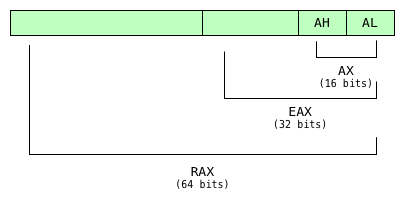
# Assembly Week 2

## Task 1: Using ‘smaller’ versions of registers

Last week we did some calculations using the e type of registers (eax,ebx etc). these are 32 bits wide. We do not always need to use arithmetic that wide, or the data we are dealing with is not that large (say a character or a single part of an RGB pixel). In this scenario we could use the smaller parts of the registers.

Quick recap:

64 32 16 8 0



For each one of our general purpose registers we can work with smaller parts of the register. Here is an example.

A close up of text on a white background

Description automatically generated

Whilst the full EAX is 0x1A2B3C4D we can access different parts and sizes of that registers and their values are shown there for AH,AH & AL. Note they are not separate registers but just smaller *parts* of the main EAX register

Load the file smallerRegisters.asm into SASM and start debugging it, showing the registers (debug menu – show registers). Single step through each line starting at line 10 and write down your prediction to what the registers will look like after executing the instruction. Do this until you hit line 19, the addition. Wait there for further orders!

|  |  |  |
| --- | --- | --- |
| **Line** | **EAX** | **EBX** |
| 10 |  | X |
| 11 |  | X |
| 12 |  | X |
| 13 |  | X |
|  |  |  |
| 15 |  |  |
| 16 |  |  |
| 17 |  |  |

When you get to the addition on line 19, write down the values that you think will be added together:

Bring up the calculator and work out what the result of the addition would be:

Now make a note of what you think the eax register will contain after the addition:

Execute the addition. Were you right? The answer and explanation are a couple of pages down.

## Task 1 Result

Note on the addition we adding BX to AX. These are 16 bit registers so we will only be dealing with the bottom 16 bits of each register.

The addition is then 0xdddd + 0xbbcc which works out to be **0x199a9**

But our destination register is only 16 bits, so the 1 is truncated off the front and 0x99a9 stored in AX

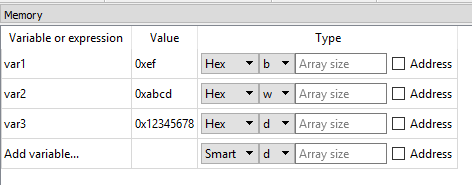
Which makes EAX equal to **0xaaaa99a9**.

So we’ve lost the most significant digit of the operation. Well kind of. If you look at the eflags register you should be able to see inside that CF is set (i.e. you can see the letters CF in the flags section). This is the carry flag. It is set when the result of an arithmetic operation overflows the size of target and so is literally the carried out most significant bit.

If you finish stepping through the code you will some PRINT\_HEX statements that we have used before, but note as we use the eax, ax and al as inputs, the size value (the first number in the statement) changes from 4 to 2 to 1 – representing the number of bytes of data we are going to output

## Task 2: Variables and Memory and Basic Addressing

Load the program variables1.asm and start to debug it (ignore the parts that say ‘add code here’ for now). Bring up the registers window and the memory window (Debug…Show Memory). Set the memory window up like this. You need to enter the variable name then change the type to Hex, and get the size right – b=byte, w=word, d=dword



You should see the value of all the variables as they have been defined in the data section.

Single step through the code and observe the variables in memory and the registers.

## Task 3: Directly interacting with Variables

Stay with variables1.asm.

At the moment we are loading a variable in from memory, adding a value to it, then writing it back out to memory. Now, in the section marked ‘add code here’ I want you to directly change the variable in memory without using registers.

For the first section (using var3), just change the value to 0xdeadbeef without using a register.

* You will probably get an assembler error the first time you try this. Refer to the powerpoint presentation section on operand sizes

For the 2nd section (using var2), add 0x4321

For the 3rd section (using var1), ‘and’ the value with 0xf  
 Try and work out what you think the value will be before trying it

For the last section (using var3, register indirect), ‘xor’ it with 0xffffffff  
 Try and work out what you think the value will be before trying it

**Note**: though we haven’t covered logical operation instructions, they are just like other mathematical operators taking a destination and a source. The instructions are **and**, **or**, **xor.**

There is another logical operator, **not** which only takes one operand and performs a bitwise NOT (each 1 is set to 0, and each 0 is set to 1)

There is no NAND, NOR instructions, you would have to execute two instructions to do this, e.g.

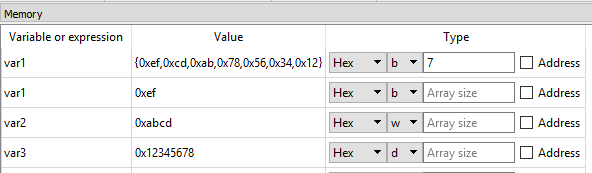
and eax,ebx  
not eax

this is the equivalent of the bitwise NAND of eax and ebx.

## Task 4: Screwing up your memory

Load variables2.asm. The start of this looks like the previous file where we define some variables

Debug and open the memory viewer like this (var1, type hex byte, 7 for array length)



At the top we can see all three variables laid out in memory



The first line of our memory debug there is showing you 7 hex bytes starting from var1. 7 because that is the size of the data we defined (var1=1 byte, var2=2 bytes, var3=3 bytes. total:7 bytes). And we can see all our variables laid out in little endian format:



var1 in red, var2 in blue and var3 in yellow.

The code is an example of what goes wrong if you do not match your machine instructions with the size of the data in memory.

Line 13 is trying to load EAX (32 bit) from var1 (which we defined as a byte) which is not a good thing to do. Seeing the layout of data in the memory, what value do you think EAX will be loaded with?

Line 14 is going to write back a 32-bit value to var1. Oops. What are the values of the variables going to be after the write happens?

var1:

var2:

var3:

Step over the code and check your results.

This is just one way your assembler code can go wrong! You are programming without a safety net!

## Task 5: Arrays

Load arrays.asm and single step debug it line by line, making sure you understand the different addressing modes. Write some notes about the way the data is accessed in each part. Refer to lecture notes if stuck.

## Task 6: Arrays and loops

Load arraysAndLoops.asm

For this task I want you create an array of 5 dwords, put in any value you like for these between 1 and 100.

You can see I have created an equ called arraySize with the value 5. We will use this to count our loops instead of hardcoding 5.

Create a loop that will iterate over the elements of the array and total them up, then write out the result.

Check your result

## Task 7: Switch Statement

Load up switch.asm

This is a bit different. It calls GET\_CHAR to read a character from the input box in this case into AL. Implement the following switch statement:

switch (al) {  
 case ‘A’:  
 print “Alpha”  
 break  
 case ‘B’:  
 print “Bravo”  
 break  
 case ‘C’:  
 print “Charlie”  
 break

default:  
 print “no match”;  
}

To make this work, when you run or debug it you will need to enter a letter into the input box before running the code or GET\_CHAR will return 0

