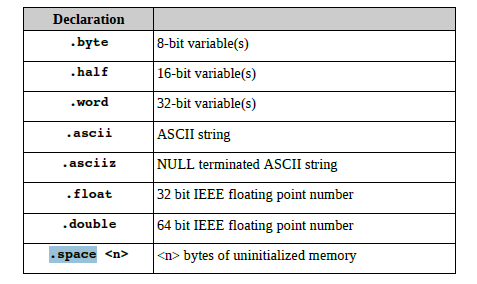
1. **Describe below in writing two basic structures used in assembly programs (as interface) to organize and access data in memory and describe the difference [**0.5 mark**].**
2. **Simple Variables**



1. Primitive Data Types

Single valued data types which are stored in memory.

X: .word 1  
Y: .float 1.0

1. Arrays

Primitive data types such as integers are single valued and cover a smaller address size. Arrays are a set of homogenous data (multiple values) of a particular length that is stored consecutively in memory.

The difference between and b is the size of the data structure; an array holds multiple data values and single data types (primitive) are much shorter in size. We can identify a particular position of an array by using an index and adding it to the address(with an offset); or we could access it by manipulating the address itself; and using an offset to determine which position in the data structure we are looking for. In this way, the addressing is slightly different, because in a primitive data structure; we only need one address to access the information.

X: .word 1,2,3,4,5

The assembler can either:

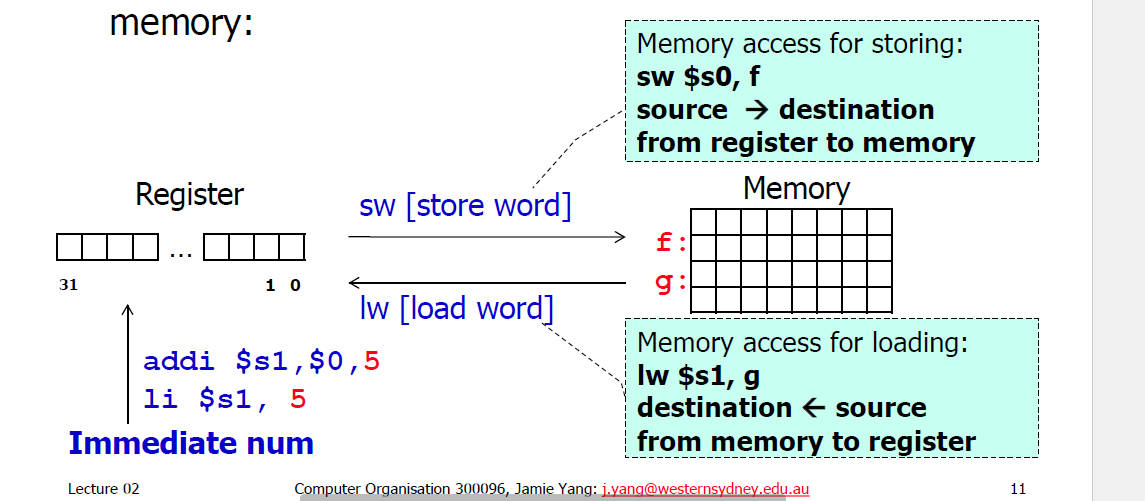
* store the information from the assembler to memory (using sw)
* load the information from memory to the assembler’s registers (using lw)

1. **Immediate Data Types**

Values which are added immediately maybe in the form of a value or of an address.   
  
The assembler can either:

* Add the value immediately to the register
* store the information from the assembler to memory (using sw)

**Difference:**

Immediate data types immediately add the value to the variables, whilst in the other way, the values or memory locations of those values are loaded and stored from memory into registers. Immediate data types are quicker since the values are not needed to be accessed and this can cause the execution process to be faster. Simple variables are also stored in memory to be accessed by the computer when it requires it.

1. **Illustrate both methods with few lines of sample code [**0.5 mark**]**
2. **Immediate Data Types:**

This can specified at the top of the program the actual arguments that these registers represent from the original program.

**addi $s1,$0,5**

**li $s1, 5**

1. **Simple Variables:**

**Example of Data Types(single 32 bit integer, or an array with 32 bit integers):**

.data

.align 2

Z: .word 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30

X: .word 0

1. Primitive Data Type **Loads address into a register, then, we can load that word into a temporary by using 0 as the offset.**

la $s2, X # $s2 has the starting address of X

lw $t1, 0($s2)

1. Array **loads base address into register, then load the offset for a particular index of the array.**  
   la $s4, Z # $s2 has the starting address of X  
   lw $s3, 0($s4) #load Z[1]

**We can see here that we can use another offset [ for instance position 3]**

la $s4, Z # $s2 has the starting address of X  
 lw $s3, 12($s4) #load Z[3]

#------------------------------------------------------------------------------------

**Here below we have a demonstration of traversing through an array data structure; and this gives an indication about how data can be accessed via the addressing methods.**

**Method for Accessing Information for Array Data Structures**

An array can be accessed and traversed using an indexing or pointer method.

1. Indexing Method:

The value of the index is multiplied by 4 and then added to the base address of the array; and then accessed (stored or loaded) afterwards.

addi $t0, $t0, 0 # I = 0

**loop:**

add $t1, $t0, $t0

add $t1, $t1, $t1 #i\*4

add $t2, $a0, $t1 #i\*4 + Arraybase

lw $t3, 0($t2)

1. Pointer Version

The base address of the array is added by four (or some other number depending on the data structure requirement) progressively, and then the data can be accessed, manipulated or organized by using this base address and the corresponding operations (or otherwise).

Addi $t0, $a0, $0 #base address

add $t1, $a1, $a1

add $t1, $t1, $t1 #size\*4

add $t2, $a0, $t1

**loop:** lw $t3, 0($t0)

addi $t0, $t0, 4 # p = p+4

slt $t4, $t0, $t2

bne $t3, $zero, loop