Number Systems

- binary, octal, and hexadecimal numbers
  - why used
- conversions, including to/from decimal
- negative binary numbers
- floating point numbers
- character codes

Radix Number Systems

- basic idea of a radix number system --
  how do we count:
- **decimal**: 10 distinct symbols, 0-9
  - when we reach 9, we repeat 0-9 with 1 in front:
    0,1,2,3,4,5,6,7,8,9, 10,11,12,13, …, 19
  - then with a 2 in front, etc: 20, 21, 22, 23, …, 29
  - until we reach 99
  - then we repeat everything with a 1 in the next position:
    100, 101, …, 109, 110, …, 119, …, 199, 200, …
- other number systems follow the same principle with a different base (radix)

Radix Number Systems

- **octal**: we have only 8 distinct symbols: 0-7
  - when we reach 7, we repeat 0-7 with 1 in front
  - 0,1,2,3,4,5,6,7, 10,11,12,13, …, 17
  - then with a 2 in front, etc: 20, 21, 22, 23, …, 27
  - until we reach **77**
  - then we repeat everything with a 1 in the next position:
  - 100, 101, …, 107, 110, …, 117, …, 177, 200, …
- decimal to octal correspondence:

D	0	1	…	7	8	9	10	11	…	15	16	17	…	23	24	…	63	64	…
O	0	1	…	7	10	11	12	13	…	17	20	21	…	27	30	…	77	100	…

Radix Number Systems

- **binary**: we have only 2 distinct symbols: 0, 1
- why?: digital computer can only represent 0 and 1
  - when we reach 1, we repeat 0-1 with 1 in front
  - 0,1, 10,11
  - then we repeat everything with a 1 in the next position:
  - 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, …
- decimal to binary correspondence:

D	0	1	2	3	4	5	6	7	8	9	10	11	12	…
B	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	…

Radix Number Systems

- **hexadecimal**: we have 16 distinct symbols: 0-9,A-F
  - when we reach 9, we continue until F
  - then repeat with 1 in front
  - 0,1, …, 9,A,B,C,D,F, **1**0,**1**1, …, **1**9,**1**A,**1**B,**1**C,**1**D,**1**E,**1**F
  - then with a 2 in front, etc: **2**0, **2**1, …, **2**9, **2**A, …, **2**F
  - until we reach **FF**
  - then we repeat everything with a 1 in the next position:
  - **1**00, 101, …, 109, 10A, …, 10F, 110, …, 11F, …, 1FF, 200, …
- decimal to hexadecimal correspondence:

D	0	1	…	7	8	9	10	11	…	15	16	17	…	25	26	…	31	32	…
H	0	1	…	7	8	9	A	B	…	F	10	11	…	19	1A	…	1F	20	…

Radix Number Systems

- purpose of different systems
- binary:
  - digital **computer** can only represent 0 and 1
- octal:
  - better for **human** use
  - very easy to **convert** to/from binary
- hexadecimal
  - also very easy to convert to/from binary
  - slightly more difficult for humans, but
  - 1 hex digit = 4 binary digits (**power of 2**: 1 byte = 2 hex)

Number System Conversions

- convert **binary ↔ octal**
- 3 bits = 1 octal digit
- use conversion table:

O	0	1	2	3	4	5	6	7
B	000	001	010	011	100	101	110	111

- Example:

  $10111010000110_2 = 10\ 111\ 010\ 000\ 110_2 = 27206_8$
- It also works with a decimal point, e.g.:

  $50.3_8 = 101\ 000.011_2$

Number System Conversions

- convert **binary ↔ hex**
- 4 bits = 1 hex digit
- use conversion table:

H	0	1	2	3	4	5	6	7
B	0000	0001	0010	0011	0100	0101	0110	0111
H	8	9	A	B	C	D	E	F
B	1000	1001	1010	1011	1100	1101	1110	1111

- Example:

  $10111010000110_2 = 10\ 1110\ 1000\ 0110_2 = 2E86_{16}$
- It also works with a decimal point, e.g.:

  $50.3_{16} = 101\ 0000.0011_2$

Number System Conversions

- to/from **decimal**
- use basic principle of radix numbers:
  each digit corresponds to a **power of the radix**
- in decimal:
  $3205.3 = 3*1000 + 2*100 + 0*10 + 5*1 + 3*0.1$
  $= 3*10^3 + 2*10^2 + 0*10^1 + 5*10^0 + 3*10^{-1}$
  In general, a decimal number can be decomposed into:
  $\ldots d_3*10^3 + d_2*10^2 + d_1*10^1 + d_0*10^0 + d_{-1}*10^{-1} + d_{-2}*10^{-2}\ldots$

Lubomir Bic 9

Number System Conversions

- a binary number can be decomposed into:
  $\ldots +d_3*2^3 + d_2*2^2 + d_1*2^1 + d_0*2^0 + d_{-1}*2^{-1} + d_{-2}*2^{-2} + \ldots$
  $\ldots +d_3*8 + d_2*4 + d_1*2 + d_0*1 + d_{-1}*0.5 + d_{-2}*0.25 + \ldots$
- convert **binary → decimal**
- learn powers of 2:
  - 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, …
  - for each binary digit that is 1 add corresponding power
- Example:
  $110101_2 = 2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53_{10}$
  $1.01_2 = 2^0 + 2^{-2} = 1 + ¼ = 1.25_{10}$

Lubomir Bic 10

Number System Conversions

- **decimal → binary:**
- decompose number into powers of 2
  - How: repeatedly subtract the largest possible power
- for each component write a 1 in corresponding digit
- Example: convert 53_{10}
  $53 – 2^5 = 53 – 32 = 21$
  $21 – 2^4 = 21 – 16 = 5$
  $5 – 2^2 = 5 – 4 = 1$
  $1 – 2^0 = 1 – 1 = 0$
- the powers are: 5, 4, 2, 0
- the binary number is: 110101_2

Lubomir Bic 11

Number System Conversions

- **summary**
  - binary ↔ octal/hex:
    grouping of 3 or 4 bits
  - octal ↔ hex:
    via binary
  - binary → decimal:
    sum up powers of 2
  - decimal → binary:
    decompose into powers of 2
  - octal/hex ↔ decimal:
    use powers of 8 or 16
    easier: via binary

Lubomir Bic 12

Negative Binary Numbers

- **signed magnitude**:
  - left-most bit is a **sign bit**
  - Example with 8 bits:
    5 = 0000 0101, –5 = 1000 0101
- **one's** complement:
  - **flip all** bits (0 becomes 1, 1 becomes 0)
  - Example: 5 = 0000 0101, –5 = **1111 1010**
  - Advantage: **subtract by adding negative number**
  - Example: 12 – 5 = 12+(–5)
    - add left-most carry to sum
  - Problem: 2 forms of zero:
    - 0000 0000 = 1111 1111

12	0000 1100
–5	+ 1111 1010
carry	1 1111 000
	0000 0110
7	0000 0111

Negative Binary Numbers

- **two's** complement:
  - one's complement + 1
  - Example: 5 = 0000 0101, –5 = 1111 1010**+1** = 1111 1011
  - add/subtract as before, but left-most carry is **discarded**

12	0000 1100
–5	+ 1111 1011
carry	1 1111 000
7	0000 0111

- drawback of two's complement: **extra** negative number
  - with 8 bits we have 2^8 = 256 possible combinations:
  - 2^7 = 128 combinations have left-most bit = 0:
    represent **zero** and **127 positive** integers
  - 2^7 = 128 combinations have left-most bit = 1:
    represent **128 negative** integers

Negative Binary Numbers

decimal	binary	decimal	1's complement	2's complement
0	0000 0000	–0	1111 1111	0000 0000
1	0000 0001	–1	1111 1110	1111 1111
2	0000 0010	–2	1111 1101	1111 1110
3	0000 0011	–3	1111 1100	1111 1101
…	…		…	…
126	0111 1110	–126	1000 0001	1000 0010
127	0111 1111	–127	1000 0000	1000 0001
128	nonexistent	–128	nonexistent	1000 0000

- eliminate –0 by shifting range down
  - +0 = –0 = 0000 0000
- –128 exists only in negative range

Finite-Precision Numbers

- number of bits determines the max/min number
- Problem:
  - assume 4 bits
  - 4 + 5 = 9

4	0100
5	0101
carry	100
?	1001

  - 1001 = **-7** → wrong result!
    9 is too large for 4 bits (largest positive integer: 0111=7)
  - **overflow – must compare sign bits to detect:**
    A and B have the **same** sign and A+B has a **different** sign

Negative Binary Numbers

- **excess N** representation
- basic idea: add N to every number ($N=2^{m-1}$ or $2^{m-1}-1$)
- Example: 8 bits
  - can represent $2^8 = 256$ different combinations (0-255)
  - **add 2^7 = 128 (or 127)** to every number to be represented

number	-128	-127	...	-1	0	1	...	127	128
excess 128 representation	0	1	...	127	128	129	...	255	
excess 127 representation		0	...	126	127	128	...	254	255

- used in floating point formats

Floating-Point Numbers

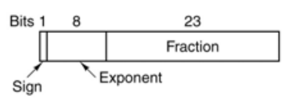
- needed for **very large** or **very small** numbers
- express numbers as $n = f * 10^e$
  - Example: $+213.1 = +2.131 * 10^2 = +0.2131 * 10^3$
  - two forms of **normalized** notation
- represent number as:

sign bit	exponent e	fraction f

  - **base** is implicit, normally 2
  - **fraction** (mantissa) is normalized; common choices:
    - 0.1 ...
    - 1. ...
  - **sign** bit applies to fraction (signed magnitude)
  - **exponent**: signed integer (2's complement, excess 2^{m-1})

Floating-Point Numbers

- exponent size vs. fraction size:
  max/min **size** of expressible numbers vs. **accuracy**
- Example: IEEE Standard
  - single precision



  - double precision

(a)

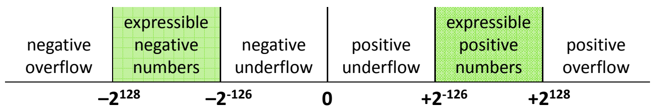
(b)

Floating-Point Numbers

- single precision standard: 32 bit word
  - 1 sign bit (0 is positive, 1 is negative)
  - 8 bits in exponent
    - **excess 127** system used: exponent ranges from -126 to +127
  - 23 bits in fraction
    - normalized to 1. ...
    - leading 1 is always present and thus **implied** (not represented)
    - I.e.: precision is **24** bits
  - **max. positive** number:
    - fraction: $1.111\ 1111 \ldots 1111 = 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + \ldots + 2^{-23}$
      $= 1 + 1/2 + 1/4 + 1/8 + \ldots$
      $= \sim 2$
    - $\max = +2 * 2^{127} = +2^{128}$
  - **min. negative** number: -2^{128}

Floating-Point Numbers

- **min. positive** number:
  - smallest fraction: 1.000 … 0000 = 2^0 = 1
  - smallest exponent: -126
  - min pos number = 1 * 2^{-126} = 2^{-126}
- **max. negative** number: -2^{-126}



Floating-Point Numbers

- **Examples: show 0.5 as floating point** (hex bit pattern)
  - change to binary and normalize: $0.5_{10} = 0.1_2 = 1.0 * 2^{-1}$
  - leading 1 is implied, fraction: $000\ 0000 \ldots 0000_2$
  - exponent: -1 in decimal is -1+127 in excess 127, i.e. 126
  - 126 = 64 + 32 + 16 + 8 + 4 + 2 = $0111\ 1110_2$

1	8 bits	23 bits
0	0111 1110	000 0000 0000 0000 0000 0000

  - 0011 1111 0000 0000 … 0000_2 = $3F000000_{16}$
- **show 42E48000 in decimal**
  - $42E48000_{16}$ = 0100 0010 1110 0100 1000 0000 0000 0000_2
  - exponent = 1000 0101_2 = 128 + 4 + 1 = 133_{10}
  - 133 is in excess 127 notation; in decimal: 133 – 127 = 6
  - mantissa: 1.11001001_2
  - result = $1.11001001_2 * 2^6$ = 1110 010.01_2 = 114.25_{10}

Representing Characters

- ASCII:
  American Standard Code for Information Interchange
  - 7-bit code: 128 characters
  - Examples
  - do not confuse chars with numbers, e.g.
    - 6: 0011 0110 (char)
    - 6: 0000 … 0000 0110 (int)
- UNICODE
  - 16-bits: 65,536 chars (symbols)
  - cover foreign languages

Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@
21	!	31	1	41	A
22	"	32	2	42	B
23	#	33	3	43	C
24	$	34	4	44	D
25	%	35	5	45	E
26	&	36	6	46	F
27	'	37	7	47	G
28	(	38	8	48	H
29	)	39	9	49	I
2A	*	3A	:	4A	J
2B	+	3B	;	4B	K
2C	,	3C	<	4C	L
2D	-	3D	=	4D	M