CSC9V4
Systems

Systems Lecture 4

Logic

Graphical User Interfaces	Higher-level Programming
Operating Systems	
Low-level Programming	
Basic Machine Architecture	
Silicon	

Logic Operators

- Other numeric operations on binary numbers
- Logical operations on binary data
- Representing floating point numbers

Multiplication/Division

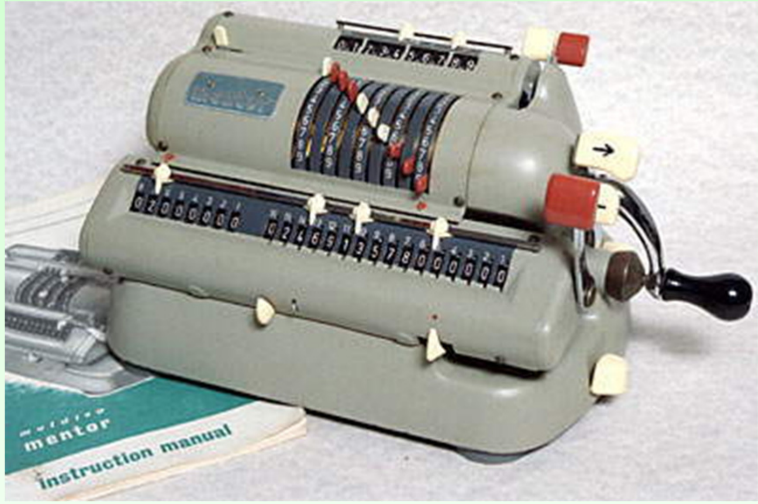
- We have looked at Negation/Addition/Subtraction

- Multiplication can be implemented through repeated addition
    - 7 * 2 = 2 + 2 + 2 + 2 + 2 + 2 + 2
  - Or rather more efficiently through addition and shifting
    - Like long multiplication

- Integer Division can be implemented by repeated subtraction. For example 20 div 3 is the number of times we can subtract 3 from 20 and leave a non-negative result.
  - So we use a loop to repeatedly subtract 3 from 20, whilst we have a non-negative result, counting how many times we execute the loop body
  - Again, shifting and subtraction is rather more efficient...

Early ALU Arithmetic

- Because of all of the above, we can say that *in principle* for integer arithmetic we need only
  - addition
  - two's complement
- Mechanical calculators relied on this:



And early computers

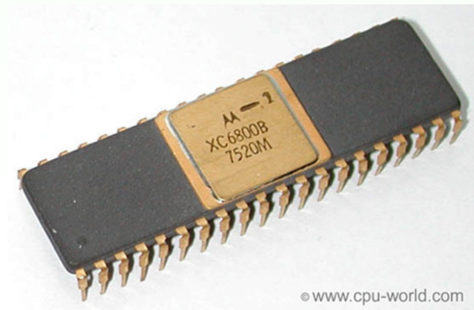
- Where hardware was large, expensive, and power-hungry

- … relied on this too

The earliest microprocessors had no multiply or divide instructions

- Because there wasn't room for the hardware on the chip !

- Multiply and divide had to be coded!
  - Using addition/subtraction and shift operators

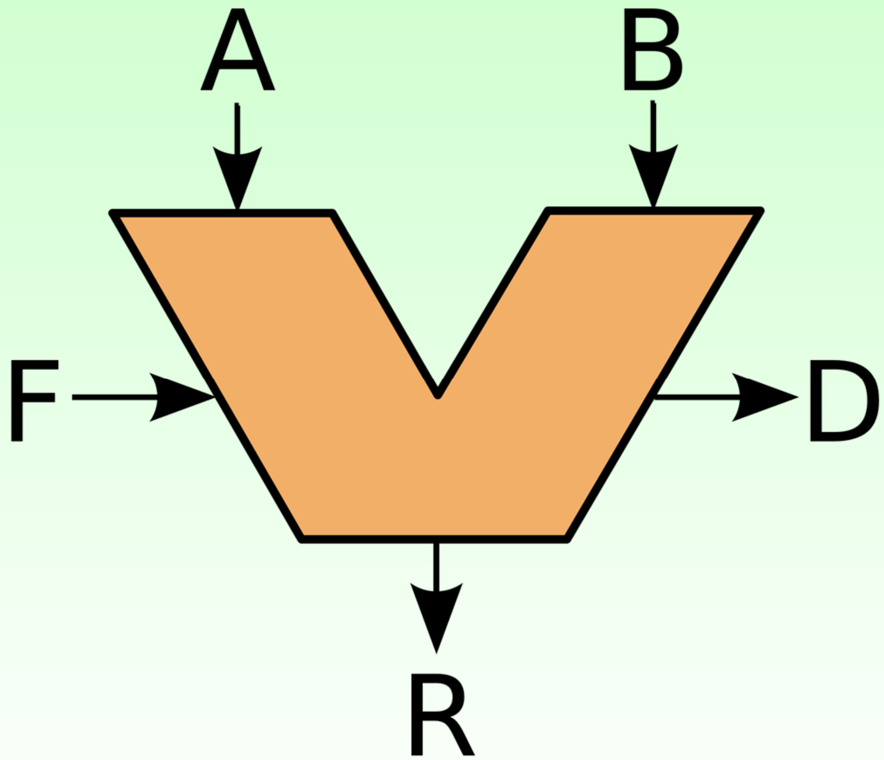


ALU operations

- All operations that alter data take place in a part of the CPU (Central Processing Unit)/processor called the Arithmetic and Logic Unit, or ALU
  - Inside an ALU, for its arithmetic capabilities, at a very minimum we need addition and an operation to flip all the bits (used for negation)
  - An ALU will also have other useful operations that are called logical operations, some of which can be used for multiplication/division
    - NOT
    - AND
    - OR
    - XOR (Exclusive Or)
    - LEFT SHIFT
    - RIGHT SHIFT
    - ROTATE

ALU

- A, B inputs
  - This is a 2-input ALU
- R result
- F function code
  - What the ALU should do to A and B to produce R
- D flags
  - Other information about the operation
  - Carry, overflow, zero, …



Logical Operation: NOT

- **NOT**
  - Set the bit in the output to be the inverse of the bit in the input
    - Truth table

x	NOT x
0	1
1	0

    - Example

```
        0010 1101
NOT x   1101 0010
```

Logical Operation: AND

- **AND**
  - Set the bit in the output only if the corresponding bits in both inputs are set.
    - Truth table

x	y	x **AND** y
0	0	0
0	1	0
1	0	0
1	1	1

    - Example

```
     0010 1101
AND  1011 1011
     ---------
     0010 1001
```

Logical Operation: OR

- **OR**
  - Set the bit in the output if *either or both* of the corresponding bits in the inputs are set.
    - Truth table

x	y	x **OR** y
0	0	0
0	1	1
1	0	1
1	1	1

    - Example

```
     0010 1101
OR   1011 1011
     ---------
     1011 1111
```

Logical Operation: XOR

- **XOR (eXclusive OR)**
  - Set the bit in the output if ***either one or the other*** of the corresponding bits in the inputs are set, but NOT BOTH
    - Truth table

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

    - Example

```
     0010 1101
XOR  1011 1011
     ---------
     1001 0110
```

Logical operations in Java/C

- Java contains operators for performing these operations on bits (bitwise operators). Try not to get them confused with *Boolean* operators.
- AND
  - Bitwise operator & AND
    - 00001111b & 00110011b results in 00000011b
  - Bitwise operator | OR
    - 00001111b | 00110011b results in 00111111b
  - Bitwise operator ^ XOR (EOR)
    - 00001111b ^ 00110011b results in 00111100b
  - Bitwise operator ~ NOT
    - ~00001111b results in 11110000b

- Note: Boolean operators are used to combine Boolean queries
  - E.g. if ((xyz >9) && (fred == 16)) {

Left Shift, Right Shift, Rotate

- **Left Shift**
  - Move the bit pattern along to the left, drop the bit shifted out. Fill space at right end with zero.
    - `0011 0111 -> 0110 1110`

- **Right Shift**
  - Move the bit pattern along to the right, drop the bit shifted out. Fill space at left end with zero.
    - `0011 0111 -> 0001 1011`

- **Rotate (right/left)**
  - As for shift, but move displaced bit to space at other end
    - 0010 1101 -> 1001 0110 (right rotate)

Applications of Bitwise Operations

- Bitwise operations can be performed very quickly by electronic circuits within the computer's ALU.

Examples:

- **Multiplication by 2**: left shift one place
- **Division by 2**: right shift one place

    `e.g. 00101101 is 45, 01011010 is 90`

- And,

```
i = 14; // Bit pattern 1110
j = i >> 1; // shifted by 1 gives 111, ie. 7 which is 14/2!
```

Code example

```
#include <stdio.h>

void showbits(unsigned int x);

int main()
{
   int j = 5225, m, n;
   printf("The decimal %d is equal to binary - ", j);
   /* prints a binary string, given an integer */
   showbits(j);

   /* the loop for right shift operation */
   for ( m = 0; m <= 5; ++m ) {
      n = j >> m;
      printf("%d right shift %d gives ", j, m);
      showbits(n);
   }
   return 0;
} // from wikipedia
```

Continued

```
void showbits(unsigned int x)
{
   int i;
   for(i=(sizeof(int)*8)-1; i>=0; i--) {
      (x&(1<<i))?putchar('1'):putchar('0');
   }
   printf("\n");
}
```

OUTPUT:

- The decimal 5225 is equal to binary – 0001010001101001
- 5225 right shift 0 gives 0001010001101001
- 5225 right shift 1 gives 0000101000110100
- 5225 right shift 2 gives 0000010100011010
- 5225 right shift 3 gives 0000001010001101
- 5225 right shift 4 gives 0000000101000110
- 5225 right shift 5 gives 0000000010100011

Historical note

- Shifting and adding was how multiplication was implemented:
  - Set result to 0
  - For bitno = 1: number of bits in word
    - If O'th bit of multiplier is 1, add multiplicand to result
    - shift multiplier one place right (// multiply by 2)
  - End for

- Essentially the same as long multiplication in decimal.

Masking

- Bitwise operations are particularly useful for extracting information (like Boolean information) which is represented by individual bits within a byte. This is known as *masking*.

- The masking pattern 'blanks out' all but one of the Boolean values:

```
Bit no:             7654 3210
                    0010 1101   (Boolean information)
            AND     0000 1000   (masking bit pattern)
                    0000 1000
```

  - in this case the one in bit 3
  - We can construct the masking pattern by left-shifting: e.g. to select bit 3 we left shift the mask 01x (=0000 0001b) three times
  - often a CPU will have some useful values like 01x in special registers for this very purpose

More masking examples

- **Testing for a negative integer**

```
        1000 1101
    AND 1000 0000          (Mask is 80x)
        1000 0000
```

- **Flip the bits** (or complement)
  - ```
        0010 1101
    XOR 1111 1111          (Mask is FFx)
        1101 0010
    ```

- **Toggle** between upper and lower case letters in ASCII (7 bits)
  - ```
        100 1011   'K'   (         4Bx)
    XOR 010 0000         (Mask is 20x)
        110 1011   'k'
    ```
  - same operation converts back (XOR is its own inverse)

The big question…

- Where do we need these low-level operations?
  - Sometimes we really do need to set bits
    - Often to control equipment
    - Switch something on or off
    - Control equipment
      - Internally and externally
  - Sometimes you really need to do things really quickly
    - Real-time operation of systems
      - Not just slow-real-time like for a human at a screen
      - Fast, difficult real-time for controlling high speed devices.
  - Sometimes power consumption has to be kept extremely low
    - Deep-space spacecraft
    - Unmanned systems far from power supplies

End of Lecture