit block will propagate a carry in to the carry out if all of the columns propagate the carry:

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

- We compute the carry out of the 4-bit block (C_i) as:

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

32-bit CLA with 4-bit blocks



Carry-Lookahead Adder Delay

- The delay of an *N*-bit carry-lookahead adder with *k*-bit blocks is:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$

  where
  - t_{pg} is the delay of the column generate and propagate gates
  - t_{pg_block} is the delay of the block generate and propagate gates
  - $t_{\text{AND\_OR}}$ is the delay from C_{in} to C_{out} of the final AND/OR gate in the *k*-bit CLA block
- The delay of an *N*-bit carry-lookahead adder is generally much faster than a ripple-carry adder for *N* > 16

Prefix Adder

- Computes generate and propagate signals for all of the columns to perform addition even faster.
- Computes *G* and *P* for 2-bit blocks, then 4-bit blocks, then 8-bit blocks, etc. until the generate and propagate signals are known for each column.
- Thus, the prefix adder has $\log_2 N$ stages.
- The strategy is to compute the carry in (C_{i-1}) for each of the columns as fast as possible and then to compute the sum:

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

Prefix Adder

- A carry is generated by being either generated in a column or propagated from a previous column.
- Define column -1 to hold C_{in}, so

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

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

  because there will be a carry out of column i-1 if the block spanning columns i-1 through -1 generates a carry.
- Thus, we can rewrite the sum equation as:

  $S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} = P_i \oplus G_{i-1:-1}$
- Goal:
  - Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, …
  - These are called the *prefixes*

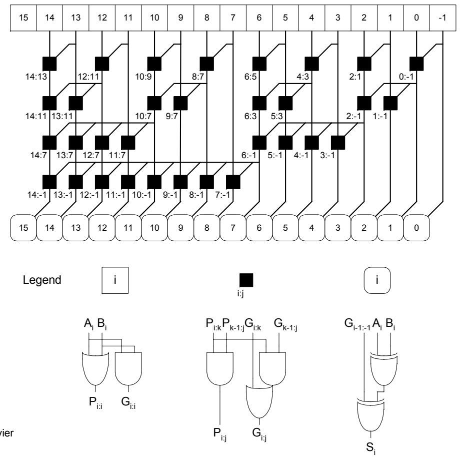
Prefix Adder

- The generate and propagate signals for a block spanning bits i:j are:

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

  $P_{i:j} = P_{i:k}P_{k-1:j}$
- In words, these prefixes describe that:
  - A block will generate a carry if the upper part (i:k) generates a carry or of the upper part propagates a carry generated in the lower part (k-1:j)
  - A block will propagate a carry if both the upper and lower parts propagate the carry.

Prefix Adder Schematic



Prefix Adder Delay

- The delay of an N-bit prefix adder is:

  $t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$

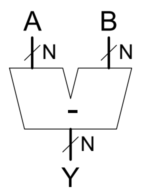
  where
  - t_{pg} is the delay of the column generate and propagate gates
  - t_{pg_prefix} is the delay of the black prefix cell (AND-OR gate)

Adder Delay Comparisons

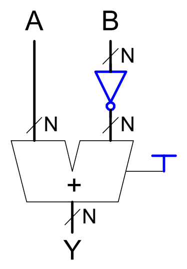
- Compare the delay of 32-bit ripple-carry, carry-lookahead, and prefix adders. The carry-lookahead adder has 4-bit blocks. Assume that each two-input gate delay is 100 ps and the full adder delay is 300 ps.

Subtracter

Symbol

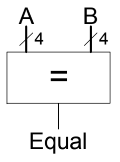


Implementation

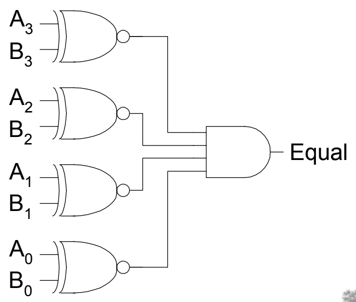


Comparator: Equality

Symbol

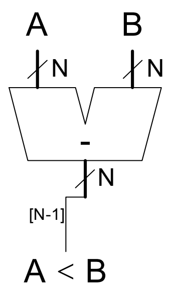


Implementation

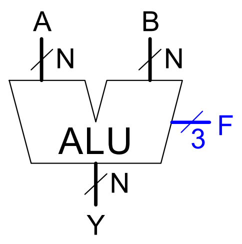


Comparator: Less Than

- For unsigned numbers

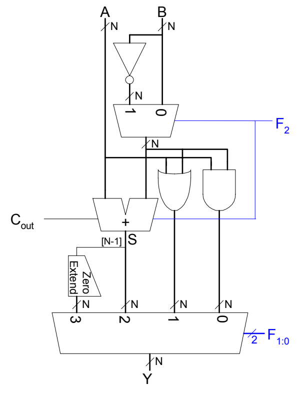


Arithmetic Logic Unit (ALU)



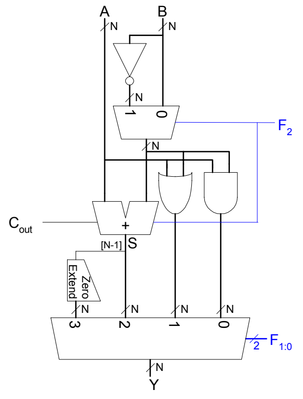
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & B
101	A | ~B
110	A - B
111	SLT

ALU Design



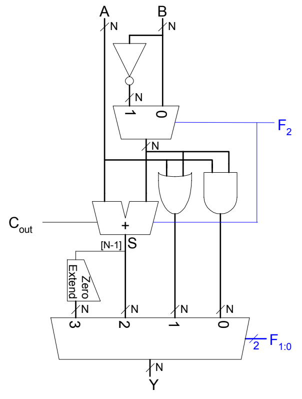
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose A = 25 and B = 32.

Set Less Than (SLT) Example

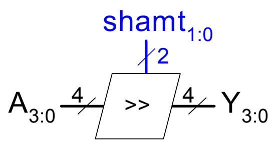


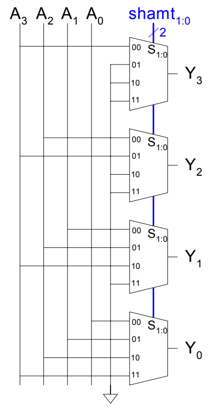
- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose A = 25 and B = 32.
  - Because A is indeed less than B, we expect Y to be the 32-bit representation of 1 (0x00000001).
  - For SLT, $F_{2:0}$ = 111.
  - F_2 = 1 configures the adder unit as a subtracter. So 25 - 32 = -7.
  - The two's complement representation of -7 has a 1 in the most significant bit, so S_{31} = 1.
  - With $F_{1:0}$ = 11, the final multiplexer selects $Y = S_{31}$ = 1.

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 =

Shifter Design





Shifters as Multipliers and Dividers

- A left shift by N bits multiplies a number by 2^N
  - Ex: 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - Ex: 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)
- The arithmetic right shift by N divides a number by 2^N
  - Ex: 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - Ex: 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

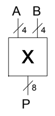
Multipliers

- Steps of multiplication for both decimal and binary numbers:
  - Partial products are formed by multiplying a single digit of the multiplier with the entire multiplicand
  - Shifted partial products are summed to form the result

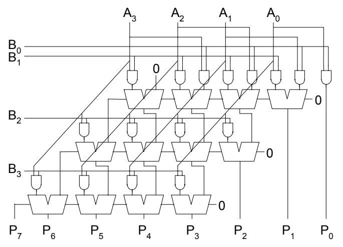
Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

230 x 42 = 9660   5 x 7 = 35

4 x 4 Multiplier



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

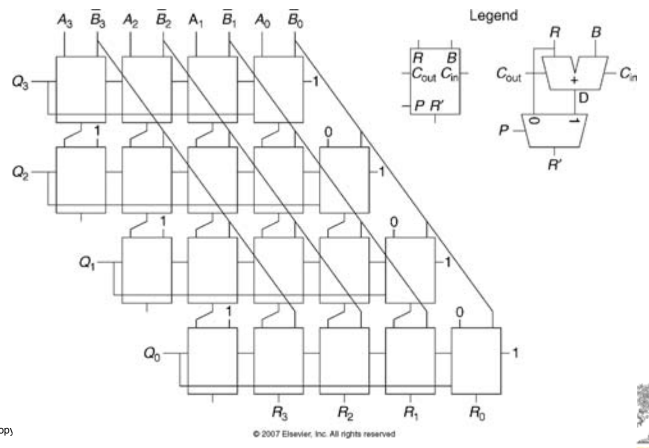


Division Algorithm

- $Q = A/B$
- R: remainder
- D: difference

```
$R = A$
for $i = N$-1 to 0
    $D = R - B$
    if $D < 0$ then $Q_i = 0$, $R' = R$        // $R < B$
    else            $Q_i = 1$, $R' = D$        // $R \geq B$
    $R = 2R'$
```

4 x 4 Divider



Number Systems

- What kind of numbers do you know how to 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:
    the binary point is fixed
  - Floating-point:
    the binary point floats to the right of the most significant 1

Fixed-Point Numbers

- Fixed-point representation of 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

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

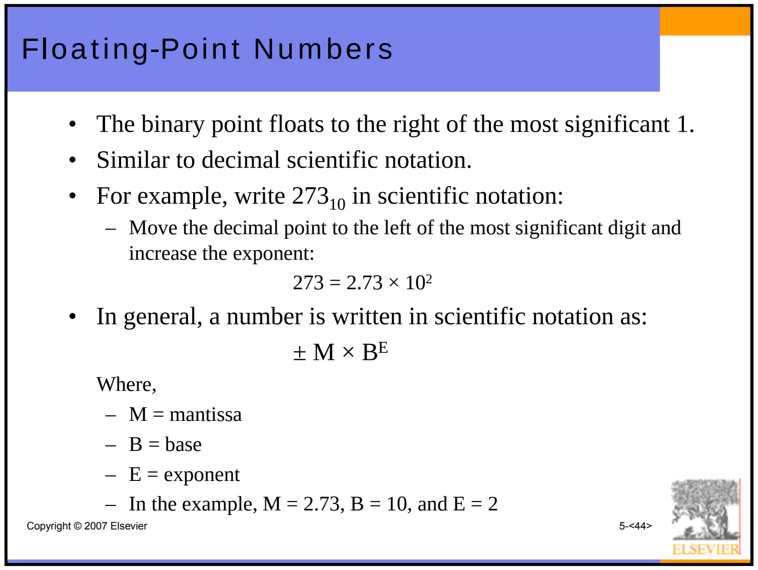
- The binary point is not a part of the representation but is implied.
- The number of integer and fraction bits must be agreed upon by those generating and those reading the number.

Fixed-Point Numbers

- Ex: Represent 6.5_{10} using an 8-bit binary representation with 4 integer bits and 4 fraction bits.

Signed Fixed-Point Numbers

- As with integers, negative fractional numbers can be represented two ways:
  - Sign/magnitude notation
  - Two's complement notation
- Represent -6.5_{10} using an 8-bit binary representation with 4 integer bits and 4 fraction bits.
  - Sign/magnitude:
  - Two's complement:
    1. +6.5:
    2. Invert bits:
    3. Add 1 ulp: ___________



Floating-Point Numbers

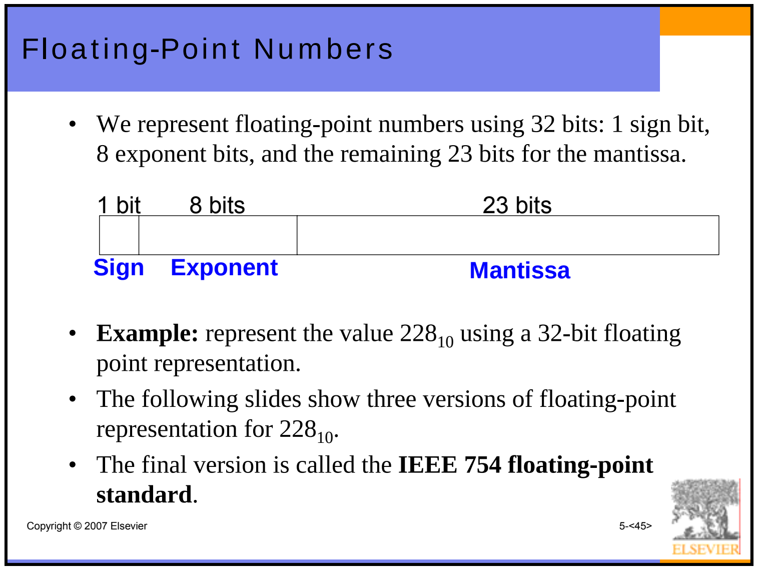
- The binary point floats to the right of the most significant 1.
- Similar to decimal scientific notation.
- For example, write 273_{10} in scientific notation:
  - Move the decimal point to the left of the most significant digit and increase the exponent:

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

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

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

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

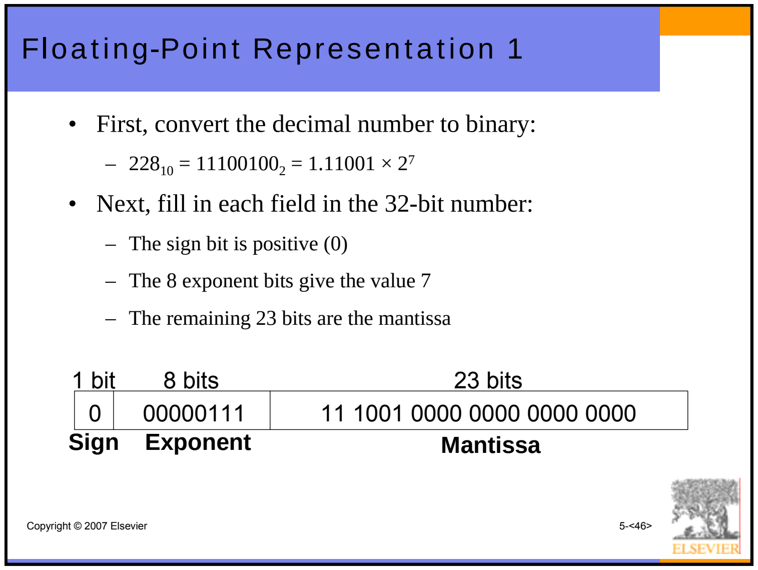


Floating-Point Numbers

- We represent floating-point numbers using 32 bits: 1 sign bit, 8 exponent bits, and the remaining 23 bits for the mantissa.

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa

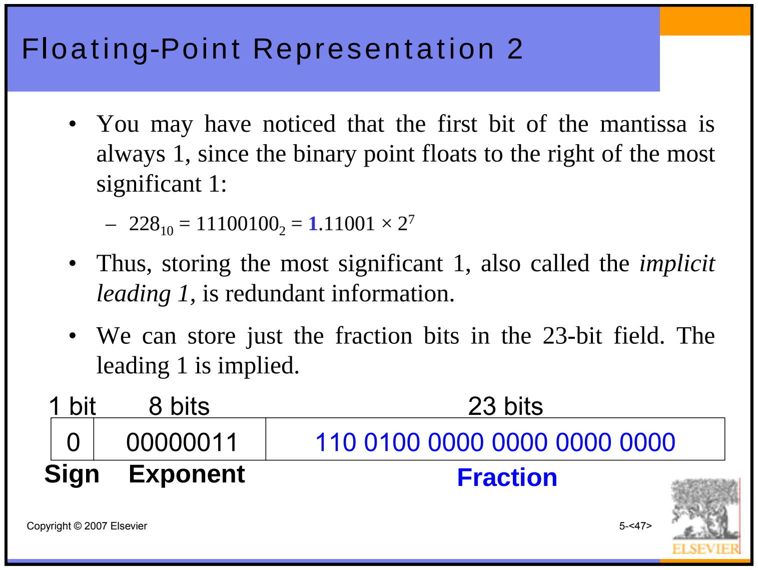
- **Example:** represent the value 228_{10} using a 32-bit floating point representation.
- The following slides show three versions of floating-point representation for 228_{10}.
- The final version is called the **IEEE 754 floating-point standard**.



Floating-Point Representation 1

- First, convert the decimal number to binary:
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Next, fill in each field in the 32-bit number:
  - The sign bit is positive (0)
  - The 8 exponent bits give 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

- You may have noticed that the first bit of the mantissa is always 1, since the binary point floats to the right of the most significant 1:
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Thus, storing the most significant 1, also called the *implicit leading 1*, is redundant information.
- We can store just the fraction bits in the 23-bit field. The leading 1 is implied.

1 bit	8 bits	23 bits
0	00000011	110 0100 0000 0000 0000 0000
Sign	Exponent	Fraction

Floating-Point Representation 3

- The final change is to store a *biased exponent.* The IEEE 754 standard uses a bias of 127.
  - Biased exponent = bias + exponent
  - For example, an exponent of 7 would be stored as:

    $127 + 7 = 134 = 0x10000110_2$
- Thus, the **IEEE 754 32-bit floating-point representation** of 228_{10} is:

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

Floating-Point Example

- Write the value -58.25_{10} using the IEEE 754 32-bit floating-point standard.
- First, convert the decimal number to binary:
  - 58.25_{10} =
- Next, fill in each field in the 32-bit number:
  - The sign bit is
  - The 8 exponent bits
  - The remaining 23 bits are the fraction bits:

1 bit	8 bits	23 bits
Sign	**Exponent**	**Fraction**

- Written in hexadecimal, this 32-bit value is:

Floating-Point Numbers: Special Cases

- The IEEE 754 standard includes special cases for numbers that are difficult to represent, such as 0 because it lacks an implicit leading 1.

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

NaN is used for numbers that don’t exist, such as $\sqrt{-1}$ or log(-5).

Floating-Point Number Precision

- Single-Precision:
  - 32-bit notation
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127
- Double-Precision:
  - 64-bit notation
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

Floating-Point Numbers: Rounding

- Overflow: when the number is too large to be represented
- Underflow: when the number is too small to be represented
- Rounding modes:
  - Down
  - Up
  - Toward zero
  - To nearest
- Example: round 1.100101 (1.578125) so that it uses 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 (× 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 × 2^1 = 1.0011 × 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..

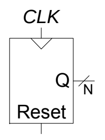
1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

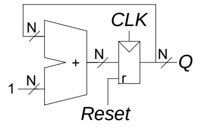
Written in hexadecimal: 0x40980000

Counters

- Increments on each clock edge.
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Counters are used in many digital systems, for example:
  - Digital clock displays
  - Program counter: used in computers to keep track of the current instruction that is executing

Symbol **Implementation**

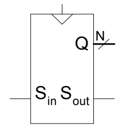


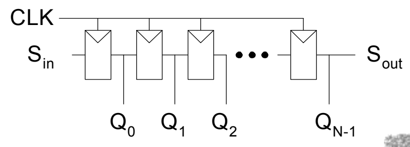


Shift Register

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

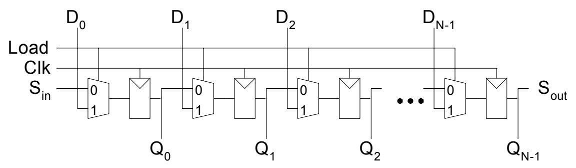
Symbol: Implementation:





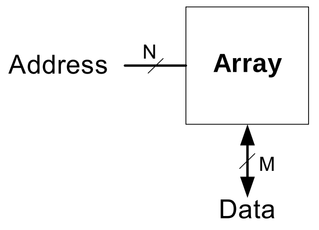
Shift Register with Parallel Load

- 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})



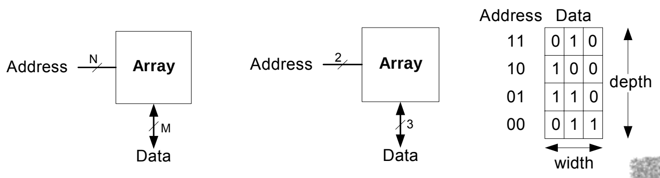
Memory Arrays

- Memory arrays efficiently store large amounts of data.
- Three common types of memory arrays:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- An *M*-bit data value can be read or written at each unique *N*-bit address.



Memory Arrays

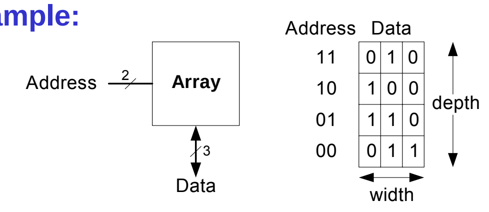
- Memory arrays are organized as a two-dimensional array of bit cells. Each bit cell stores one bit.
- An array with *N* address bits and *M* data bits has 2^N rows and *M* columns. Each row of data is called a word.
  - Depth: number of rows in a memory array
  - Width: number of columns in a memory array (the word size)
  - Array size is given as depth × width



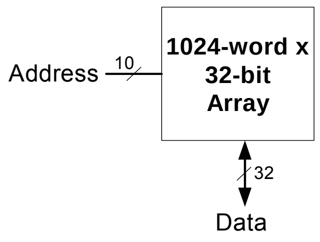
Memory Array: Example

- The memory array below is a $2^2 \times 3$-bit array.
- The word size is 3-bits.
- For example, the 3-bit word stored at address 10 is 100.

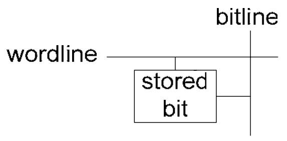
Example:



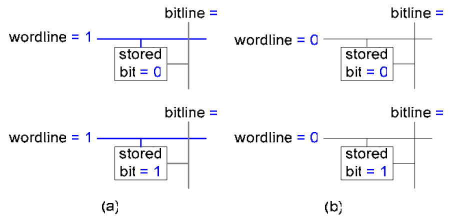
Memory Arrays



Memory Array Bit Cells

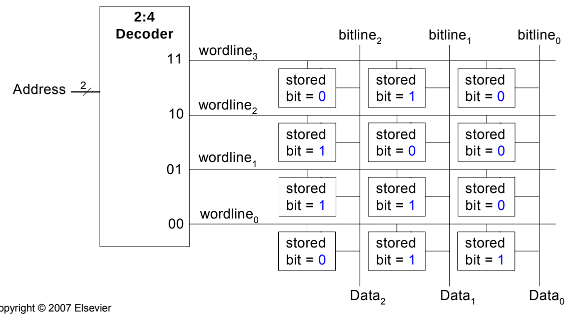


Example:



Memory Array

- The wordline, similar to an enable, allows a single row in the memory array to be read or written at once.
- Each wordline corresponds to a unique address – only one wordline is HIGH at any given time



Types of Memory

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

RAM

- Random access memory
  - Volatile: loses its data when the power is turned off
  - Can be read and written quickly
  - Main memory in your computer is RAM (specifically, DRAM)
  - Historically called *random* access memory because any data word can be accessed as easily as any other (in contrast to sequential access memories such as a tape recorder).

Types of Memory

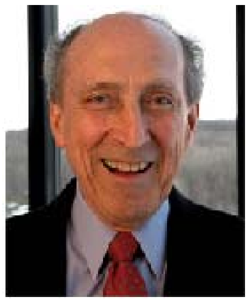
- Read only memory (ROM)
  - Nonvolatile: retains its data when power is turned off
  - Can be read quickly, but writing is impossible or slow
  - Flash memory in cameras, thumb drives, and digital cameras are all ROMs
  - Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash.

Types of RAM

- The two main types of RAM are:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
- They differ in how they store data:
  - DRAM uses a capacitor
  - SRAM uses cross-coupled inverters

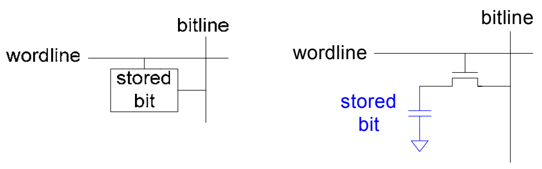
Robert Dennard, 1932 -

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

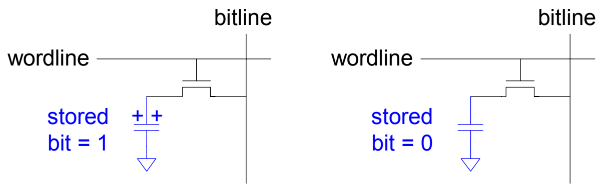


DRAM

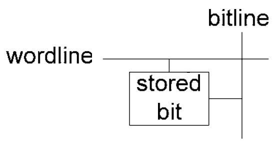
- Data bits are stored on a capacitor.
- DRAM is called *dynamic* because the value needs to be refreshed (rewritten) periodically and after being read because:
  - Charge leakage from the capacitor degrades the value
  - Reading destroys the stored value

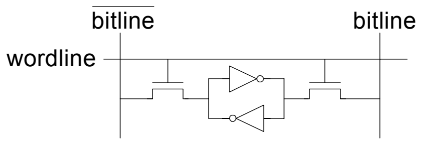


DRAM

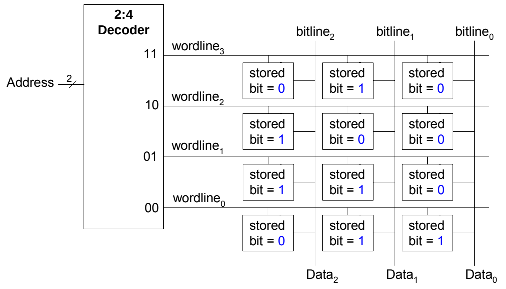


SRAM

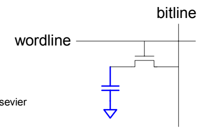




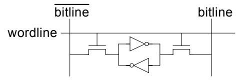
Memory Arrays



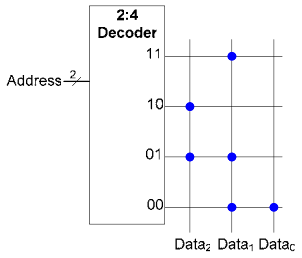
DRAM bit cell:

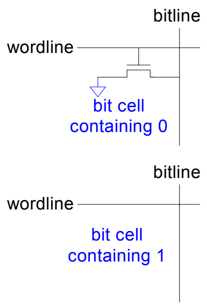


SRAM bit cell:



ROMs

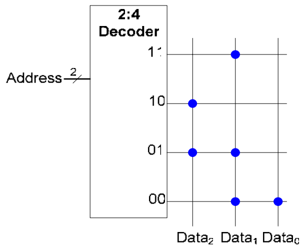




Fujio Masuoka, 1944-

- Developed memories and high speed circuits at Toshiba from 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

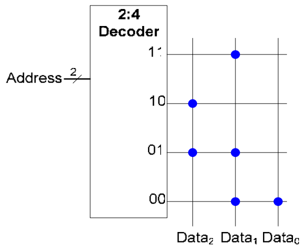


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

ROM Logic



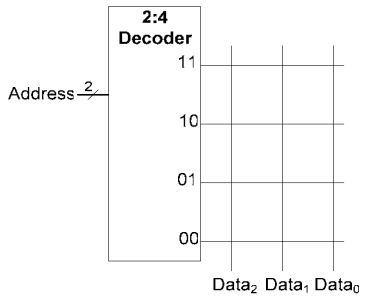
$$Data_2 = A_1 \oplus A_0$$

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

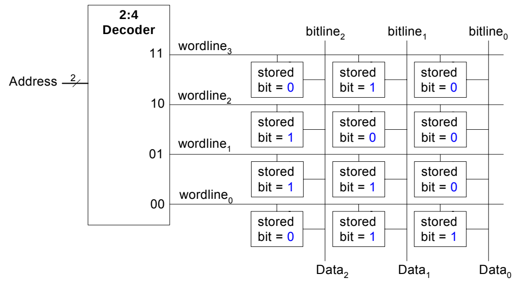
$$Data_0 = \overline{A_1}\,\overline{A_0}$$

Example: Logic with ROMs

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



Logic with Memory Arrays



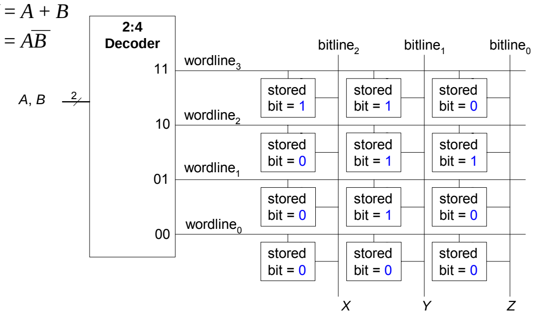
$$Data_2 = A_1 \oplus A_0$$

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

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

Logic with Memory Arrays

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

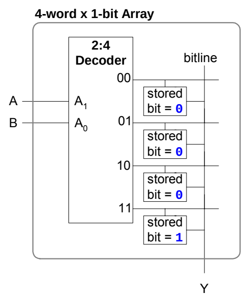


Logic with Memory Arrays

- Memory arrays used to perform logic are called *lookup tables* (LUTs).
- The user looks up the value of the output at each input combination (address).

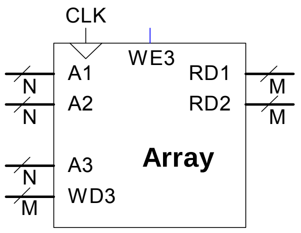
Truth Table

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)
- Small multi-ported memories are called *register files*



Verilog Memory Arrays

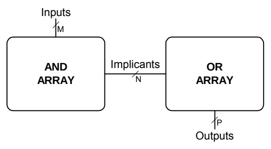
```
// 256 x 3 memory module with one read/write port
module dmem( input          clk, we,
             input  [7:0]   a
             input  [2:0]   wd,
             output [2:0]   rd);

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

  assign rd = RAM[a];

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

endmodule
```

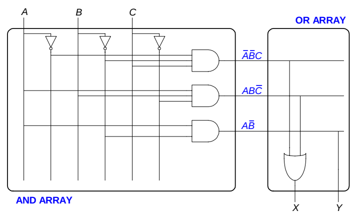
Logic Arrays

- Programmable logic arrays (PLAs)
  - AND array followed by OR array
  - Perform combinational logic only
  - Fixed internal connections
- Field programmable gate arrays (FPGAs)
  - Array of configurable logic blocks (CLBs)
  - Perform combinational and sequential logic
  - Programmable internal connections

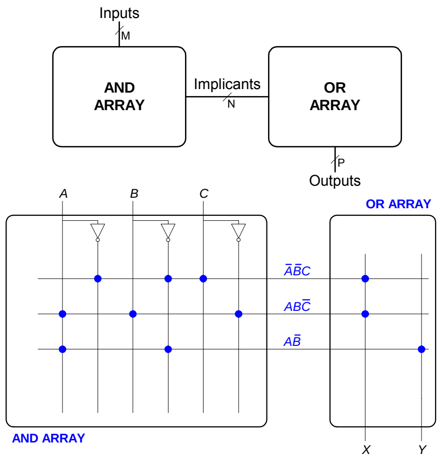
PLAs

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





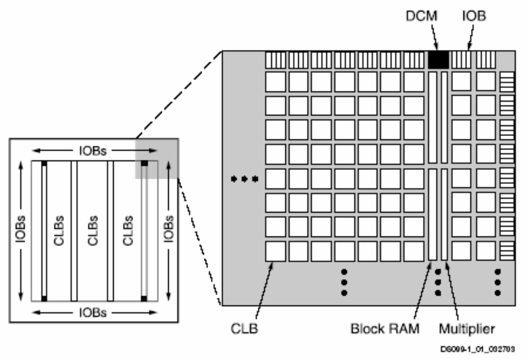
PLAs



FPGAs

- Composed of:
  - CLBs (Configurable logic blocks): to perform logic
  - IOBs (Input/output buffers): to interface with outside world
  - Programmable interconnection: to connect CLBs and IOBs
  - Some FPGAs include other building blocks such as multipliers and RAMs

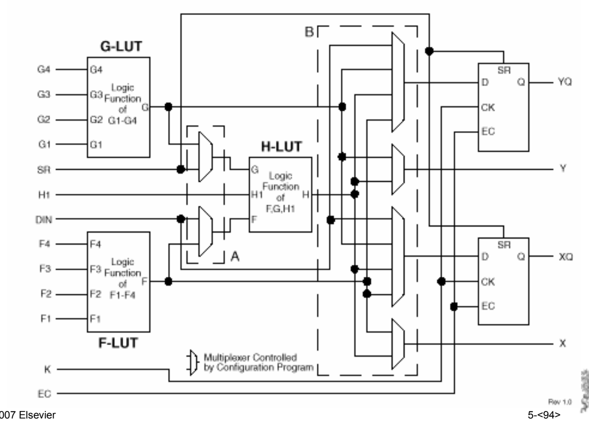
Xilinx Spartan 3 FPGA Schematic



CLBs

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

Xilinx Spartan CLB

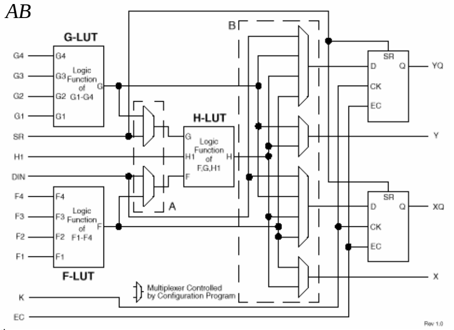


Xilinx Spartan CLB

- The Spartan CLB has:
  - 3 LUTs:
    - F-LUT (2^4 x 1-bit LUT)
    - G-LUT (2^4 x 1-bit LUT)
    - H-LUT (2^3 x 1-bit LUT)
  - 2 registered outputs:
    - *XQ*
    - *YQ*
  - 2 combinational outputs:
    - *X*
    - *Y*

CLB Configuration Example

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

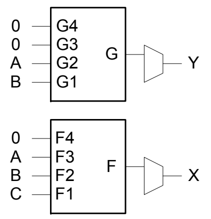


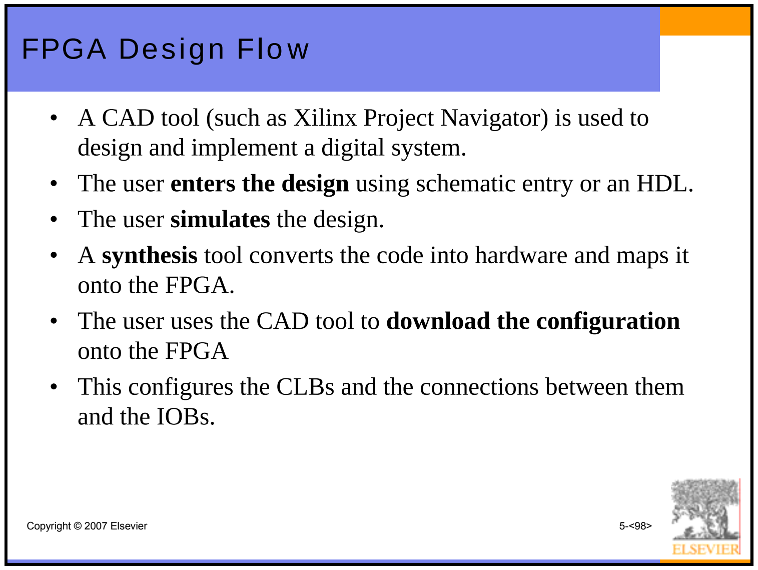
CLB Configuration Example

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

	(A)	(B)	(C)	(X)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
x	1	1	1	0

		(A)	(B)	(Y)
G4	G3	G2	G1	G
X	X	0	0	0
X	X	0	1	0
X	X	1	0	1
X	X	1	1	0





FPGA Design Flow

- A CAD tool (such as Xilinx Project Navigator) is used to design and implement a digital system.
- The user **enters the design** using schematic entry or an HDL.
- The user **simulates** the design.
- A **synthesis** tool converts the code into hardware and maps it onto the FPGA.
- The user uses the CAD tool to **download the configuration** onto the FPGA
- This configures the CLBs and the connections between them and the IOBs.