Fundamentals of Electrical and Electronic Engineering

400462

Digital Electronics

Dr Ian M. Bell

Numerical Data and Arithmetic Circuits

Representation of numbers in logic circuits

- Coding to represent numbers in general
- Representing negative numbers

Basic arithmetic operations very important in digital systems

- Addition
- Subtraction
- Multiplication
- Division

The other three can be derived from addition

• Subtraction	Addition of negative numbers
• Multiplication	Repeated addition
• Division	Repeated subtraction

Comparison operations are also very important

- Equals
- Greater than
- Less than

Why?
Numerical/mathematical operations are performed in wide range of digital systems from spreadsheets and graphics on computers to robotic control systems and digital signal processing of audio and video

Numerical Data and Arithmetic Circuits

Representation of numbers in logic circuits

- Coding to represent numbers in general
- Representing negative numbers

Basic arithmetic operations very important in digital systems

- Addition
- Subtraction
- Multiplication
- Division

The other three can be derived from addition

- Subtraction — Addition of negative numbers
- Multiplication — Repeated addition
- Division — Repeated subtraction

Comparison operations are also very important

- Equals
- Greater than
- Less than

Numbers in Digital Systems

- Digital systems represent / analyse information in discrete form
  - “1s and 0s” not analogue voltages
- A collection of *N* binary logic signals (on wires or in registers) together can represent 2^N different codes
- A code a simply a particular pattern of 1s and 0s (e.g. 0110 or 1110)
  - We can define which code represents which numerical value
- 2^N codes can represent (up to) 2^N possible numbers (numerical values)
- A straightforward and commonly used code for numbers is to use the *N* logic signals to represent the digits in an *N*-digit unsigned base-2 integer
  - Often called “Binary”
  - Truth table rows often in binary order
- Binary is not the only coding system for integers in use (e.g. Gray Code and BCD are an alternatives)

Binary Numbers

Number in base 2:

$$N_2 = d_n\, 2^n + d_{n-1}\, 2^{n-1} + \ldots\, d_1\, 2^1 + d_0\, 2^0 + d_{-1}\, 2^{-1} + d_{-2}\, 2^{-2} + \ldots + d_{-m}\, 2^{-m}$$

d_n may only be 0 or 1 and is a ***bi***nary digi***t*** or **bit**

$1_{10} = 0001_2$ $\quad 3_{10} = 0011_2$ $\quad 8_{10} = 1000_2$ $\quad 0.5_{10} = 0.1_2$

LSB – least significant bit

MSB – most significant bit

Positive Powers of Two (whole numbers)									Negative Powers of Two (fractional number)					
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
256	128	64	32	16	8	4	2	1	1/2 0.5	1/4 0.25	1/8 0.125	1/16 0.0625	1/32 0.03125	1/64 0.015625

Examples

What is the binary representation of decimal **98** using 8-bits?

What is $\mathbf{10110100_2}$ in decimal?

Various Codes for Numbers 0 to 15

Decimal	Binary	BCD	Gray Code
0	0000	0000 0000	0000
1	0001	0000 0001	0001
2	0010	0000 0010	0011
3	0011	0000 0011	0010
4	0100	0000 0100	0110
5	0101	0000 0101	0111
6	0110	0000 0110	0101
7	0111	0000 0111	0100
8	1000	0000 1000	1100
9	1001	0000 1001	1101
10	1010	0001 0000	1111
11	1011	0001 0001	1110
12	1100	0001 0010	1010
13	1101	0001 0011	1011
14	1110	0001 0100	1001
15	1111	0001 0101	1000

Numerical Coding

Binary

- Code follows base-2 unsigned integers
- Leads to efficient circuits for data processing

BCD – Binary Coded Decimal

- Each four-bit group represents a decimal digit using binary code
- Generally less efficient circuitry than binary
- Suitable for relatively straightforward processing (e.g. counting) particularly when data is to be displayed directly in decimal (e.g. on seven segment display)

Gray Code

- Only one bit changes between successive values
- Used in special circumstances to avoid errors as number values change
- Output from mechanical position sensors
- Avoiding errors due to timing differences along paths taken by each bit
- Karnaugh map rows and columns are in gray code order

Numbers in Digital Systems

Use of unsigned binary integers does not cover

- **Negative values**
  - Use 2's complement code – discussed later
- **Fractional values**
  - But we can expand the digits beyond the binary point
- **Very large and very small values**
  - Would required too many bits as an integer or fraction,
  - Such numbers have to be coded in a floating point format
  - e.g. 6.626×10^{-34} (decimal floating point)

Numbers in Digital Systems

Binary not very convenient for humans to use (or paper or on screen)

Binary numbers often written using base-8 (octal) or base-16 (hex)

Octal – uses groups of 3 bits from binary word
– mainly historical, not widely used now

Hexadecimal – uses groups of 4 bits from binary word
– widely used

Number Base (radix)	*Available Digits*
2 (binary)	0,1
8 (octal)	0,1,2,3,4,5,6,7
10 (decimal)	0,1,2,3,4,5,6,7,8,9
16 (hexadecimal)	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

- Any multi-bit digital code / data can be written in octal or hex for convenience
- It does not have to represent a number (e.g. it could be processor instruction codes, text or other data)

Numbers 0 to 15 in Number Bases 10, 2, 8 and 16

Decimal	Binary	Octal	Hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Example of binary data written using hexadecimal

Binary	0100001011011010
Groups of 4 bits	0100 0010 1101 1010
Hex	42DA

Example 2

Binary 1110010110101001

Hex

Numerical Data and Arithmetic Circuits

Representation of numbers in logic circuits

- Coding to represent numbers in general
- Representing negative numbers

Basic arithmetic operations very important in digital systems

- Addition
- Subtraction
- Multiplication
- Division

The other three can be derived from addition

• Subtraction	Addition of negative numbers
• Multiplication	Repeated addition
• Division	Repeated subtraction

Comparison operations are also very important

- Equals
- Greater than
- Less than

Adder Circuits

Adder circuits can be based on classic long addition

Here is an example of long addition in decimal

86 + 57 = 143

$$
\begin{array}{cccc}
 & & 8 & 6 \\
+ & & 5 & 7 \\
 & 1 & 1 & 0 \\
\hline
 & 1 & 4 & 3 \\
\hline
\end{array}
$$

6 + 7, the result is 13, so the first digit is 3 and we carry 1

8 + 5 + 1 (the carry), the result is 14, so the second digit is 4 and we carry 1

Adder Circuits

The same long addition process can be used with binary numbers

0110 + 0011 = 1001

Adder Circuits – Example

Perform the following long addition in binary

(written out as on the previous slide)

1010 + 1011

Adder Circuits

- An adder circuit can be based on binary long addition
- Each block in the circuit handles one digit
- Each block adds two 1-bit numbers and also has a carry in and carry out

0110 + 0011 = 1001

	0	1	1	0
+	0	0	1	1
0	1	1	0	0
0	1	0	0	1

1-bit Adders

XOR gate

- Adds two one-bit binary numbers and gives the "units" result
- No carry in
- No carry out

Half adder

- Adds two 1-bit binary numbers and gives a 2-bit binary result
- The 2s output usually referred to as the carry-out
- No carry-in

Full adder

- Adds two 1-bit binary numbers, plus carry in and gives a 2-bit binary result
- The 2s output is usually referred to as the carry-out

To add multi-bit numbers we need carry-in and carry-out so full adders are required

1-bit Adders

Truth tables for a basic 1-bit adder S = A + B

A and B single bit, S is two bits (S_1S_0)

Binary Addition Truth Tables

A_0	B_0	S_0
0	0	0
0	1	1
1	0	1
1	1	0

A_0	B_0	S_1
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Equations

$$S_0 = A_0 \oplus B_0 \qquad S_1 = A_0 B_0$$

1-bit Adders

1-bit half adder implemented using XOR and AND gate

This is a half adder….

…there is no carry in or out

… we need 2 to make a full adder

Full Adder

The full adder comprises two half adders and an XOR gate

The full adder adds two 1-bit binary numbers plus a carry in.

It produces 1-bit sum output and a carry output

- The full adder is the addition block in the “long addition” circuit shown earlier

4-Bit Adder

The adder circuit in more detail

- More stages can be added to add number with more bits
- The basic structure is just repeated *N* times for *N* bits

Ripple Carry Adder– Speed Problem

This circuit is called a **ripple carry adder** because the carry signal propagates ("ripples") through the adder from the least significant bit to the final carry out

In a large adder this long signal path had a large delay, limiting the speed of the adder

Carry Look-Ahead (CLA)

- Aims to reduce the total carry delay to produce a faster adder
- The final carry for an adder section (e.g. 4 bits) is produced by a separate CLA circuit
- The carry from the (say) 4th bit will occur much earlier than in a standard adder
- When CLA stages are linked together in large adders the speed up is very significant

4-bit binary full adder

HEF4008B
MSI

DESCRIPTION

The HEF4008B is a 4-bit binary full adder with two 4-bit data inputs (A_0 to A_3, B_0 to B_3), a carry input (C_{IN}), four sum outputs (S_0 to S_3), and a carry output (C_{OUT}). The IC uses full look-ahead across 4-bits to generate C_{OUT}. This minimizes the necessity for extensive look-ahead and carry-cascading circuits.

Fig.1 Functional diagram.

Fig.2 Pinning diagram.

PINNING

A_0 to A_3	data inputs
B_0 to B_3	data inputs
S_0 to S_3	sum outputs
C_{IN}	carry input
C_{OUT}	carry output

TRUTH TABLE (one adder)

C_{IN}	A	B	C_{OUT}	S
L	L	L	L	L
L	L	H	L	H
L	H	L	L	H
L	H	H	H	L
H	L	L	L	H
H	L	H	H	L
H	H	L	H	L
H	H	H	H	H

If all gates have a delay of 2 ns what are the worse case delays for the ripple carry out and look ahead carry out?

You will need to complete the circuit for the ripple carry out do to this

Adders and Subtractors

We have adder circuits…

But how to perform subtraction?

This raises the issue of how we represent negative numbers in arithmetic circuits.

Numerical Data and Arithmetic Circuits

Representation of numbers in logic circuits

- Coding to represent numbers in general
- Representing negative numbers

Basic arithmetic operations very important in digital systems

- Addition
- Subtraction
- Multiplication
- Division

The other three can be derived from addition

• Subtraction	Addition of negative numbers
• Multiplication	Repeated addition
• Division	Repeated subtraction

Comparison operations are also very important

- Equals
- Greater than
- Less than

Negative Numbers – Sign and Magnitude

- We usually write numbers as sign and magnitude

  `+3  -8  +72  -109` etc

- We can use a sign bit in a similar way for binary numbers in circuits, e.g. as sign bit plus 8 magnitude bits:

```
0 0000 0011
1 0000 1000
0 0101 1000
1 0110 1101
```

- Unfortunately this approach does not lead to straightforward design of arithmetic circuits
- Another approach is needed

Modulo Arithmetic

Example – modulo 5

```
1 + 1 = 2

4 + 1 = 0

3 + 3 = 1

3 – 3 = 0

3 – 4 = 4
```

Note that 5 and 0 are congruent in modulo 5

In general for module *n* the largest number is *n*-1 but in some cases *n* is used, in particuar in stating time of day where for example 12:00 is used instead of 00:00 in the 12-hour clock system (where hours are modulo 12)

Modulo arithmetic is relevant to digital circuits because numerical values are usually limited to a fixed number of bits

4 bit binary data coded as unsigned integers

One code for 0

One abrupt transition: 15 to 0 (at the extreme values)

4 bit binary data coded as sign-magnitude signed integers

Two codes for zero +0 and -0

Two abrupt transactions: -7 to +0 and 7 to -0

Codes do not go smoothly through 0

Cannot add/subtract N by moving forwards/backwards N steps for all in-range values

4 bit binary data coded as 2s compliment signed integers

One code for 0

One abrupt transition: 8 to -7 (at the extreme values – like unsigned)

Codes go smoothly through 0 (-1 to 0 to +1)

Can add/subtract N by moving forwards/backwards N steps for all in-range values (like unsigned)

Negative Numbers - Complements

- Subtraction performed by adding a number's complement
- There are various forms of complement
- The **Radix Complement** is found by taking the next power of the number base (radix) and subtracting the number

Example using base 10, 10's Complement

-45 in base 10, two digit numbers, sign-magnitude format

Next power is 100 (three digits)

100 – 45 is 55

55 is the 10's complement of -45

Negative Numbers – Complements

Example using base 2, 2's Complement

-0110 in binary, 4 digit (bit) sign-magnitude numbers

Next power of two is 10000 (5 bits)

10000 – 0110 is 1010 *(16 – 6 = 10)*

1010 is the 4-bit 2's complement of -0110

- obtain the 2's complement we can
- **Invert all the bits and add 1** e.g. 0110➔1001➔1010
- This is easy to implement as a logic circuit

Subtraction Using Radix Complements

Example in base 10 (10's complement, 2 digits)

47 – 23

complement of 23 is 100 – 23 ➔ 77

so –23 is represented by 77

47 – 23

47 + (-23)

47 + 77 = 124

Use modulo 100 so 124 ➔ 24

Result is 24

This 2 digit 10's complement number is positive so result in sign-magnitude format is the same (24).

Subtraction Using Radix Complements

Example in base 10 (10's complement, 2 digits)

25 – 32

complement of 32 is 100 – 32 ➔ 68

so –32 is represented by 68

25 – 32

25 + (-32)

25 + 68 = 93

Result is 93

93 represents a negative number in 2-digit 10's complement

To convert to sign-magnitude: 100 – 93 ➔ 7

Result in sign-magnitude format is –7