# 7. Data Structures: Arrays

# Data Structures

- A data structure is a means of organising many data items into an aggregate, so that we can
  - Operate on the aggregate as a whole
  - Gain access to individual data items within the aggregate
  - Arrays, stacks, lists, trees, graphs, and many more…
- A variable which just holds a single value (a number, or a character) is sometimes called a simple variable or a scalar variable.
- Simplest structure is an array.

# Arrays

- An array is a collection of data where
  - Every element has same type (e.g. integer)
  - Elements don't have individual names, but are accessed by indices
  - An index is an integer from *0, 1, ..., N-1* where *N* is the size of the array
- Examples: vectors, matrices, character strings, ...
- The elements of the array do not have individual names
  - There may be too many of them!
  - At the time the program is written, may not know how many there will be.
- Instead, we can place the data in a sequence $x_0, x_1, x_2, \ldots, x_n$ and refer to an individual element by its index.
- Notation
  - Mathematical $x_i$
  - Programming x[i]

# Representation of Arrays

- An array is held as a sequence of words in memory, with consecutive addresses.
- Suppose X is the name of the array. Then X is the label (address) of the location that contains X[0], next location contains X[1], and so on. X is the base address.
- In the Sigma16 assembly language, arrays are established as follows

Example: suppose we have an integer x with value 2, an array Y = {11,-3,4,28}, and an integer z with unspecified value. These could be defined thus…



```
x       DATA    2       ; the integer variable x
Y       DATA    11      ; this is y[0]
        DATA    -3      ; here is y[1]
        DATA    4       ; and y[2]
        DATA    28      ; and y[3], last element
z       DATA    $0000   ; the integer variable z
```

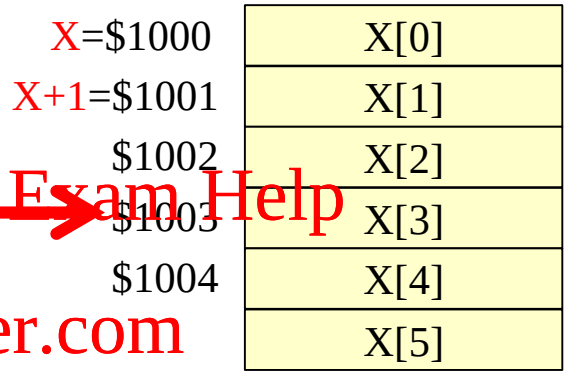
- To create an array in memory, need to allocate space for every element
  - If there are 4 elements, you need four DATA statements, even if no initial values
  - Only the first element gets a label (the name of the array)
  - A real assembly language would allow us to allocate arrays of any size with one statement (assuming elements do not need to be individually initialised).

# Effective Addresses

- How do we access x[i]?
  - Word that contains *x*[*i*] has address *x*+*i*.
  - Value of label *x* is the address of *x*[0]
- Recall: LOAD and STORE instructions specify memory address as *label*[*Rn*]
- The *effective address* (EA) is the value of *label* + contents of register *Rn*
- Lots of flexibility, for example in:
  - X[R0] the EA is X (R0 contains 0)
  - 0[R4] the EA is the contents of R4
  - X[R3] the EA is X + R3.
- Notice that to calculate the address of X[i] the CPU needs to add addresses:
  - The address where array X starts in memory, plus the value of the index I
  - This is address arithmetic performed, as usual, in binary.
  - Addresses always treated as non-negative but index can be negative. E.g. if register R1 has content $FFFF and label is $0008, label[R1] evaluates to $0007

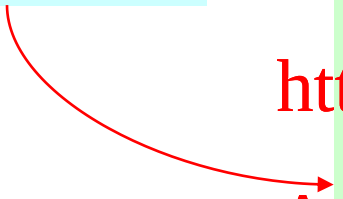
# Example

- Consider the address X[R1] where X is the base address of the array and R1 contains value 3. Suppose X=$1000. Then X[R1] will access address $1003 which contains element X[3] of the array.
- If R1 contains value 4, X[R1] is $1004 which contains X[4]



# Indexing and Effective Address

```
a = x[i];
y[i] = x[i];
```



```
; a = x[i]
        LOAD    R1,i[R0]        ; R1 = i
        LOAD    R2,x[R1]        ; R2 = x[i]
        STORE   R2,a[R0]        ; a = x[i];

; y[i] = x[i]
        LOAD    R1,i[R0]        ; R1 = i
        LOAD    R2,x[R1]        ; R1 = x[i]
        STORE   R2,y[R1]        ; y[i] = x[i]
```

# Unassessed Exercise

Example 1: Array Sum. Given an array x[0], ..., x[n-1], write an assembly language program to compute the sum of the elements.

Test the program with a 5 element array as follows:

{17, 2, 150, -3, 25}



Example 2: Array Max. Suppose an array X, of arbitrary size, contains a sequence of non-negative numbers in memory. The first negative number marks the end of the data (this representation is sometimes used for strings). Write a program to find the largest element and store it in a variable, `max`.





Test the program with a 5 element array as follows:

{2, 42, 224, 19, 4, -1}