School of Computing Science
Simon Fraser University

CMPT 120 - Introduction to Computing Science and Programming

Term: Fall 2012-3

Instructor: Bill Havens

NOTE: section numbers ("§") refer to the CMPT-120 Study Guide.

Notes 9: Binary Arithmetic

1. (§2.6) Binary Number Representation

- Modern computers are "binary digital computers" meaning that they compute using binary numbers.
- What are binary numbers?
- Definition: a **binary number** is a number composed of only the digits 0 and 1 using a positional number representation.
- Examples:

```
0, 111, 011011001
```

- Number systems are characterized by the number of digits used to represent values, called the **base** of the number system.
- Binary number are base-2
- While ordinary numbers using 10-digits ("0", "1", ..., "9") are base-10.
- Examples:

```
0, 128, 99999
```

- Note the difference between a number and its representation in some base.
- Every number can be represented in any base
- Examples:

$0_{10} = 0_2$

$129_{10} = 10000001_2$

$99999_{10} = 11000011010011111_2$

where we use the subscripts following the numbers to indicate in which base they are represented.

- Other popular bases include:
  - base-16 (called "hexadecimal" and pioneered by IBM)
    `digits = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`
  - base-8 (called "octal" and pioneered by Digitial Equipment Corp - nay Oracle)
    `digits = 0,1,2,3,4,5,6,7`

- Note that these bases are really binary "under the hood"
- Definition: a **positional number representation** represents arbitrarily large numbers using a fixed alphabet of digits organized such that digits (read right-to-left) represent successively higher orders of magnitude (of the base).
- Example: base-10

```
129  = 1*10^2 + 2*10^1 + 9*10^0
     = 100 + 20 + 9
     = 129
```

- Example: base-2

```
10000001 = 1*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 1*2^0
         = 128 + 0 + 0 + 0 + 0 + 0 + 0 + 1
         = 129
```

- Why do computers use binary arithmetic?
- Early computers used decimal arithmetic instead
- More natural for people (why?)
- How many digits do you have?
- Decimal computers used a number representation called "binary coded decimal" (BCD)
- BCD representation (using 4 binary digits)

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
1010 = error
1011 = error
1100 = error
1101 = error
1110 = error
1111 = error
```

- So the BCD representation wasted a lot of memory for illegal values
- Modern computers ALL use binary number representation
- Conversion from decimal to binary and back to decimal implemented as I/O functions

- See today's laboratory assignment
- BCD still popular in business/financial software
- Bad ideas never die!
- See: http://en.wikipedia.org/wiki/Binary-coded_decimal

2. Bits and Bytes

- Each piece of binary data is called a **bit** (smallest possible piece)
- In modern machines, bits are grouped into 8-bit pieces called **bytes** (for convenience)
- Computer arithmetic performed in 16-bit, 32-bit or now 64-bit chunks called **words**.
- First personal computer (Altair homebrew kit) used an 8-bit Intel 8080 chip.
- Memory sizes also expressed in bytes
- All sizes as powers of 2
- Examples:
    - kilobyte = 2^{10} bytes = 1024
    - megabyte = 2^{20} = 1048576
    - gigabyte = 2^{30} = 1073741824

- Why represent information in computers using binary data? Why not natural base-10?
- (1) Because binary data is robust in memory
    - magnetic polarity in hard disks
    - electrical charge in flash memory
    - electrical voltage in RAM memory
    - electrical voltage in CPU chips
    - pulses of light in optical fibers
- (2) Boolean logic is simple for implementation in hardware
    - arithmetic for boolean number is much simpler than base-10 arithmetic
    - can be implemented using simple boolean logic gates
    - examples: AND, OR, XOR, NOT, ...
    - logic gates are easy to make on integrated circuit chips (and cheap!)
    - we shall see how to perform binary arithmetic later . . .

3. Binary Arithmetic

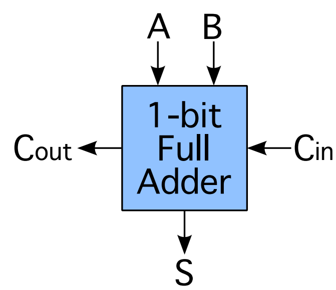
- Lets practice with binary addition
- Consider the following example:

```
 1010
+0100
______
 1110
```

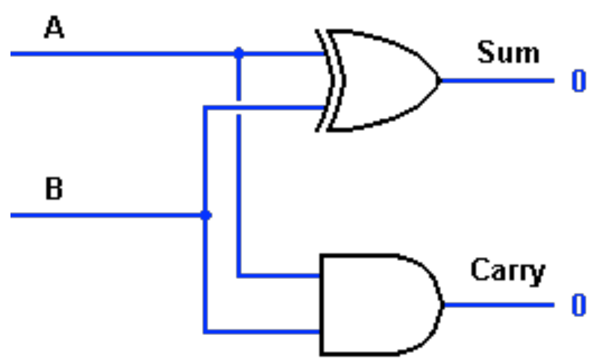
• Now an example with carries

```
  1101
 +0101
 _____
 10010
```

• Exactly the same as decimal arithmetic (grade 3?)
• How does the computer CPU chip implement binary addition?



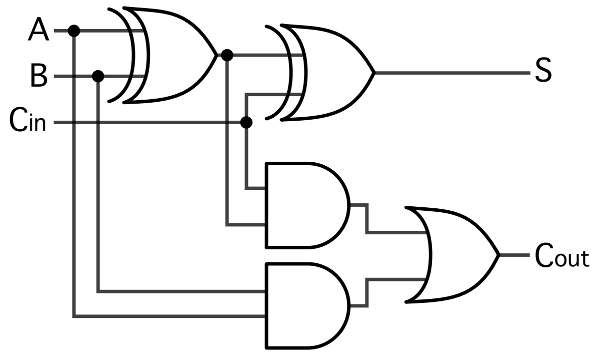
• Using boolean logic circuits (note the transition from arithmetic to logic)
• Here is the logic circuit for a 1-bit adder
• see: http://www.circuitstoday.com/half-adder-and-full-adder



• Logic:

```
Carry = AND(A, B)
Sum = ExclusiveOR(A, B)
```

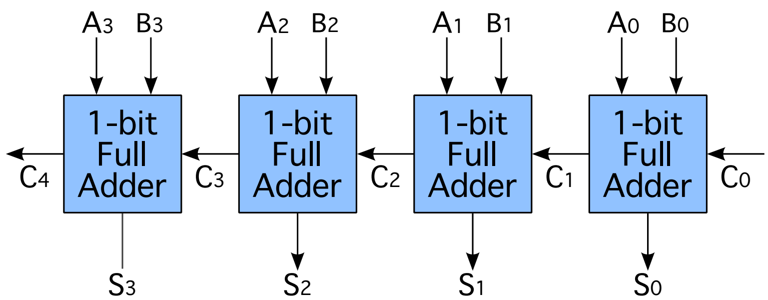
• Example: show sum and carry logic for binary addition
• Above is actually called a "half-adder" because it has no input for the carry bit. Need to use two half-adders to make a "full-adder" for each bit in the binary number.
• Here is a full-adder circuit (for curiosity sake only):



- These details are not important for our purposes

CPUs

- We could "glue" 8 or 16 or more of these things (full adders) together to add larger numbers.



- This is how the CPU implements arithmetic
- *Observation*: building a modern computer CPU using boolean logic is straightforward

Simulation

- Lets simulate the logic of the full-adder in software.
- Full 8bit adder coded in Python

```
# full adder for 8bit binary arithmetic
# arguments are all 8-bit lists with high-order bit leftmost

def adder8bit(x, y, sum):
    carry = 0
    for i in range(7,-1,-1):
        sum[i] = (x[i] + y[i] + carry) % 2      # XOR gate
        carry =  (x[i] + y[i] + carry) / 2      # AND gate
    return carry                                # overflow
```

- Simulates XOR and NAND gates using modulus arithmetic and integer division
- Examples: show addition of lists of binary numbers as registers

```
x = [00000011]        # 3 (base10)
y = [00000010]        # 2 (base10)
z = [00000000]        # 0 (base10)
adder8bit(x, y, z)
```

4. Signed vs Unsigned Numbers

- Above we considered **unsigned** arithmetic only (all positive)
- Java, C, C++ all provide operations for unsigned arithmetic
- How can we represent positive AND negative numbers?
- Called **signed arithmetic**
- How can we do subtraction, multiplication and division?
- Similar boolean logic circuits do it all!
- Handling negative numbers is particularly interesting
- Two approaches:
- (1) attach a **sign bit** to each number to indicate whether positive or negative
  - Example

```
+0100
-0010
_____
+0010
```

  - Sign bit is just the left-most (high-order) bit in the binary number
  - But this approach is awkward
  - Logic is complicated to implement
- (2) use a clever scheme called **2's-complement arithmetic**
- note that for n-bits there are 2^n possible patterns (permutations)
- for 8-bits there are $2^8 = 256$ possible numbers to represent
- in unsigned arithmetic the numbers are 0, 1, ..., 255
- BUT we could allocate half for positive and half for negative as follows:
- possibles values are:

$$-2^{n-1}, \ldots, -1, 0, +1, \ldots, +2^{n-1}-1$$

- for 8bits this scale is:

$$-128, \ldots, -1, 0, +1, \ldots, +127$$

- note that zero is considered positive

- But those hardware engineers are even more clever!
- Negative values above are encoded different than positive values (called 2's complement)
- Negative values are coded as follows:
  - all the bits of the number are negated (flipped) from their positive version
  - 1 is added to the result
- Example:

```
0101        = +5
1010        = all bits flipped
1011        = -5 by adding one
```

- Advantages
  - testing for negative value is easy (high order bit = 1)
  - for positive numbers, the signed and unsigned versions are the same
  - only one representation for zero
  - addition and subtraction work the same (without testing the sign bit)
- Example: signed addition

```
1011      = -5
0100      = +4
____
1111      = -1
```

- which can be seen by converting back to unsigned

```
1111
  -1      # subtract 1
____
1110      # now flip bits
0001      = 1
```

5. Conversions to/from Binary

- Already seen how to convert binary to a decimal above
- But as an algorithm for any base its similar

```
given a binary number x
let sum = 0
for the binary digit d scanning x from left to right:
   sum = sum * 2 + d
return sum
```

- Example: convert 0101 to decimal

```
sum = 0
sum = sum * 2 + 0 = 0
sum = sum * 2 + 1 = 1
sum = sum * 2 + 0 = 2
sum = sum * 2 + 1 = 5
sum = 5
```

- Converting from a decimal value to binary
- Need to repeatedly find the lowest order bit and shift the remainder
- Similar to the 8-bit adder above
- Algorithm:

```
given a decimal number x
repeat:
    next digit = x % 2          # find low order bit
    x = x /2                    # shift number to the right
    until x = 0
```

- Example: convert decimal 3 to binary

```
x = 3
next digit = 3 % 2 = 1      # next binary digit = 1
x = x / 2 = 3 / 2 = 1
next digit = 1 % 2 = 1      # next binary digit = 1
x = x / 2 = 1/2 = 0
halt
```

6. Floating-Point Numbers

- Floating point numbers are a different "kettle of fish"
- What are they for?
- Compare: counting versus distance
- Counting uses integers
- Distance measure is a real value (arbitrarily small or large)
- Approximated in computer arithmetic using floating point representation
- Basic Idea: Use **scientific notation** to extend dynamic range
- Examples in decimal:

$0.67788 * 10^{28}$ # large number

$0.1 * 10^{-200}$ # very small number

- Scientific notation represents real numbers as:

```
(<exp>, <fraction>)
where <exp> is a signed integer
and <fraction> is a signed integer fraction with the radix
point at the far left
```

7. Characters and Strings

- Characters are stored in memory as binary patterns.
- Each character has a unique (assigned) bit pattern.
- Called a **character code**
- Patterns are organized to facilitate sorting
- Various character codes are defined.
- Examples: EBCDIC, ASCII, Unicode
- EBCDIC
    - defined by IBM for their IBM-360 series computers
    - circa 1963
    - very complicated extension of BCD numbering
    - see http://en.wikipedia.org/wiki/Extended_Binary_Coded_Decimal_Interchange_Code
- ASCII = American Standard Code for Information Interchange
    - circa 1963
    - popular standard for many years
    - 7-bit code
- ASCII table:

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

- Unicode
  - Successor to ASCII
  - contains as a subset
  - used by Java exclusively
  - 16-bit code
  - supports many languages / character sets