CS354: Machine Organization and Programming

Lecture 7
Friday the September 18th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

1. Questions about Assignment 1?
2. Come meet us at office hours for hands-on help. <2 students show up every hours now.
3. Start Early! Assign 1 is much much harder than Assign 0!
4. Hands-on overview of File I/O and related C Programming aspects relevant to P1 during lecture?

Lecture Overview

1. Integer Arithmetic (Addition, Subtraction, Multiplication, Division, Sign Extension, Logical Operations)
2. Data Representation (Floating Point)

Unsigned Representation

$B2U_w(x_{vec}) = Sum_{i=0->w-1}\ x_i.2^i$

$B2U_4([0101]) = 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 5$

$B2U_w$ is a bijection:
- associates a unique value to each bit vector of length w
- each integer between **0 and 2^w-1** has a unique binary representation as a bit vector of length w

Two's complement Representation

$B2T_w(x_{vec}) = -x_{w-1}2^{w-1} + Sum_{i=0->w-2}\ x_i 2^i$

$B2T_4([1011]) = -1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = -5$

$B2T_w$ is a bijection:
- associates a unique value to each bit vector of length w
- each integer between **-2^{w-1} and $2^{w-1}-1$** has a unique binary representation as a bit vector of length w

Conversion from 2's complement to unsigned

Rule: The numeric values might change but the bit patterns do not.

$T2U_w(x)$ equals:

$x+2^w$, if $x < 0$

x, if $x >= 0$

2's Complement Addition

Of two signed 2'complement w bit values X & Y

X + Y equals:

- $X+Y-2^w$, if $2^{w-1} <= (X+Y)$ Positive overflow
- $X+Y$, if $-2^{w-1} <= (X+Y) < 2^{w-1}$ Normal
- $X+Y+2^w$, if $(X+Y) < -2^{w-1}$ Negative overflow

Two's Complement Addition

	1	1	1	1	1	1	1	0	( )
+	0	0	0	0	0	0	0	1	( )
									( )

	1	1	1	1	0	0	0	0	( )
+	0	0	1	1	0	0	0	0	( )
									( )

Two's Complement Addition

	1	1	1	1	1	1	1	0	(-2)
+	0	0	0	0	0	0	0	1	(1)
	1	1	1	1	1	1	1	1	(-1)

	1	1	1	1	0	0	0	0	(-16)
+	0	0	1	1	0	0	0	0	(48)
	0	0	1	0	0	0	0	0	(32)

Overflow

The condition in which the result of an arithmetic operation cannot fit into the fixed number of bits available.

For example:

+8 cannot fit into a 3-bit, unsigned representation. It needs 4 bits: `1000`

Overflow Detection

- Most architectures have hardware that *detects* when overflow has occurred (for arithmetic operations).
- The detection algorithms are simple.

Unsigned Overflow Detection

6-bit examples:

```
  0 0 1 1 1 1
+ 0 0 1 1 1 1
-------------
```

```
  1 1 1 1 1 1
+ 0 0 0 0 0 1
-------------
```

```
  1 0 0 0 0 0
+ 1 0 0 0 0 0
-------------
```

Carry out from msbs is overflow in unsigned

Unsigned Overflow Detection

6-bit examples:

$$
\begin{array}{cr}
 & 0\ 0\ 1\ 1\ 1\ 1 \\
 + & 0\ 0\ 1\ 1\ 1\ 1 \\
\hline
0 & 0\ 1\ 1\ 1\ 1\ 0
\end{array}
$$

No Overflow

$$
\begin{array}{cr}
 & 1\ 1\ 1\ 1\ 1\ 1 \\
 + & 0\ 0\ 0\ 0\ 0\ 1 \\
\hline
1 & 0\ 0\ 0\ 0\ 0\ 0
\end{array}
$$

Overflow!

$$
\begin{array}{cr}
 & 1\ 0\ 0\ 0\ 0\ 0 \\
 + & 1\ 0\ 0\ 0\ 0\ 0 \\
\hline
1 & 0\ 0\ 0\ 0\ 0\ 0
\end{array}
$$

Overflow!

Carry out from msbs is overflow in unsigned

<u>Two's Complement</u> Overflow Detection

When adding 2 numbers of like sign

+ to +

- to -

and the sign of the result is different!

$$\begin{array}{rr} & + \\ + & + \\ \hline & - \end{array} \qquad\qquad \begin{array}{rr} & - \\ + & - \\ \hline & + \end{array}$$

Overflow! Overflow!

Addition

Overflow detection: 2's complement

6-bit examples

```
  111111 ( )
+ 111111 ( )
--------
         ( )
```

```
  100000 ( )
+ 011111 ( )
--------
         ( )
```

```
  011111 ( )
+ 011111 ( )
--------
         ( )
```

Addition

Overflow detection: 2's complement

6-bit examples

```
  111111 (-1)
+ 111111 (-1)
  ------
  111110 (-2)
```

```
  100000 (-32)
+ 011111 (31)
  ------
  111111 (-1)
```

```
  011111 (31)
+ 011111 (31)
  ------
  111110 (-2)
```

Subtraction

basic algorithm is like decimal…

$$0 - 0 = 0$$
$$1 - 0 = 1$$
$$1 - 1 = 0$$
$0 - 1 = ?$ BORROW!

```
  111000
- 010110
--------
```

Subtraction

basic algorithm is like decimal…

0 – 0 = 0
1 – 0 = 1
1 – 1 = 0
0 – 1 = ? BORROW!

		Unsigned	Two's complement
	111000	56	-8
–	010110	22	22
	100010	34	-30

Subtraction

For two's complement representation

- The implementation redefines the operation:

  `a - b` **becomes** `a + (-b)`

- This is a 2-step algorithm:
  1. "take the two's complement of `b`"

     (common phrasing for: find the additive inverse of `b`)
  2. do addition

2's Complement Inverse

Additive inverse of a 2'complement w bit value X equals:

-2^{w-1}, if $x = -2^{w-1}$

$-X$, if $X > -2^{w-1}$

2's Complement Inverse: Easy Techniques

1) **Toggle all bits and then add 1:**

E.g. Inverse of 0101 (5) is 1011 (-5)

Inverse of 1000 (-8) is 1000 (-8)

2) **Toggle all bits until (not including) the rightmost 1 bit:**

E.g. Inverse of 0111 (7) is 1001 (-7)

Inverse of 1010 (-6) is 0110 (6)

Subtraction

6-bit, 2's complement examples

```
  001111 (  )
- 111100 (  )
  ______
```

```
  000010 (  )
- 011100 (  )
  ______
```

Subtraction

6-bit, 2's complement examples

```
  001111  (15)        001111   15
- 111100  (-4)       +000100    4
  ------             -------
  010011   19         010011   19
```

```
  000010  ( )
- 011100  ( )
  ------
```

Subtraction

6-bit, 2's complement examples

001111	(15)	001111	15
− 111100	(-4)	+000100	4
010011	19	010011	19

000010	(2)	000010	2
− 011100	(28)	+100100	-28
		100110	-26

Multiplication

$0 \times 0 = 0$

$0 \times 1 = 0$

$1 \times 0 = 0$

$1 \times 1 = 1$

- Same algorithm as decimal…
- There is a precision problem

$$\begin{array}{rr} & n \text{ bits} \\ * & n \text{ bits} \\ \hline \multicolumn{2}{l}{n + n \ \text{ bits may be needed}} \end{array}$$

In HW, space is always designated for a larger precision product.

$$\begin{array}{r} \text{32 bits} \\ * \quad \text{32 bits} \\ \hline \text{64 bits} \end{array}$$

Unsigned Multiplication

```
    01111
*   01101
_________
```

Unsigned Multiplication

```
      01111     15
*     01101     13
----------
       01111
          0
     01111
    01111
       0
----------
    11000011
  128+64+2+1    195
       00011      3
```

Unsigned Multiplication

```
        11111
*       11111
_____________
```

Unsigned Multiplication

```
      11111      31
*     11111      31
-----------
      11111
     11111
    11111
   11111
  11111
-----------
 1111000001      961
```

512+256+128+64+1 = 961

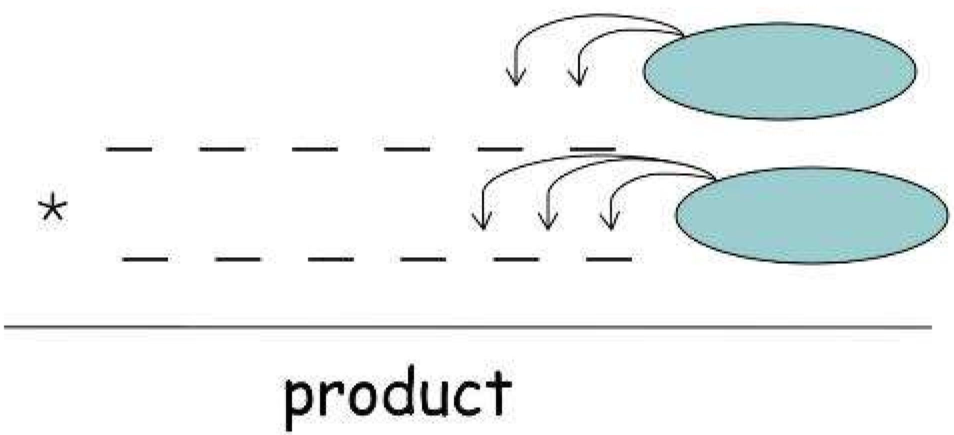
00001 1

Two's Complement

- Slightly trickier: must sign extend the partial products (sometimes!)

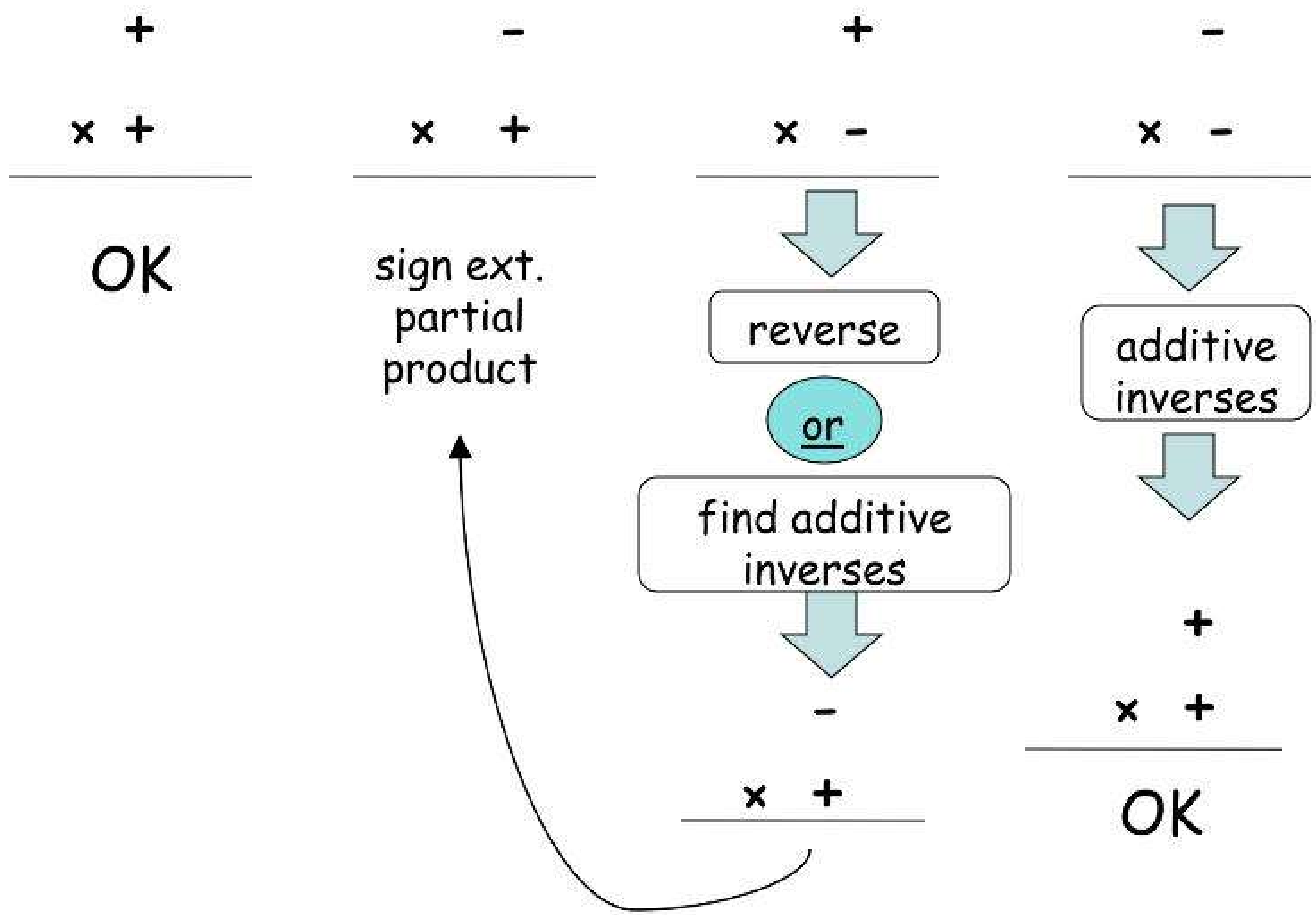
OR

Sign extend multiplier and multiplicand
to full width of product



And, *use only exact number of lsbs of product*

Multiplication



Unsigned Division

11 | 11001

25/3

Unsigned Division

```
         1000   (8)
     ┌──────────
  11 │ 11001
       11
       ──
        0001
```

25/3

Sign Extension

The operation that allows the same 2's complement value to be represented, but using more bits.

```
      0 0 1 0 1  (5 bits)
_ _ _ 0 0 1 0 1  (8 bits)

        1 1 1 0  (4 bits)
_ _ _ _ 1 1 1 0  (8 bits)
```

Sign Extension

The operation that allows the same 2's complement value to be represented, but using more bits.

```
      0 0 1 0 1 (5 bits)
0 0 0 0 0 1 0 1 (8 bits)

        1 1 1 0 (4 bits)
1 1 1 1 1 1 1 0 (8 bits)
```

Zero Extension

The same type of thing as sign extension, but used to represent the same unsigned value, but using more bits

```
      0 0 1 0 1 (5 bits)
_ _ _ 0 0 1 0 1 (8 bits)

        1 1 1 1 (4 bits)
_ _ _ _ 1 1 1 1 (8 bits)
```

Zero Extension

The same type of thing as sign extension, but used to represent the same unsigned value, but using more bits

```
      0 0 1 0 1 (5 bits)
0 0 0 0 0 1 0 1 (8 bits)

        1 1 1 1 (4 bits)
0 0 0 0 1 1 1 1 (8 bits)
```

Truth Table for a Few Logical Operations

X	Y	X and Y	X nand Y	X or Y	X xor Y
0	0	0	1	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	0

Logical Operations

Logical operations are done bitwise on every computer

Invented example:

Assume that `X`, `Y`, and `Z` are 8-bit variables

```
and Z, X, Y
```

If

`X` is `0 0 0 0 1 1 1 1`

`Y` is `0 1 0 1 0 1 0 1`

then

`Z` is `_ _ _ _ _ _ _ _`

To selectively clear bit(s)

- clear a bit means make it a `0`
- First, make a **mask**:

  (the generic description of a set of bits that do whatever you want them to)
- Within the mask,
  - `1`'s for unchanged bits
  - `0`'s for <u>cleared</u> bits

To clear bits numbered 0,1, and 6 of variable `X`

```
mask  1 . . 1 0 1 1 1 1 0 0
```

and use the instruction

```
and result, X, mask
```

To selectively **set** bit(s)

- set a bit means make it a `1`
- First, make a **mask**:
  - `0`'s for unchanged bits
  - `1`'s for set bits

To set bits numbered 2,3, and 4 of variable `X`

```
mask  0 . . 0 0 0 1 1 1 0 0
```

and use the instruction

```
or result, X, mask
```

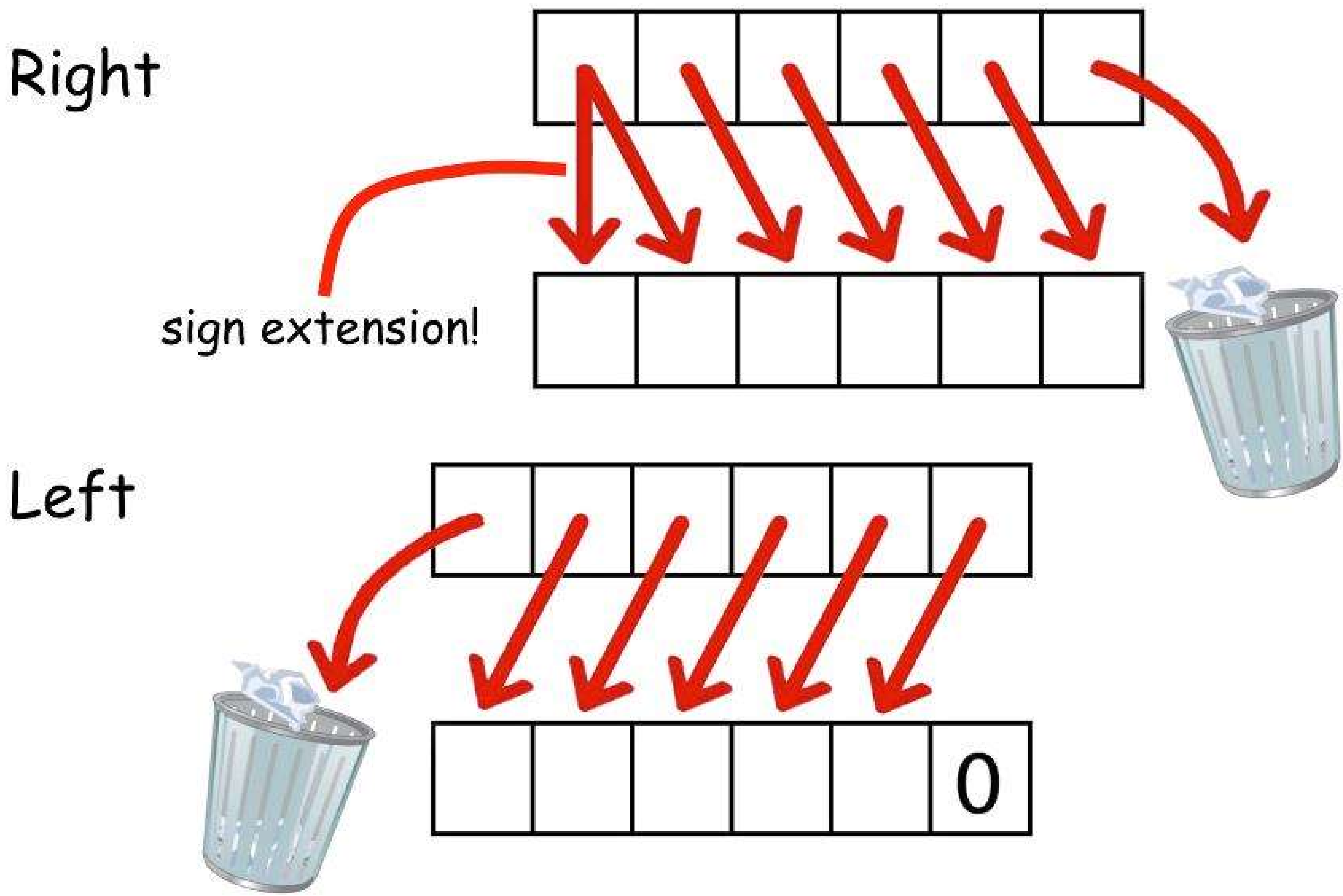
Shift

Moving bits around

1) arithmetic shift
2) logical shift
3) rotate

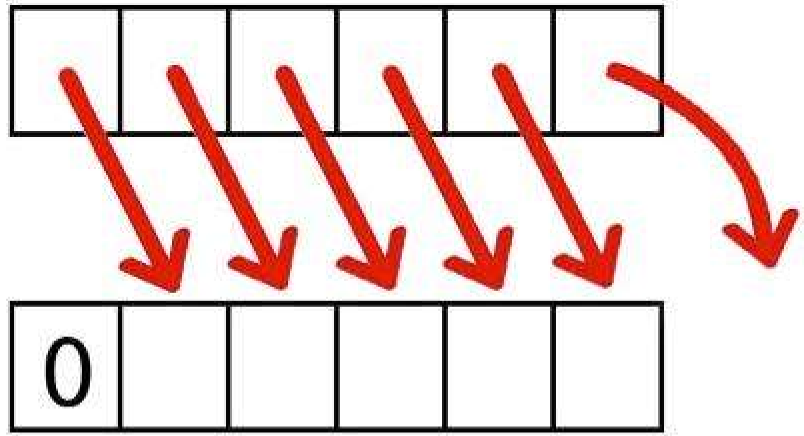
Bits can move right or left

Arithmetic Shift

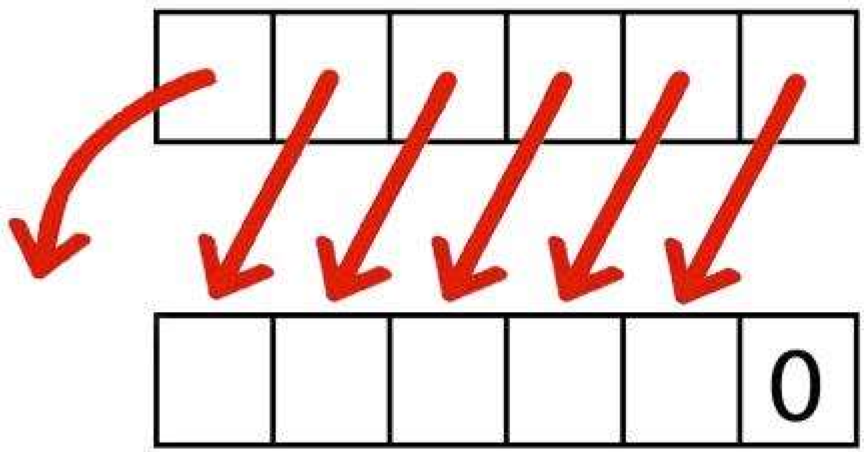


Logical Shift

Right



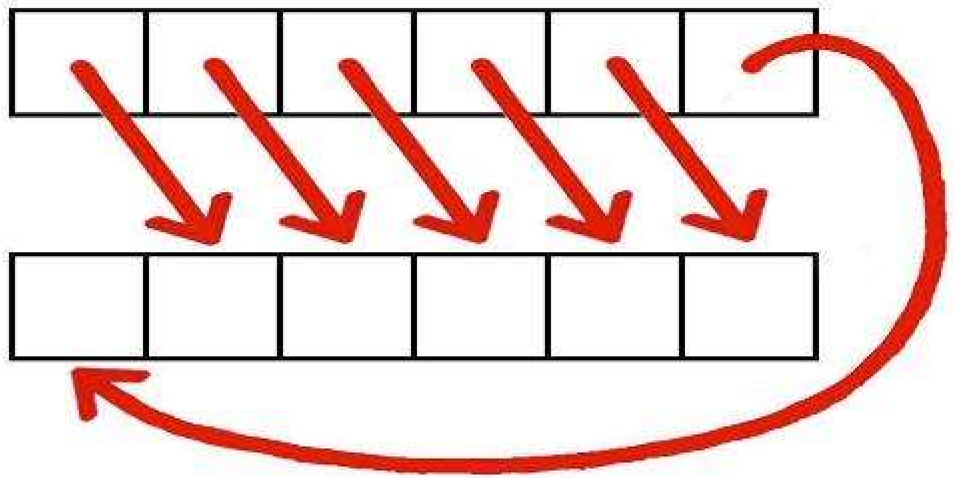
Left



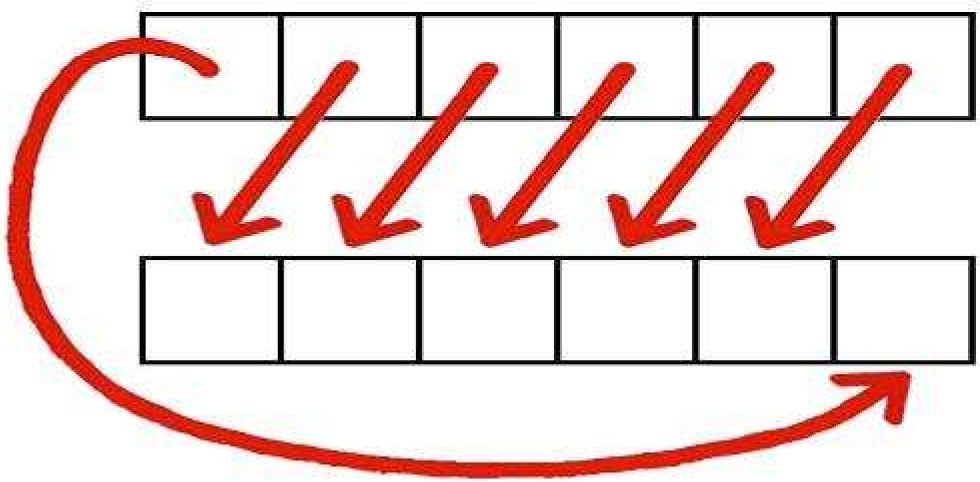
Logical left is the same as arithmetic left.

Rotate

Right



Left



No bits lost, just moved

- Assume a set of 4 chars. are in an integer-sized variable (X).
- Assume an instruction exists to print out the character all the way to the right…

X	‘A’	‘B’	‘C’	‘D’

```
putc X
```

(prints D)

- Invent instructions, and write code to print ABCD, without changing X.