Number Systems and Logic

UCT Dept of Computer Science
CSC116 2005

Number Representations

- Numeric information is fundamental in computers – they encode data and instructions.
- Numerical data is stored efficiently for humans and computers.
  - Usually stored in computer formats and converted for humans!
- Notation: N_r for number N using radix r
  - radix refers to number of possible symbols for each digit.
- General radix *r* number representation:
    - $d_p d_{p-1} d_{p-2} \ldots d_2 d_1 d_0 . d_{-1} d_{-2} \ldots d_{-q}$
  - Numeric value is: $\Sigma_{i=-q \ldots p} \, d_i r^i$

Decimal Codes

- Common representation for humans.
- Radix = 10.
  - Possible values for digits: 0-9
- Example:

  $1041.2_{10} = 1*10^3 + 0*10^2 + 4*10^1 + 1*10^0 + 2*10^{-1}$
- n-digit decimal number: 0_{10} to $(10^n-1)_{10}$

Binary Codes

- Computers use presence/absence of voltage.
- Radix = 2.
  - Possible values for digits: 0 and 1
- Example:
  - $1001.1_2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} = 9.5_{10}$
  - $11.01 = 1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} = 3.25_{10}$
- n-bit binary number: 0_{10} to $(2^n-1)_{10}$
- Largest 8-bit (unsigned) number: $11111111_2 = 255_{10}$

Decimal to Binary Conversion

- Algorithm:

```
quot = number; i = 0;
repeat until quot = 0
    quot = quot/2;
    digit i = remainder;
    i++;
```

- Example:
  - Convert 37_{10} to binary.
  - Calculation:
    - 37/2 = 18 rem 1 least sig. digit
    - 18/2 = 9 rem 0
    - 9/2 = 4 rem 1
    - 4/2 = 2 rem 0
    - 2/2 = 1 rem 0
    - 1/2 = 0 rem 1 most sig. digit
  - Result:
    - $37_{10} = 100101_2$

Quick Decimal to Binary

- Algorithm:
  - Let i = largest power of 2 less than or equal to the number.
  - $N = N - 2^i$
  - Set digit i of result to 1
  - repeat until N = 0
- Example:
  - Convert 37_{10} to binary.
  - Calculation:
    - N >= 32, so digit 5 is 1 and N=5
    - N >= 4, so digit 2 is 1 and N=1
    - N >= 1, so digit 0 is 1 and N=0
  - Result:
    - $37_{10} = 100101_2$
- Note: Use this only to check your answers!

Converting Fractional Numbers

- Algorithm

```
i = 0
repeat until N == 1.0 or i == n
    N = FracPart(N);
    N *= 2;
    digit i = IntPart(N);
    i++
```

- Example
  - Convert 0.125_{10} to binary
  - Calculation:
    - 0.125*2 = 0.25; IntPart = 0 most significant digit
    - 0.250*2 = 0.50; IntPart = 0
    - 0.500*2 = 1.00; IntPart = 1 least significant digit
  - Result:
  - $0.125_{10} = 0.001_2$
- Convert integer and fractional parts separately.
- Many numbers cannot be represented accurately:
  - $0.3_{10} = 0.0[1001]..._2$ (bracket repeats, limited by bit size)

Binary Addition

- Adding binary numbers:
  - 1+0 = 0+1 = 1
  - 0 + 0 = 0
  - 1 + 1 = 0 carry 1
- Possibility of **overflow**

  Add 109_{10} to 136_{10}:

  $01101101_2 + 10001000_2 = 11110101_2 = 245_{10}$

  Add 254_{10} to 2_{10}:

  $11111110_2 + 00000010_2 = [1]00000000_2 = 256_{10}$
- We only have 8 bits to store answer...so it's zero!
- Program can generate an "exception"' to let us know.
- Usually number of bits is quite large: e.g., MIPS R4000 32-bits.

Signed Numbers

- Can use left-most bit to code sign.
  - 0=positive and 1=negative
  - Gives symmetric numbers from $-(2^7-1)\ldots 2^7-1$ AND two zeros!
  - Addition not straight forward (bad for hardware implementors).
- This is nonsensical and wasteful: can use extra bit pattern.
- Try *one's complement* representation*:*
  - negative numbers obtained by flipping signs.
  - positive numbers unchanged.
  - e.g. -5_{10} = complement(00000101_2) = 11111010_2
- Left-most bit still indicates sign.

1's Complement Addition

- Using 1's Complement, it is easier to subtract number: complement one number and add.
- Example
  - Calculate 5-6.
  - 5-6
    = 00000101 + complement(00000110)
    = 00000101 + 11111001
    = 11111110
    = complement(00000001)
    = -1_{10}
- A carry is added into right-most bit.
- Can still overflow: can check sign bits.
  - Only numbers with same sign can overflow.
  - Check: if input sign != output sign then overflow.

1's Complement Example

- Evaluate 10–7 using 8-bit one's complement arithmetic:

  10 - 7
  = 00001010 + complement(00000111)
  = 00001010 + 11111000
  = 00000010 carry 1
  = 00000010 + 00000001
  = 00000011
  = 3_{10}

2's Complement Addition

- 1's Complement still has two zeros.
- An alternative to address this is *two's complement*
  - Complement then add 1
  - Our number range now asymmetric: $-2^7 \ldots 2^7-1$
  - Used extra zero bit pattern
    - 2 = 0010, 1 = 0001, 0 = 0000, -1 = 1111, 2 = 1110
- Now when we add, discard carry

  10 - 7
  = 00001010 + 2complement(00000111)
  = 00001010 + 11111001
  = 00000011 carry 1 (discard)
  = 3
- Same overflow test can be used.

Binary Coded Decimal

- Can use Binary Coded Decimal (BCD) to represent integers.
  - Map 4 bits per digit (from 0000 to 1001)
- Wasteful: only 10 bit patterns required – 6 are wasted. Binary code is more compact code.
  - Example:
    - $256_{10} = 100000000_2 = 0010\ 0101\ 0110_{BCD}$
    - … 9 vs 12 bits in this case
- Not practical; complicates hardware implementation.
  - How to add/subtract, deal with carries etc?

Octal and Hexadecimal

- Base 8 (octal) and base 16 (Hexadecimal) are sometimes used (both are powers of 2).
- Octal (0NNN...N) uses digits 0-7
- Hex (0xNNN...N) uses "digits" 0-9,A-F
- Examples: $17_{10} = 10001_2 = 21_8 = 11_{16}$
- Conversion as for decimal to binary:
  - divide/multiply by 8 or 16 instead
- Binary to octal or hexadecimal:
  - group bits into 3 (octal) or 4 (hex) from LS bit.
  - pad with leading zeros if required.

Octal/Hexadecimal Conversion 1/2

- 0100011011010111_2
  - = (000) (100) (011) (011) (010) (111)
  - = 43327_8
  - = (0100) (0110) (1101) (0111)
  - = $46D7_{16}$
- Note padding at front of number.

Octal/Hexadecimal Conversion 2/2

- To convert from hex/octal to binary: reverse procedure
  - $FF_{16} = (1111)(1111)_2$
  - $377_8 = (011)(111)(111)_2$
- NOTE: for fractional conversion, move from left to right and pad at right end:
  - $0.11001101011_2 = 0.\ (110)\ (011)\ (010)\ (110)$
    $= 0.6326_8$
  - $0.11_2 = 0.(110)_2 = 0.6_8$
- Convert fractional/integer part separately.
- When converting to hex/oct it may be easier to convert to binary first.

Floating Point Numbers

- Fixed point numbers have very limited range (determined by bit length).
- 32-bit value can hold integers from -2^{31} to $2^{31}-1$ or smaller range of fixed point fractional values.
- Solution: use *floating point* (scientific notation)
  Thus $0.0000000000000976 \Rightarrow 9.76*10^{-14}$
- Consists of two parts: **mantissa** & **exponent**
  - **Mantissa:** the number multiplying the base
  - **Exponent:** the power
- The **significand** is the part of the mantissa after the decimal point

Floating Point Range

- Range of numbers is very large, but accuracy is limited by significand.
- So, for 8 digits of precision,
  $976375297321 = 9.7637529*10^{11}$,
  and we lose accuracy (**truncation error**)
- We can normalise any floating point number:
  $34.34*10^{12} = 3.434*10^{13}$
- Shift point until only one non-zero digit is to left.
  - add 1 to exponent for each left shift
  - subtract 1 for each right shift

Binary Floating Point

- We can use FP notation for binary: use base of 2

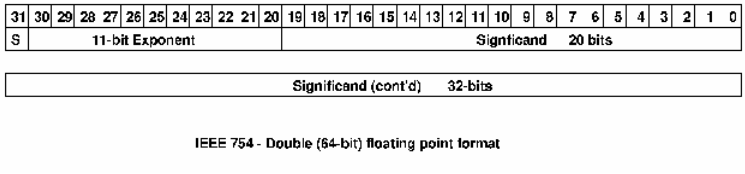
  $0.11001*2^{-3} = 1.11001*2^{-4} = 1.11001 * 2^{11111100}$ (2's complement exponent)
- For binary FP numbers, normalise to:

  $1.xxx…xxx*2^{yy…yy}$
- Problems with FP:
  - Many different floating point formats; problems exchanging data.
  - FP arithmetic not associative: x + (y + z) != (x + y) + z

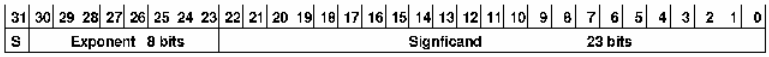
Floating Point Formats

- IEEE 754 format introduced: single (32-bit) and double (64-bit) formats; standard!
  - Also extended precision - 80 bits (long double).
- Single precision number represented internally as
  - sign bit
  - followed by exponent (8-bits)
  - then the fractional part of normalised number (23 bits)
- The leading 1 is implied; not stored.
- Double precision
  - has 11-bit exponent and
  - 52-bit significand
- Single precision range: $2*10^{-38}$ to $2*10^{38}$
- Double range: $2*10^{-308}$ to $2*10^{308}$

Floating Point Templates



IEEE 754 - Double (64-bit) floating point format



IEEE 754 - Single (32-bit) floating point format

Floating Point Exponents

- The exponent is "biased'": no explicit negative number.
- Single precision: 127, Double precision 1023
- So, for single precision:

  exponent of 255 is same as 255-127 = 128, and 0 is 0 - 127 = -127 (can't be symmetric, because of zero)
- Most positive exponent: 111...11, most negative: 00....000
- Makes some hardware/logic easier for exponents (easy sorting/compare).
- Numeric value of stored IEEE FP is actually:

$$(-1)^S * (1 + \text{significand}) * 2^{\text{exponent - bias}}$$

Real to IEEE754 Single

- Example
  - Convert -0.75 to IEEE Single Precision
- Calculation
  - Sign is negative: so S = 1
  - Binary fraction:
    0.75*2 = 1.5 (IntPart = 1)
    0.50*2 = 1.0 (IntPart = 1), so $0.75_{10} = 0.11_2$
  - Normalise: $0.11*2^0 = 1.1*2^{-1}$
  - Exponent: -1, add bias(127) = 126 = 01111110;
- Answer: 1 01111110 100…000000000
  s 8 bits 23 bits

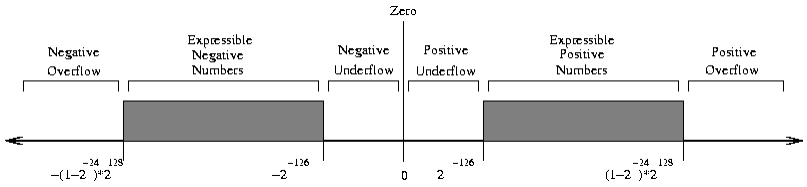
IEEE754 Single to Real

- Example
  - What is the value of the FP number:
    1 10000001 10010000000000000000000
- Calculation
  - Negative number (s=1)
  - Biased exponent: 10000001 = 128+1 = 129
  - Unbiased exponent = 129-127 = 2
  - Significand: 0.1001 = 0.5+0.0625 = 0.5625
- Result = $-1 * (1 + 0.5625)*2^2 = -6.25_{10}$

IEEE 754 Special Codes

- IEEE 754 has special codes for zero, error conditions (0/0 etc).
- **Zero**: exponent and significand are zero.
- **Infinity**: exp = 1111...1111, significand = 0
- **NaN** (not a number): 0/0; exponent = 1111...1111, significand != 0
- Underflow/overflow conditions:

Range of Single Precision FPs



FP Operations and Errors

- Addition/Subtraction: normalise, match to larger exponent then add, normalise.
- Error conditions:
  - **Exponent Overflow** Exponent bigger than max permissable size; may be set to "infinity".
  - **Exponent Underflow** Negative exponent, smaller than minimum size; may be set to zero.
  - **Significand Underflow** Alignment may causes loss of significant digits.
  - **Significand Overflow** Addition may cause carry overflow; realign significands.

Character Representations

- Characters represented using "character set".
- Examples:
  - ASCII (7/8-bit)
  - Unicode (16-bit)
  - EBCDIC (9-bit)
- ASCII - American Standard Code for Information Interchange
  - Widely used; 7-bits used for std characters etc.; extra for parity or foreign language.

Character Codes

- ASCII codes for Roman alphabet, numbers, keyboard symbols and basic network control.
- Parity-bit allows error check (crude) as opposed to Hamming codes, which can be self-correcting.
- Unicode is very popular today: subsumes ASCII, extensible, supported by most languages and OSes.
  - Handles many languages, not just Roman alphabet and basic symbols.

Bit/Byte Ordering

- Endianness: ordering of bits or bytes in computer
  - Big Endian: bytes ordered from MSB to LSB
  - Little Endian: bytes ordered from LSB to MSB
- Example: how is Hex A3 FC 6D E5 (32-bit) represented?
  - Big Endian: A3FC6DE5 (lowest byte address stores MSB)
  - Little Endian: E56DFCA3 (lowest byte address stores LSB)
- Problems with multi-byte data: floats, ints, etc
- MIPS Big Endian, Intel x86 Little Endian
- Bit ordering issues as well: endian on MSb/LSb
- Can check using bitwise operators...

Boolean Algebra & Logic

- Modern computing devices are digital rather than analog
  - Use two discrete states to represent all entities: 0 and 1
  - Call these two logical states **TRUE** and **FALSE**
- All operations will be on such values, and can only yield such values.
- George Boole formalised such a logic algebra as "Boolean Algebra".
- Modern digital circuits are designed and optimised using this theory.
- We implement "functions" (such as add, compare, etc.) in hardware, using corresponding Boolean expressions.

Boolean Operators

- There are 3 basic logic operators

Operator	**Usage**	**Notation**
AND	A AND B	A.B
OR	A OR B	A+B
NOT	NOT A	$\overline{A}$

- A and B can only be TRUE or FALSE
- TRUE represented by 1; FALSE by 0

Truth Tables

- Use a truth table to show the value of each operator (or combinations thereof).
  - AND is TRUE only if both args are TRUE
  - OR is TRUE if either is TRUE
  - NOT is a unary operator: inverts truth value

A	B	F=A.B	F=A+B	F = $\bar{A}$	F=$\bar{B}$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

NAND, NOR and XOR

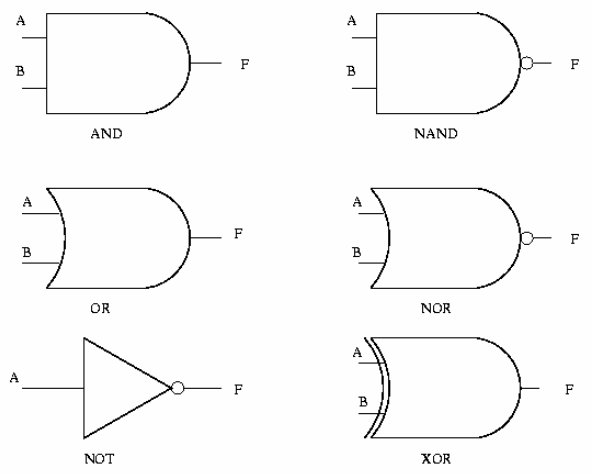
- NAND is FALSE only both args are TRUE [NOT (A AND B)]
- NOR is TRUE only if both args are FALSE [NOT (A OR B)]
- XOR is TRUE is either input is TRUE, but not both

A	B	F=$\overline{A.B}$	F=$\overline{A+B}$	F=A⊕B
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

Logic Gates

- These operators have symbolic representations: "logic gates".
- These are the building blocks for all computer circuits.
- Specify arbitrary F using truth table; then derive Boolean expression.

Logic Gate Symbols



Finding a Boolean Representation

- F = F(A,B,C); F called "output variable"
- Find F values which are TRUE:
  - So, if A=0, B=1, C=0, then F = 1.
  - So, $F_1 = \bar{A}.B.\bar{C}$
  - That is, we know our output is TRUE for this expression (from the table).
  - Also have $F_2 = \bar{A}.B.C$ & $F_3 = A.B.\bar{C}$
  - F TRUE if F_1 TRUE or F_2 TRUE or F_3 TRUE
  - $\Rightarrow F = F_1 + F_2 + F_3$

Cases for F FALSE follows from F TRUE

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Algebraic Identities

- *Commutative*: A.B = B.A and A+B = B+A
- *Distributive*:
  - A.(B+C) = (A.B) + (A.C)
  - A+(B.C) = (A+B).(A+C)
- *Identity Elements*: 1.A = A and 0 + A = A
- *Inverse*: $A.\bar{A} = 0$ and $\bar{A} + A = 1$
- *Associative*:
  - A.(B.C) = (A.B).C and A+(B+C) = (A+B)+C
- *DeMorgan's Laws*:
  - $\overline{A.B} = \bar{A} + \bar{B}$ and
  - $\overline{A+B} = \bar{A}.\bar{B}$