Computer Organization and Assembly Language - Lab
Fall 2020

Faisal Khan: faisal.khan@nu.edu.pk

Modes of Addressing:

There are different modes of addressing in assembly.

1. Direct Addressing
2. Indirect Addressing
3. Register + Offset Addressing

Direct Addressing:

In direct addressing, we simply give the address of our memory location i.e. our variable.

Example 1.1:

```
num1: dw 5
num2: dw 15
sum: dw 0

1. mov ax, [num1]
2. mov bx, [num2]

3. add ax, bx

4. mov [sum], ax
```

In (1), we are not directly passing the value 5 to ax, instead we are giving the address of memory location ‘num1’ which has our data. Exactly the same is happening in (2). In (4), we are passing the sum to our variable ‘sum’ for it to be stored.

It will be seen as:

```
1. mov ax, 5
```

```
2. mov bx, 15
4. mov sum, 20
```
Example 1.2:

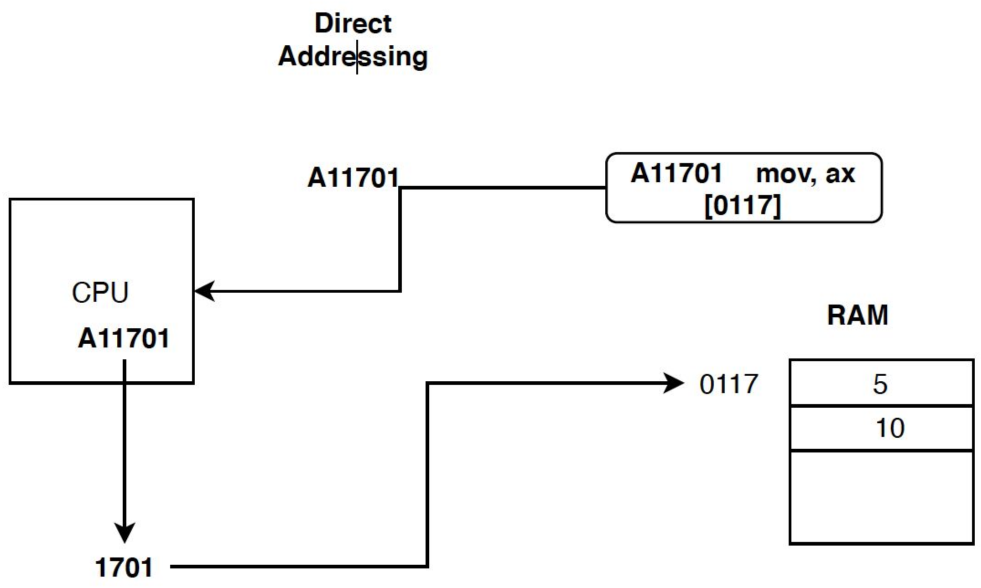
We can also use just one reference point/variable instead of multiples to do the same thing that we did in *Example 01.*

```
num1: dw 5
num2: dw 15
sum: dw 0

1. mov ax, [num1]
2. mov bx, [num1+2]

3. add ax, bx

4. mov [num1+4], ax
```

In (1) we are passing the memory location’s address so our data i.e. 5 can be moved in ax. As ‘num1’ is a double word so it needs to jump 2 memory locations ahead to get to ‘num2’. ‘num1+2’ is actually ‘num2’. In (4), ‘num1+6’ is ‘sum’ so our sum is being moved into ‘sum’ actually.
(If our variable’s datatype was ‘word’ we would have to jump ‘+4’. ‘+’ takes us to the next memory location and ‘-’ takes us to the previous memory location.)
(If the datatype was ‘db’ we would have to jump ‘+1’)

It will be seen as:

```
1. mov ax, 5
2. mov bx, 15
4. mov sum, 20
```



Indirect Addressing:

In indirect addressing, we use registers instead of variables. It’s mainly used when the data we want to work on is large and not just four to five values.

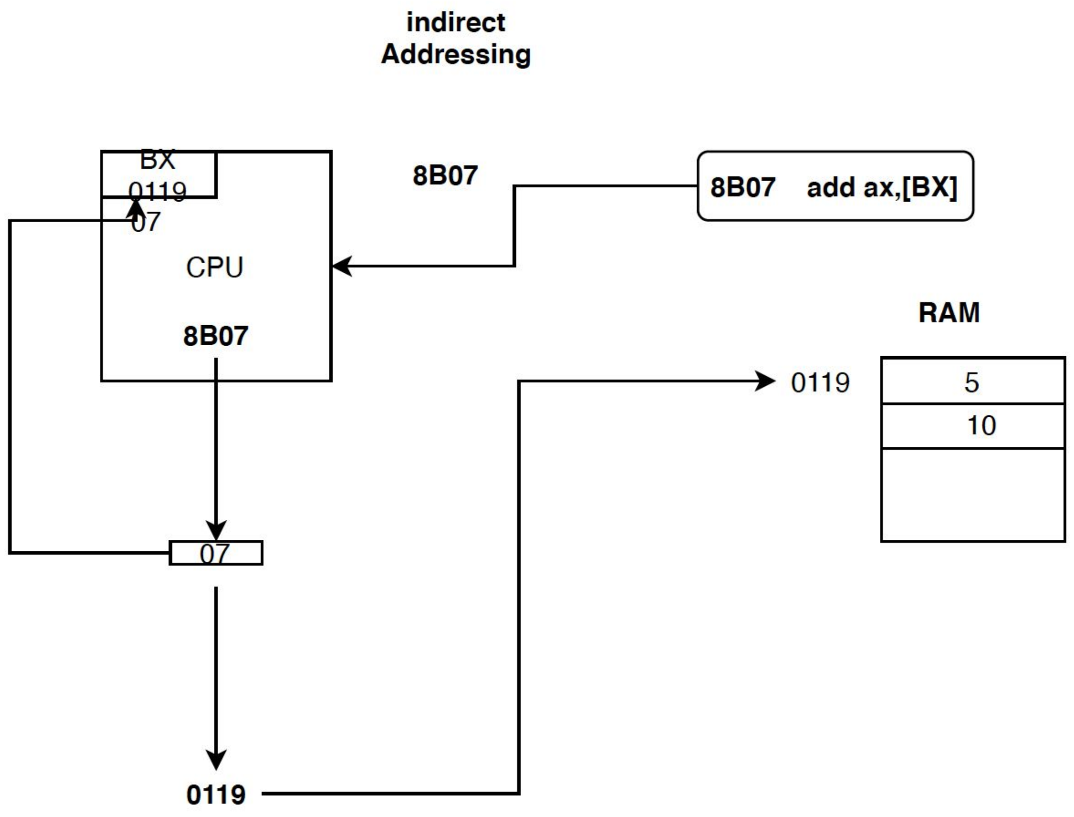
Example 2.1:

```
num1: dw 1, 2, 3, 4, 5 (...)
sum: dw 0

1. mov bx, num1
2. mov cx, 5
3. mov ax, 0

4. add ax, [bx]
5. add bx, 2
6. sub cx, 1
7. jump to (4)

8. mov [sum], ax
```

As you can tell ‘num1’ in this example is an array and the ‘...’ shows that the array can be greater than 5 elements in this example. In (1), we are moving the offset/address of num1 in bx. In (2) we are moving ‘10’ in cx. This will act like a loop.
(It’s not a good practice but since you are not familiar with loops in assembly right now this little shortcut would do.)
In (3) we are emptying our ax register because this is where the sum will take place. (There are two ways to empty a register: *‘mov eax, 0’* or *‘xor eax, eax’* but ‘mov’ is a good practice.)
In (4) we are adding the value stored in bx to ax. In (5) we are incrementing bx by 2 because our datatype of array is ‘dw’ so the jumps would be ‘+2’. In (6) we are decrementing cx by 1 which looks like we are done with a round of loop. In (7) we are going back to instruction (4) to add up the next number in ax, incrementing bx by 2 to point to next memory location/value of array, decrementing cx, jumping back to (4). This will go on and on until our cx is left with ‘0’. It will mean our loop has finished and all the data in our array has been added up in ax.
(You will become familiar with ‘jump’ instructions later)
In (8) we are moving the sum of our array in variable ‘sum’.

In first cycle of loop it’s seen as:

```
1. mov bx, (offset of num1 array)  ;bx pointing to array base i.e ‘1’
2. mov cx, 5
3. mov ax, 0  ;emptying ax

4. add ax, 1
5. add bx, 2  ;bx now pointing to ‘2’
6. sub cx, 1  ;cx = 4
7. jump to (4)

8. mov [sum], ax
```



Register + Offset Addressing:

In register + offset addressing, different combinations of direct and indirect addressing are used.

Example 3.1:

```
num1: dw 1, 2, 3, 4, 5 (...)
sum: dw 0

1. mov bx, 0
2. mov cx, 5
3. mov ax, 0

4. add ax, [num1+bx]
5. add bx, 2
6. sub cx, 1
```

```
7. jump to (4)

8. mov [sum], ax
```

You might be able to understand already what's happening in this example.
In Example 1.2, we manually incremented bx by '2'. In this Example we are doing that with the help of bx. This part is *Direct Addressing.*
In Example 2.1, bx was acting like the pointer on the array. In this Example, it's there only for incrementing. This part is *Indirect Addressing.*

In first cycle of loop it's seen as:

```
1. mov bx, 0
2. mov cx, 5
3. mov ax, 0

4. add ax, 1
5. add bx, 2  ;bx = 2
6. sub cx, 1  ;cx = 4
7. jump to (4)

8. mov [sum], ax
```

Errors:

You have to keep in mind the datatypes. In assembly, this is very crucial as your program may not run or output dummy values if wrong datatypes are used.

Options:

1. Byte is stored in 'al' or 'ah'.
2. Word is be stored in 'ax'
3. DoubleWord is stored in 'eax'

Memory to memory movements are not allowed.

Example:

```
mov [num1], [num2]
```

Some instructions are considered ambiguous which means that they are not clear enough for assemblers.

Example:

```
mov [num1], 5
```

This is because assembler doesn't know if 'num1' is a byte or a word.
To avoid this, it's advised to declare datatype with variable.
Example:

```
mov byte [num1], 5
mov word [num2], 15
```

Reading Assignment:

Pages 27 - 28

Direct, Based Register Indirect, Indexed Register Indirect, Based Register Indirect + Offset, Indexed Register Indirect + Offset, Base + Index, Base + Index + Offset.
