Algebra

- What is common between the following?
  - $1 + 1 = 2$
  - $2 + 3 = 5$
  - $6 + 7 = 13$
- Answer:
  - $a + b = c$

What is the benefit of it?

Benefits Of Algebra

1. Readability
   – Which one is easier to read?
   I. The square feet of an area is the length multiplied by the width
   II. length * width = area
2. Easier calculation
   – Sample question,
      - We have ducks and lambs
      - There is a total of 50 heads
      - There is a total 150 feet
      - How much ducks and how much lambs do we have?

Solution

1. Each duck has 1 head and 2 feet
2. Each lamb has 1 head and 4 feet

3. We got
   I. Ducks + Lambs = 50 (heads)
   II. (Ducks * 2) + (Lambs * 4) = 150 (feet)

4. Divide the second equation by 2

$$\frac{(\text{Ducks} * 2) + (\text{Lambs} * 4) = 150}{/2}$$

$$(\text{Ducks}) + (\text{Lambs} * 2) = 75$$

5. Subtract the first equation

$$\begin{array}{rl} & (\text{Ducks}) + (\text{Lambs} * 2) = 75 \\ - & \text{Ducks} + \text{Lambs} = 50 \\ \hline & \text{Lambs} = 50 \end{array}$$

Algebra Functions

A function in algebra is:

1. A formula
2. That returns a value
3. That can take arguments
4. The return value is always the same, if the arguments are the same

Example:

1. Add(a,b) is a function
2. CurrentTime() is not a function
3. Void DoSomethingNoReturn is not a function

Algebra function syntax:

area = f(length, width) = length * width

Functions In Computers

- Function vs Sub
  - Databases separate between a function (that returns a value) and stored procedures (that is like a subprogram but doesn’t return a value)
  - Also basic distinguishes between a function that returns a value and a “sub” (subroutine or subprogram) that doesn’t return a value
  - Other imperative languages consider everything as functions
- Same return value
  - MySQL has a keyword “deterministic”
  - In functional languages a variable is not changeable, and a function always returns the same value (as in algebra)
    - A function with no arguments is in fact a constant

Boolean Algebra

- True = 1
- False = 0

- True AND True = True
- True AND False = False

- True OR False = True

- True Xor True = False
- True Xor False = True

- Not True = False
- Not False = True

Computers And Boolean Algebra

1. If and while statements

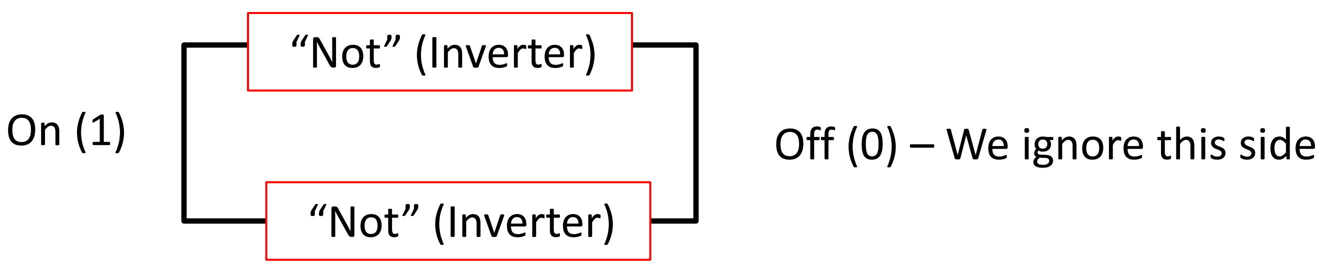
2. The hardware logic gates are based on “Nand” and “Not” circut transistors

Computer Language history

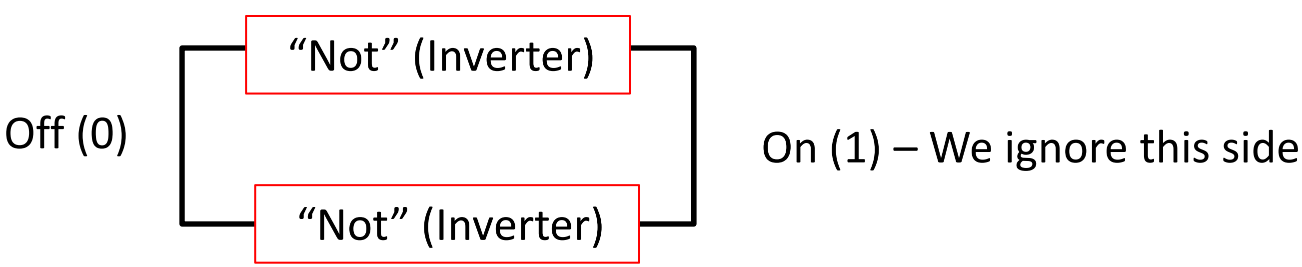
- Machine Language
  - Assembly Language
    - Fortran (FORmula TRANsalator)
      - Algol
        - » B
          - C (UNIX) by K&R
            - C++ [c = c + 1]
            - Java
            - C#
            - JavaScript
            - PHP
    - Basic
      - VB6
      - VBA
      - VBS
      - VB.Net

Sample Memory Transistor

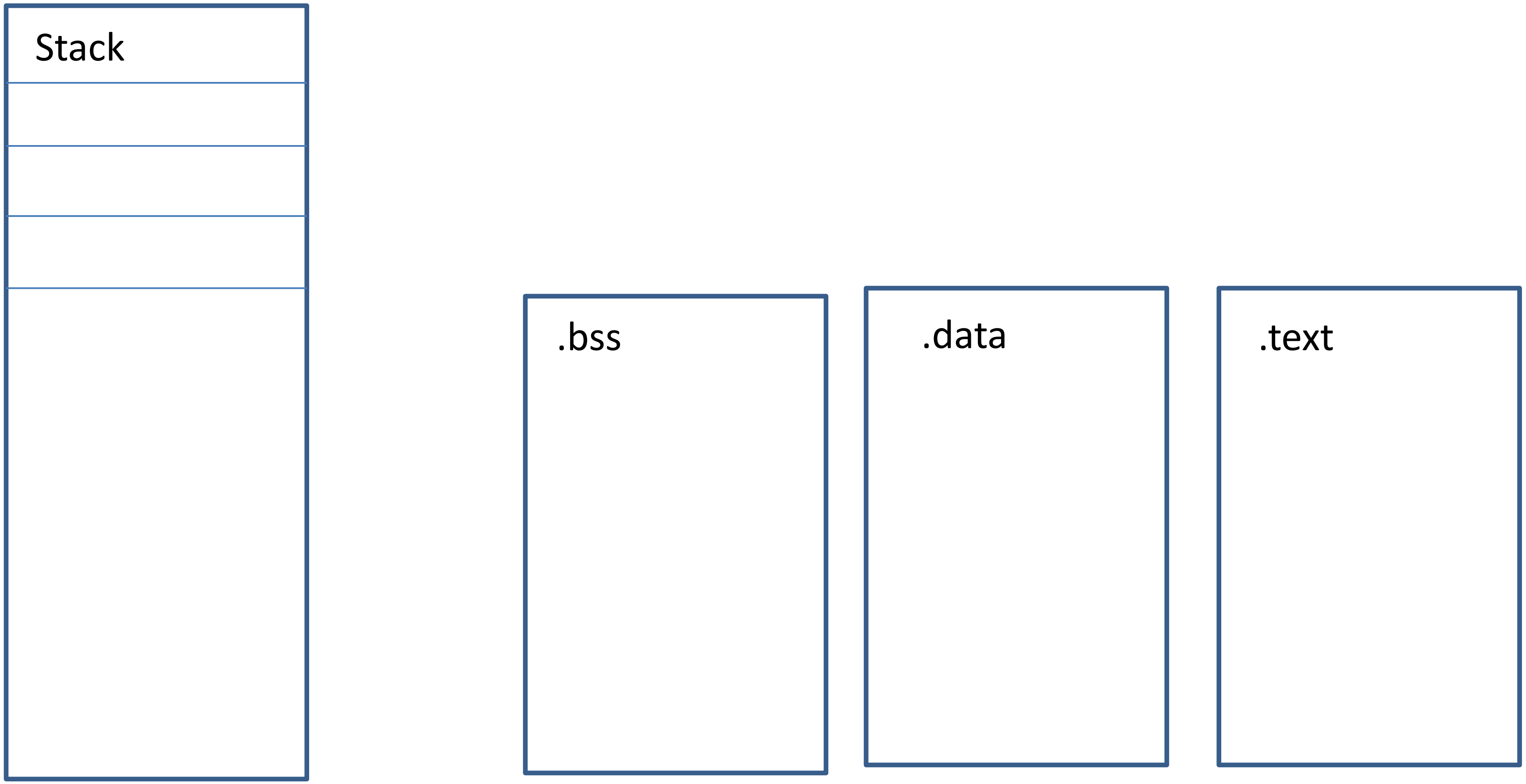
Chip Is On (1)



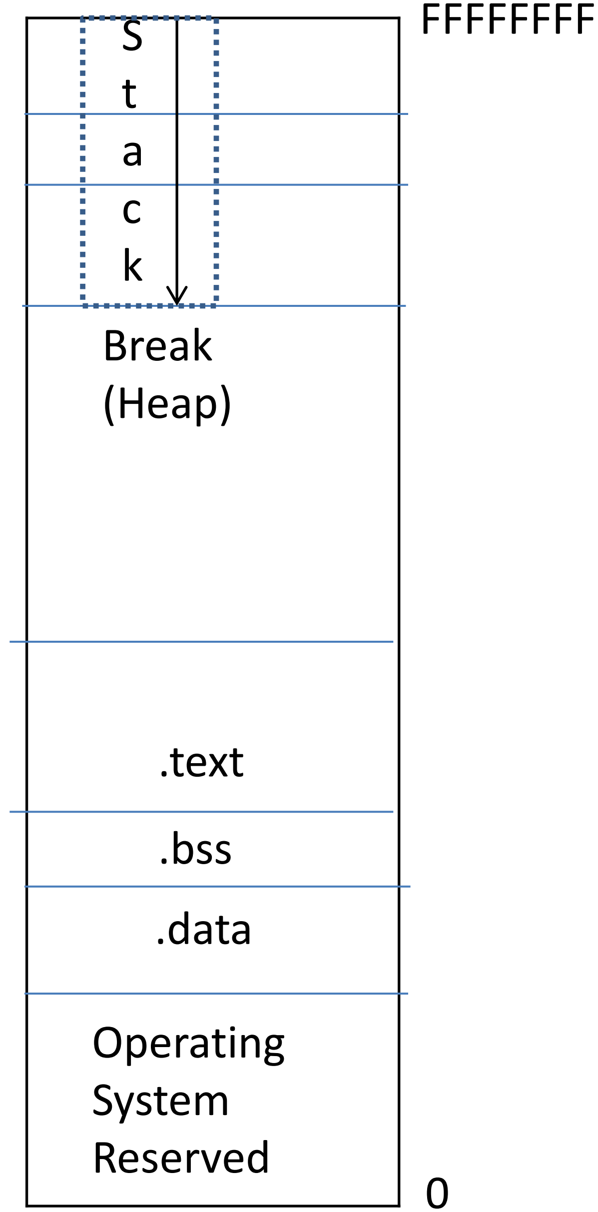
Chip Is Off (0)



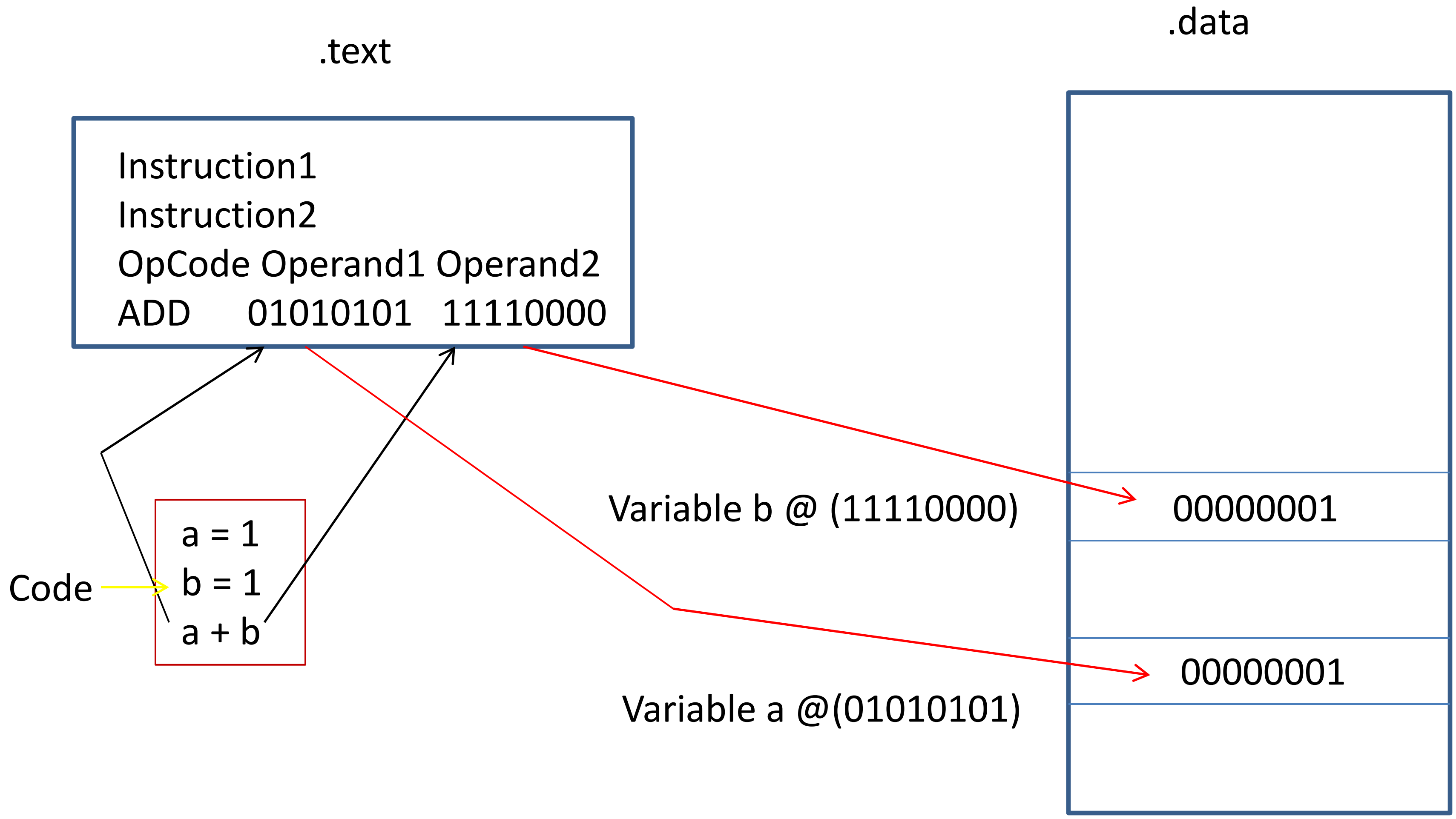
Program Internal Sections



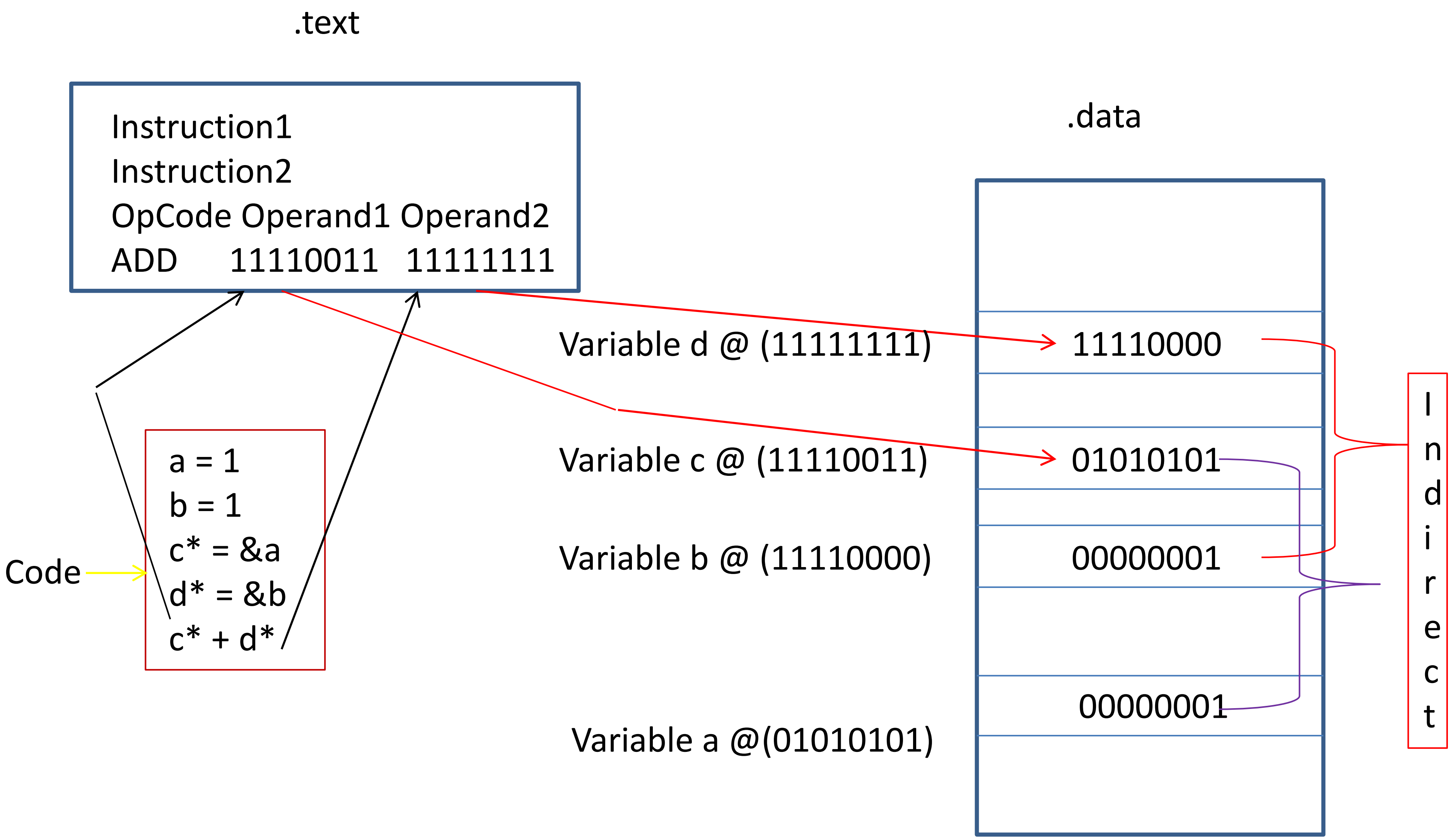
Virtual Memory Layout



How The Sections Work



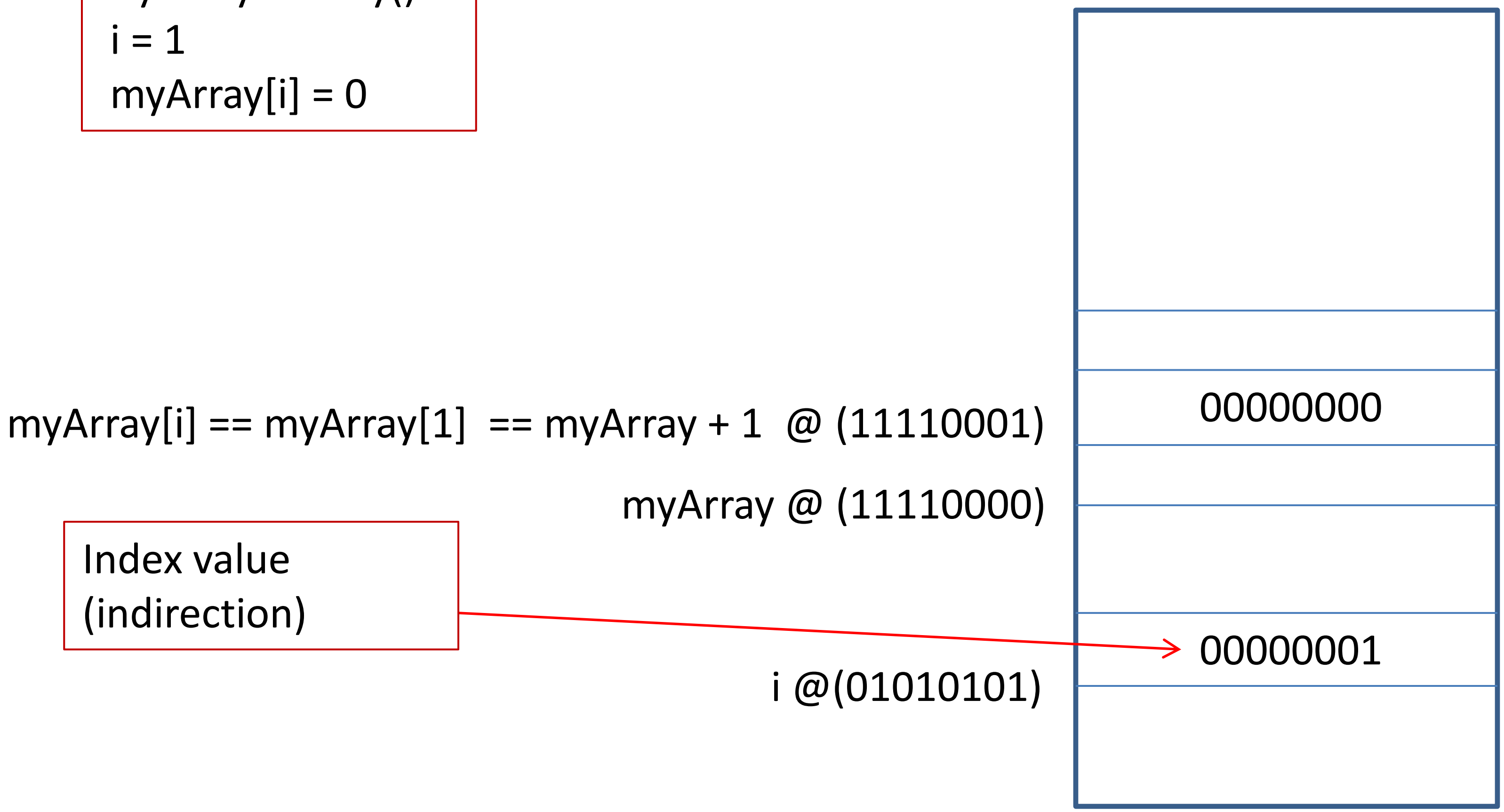
Indirection



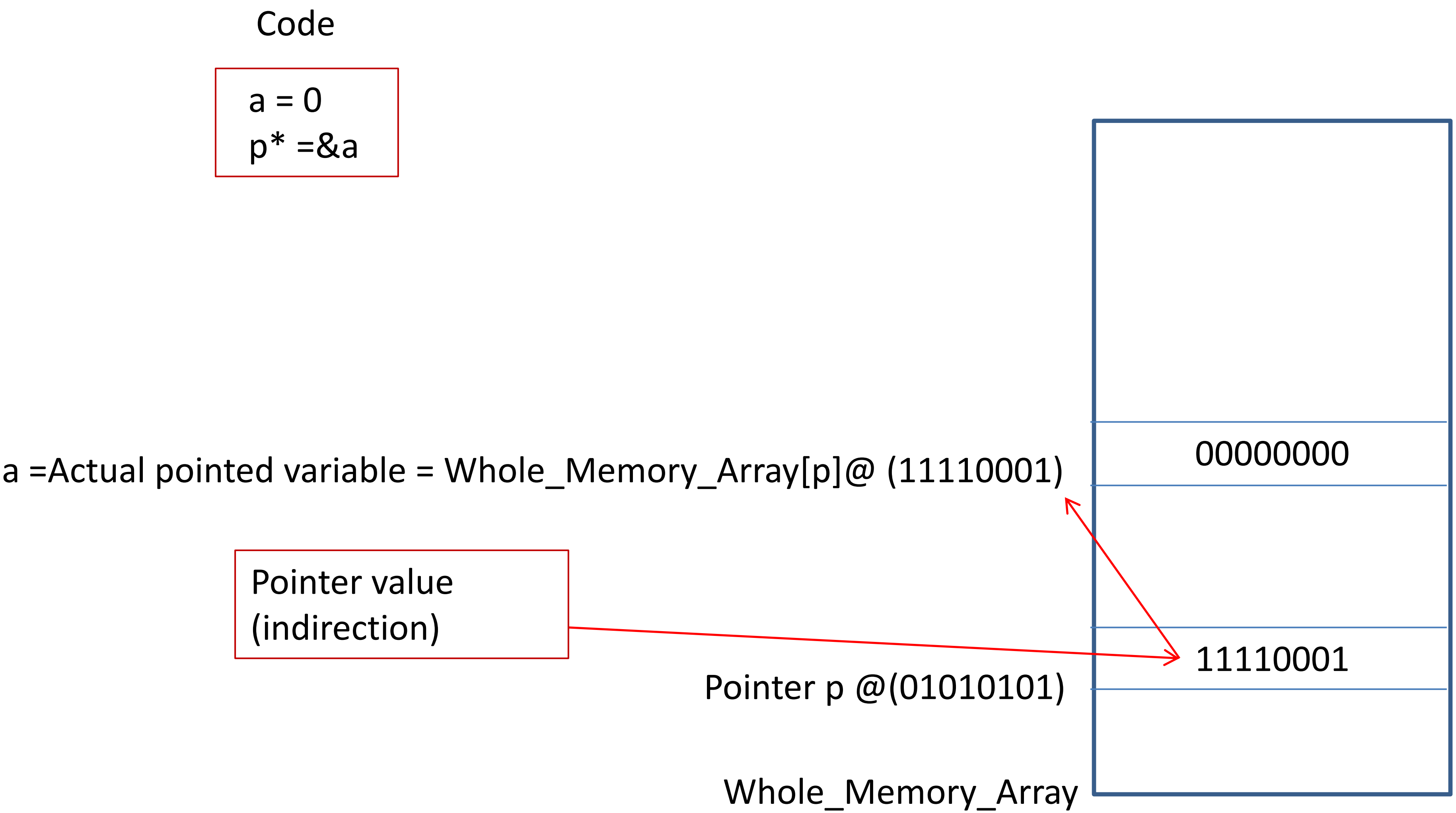
Example A “**C**” “Array”

Code

```
myArray = array()
 i = 1
 myArray[i] = 0
```

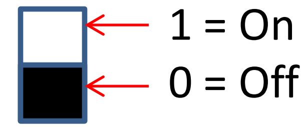


Back To Memory

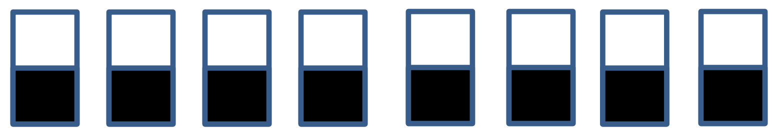


Intro To Binary

Switch = Bit



8 Switches = 8 Bits = Byte



We can have only 0 or 1
Is there any way to get real numbers?

Lets Take an Example From Regular (Decimal) Numbers

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- ?????

We have only 9 numerals
So how do we proceed?

Lets Take an Example From Regular (Decimal) Numbers

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- ?????

- 10
- 11
- 12
- …
- 20
- 21
- ….
- 30
- ….
- 99
- ?????

- 100
- 101
- ….
- 200
- …..

The answer is combination

Back To Binary

- 0
- 1
- ???

- 10
- 11
- ?????

- 100
- 101
- 110
- 111
- ??????

- 1000
- 1001
- 1010
- 1100
- 1101
- 1110
- 1111
- ??????

The answer is combination

Binary Compared To Decimal

Decimal	Binary	Octal
0	00000000	0
1	00000001	1
2	00000010	2
3	00000011	3
4	00000100	4
5	00000101	5
6	00000110	6
7	00000111	7
8	00001000	10
9	00001001	11
……	…….	……

Negative Numbers

- Rule 1: Negative has a 1 in front

  Ex: → 100000001

- Rule 2: The 2's complement

  1. The 1's Complement – Xor all bits

     Ex: 000000001 - (Decimal "1")

     Xor: 111111110

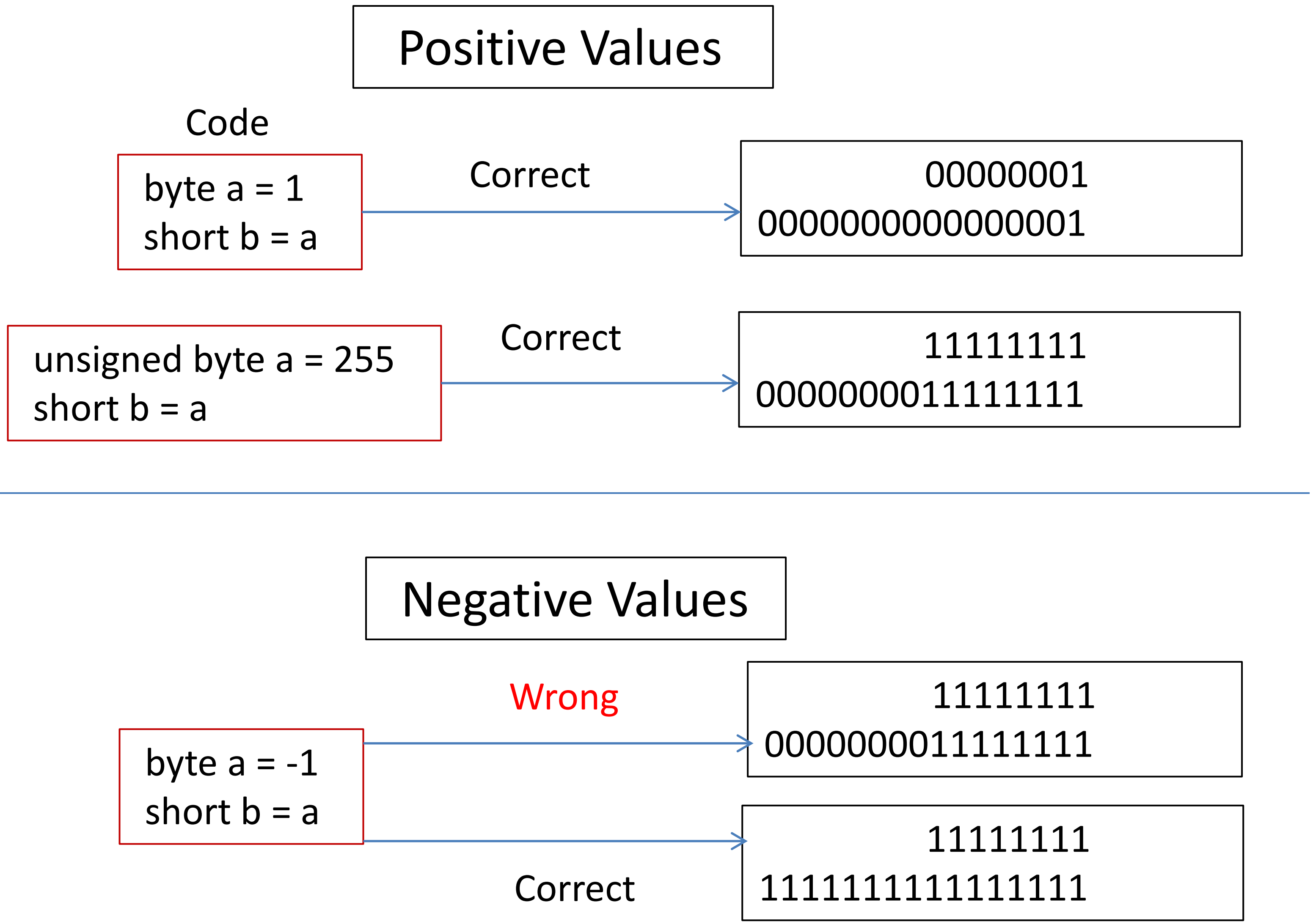
  2. The 2's Complement – Add 1

     Ex: 000000001 - (Decimal "1")

     Xor: 111111110

     Add 1: 111111110 - (Decimal "-1")

Converting Between Smaller And Larger Types



Identifiers Turned Into Memory Addresses

1. The identifiers that are being turned into memory addresses:
   - Global Variables
   - Functions
   - Labels
2. The identifiers that are NOT being turned into memory addresses, (are only used to measure the size to reserve):
   - Custom types
   - Struct
   - Class names
3. The identifiers that are used as offsets:
   - Array index
   - Class (and struct) field (NOT function) member
   - Local variables

Variables

Variables are a “higher language – human readable” name for a memory address

- Size of the reserved memory is based on the type
- There might be the following types

1. **Built-in type** – which is just a group of bytes, based on the compiler and/or platform
2. **Custom type** – which is just a series of built in types
3. **Array** (in C and C++) – which is just a series of the same built-in type (but no one keeps track of the length, so it is the programmers job)
4. **Pointer** – system defined to hold memory addresses
5. **Reference** – a pointer without the possibility to access the memory address
6. **Handle** – a value supplied by the operating system (like the index of an array)
7. **Typedef and Define** – available in C and C++ to rename existing types

Labels

Labels are a "higher language – human readable" name for a memory address

- There is no size associated with it
- Location
  - In assembly it might be everywhere
    - In fact in assembly it is generally the only way to declare a variable, function, loop, or "else"
    - In Dos batch it is the only way to declare a function
  - In C and C++ it might be only in a function – but is only recommended to break out of a nested loop
  - In VB it is used for error handling "On error goto"
  - In Java it might only be before a loop

Label Sample In Assembly

```
.data
      .int
            var1:
                  1
            var2:
                  10
.text
      .global
      start:
            mov var1 %eax
            call myfunc
            jmp myfunc
      myfunc:
            mov var2 %ebx
            add %eax %ebx
            ret
```

Label Sample In Java (Or C)

```
outerLabel:
    while(1==1)
    {
        while(2==2)
        {
            //Java syntax
            break outerLabel;

            //C syntax (not the same as
before, as it will cause the loop again)
            goto outerLabel;
        }
    }
```

Boolean Type

- False == 0 (all switches are of)
- True == 1 (switch is on, and also matches Boolean algebra)
- All other numbers are also considered true (as there are switches on)
- There are languages that require conversion between numbers and Boolean (and other are doing it behind the scenes))

However TWO languages are an exception

Why Some Languages Require conversion

- Consider the following C code, all of them are perfectly valid:

```
if(1==1) //true
if(1)    //true as well
if(a)    //Checks if “a” is non-zero
if(a==b) //Compares “a” to “b”
if(a=b)  //Sets “a” to the value of “b”, and then
         //checks “a” if it is non-zero, Is this by
         //intention or typo?
```

- However in Java the last statement would not compile, as it is not a Boolean operation

- For C there is some avoidance by having constants to the left, ie.

`if(20==a)` Instead of `if(a==20)`

Because

`if(20=a)` Is a syntax error

While

`if(a=20)` Is perfectly valid

Implicit Conversion

- However some languages employ implicit conversion
- JavaScript considers
  - `If (1)   : true`
  - `If (0)   : false`
  - `If (“”) : false`
  - `If (“0”) : true`
- `Php Considers`
  - `If (“0”) : false`
  - `If (array()) : false`

Boolean In Visual Basic

- True in VB = -1
- To see why let us see it in binary
  - 1 = 00000001
  - 1’s complement = 1111111110
  - 2’s complement = 1111111111
- So all switches are on

But why different than all others?

To answer that we need to understand the difference between Logical and Bitwise operators, and why do Logical operators short circuit?

Logical vs bitwise

- Logical
  - Step 1 – Check the left side for true
  - Step 2 – If still no conclusion check the right side
  - Step 3 – Compare both sides and give the answer
- Bitwise
  - Step 1 – Translate both sides into binary
  - Step 2 – Compare both sides bit per bit
  - Step 3 – Provide the soltuion

Example Bitwise vs Logical

- Example 1

If 1==1 AND 2==2

	Logical And	Bitwise And
Step 1:	1==1 is true	1==1 is true=00000001
Step 2:	2 ==2 is true	2==2 is true=00000001
Step 3:		
	True	00000001
And	True	00000001
	= True	00000001

More Examples

- Example 2

`If 1==2 AND 2==2`

	Logical And	Bitwise And
Step 1:	1==2 is false	1==2 is false=00000000
Step 2:	return false	2==2 is true =00000001
Step 3:	N/A	00000000 AND 00000001 00000000

- Example 3

`If 1 AND 2`

Step 1:	1 is True	00000001
Step 2:	2 is True	00000010
Step 3:	True	00000000

Bitwise vs Logical Operators

<table>
<tr><th>Operator</th><th>C</th><th>Basic</th><th>VB.Net</th></tr>
<tr><td colspan="4">(Logical)</td></tr>
<tr><td>Logical AND</td><td>&&</td><td>N/A</td><td>AndAlso</td></tr>
<tr><td>Logical OR</td><td>||</td><td>N/A</td><td>OrElse</td></tr>
<tr><td>Logical NOT</td><td>!</td><td>N/A</td><td>N/A</td></tr>
<tr><td colspan="4">(Bitwise)</td></tr>
<tr><td>Bitwise AND</td><td>&</td><td>AND</td><td>AND</td></tr>
<tr><td>Bitwise OR</td><td>|</td><td>OR</td><td>OR</td></tr>
<tr><td>Bitwise XOR</td><td>^</td><td>XOR</td><td>XOR</td></tr>
<tr><td>Bitwise NOT</td><td>~</td><td>NOT</td><td>NOT</td></tr>
</table>

Back To VB

- Since we have only a bitwise NOT we have to make sure it works on Boolean

NOT On 1

1 = 00000001 = True
NOT = 11111110 = True

NOT On -1

-1 = 11111111 = True
NOT = 00000000 = False

Beware Of The Win32 API

Boolean In Bash Shell Scripting

```
# if(true) then echo “works”; fi
# works
#
# if(false) then echo “works”; fi
#
# if(test 1 –eq 1) then echo “works”; fi
# works
#
# if(test 1 –eq 2) then echo “works”; fi
#
# echo test 1 –eq 1
#
# test 1 –eq 1
# echo $?
# 0
# test 1 –eq 2
# echo $?
# 1
#
# true
# echo #?
# 0
# false
# echo #?
# 1
```