Manipulating Binary Information in Computers

- An operation manipulates the **fixed-width** binary representation of information
  - It combines **n-bit values** to get **n-bit results**
- Basic Arithmetic Operations : add, subtract, multiply, divide
- Basic Logic Operations : and, or, xor

Arithmetic Operations : Binary Addition and Subtraction

- These operations perform a **bitwise** add/subtract of values of width n to give a result of width n

- 8-bit Unsigned Integer Examples:

$$\begin{array}{rl} & 117_{10} = 0111\ 0101_2 \\ + & 99_{10} = 0110\ 0011_2 \\ \hline & 216_{10} = \mathbf{1101\ 1000}_2 \end{array}$$

$$\begin{array}{rl} & 133_{10} = 1000\ 0101_2 \\ - & 51_{10} = 0011\ 0011_2 \\ \hline & 82_{10} = \mathbf{0101\ 0010}_2 \end{array}$$

Arithmetic Operations : Binary Addition and Subtraction

Signed is now being used to mean 2's complement signed

8-bit Signed Integer Examples:

- The computer does exactly the same thing for **2's complement** signed integers! ☺

$$\begin{array}{rl} & -117_{10} = 1000\ 1011_2 \\ + & 99_{10} = 0110\ 0011_2 \\ \hline & -18_{10} = \mathbf{1110\ 1110_2}\ (0001\ 0001_2 + 1 = 12h) \end{array}$$

$$\begin{array}{rl} & -32_{10} = 1110\ 0000_2 \\ - & 5_{10} = 0000\ 0101_2 \\ \hline & -37_{10} = 1101\ 1011_2\ (\mathbf{0010\ 0100_2 + 1 = 25h}) \end{array}$$

Arithmetic Operations : Binary Addition and Subtraction

Computers often implement subtraction using “negate and add”

$$X - Y = X + (-Y)$$

Example : $32 - 65 = 32 + (-65)$

$$\begin{array}{rrl} & 32_{10} = & 0010\ 0000_2 \\ + & -65_{10} = & 1011\ 1111_2\ (\ 0100\ 0001_2 + 1) \\ \hline & -33_{10} = & 1101\ 1111_2\ (\ \mathbf{0010\ 0000_2 + 1 = 21h}) \end{array}$$

$$\begin{array}{rrl} & -32_{10} = & 1110\ 0000_2 \\ + & (-5)_{10} = & 1111\ 1011_2 \\ \hline & -37_{10} = & \mathbf{(1)\ 1101\ 1011_2} \end{array}$$

Overflow

- **Overflow** occurs when result of operation **outside the range** that can be represented
  - Problem arising due to **limited range** of fixed-width representation.
  - Result still produced: **meaningless**.

- 8-bit Unsigned Integer Example:

Question : What is the range of an 8-bit unsigned number ?

$$\begin{array}{rrl} & 255_{10} = & 1111\ 1111_2 \\ + & 1_{10} = & 0000\ 0001_2 \\ \hline 256\ ?? & 0_{10} = & \mathbf{(1)\ 0000\ 0000_2} \end{array}$$

We need 9 bits to represent result

CARRY

  - OVERFLOW OCCURRED! (fixed 8-bits)
  - In this case (unsigned) : carry @ MSB is important in the INTERPRETATION of the result.

Addition and Subtraction Overflow

Is that the only interpretation of the example?

$$\begin{array}{rr} & 1111\ 1111_2 \\ + & 0000\ 0001_2 \\ \hline (1) & 0000\ 0000_2 \end{array}$$

Same binary pattern !

- What if the values are interpreted as 8-bit signed integers ?
  - The result is correct (– 1 + 1 = 0). **NO OVERFLOW**.
  - In this case (signed), carry at MSB still occurs but is not important to the interpretation!

Addition and Subtraction Overflow

Another example : With Borrow

8-bit result $32_{10} = 0010\ 0000_2$

$- \quad 65_{10} = 0100\ 0001_2$

$-33_{10} = 1\ \ 1101\ 1111_2$

8-bit result

(+223 and Borrow 1 if interpreted as Unsigned)

- If values interpreted as unsigned, the borrow **implies overflow** (actually, underflow)
- If values interpreted as signed, no overflow; **ignore the borrow**.

Addition and Subtraction Overflow

Overflow depends on the *interpretation* of the values.

Another example:	**unsigned**	**signed**
$0111\ 1111_2$	127	127
+ $0000\ 0001_2$	+ 1	+ 1
$1000\ 0000_2$	128	– 128

OVERFLOW … even though there is no carry outside of fixed width!

Overflow Cookie Cutters

Unsigned: Carry or borrow means overflow

Signed : Ignore carry or borrow

Overflow occurred if : positive + positive = negative

(positive – negative = negative)

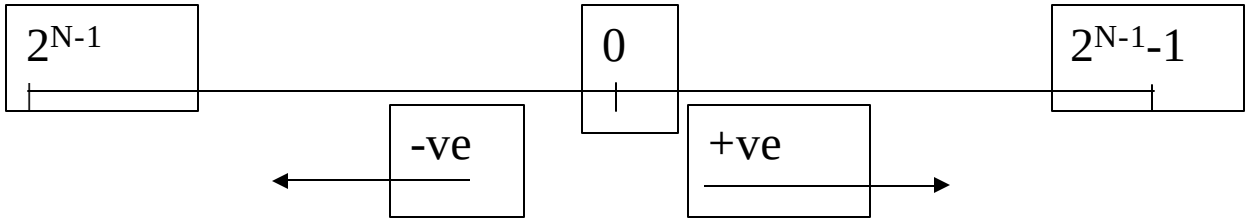
negative + negative = positive

(negative – positive = positive)

Overflow is impossible if :

positive + negative (positive – positive)

negative + positive (negative – negative)



Logical Operations - AND

Perform bit-wise logic operations (ELEC2607!)

AND

1-bit truth table:

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

8-bit example:

```
        1011 0110
AND     1100 0011
        ---------
        1000 0010
```

Logical Operations - OR

1-bit truth table:

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

8-bit example:

```
        1011 0110
OR      1100 0011
        ---------
        1111 0111
```

Logical Operations - XOR

1-bit truth table:

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

8-bit example:

```
          1011 0110
XOR       1100 0011
          ---------
          0111 0101
```

Logical Operations : Shift and Rotate

- Shift and Rotate Operations: move bits to the left or to the right
- Difference? Treatment of the most and least significant bits
  - Rotates : Bits put **into the other side** (circular storage)
  - Shifts : Bits injected on one end and **"drop"** off the other end.

- Example : Shift Left

Value before: 1001 1010

Value after being shifted left 4 bits: 1010 0000

What happened to the upper 4 bits?

What value was injected ?

Arithmetic versus Logical Shifts

- When shifting left, **zero** is always injected.
- When shifting right, two versions.
  - Difference ? Value injected.

- Logical Shift: value treated as a logical or unsigned value.
  - **Zero** is inserted (Shift-right is same as shift-left)
- Arithmetic Shift: value treated as a signed value.
  - **MSB** is injected when shifting right. Why ?

- Example : Arithmetic Shift Right 2 bits

Value before: 1001 1010
Value after shift-right-by-2: 1110 0110

MSB = 1 so inject 1's

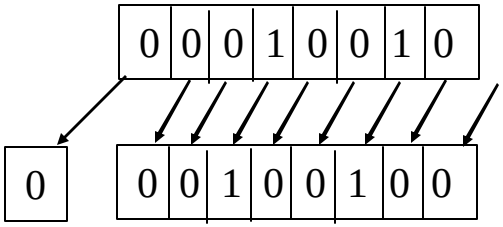
Rotate Operations

- Rotates: shift bits; bit shifted “out” of the value gets injected as new bit
  - All bits in original value are saved

- Example : Rotate right 3 bits

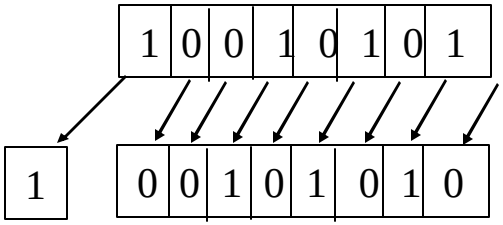
Value before :	1001 1110	0 rotated “out”and is injected as MSB
Rotated 1st bit:	0100 1111	
Rotated 2nd bit:	1010 0111	1 is rotated out
Rotated 3rd bit:	1101 0011	1 is rotated out

Example Suppose that we shift the number 12h left by 1.



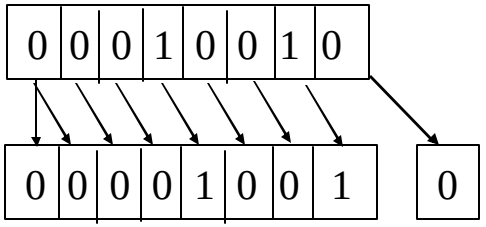
Before : 12h
After :

Example Suppose that we shift the number 95h left by 1



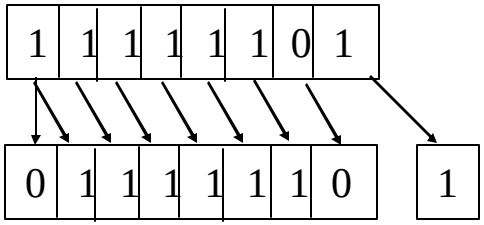
Before : 95h
After :

Example Suppose that we arithmetic/logical-shift the number 12h right by one.



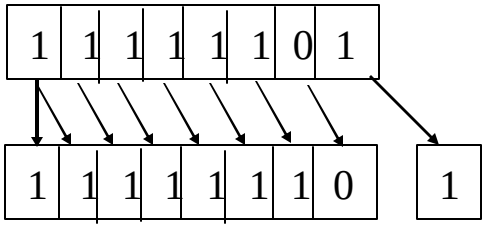
Before : 12h
After :

Example Suppose that we logical-shift the number EDh right by one.



Before : FDh
After :

Example Suppose that we arithmetic-shift the number EDh right by one.



Before : FDh
After :