Take Home Test 1: Comparison ISA

Shirong Zheng
Professor Izidor Gertner
March 19, 2017
CSC34200-G Spring 2017

Contents

❖1.Objective

The objective of take home test 1 is going to using the assembly code and C++ code which provide by the professor to analysis in MIPS (assembly compiler which use MARS based on the Java JDK), Microsoft Visual Studio in 32 bit Window(debugger analysis) and GDB/GCC in 64 bit LINUX compiler. This process we called comparison ISA.

❖2.Part I Analysis in MIPS

The below is the 2-2_1 assembly code. That assign these values 1-5 to variables a-e. Then assign the assembly character, $s0-$s4 to theses variables. In the Add part, “add $s0, $s1, $s2” means “a=b+c”. In the Subtraction part, “sub $s3, $s0, $s4” means “d=a-e”. Then store word(sw) of each operation. The lw means load word.

MIPS comes with 32 general purpose registers named $0. . . $31.

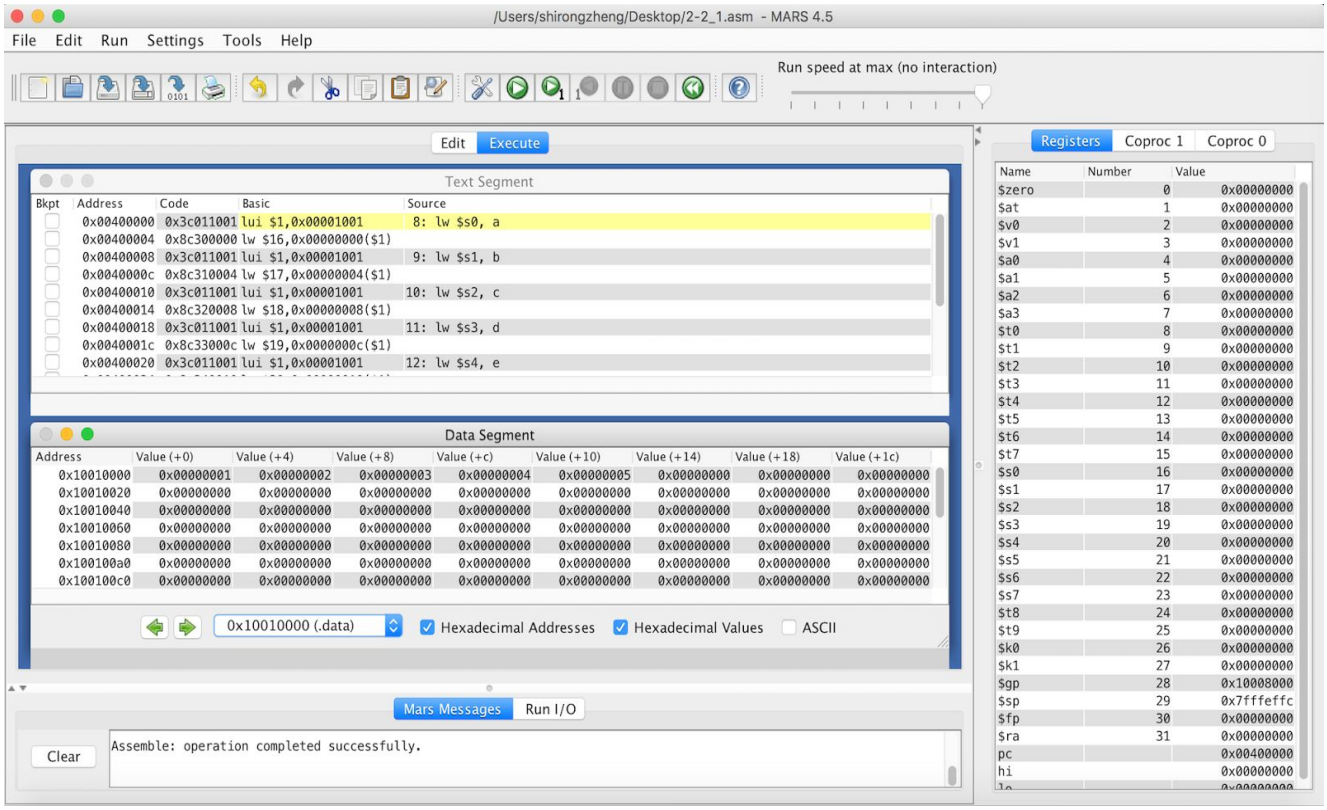
2-2_1.asm

```
1   .data
2   a: .word 1
3   b: .word 2
4   c: .word 3
5   d: .word 4
6   e: .word 5
7   .text
8   lw $s0, a
9   lw $s1, b
10  lw $s2, c
11  lw $s3, d
12  lw $s4, e
13  # a = b + c
14  add $s0, $s1, $s2
15  sw $s0, a
16  # d = a - e
17  sub $s3, $s0, $s4
18  sw $s3, d
```

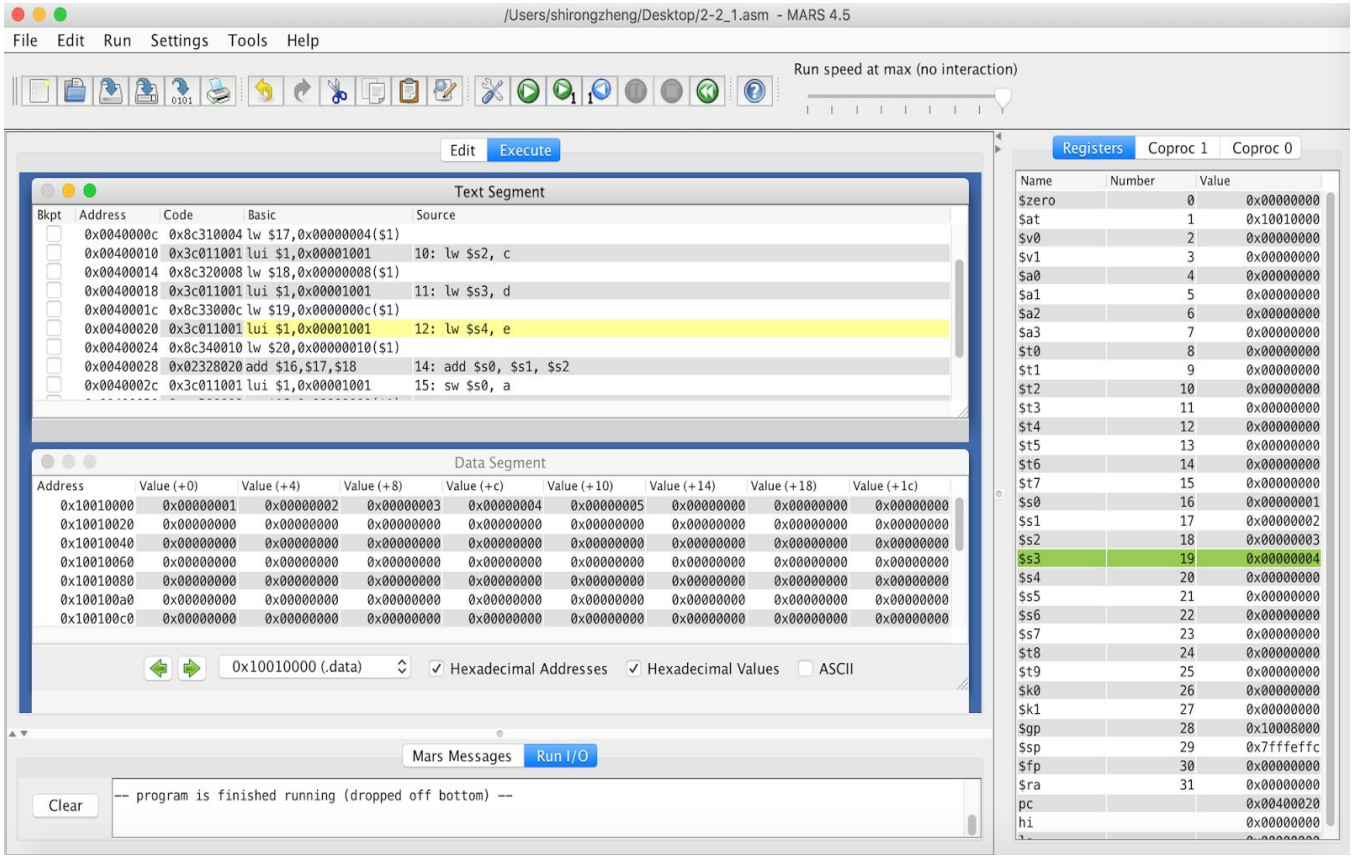
2-2_1.asm

The below is my screenshot after I compile my 2-2_1.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate.

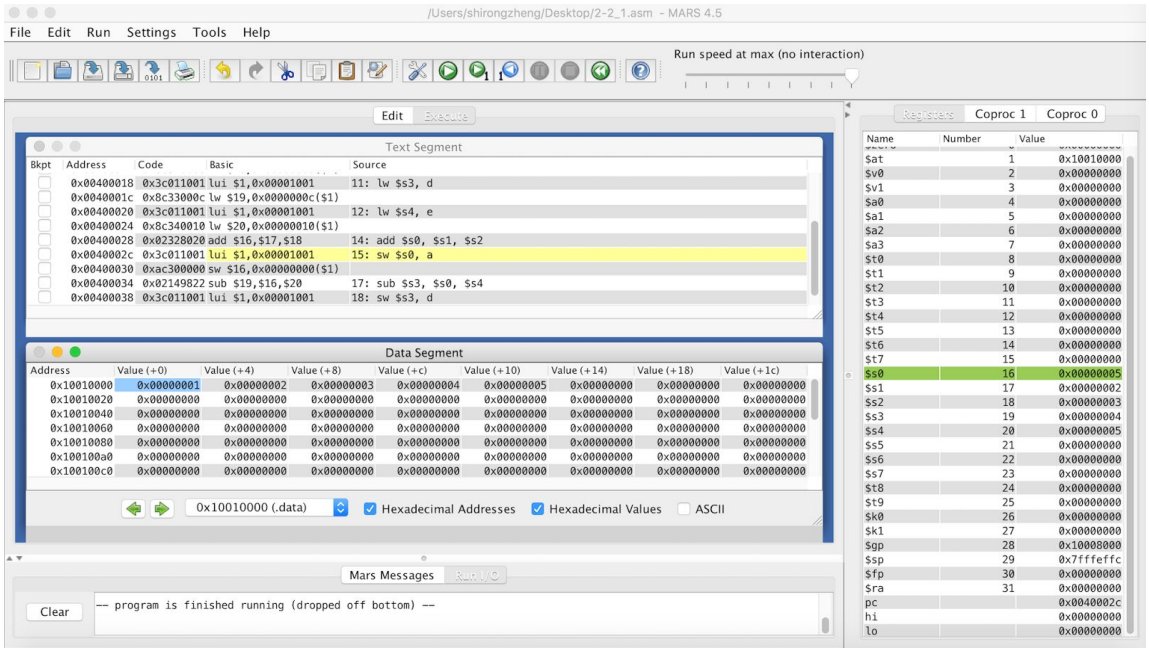
The address of variable “a” is 0x10010000 which has the decimal value of 1. The address of variable “b” is 0x10010004 which has the decimal value of 2. The address of variable “c” is 0x10010008 which has the decimal value of 3. The address of variable “d” is 0x1001000c which has the decimal value of 4. The address of variable “e” is 0x10010010 which has the decimal value of 5. (Follow the Address 0x1001000, then add values +0, +4,+8,+c, +10. That we could say the address of going from 0x1001000 to 0x10010010).



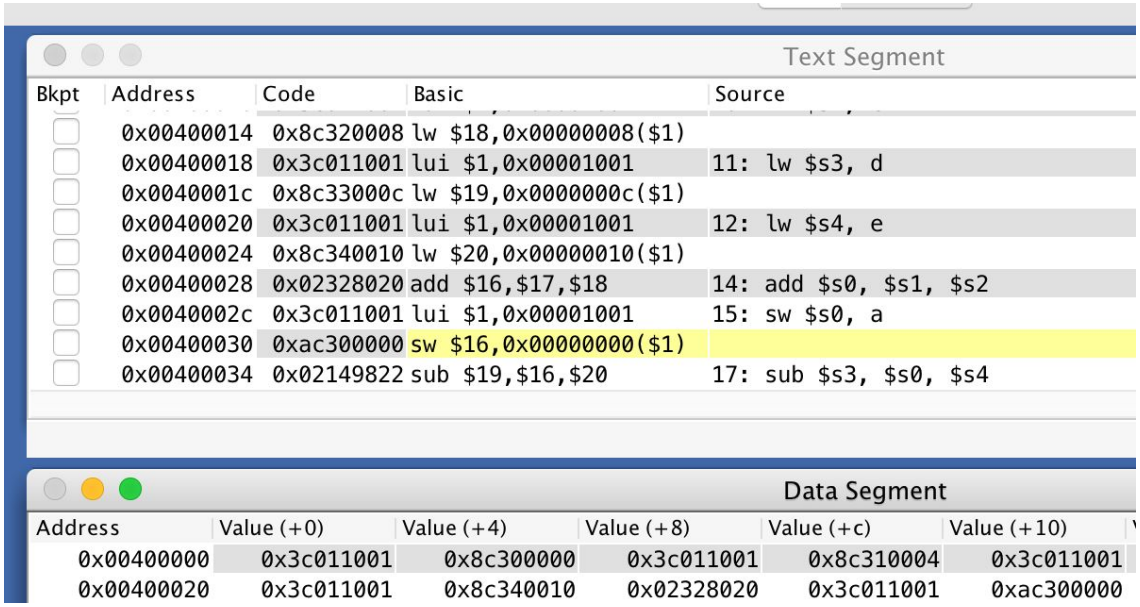
In registers, All values from $s0-$s4 are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text is 0x0040020 in e.



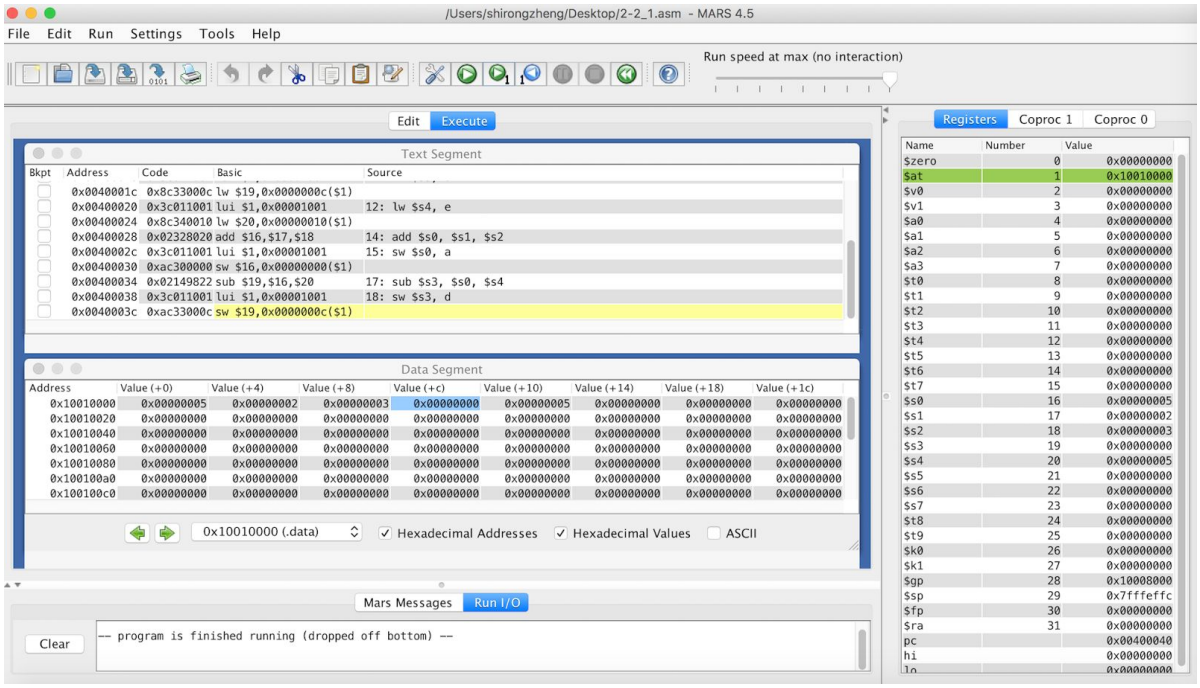
$s1 and $s2 represent variable b and c, $s0 is a. The value of $s1 and $s2 is 2 and 3. Then convert to register value is 0x00000002 and 0x00000003. The $s0(a) is 0x00000005.



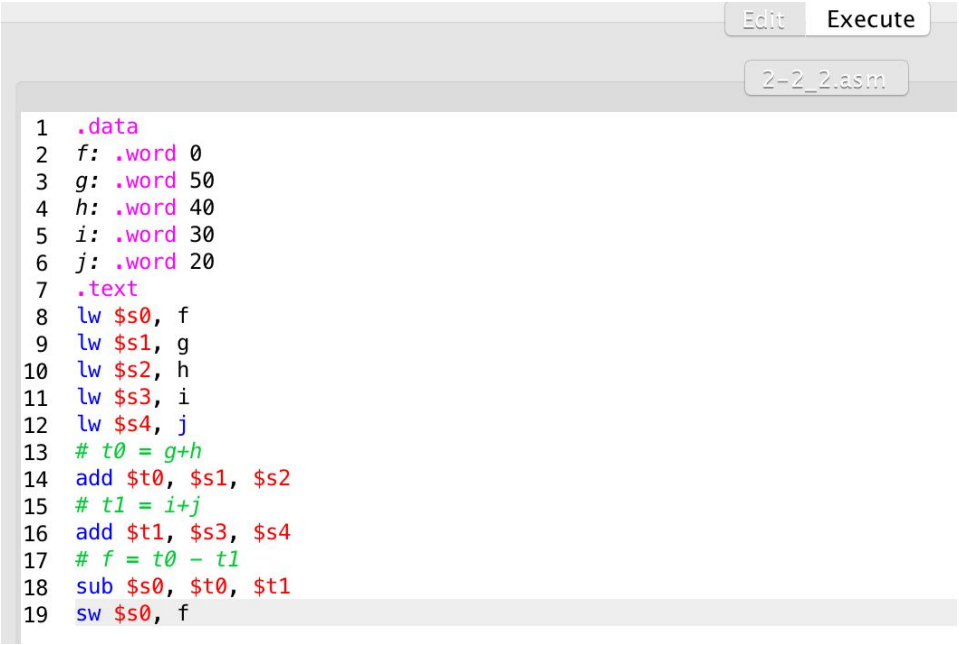
After compile and run it, then new $16 was assign to the $s0, the original address is 0x00400020, then add value +10 is 0xac300000. It store the address as the static values.



In the Subtraction part, the new value of $s0(a) is 5, then $s4(e) is 5. 5-5=0, then the value is 0x00000000. Then the register value 0x10010000 will convert to 0 value.



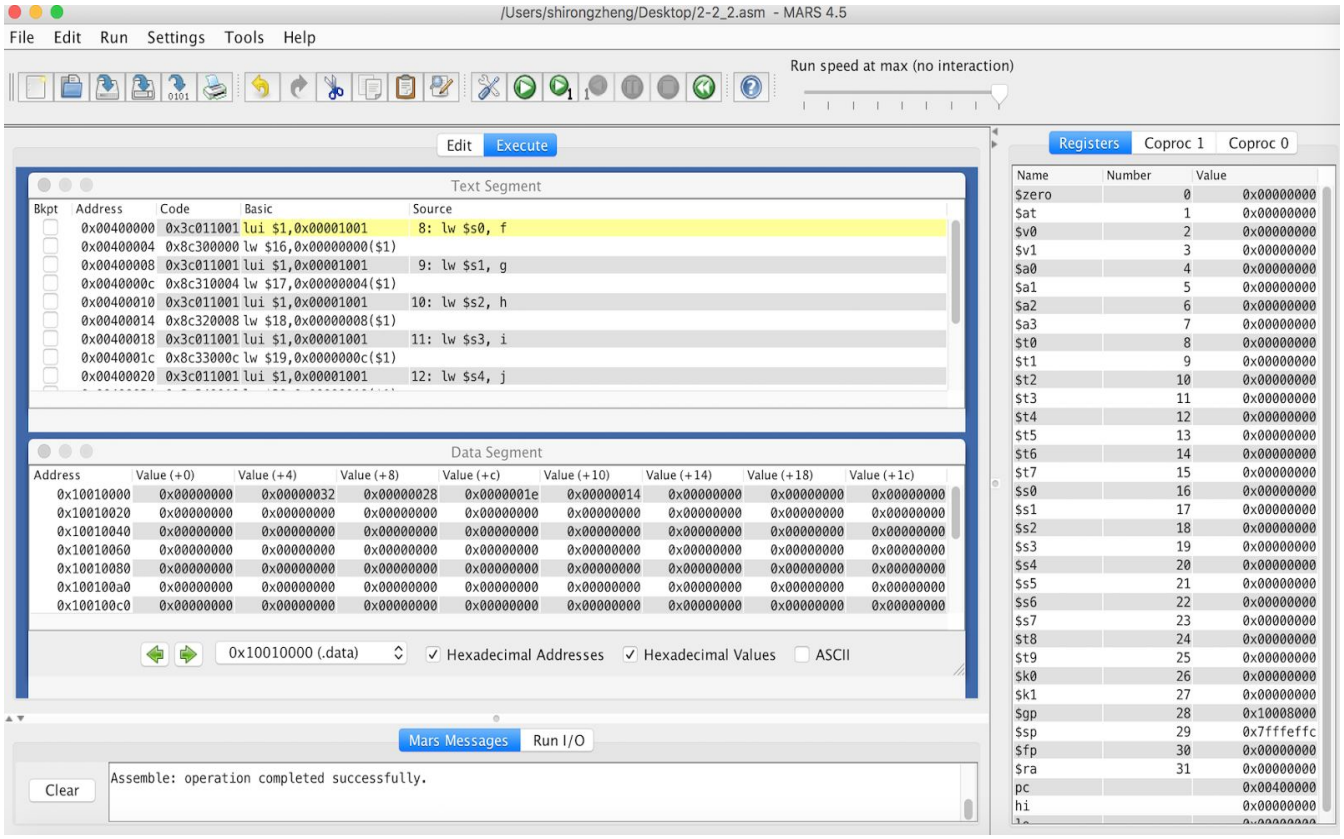
The below is my 2-2_2.asm code. I assign values 0, 50, 40, 30, 20 to variables f, g, h, i, j. Then assign the assembly character, $s0-$s4 to theses variables. In the Add part, “add $t0, $s1, $s2” means “t0=g+h”. In other Add part, “sub $t1, $s3, $s4” means “t0=i+j”. In the final Subtraction part, “sub $s0, $t0, $t1 means “f=t0-t1”.Then store word(sw) of each operation. The lw means load word.



2-2_2.asm

The below is my screenshot after I compile my 2-2_2.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate.

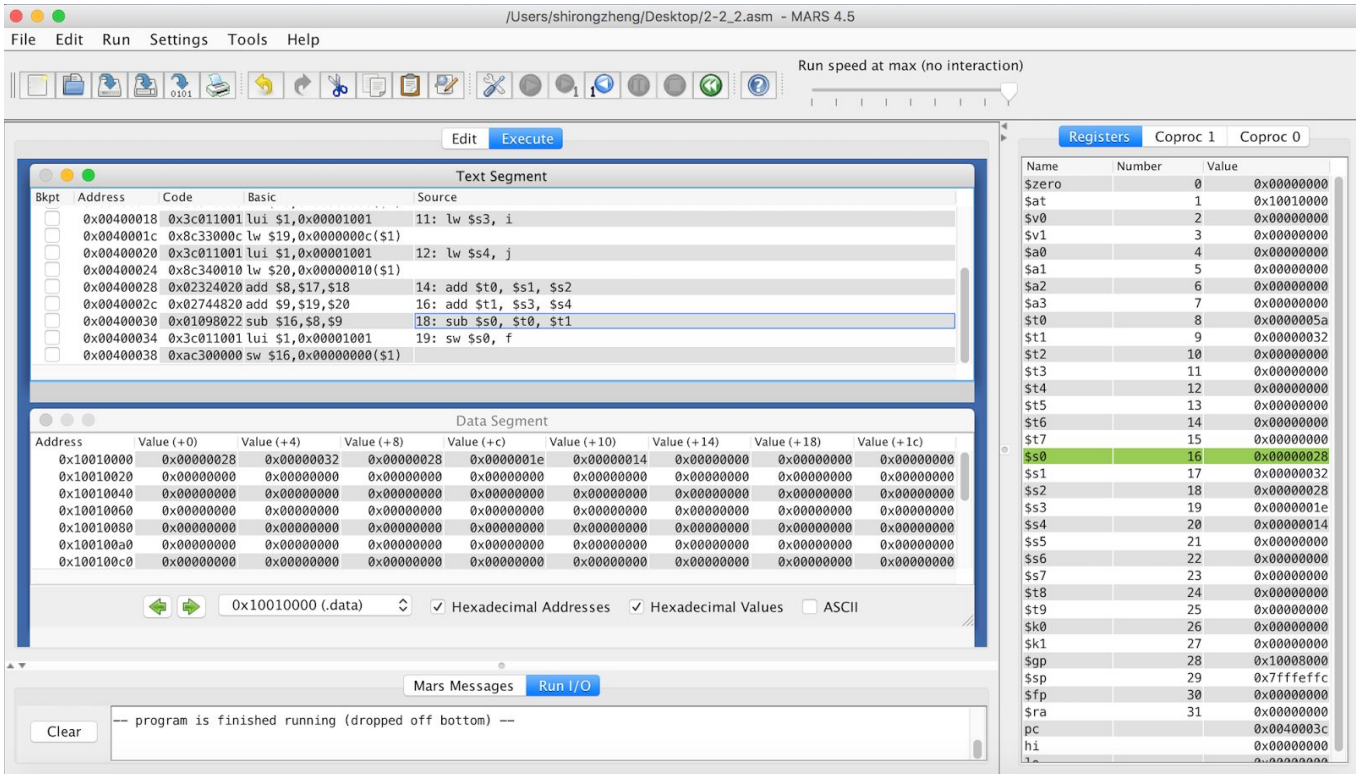
The address of variable “f” is 0x10010000 which has the decimal value of 0. The address of variable “g” is 0x10010004 which has the decimal value of 50. The address of variable “h” is 0x10010008 which has the decimal value of 40. The address of variable “i” is 0x1001000c which has the decimal value of 30. The address of variable “j” is 0x10010010 which has the decimal value of 20. (Follow the Address 0x1001000, then add values +0, +4,+8,+c, +10. That we could say the address of going from 0x1001000 to 0x10010010).



In registers, All values from $s0-$s4 are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text.

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Let's say $t0=$s1+$s2, which is 0x00000032+0x00000028=0x00000060. $t1=$s3+$s4, then the sum is 0x00000032. We use $t0-$t1=0x00000028, this is the memory address of static variable in the register $s0. The $at is the instruction to load the address of static variable to number 1(0x10010000). It will add the value +8 to this address, (0x10010008).



The below is my 2-3_1.asm code. I assign values 0, 22, 0-100, 100to variables g, h, A, size. Then store word(sw) of each operation. The lw means load word, then la means load address.

2-3_1.asm

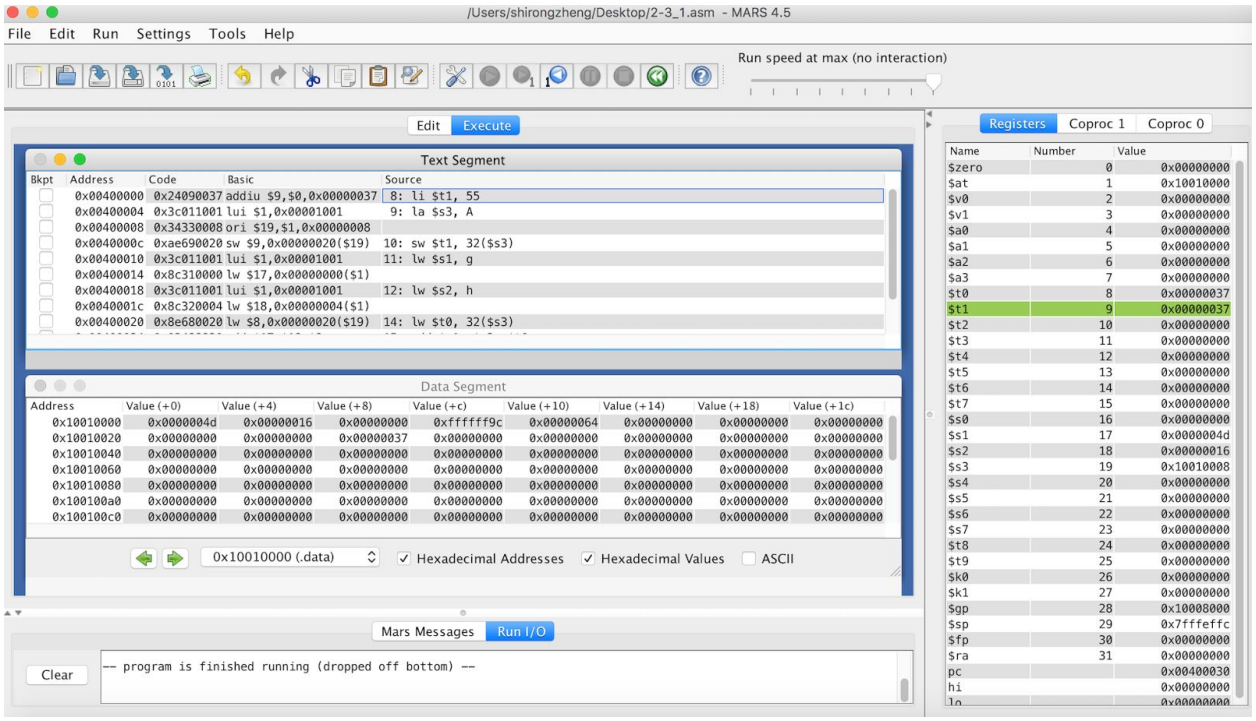
```
1   .data
2  g: .word 0
3  h: .word 22
4  A: .word 0-100
5  size: .word 100
6  .text
7  #just to set A[8] to 55
8  li $t1, 55
9  la $s3, A
10 sw $t1, 32($s3)
11 lw $s1, g
12 lw $s2, h
13 #loading the value of A[8] into t0
14 lw $t0, 32($s3)
15 add $s1, $s2, $t0
16 sw $s1, g
```

2-3_1.asm

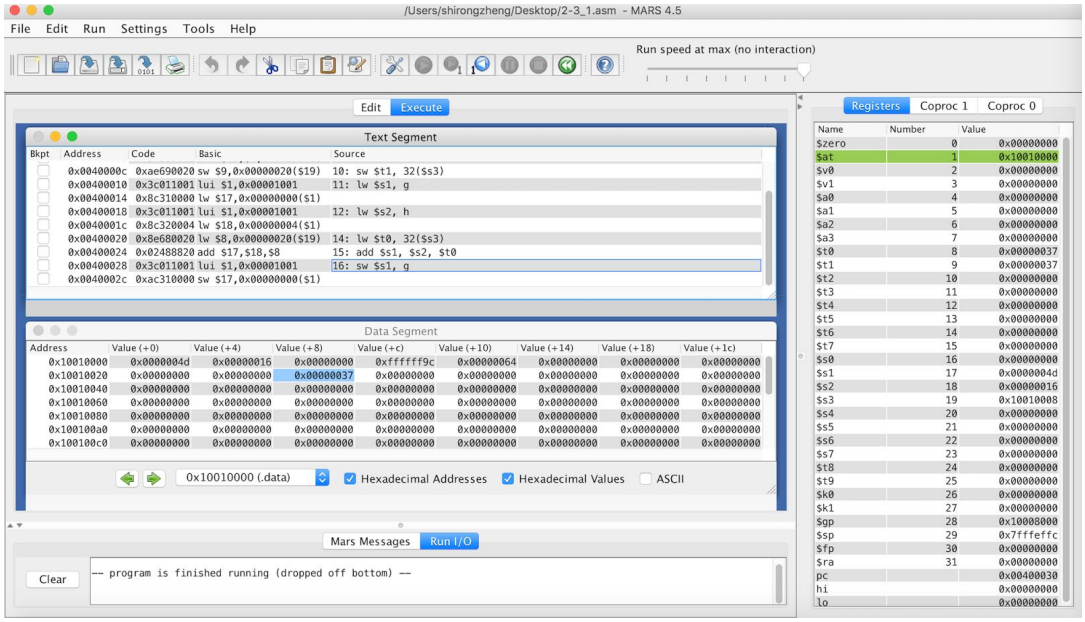
The below is my screenshot after I compile my 2-3_1.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate. The li means load immediate. That set the array A[8] to 55.

The address of variable “g” is 0x10010000 which has the decimal value of 0. The address of variable “h” is 0x10010004 which has the decimal value of 22. The address of array “A” is 0x10010008 which has the decimal value of 0. The address of variable “size” is 0x1001000c has the decimal value of 100. (Follow the Address 0x1001000, then add values +0, +4,+8,+c. That we could say the address of going from 0x1001000 to 0x1001000c). In registers, All values

are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text, which is 0x0040002c.



This base address of array “A” will be stored in $s3. From the register, the value $s3 had the value of 0x10010008. We use math way to add $s2 and $s0, then store it as $s1. In math: $s1=$s2:0x00000016+$s0:0x00000000, and the result of $s1 is 0x00000016. I could get the same answer after I compile it.



The below is my 2-3_2.asm code. I assign values 25, 0-100, 100 to variables h, A, size. The A is array in this code. Then store word(sw) of each operation. The lw means load word. The li means load immediate. The la means load address.

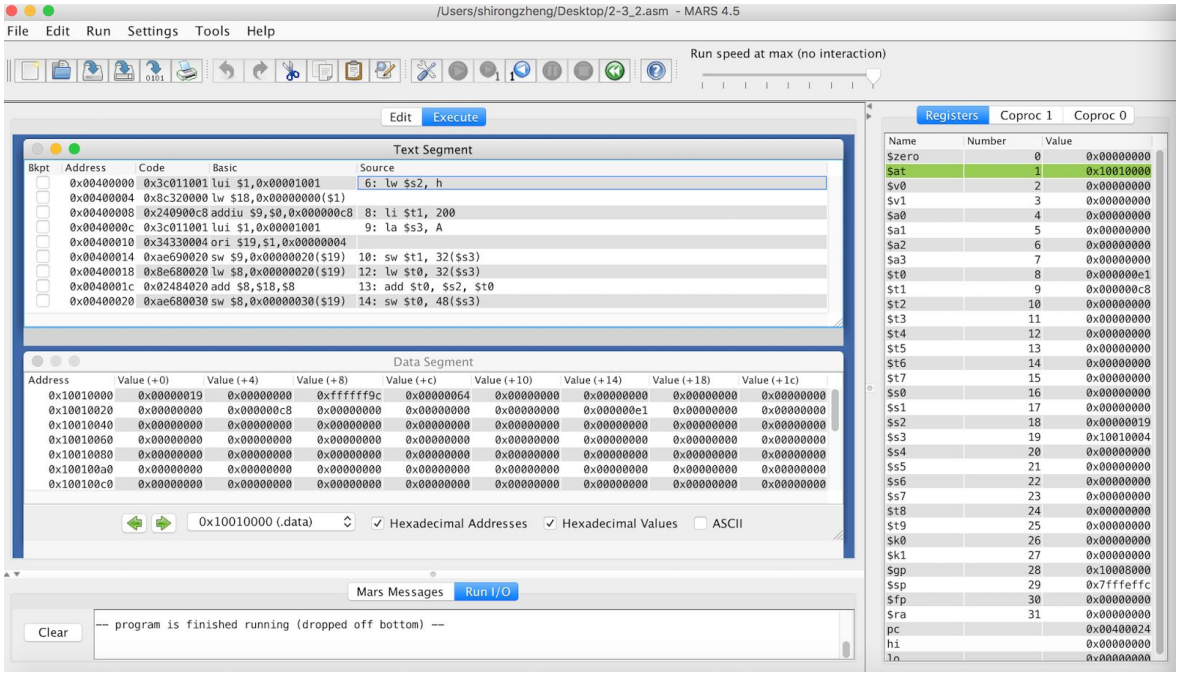
2-3_2.asm

```
1  .data
2  h: .word 25
3  A: .word 0-100
4  size: .word 100
5  .text
6  lw $s2, h
7  #initializing A[8] to 200
8  li $t1, 200
9  la $s3, A
10 sw $t1, 32($s3)
11 #A[12] = h + A[8]
12 lw $t0, 32($s3)
13 add $t0, $s2, $t0
14 sw $t0, 48($s3)
```

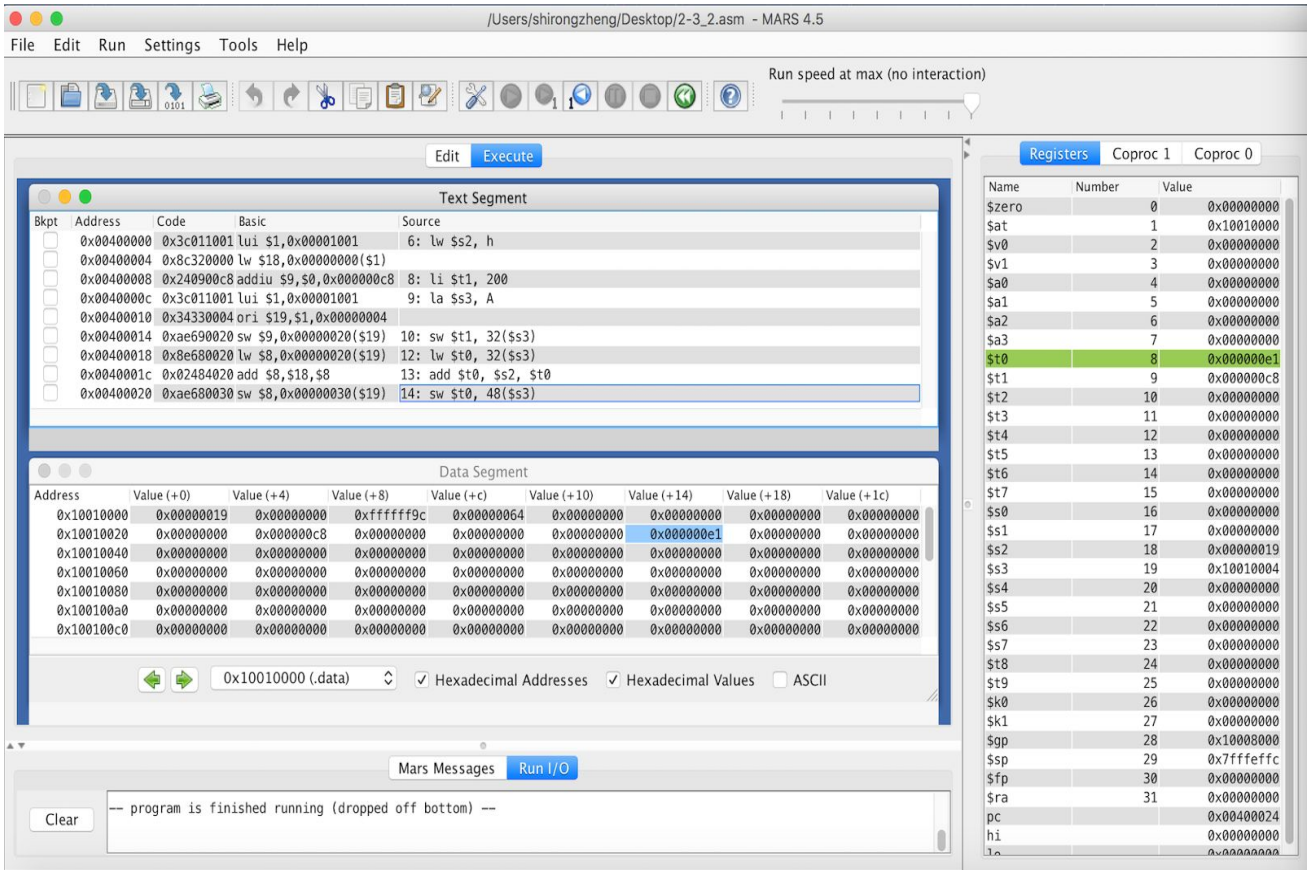
2-3_2.asm

The below is my screenshot after I compile my 2-3_2.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate. The li means load immediate. That set the array A[8] to 200.

The address of variable “h” is 0x10010000 which has the decimal value of 0. The address of array “A” is 0x10010004 which has the decimal value of 0. The address of variable “size” is 0x1001000c has the decimal value of 100. (Follow the Address 0x1001000, then add values +0, +4,+8,+c. The value+8 will be assign empty, then it’s address is 0xffffff9c. That we could say the address of going from 0x1001000 to 0x1001000c). In registers, All values are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text, which is 0x00400024.



The sw $t0= memory [$s3+48]. That add $s2 and $t0, the sum is $t0. In math part, 0x00000019 + $t0: 0x000000c8 = 0x000000e1. When the system load $s which has address 0x00001001. The $s3 has address 0x00000004 and which located in array A[5]. The $s3 is 0x10010004 in register. The base address is memory[A+32] which is A + 32 and the corresponding address is 0x10010004 + 32 = 0x10010024. We say it is located in the $t0 in line 12 of the code.



The below is my 2-5_2.asm code. I assign values 20, 0-400, 400 to variables h, A, size. The A is array in this code. Then store word(sw) of each operation. The lw means load word. The li means load immediate. The la means load address.

2-5_2.asm

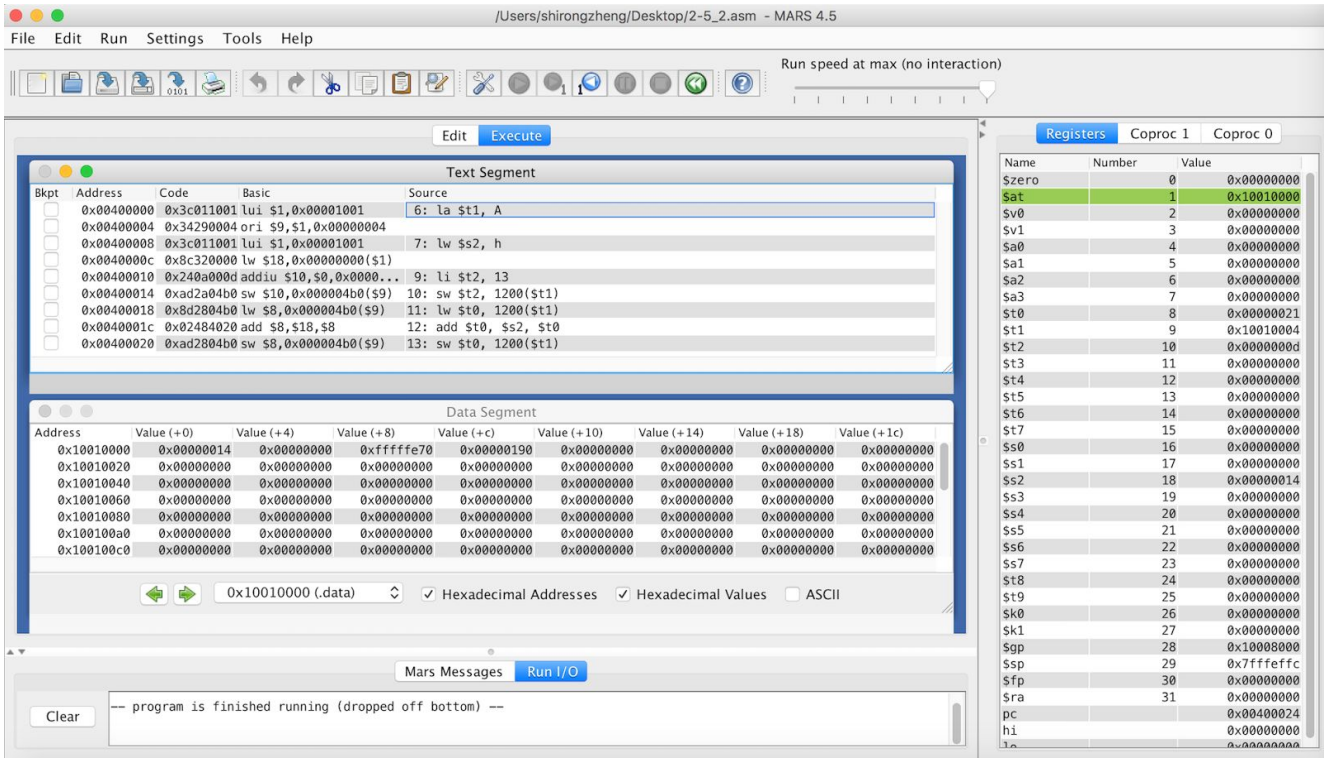
```
1  .data
2  h: .word 20
3  A: .word 0-400
4  size: .word 400
5  .text
6  la $t1, A
7  lw $s2, h
8  #initialzing A[300] to 13
9  li $t2, 13
10 sw $t2, 1200($t1)
11 lw $t0, 1200($t1)
12 add $t0, $s2, $t0
13 sw $t0, 1200($t1)
```

2-5_2.asm

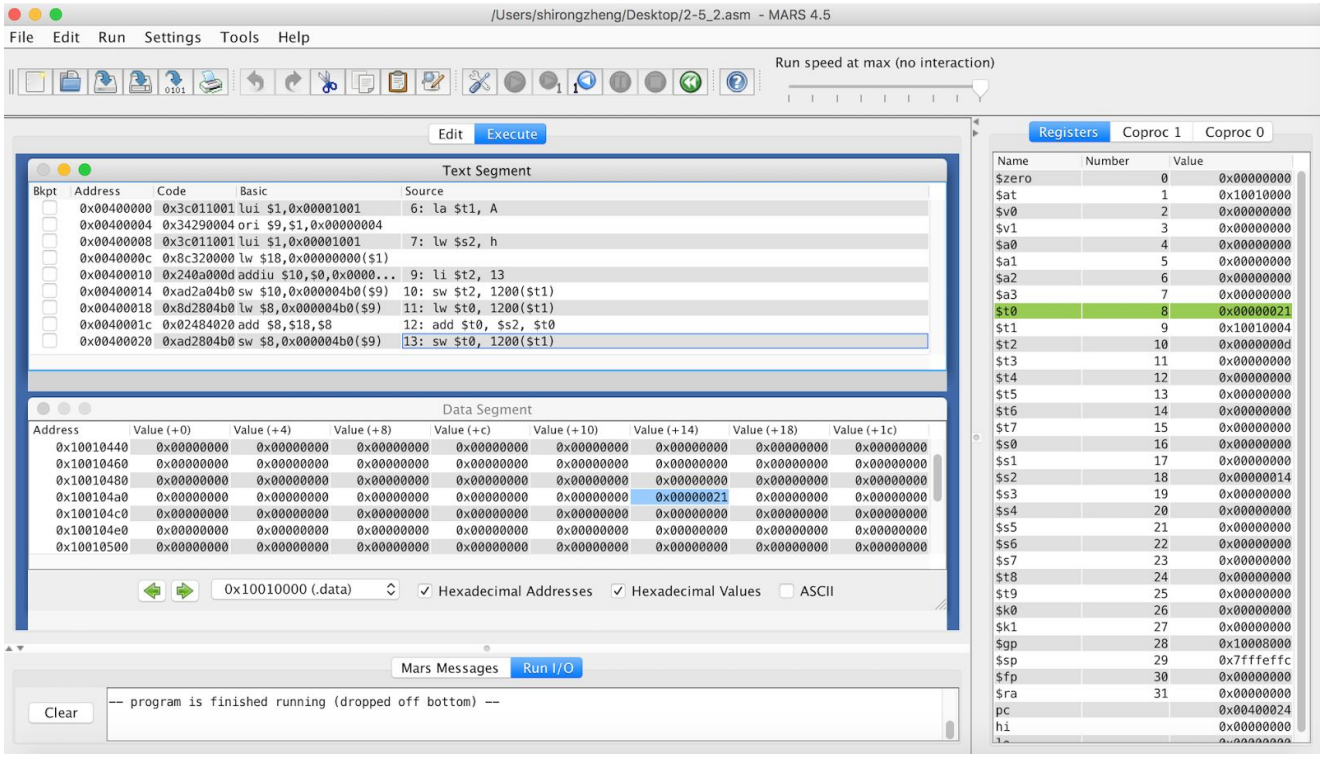
The below is my screenshot after I compile my 2-5_2.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate. The li means load immediate. That set the array A[300] to 13.

The address of variable "h" is 0x10010000 which has the decimal value of 0. The address of array "A" is 0x10010004 which has the decimal value of 0. The address of variable "size" is 0x1001000c has the decimal value of 100. (Follow the Address 0x1001000, then add values +0, +4,+8,+c. The value+8 will be assign empty, then it's address is 0xfffffe70. That we could say the address of going from 0x1001000 to 0x1001000c). In registers, All values are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to

the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text, which is 0x00400024.



The sw $t0= memory [$s1+1200]. The program add $s2 and $t0, the new memory address will store in the new $t0. The $t0+$s2 is 0x0000000d + 0x00000014 = 0x00000021. The new $t0 has this value stored in the system after I compile it. Try to verify in the register. In the final process, we change the array A[30] to 0x00000021.



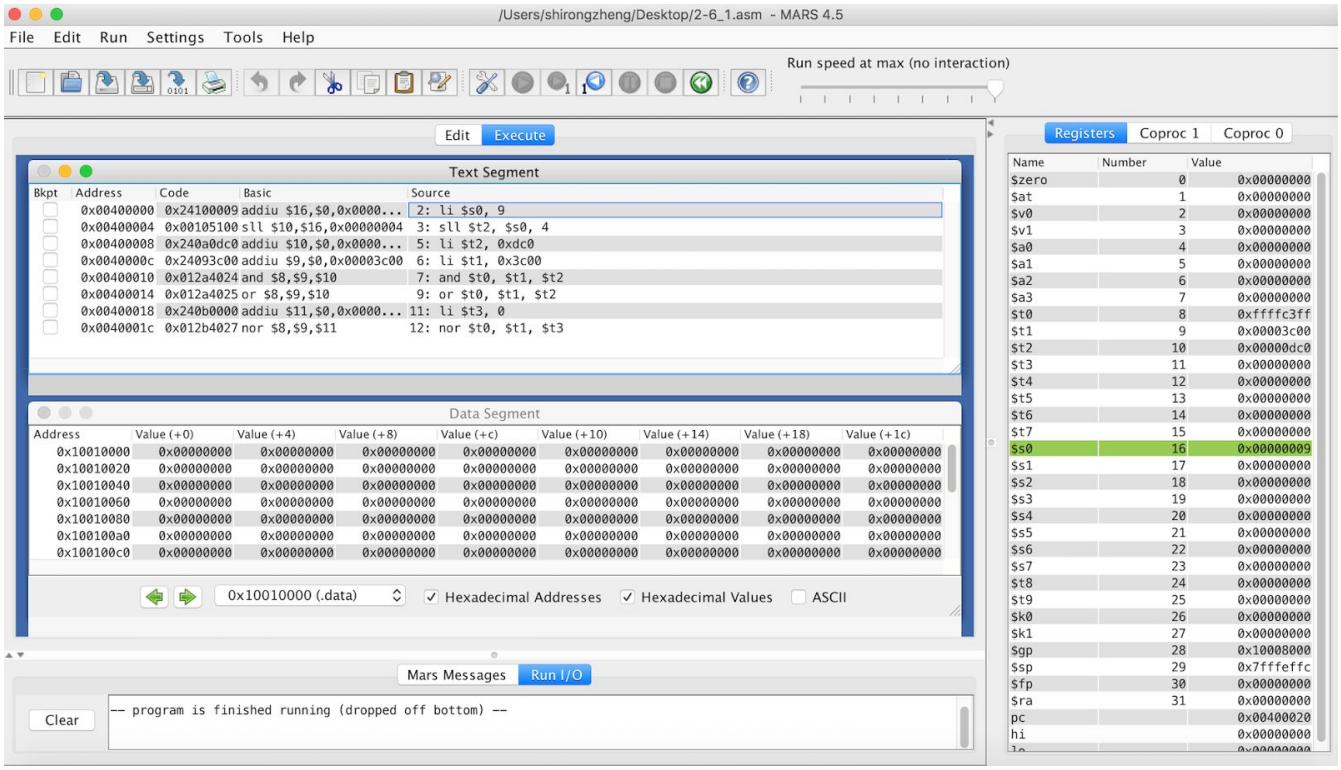
The below is my 2-6_1.asm code. The li means load immediate. “sll $t2, $s0, 4” means $t2=$s0<<4. “ and $t0, $t1, $t2” means $t0=$t1 & $t2. “nor $t0, $t1, $t3” means $t0 ~($t1 | $t3).

2-6_1.asm

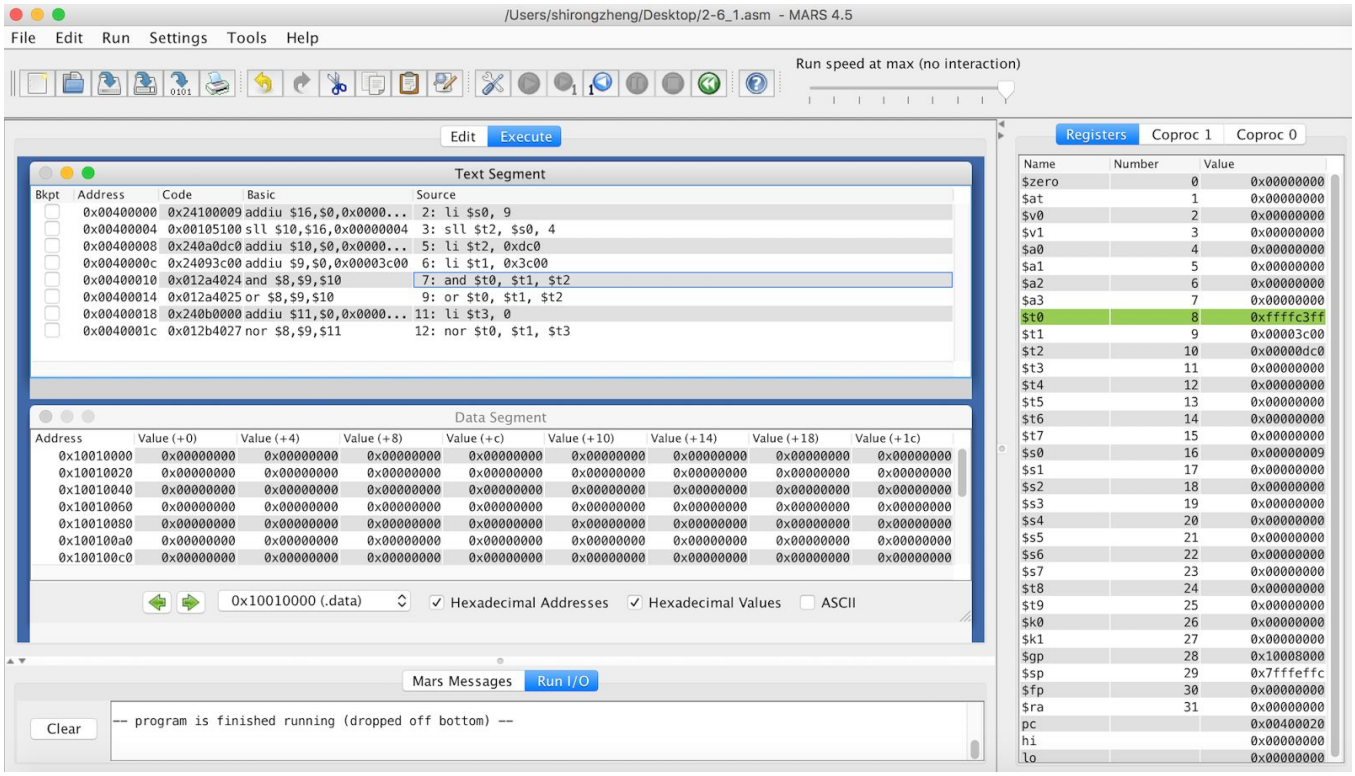
```
1  # left shift
2  li $s0, 9
3  sll $t2, $s0, 4
4  # AND
5  li $t2, 0xdc0
6  li $t1, 0x3c00
7  and $t0, $t1, $t2
8  # OR
9  or $t0, $t1, $t2
10 # NOR
11 li $t3, 0
12 nor $t0, $t1, $t3
```

2-6_1.asm

In this code, we don't assign any value to that, because there are not initial static variables. In registers, All values are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text, which is 0x00400020.



This assembly code is going to find the AND, OR, NOR operations. For example, the AND operation. The system finds the $t0=$t1 & $t2, $t1 is 0x00003c00, $t2 is 0x00000dc0. After the AND operation, $t0 is 0xffff3cff. All the answers could verify in the register window.



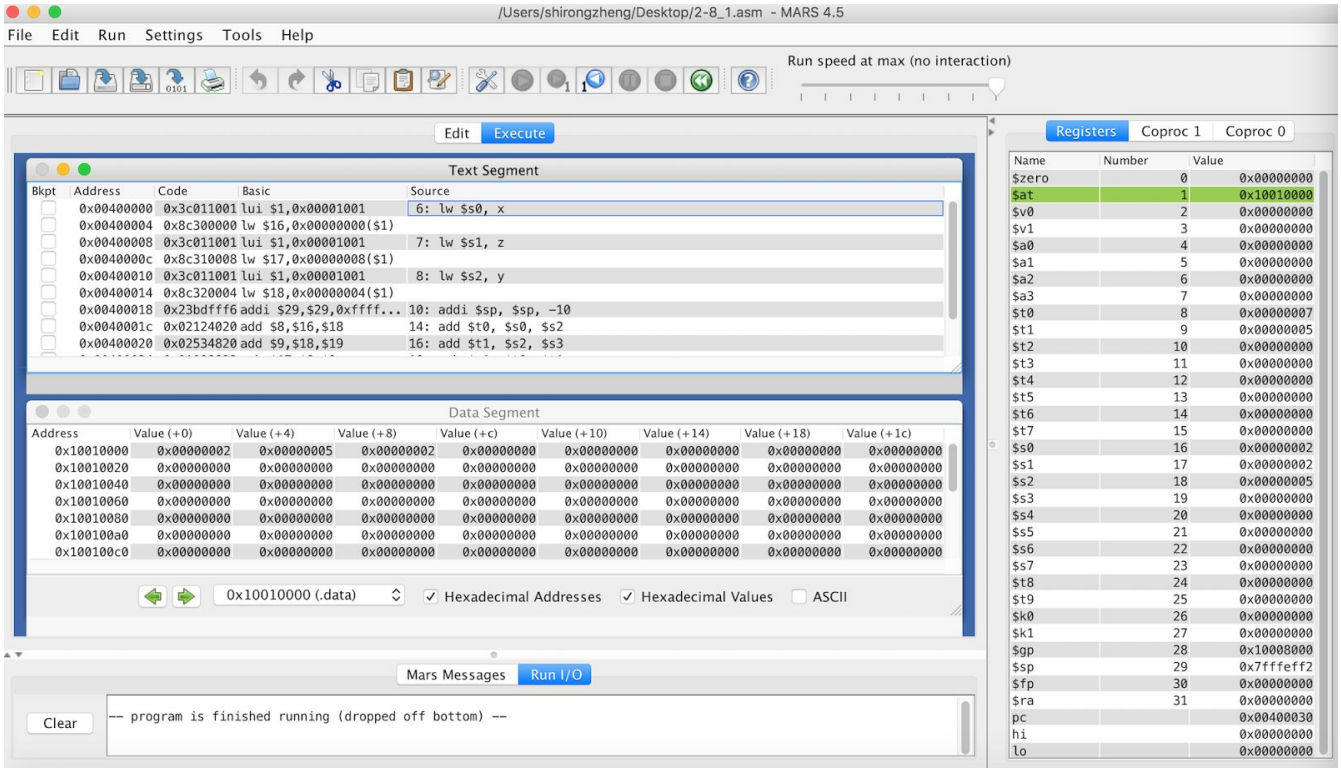
The below is my 2-8_2.asm code. The lw mean load address. The addi in myadd function is $sp=$sp+(-10). Then math function, z=(0+x)-(0+b).

2-8_1.asm

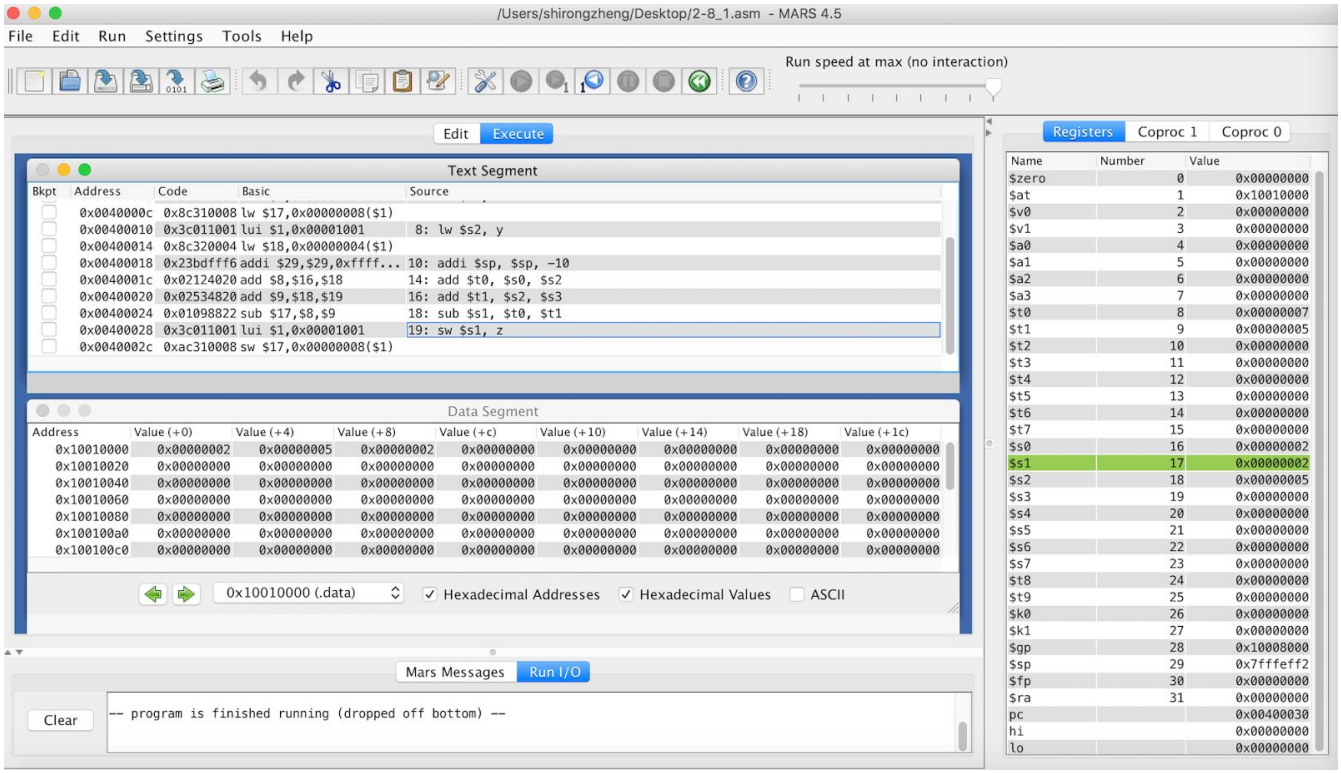
```
1  .data
2  x: .word 2
3  y: .word 5
4  z: .word 0
5  .text
6  lw $s0, x
7  lw $s1, z
8  lw $s2, y
9  myadd:
10 addi $sp, $sp, -10
11
12
13 # t0 = 0+x
14 add $t0, $s0, $s2
15 # t1 = b+0
16 add $t1, $s2, $s3
17 # z= t0 - t1
18 sub $s1, $t0, $t1
19 sw $s1, z
```

2-8_1.asm

The address of variable “x” is 0x10010000 which has the decimal value of 2. The address of array “z” is 0x10010004 which has the decimal value of 0. The address of variable “y” is 0x10010008 has the decimal value of 5. In registers, All values are 0X0000000 in register. The $gp means global pointer which has value 0x10008000, it points to the middle of the 64K block of memory in the static data segment. The $sp means stack pointer which has value 0x7fffeff2, it points to last location on the stack. The pc is program counter, it shows the current address of text, which is 0x00400030.



After compile, x is $s0 which has address value 0x00000002, y is $s2 which has address value 0x00000005. $t0 is $s0+$s2 and equal to 0x00000007. $t1 is $s2+$s3 and equal to 0x00000005. The z ($s1) is $t0-$t1 and equal to 0x00000002.



❖3. Part II Analysis in x86 Intel on Windows 32-bit

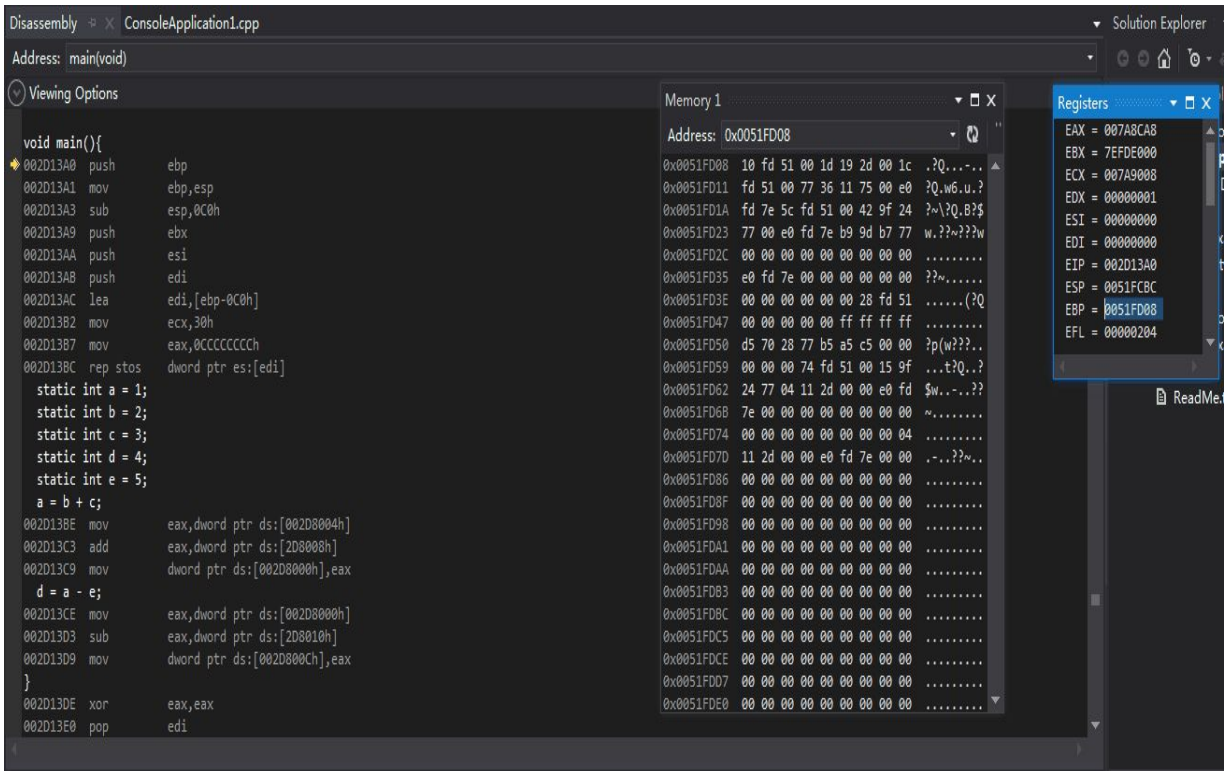
First I need to create a Console Application in the 32-bit Visual Studio in Windows 7.

The below is my 2-2_1.cpp file. I will assign some constant values to my static variables.

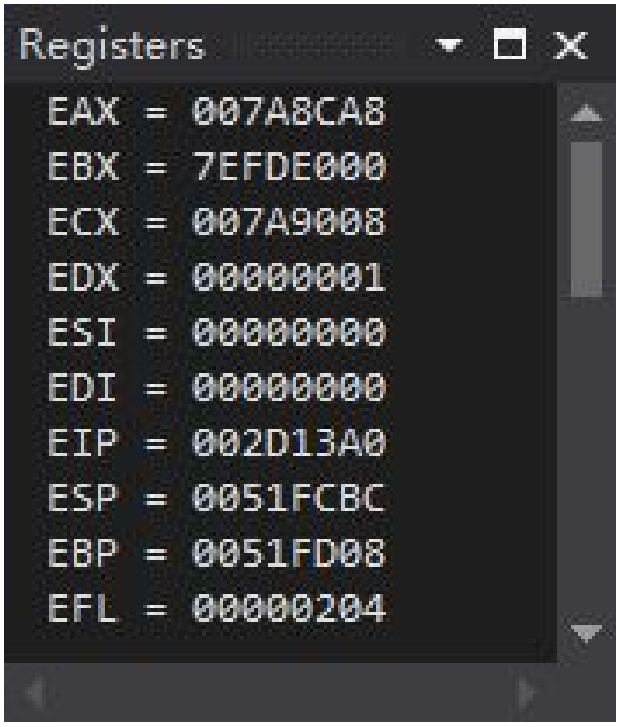
```
void main(){
    static int a = 1;
    static int b = 2;
    static int c = 3;
    static int d = 4;
    static int e = 5;
    a = b + c;
    d = a - e;
}
```

2-2_1.cpp

First , I evaluate $ebp, which has address 002D13A0, it's EBP = 0051FD08. We could find the memory is 10 fd 51 00 1d 19 2d 00 1c in Memory 1 window.



In the register, the stack pointer which is ESP=0051FCBC.



The below code shows the mini way to operate the function.

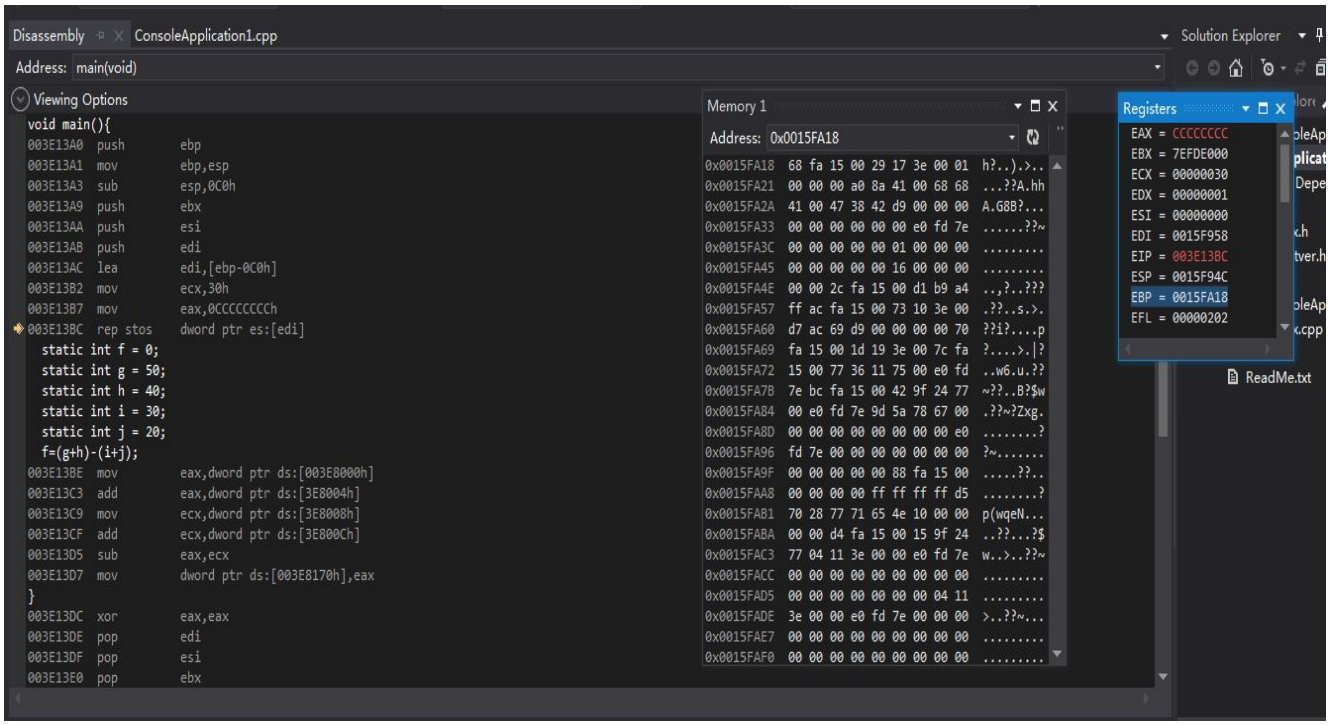
```
002D13DE  xor         eax,eax
002D13E0  pop         edi
002D13E1  pop         esi
002D13E2  pop         ebx
002D13E3  mov         esp,ebp
002D13E5  pop         ebp
002D13E6  ret
```

The below is my 2-2_2.cpp file. I will assign some constant values to my static variables.

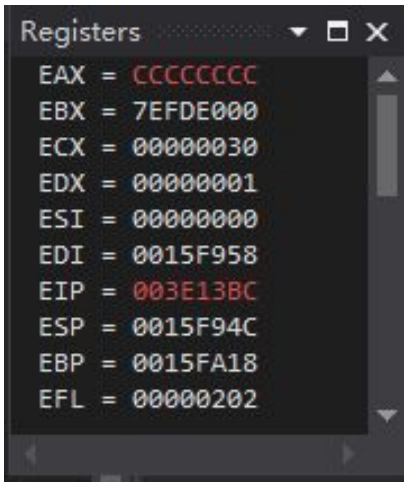
```
void main(){
    static int f = 0;
    static int g = 50;
    static int h = 40;
    static int i = 30;
    static int j = 20;
    f=(g+h)-(i+j);
}
```

2-2_2.cpp

First , I evaluate the value of f, which has address 003E13BC, it's EBP =0015FA18. We could find the memory is 68 fa 15 00 29 17 3e 00 01in Memory 1 window.



In the register, the EAX=CCCCCCCC, that is accumulator register.



In the address 003E13A1, that move the 4 bytes in memory at the address contained in ESP into EBP. ESP has address 0x0015F94C, EBP has address 0x0015FA18. ESP is the current stack pointer, which will change any time a word or address is pushed or popped onto/off off the stack. EBP is a more convenient way for the compiler to keep track of a function's parameters and local variables than using the ESP directly.

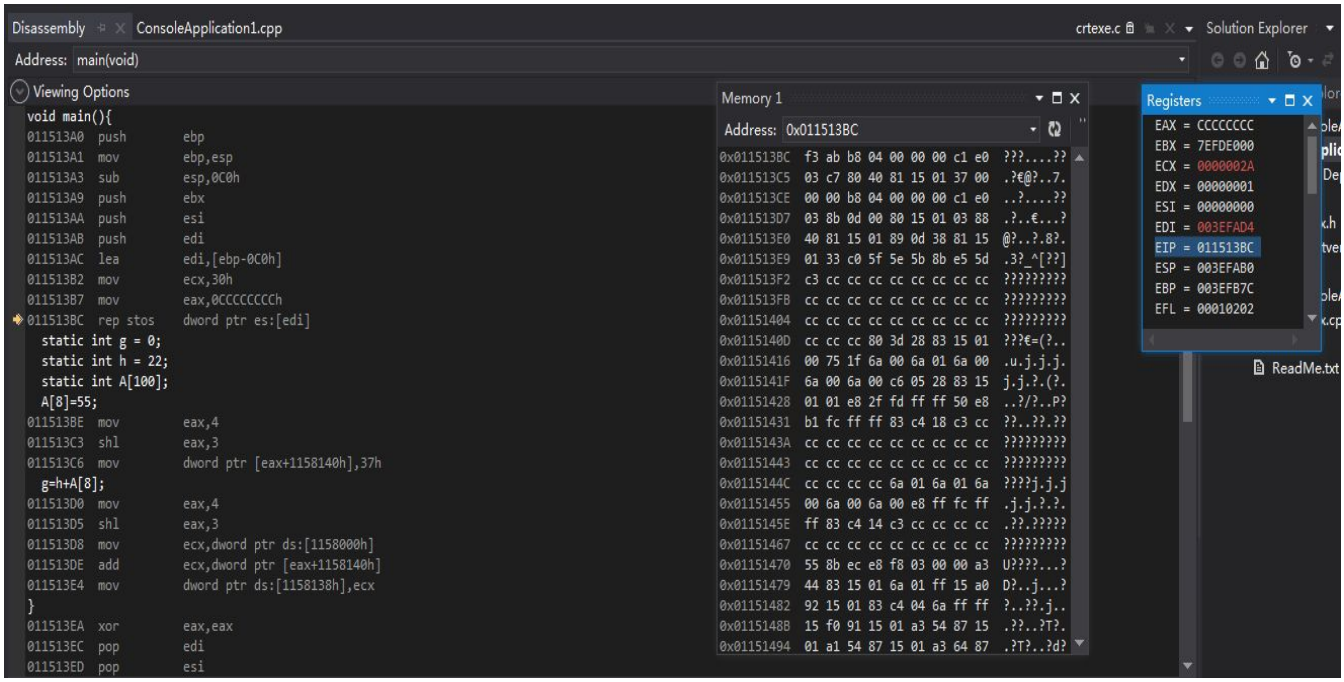
```
003E13A0  push        ebp
003E13A1  mov         ebp,esp
003E13A3  sub         esp,0C0h
003E13A9  push        ebx
003E13AA  push        esi
003E13AB  push        edi
003E13AC  lea         edi,[ebp-0C0h]
003E13B2  mov         ecx,30h
003E13B7  mov         eax,0CCCCCCCCh
003E13BC  rep stos    dword ptr es:[edi]
```

The below is my 2-3_1.cpp file. I will assign some constant values to my static variables and also declare the array A[100].

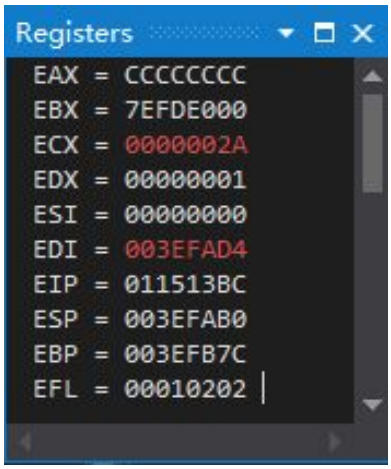
```
void main(){
    static int g = 0;
    static int h = 22;
    static int A[100];
    A[8]=55;
    g=h+A[8];
}
```

2-3_1.cpp

After disassembly, The EIP register always contains the address of the next instruction to be executed. You cannot directly access or change the instruction pointer. However, instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer. EIP=011513BC, it has memory value f3 ab b8 04 00 00 00 c1 e0 in Memory 1 window. In address 0x0115138C, the "dword ptr" part is called a size directive. This page explains them, but it wasn't possible to direct-link to the correct section. Basically, it means "the size of the target operand is 32 bits", so this will bitwise-AND the 32-bit value at the address computed by taking the contents of the ebp register and subtracting four with 0.



The stack pointer has the address 0x003EFAB0.

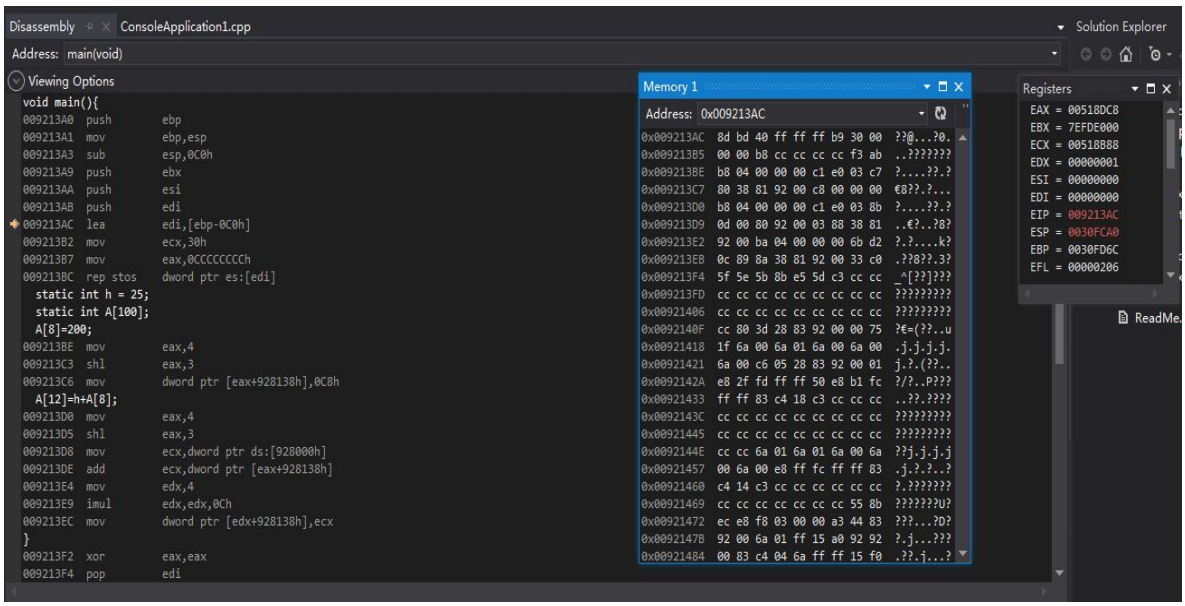


The below is my 2-3_2.cpp file. I will assign constant value 25 to my static variable h and also declare the array A[100].

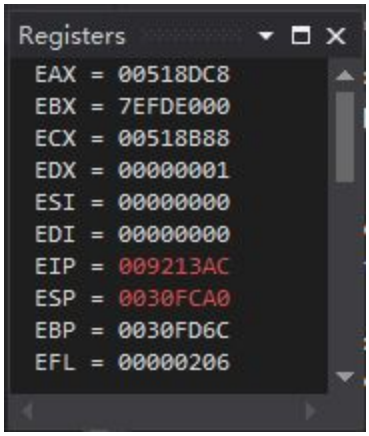
```
void main(){
    static int h = 25;
    static int A[100];
    A[8]=200;
    A[12]=h+A[8];
}
```

2-3_2.cpp

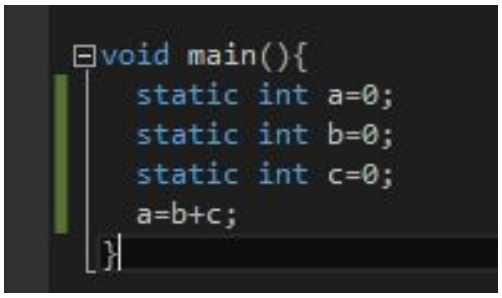
After disassembly, The EIP which instruction pointer to show the address is 0x009213AC, and it has the memory value 8d bd 40 ff ff ff b9 30 00 in Memory 1 window. It's the equivalent of "eax = edx + eax * 1". Generally, lea (address expression), register means "compute the address expression and change the register value to that"; other instructions use address expressions for memory access, i.e. mov (address expression), register means "compute the address expression and load the value from the resulting address into the register".



The below is my register window of this code. EFL=00000206.



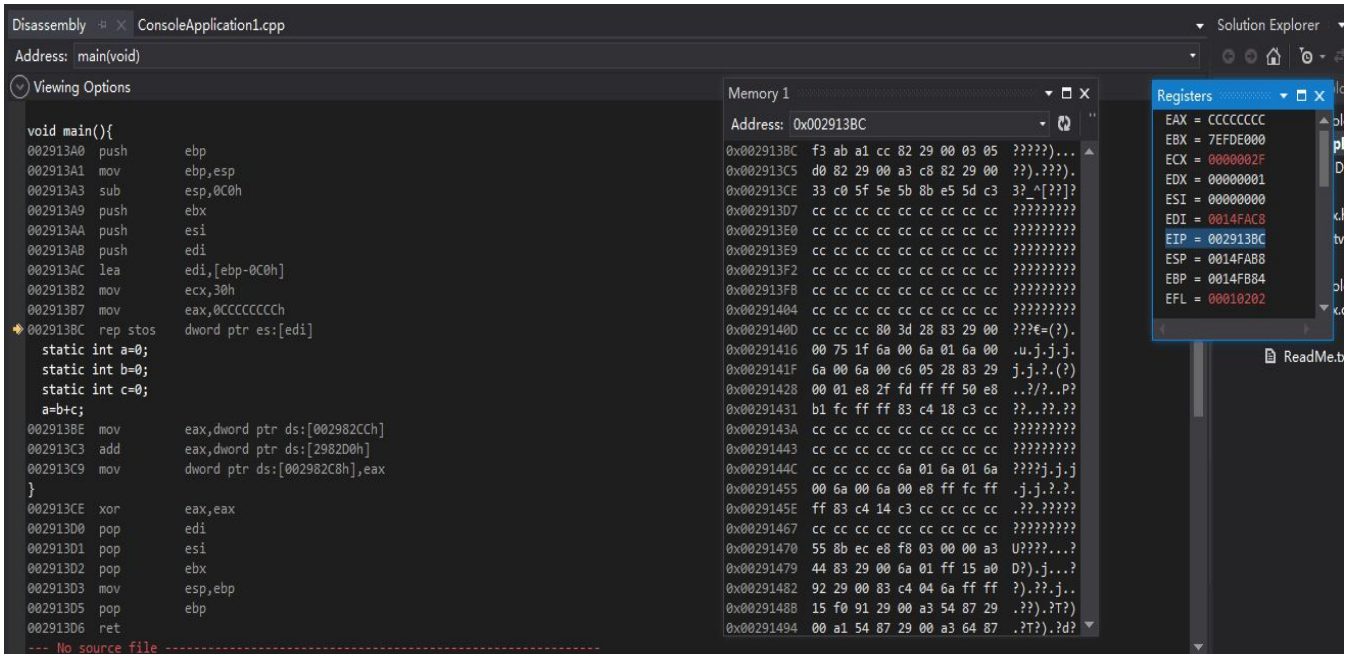
The below is my 2-5_1.cpp file. I will assign constant value 0 to my static variables a-c.



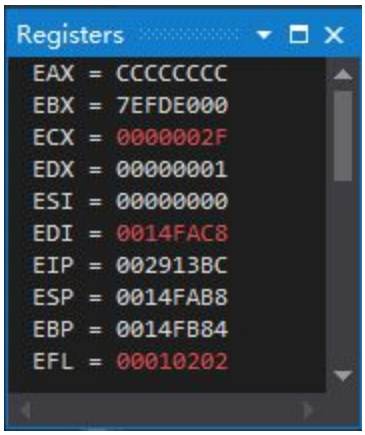
2-5_1.cpp

After disassembly, The EIP register always contains the address of the next instruction to be executed. You cannot directly access or change the instruction pointer. However, instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer. EIP=002913BC, it has memory value f3 ab a1 cc 82 29 00 03 05 in Memory 1 window. In EIP address, the "dword ptr" part is called a size directive. This page explains them, but it wasn't possible to direct-link to the correct section. Basically, it means "the size of

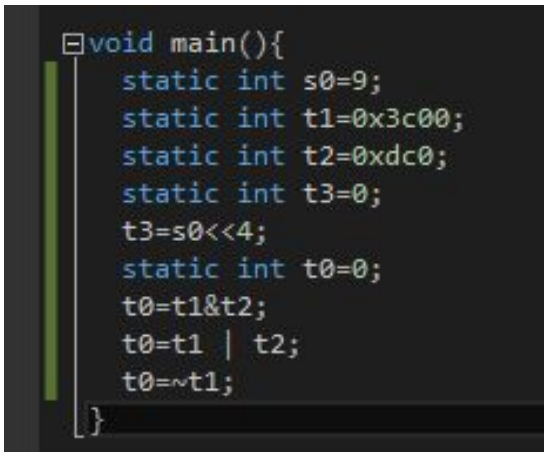
the target operand is 32 bits", so this will bitwise-AND the 32-bit value at the address computed by taking the contents of the ebp register and subtracting four with 0.



In the register, the EAX=CCCCCCCC, that is accumulator register. The leftmost one is the source. For example, movl %edx, %eax means Move the contents of the edx register into the eax register. For another example, addl %edx,%eax means Add the contents of the edx and eax registers, and place the sum in the eax register.

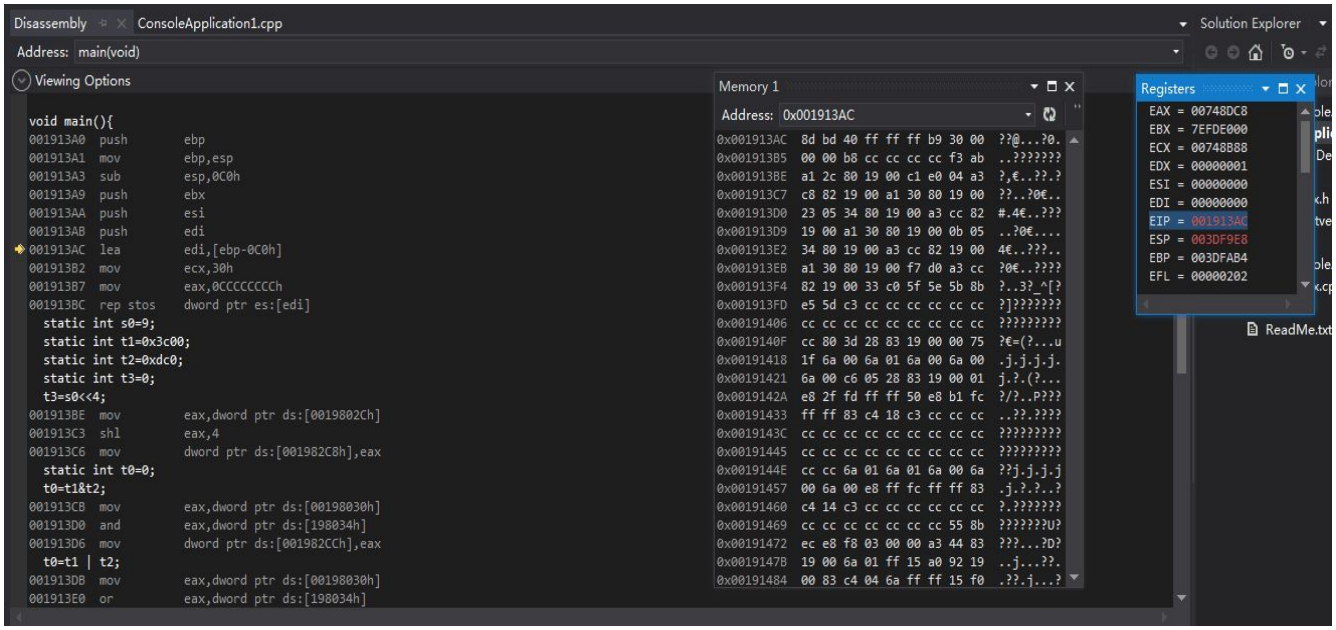


The below is my 2-6_1.cpp file.

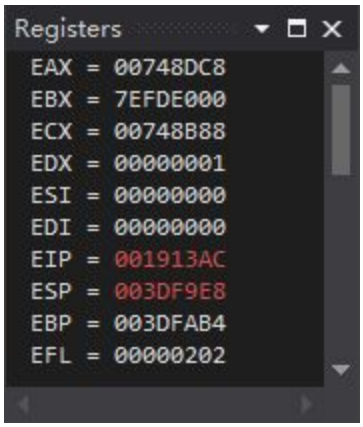


```
void main(){
    static int s0=9;
    static int t1=0x3c00;
    static int t2=0xdc0;
    static int t3=0;
    t3=s0<<4;
    static int t0=0;
    t0=t1&t2;
    t0=t1 | t2;
    t0=~t1;
}
```

After disassembly, The EIP which instruction pointer to show the address is 0x001913AC, and it has the memory value 8d bd 40 ff ff ff b9 30 00 in Memory 1 window. It's the equivalent of "eax = edx + eax * 1". Generally, lea (address expression), register means "compute the address expression and change the register value to that"; other instructions use address expressions for memory access, i.e. mov (address expression), register means "compute the address expression and load the value from the resulting address into the register".

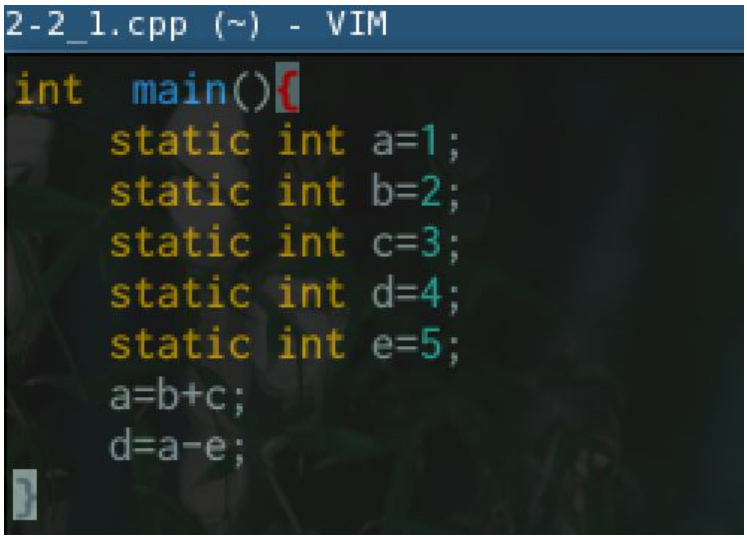


The EDX which is data register which has memory address 0x00000001. The Memory Data Register (MDR) or Memory Buffer Register (MBR) is the register of a computer's control unit that contains the data to be stored in the computer storage (e.g. RAM), or the data after a fetch from the computer storage.



❖4.Part III Analysis in LINUX, gdb/gcc 64 bit

I should declare main function as int from original void type.
The below code is my 2-2_1.cpp. I assign values from 1-5 to static variable a-e.



The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca. Then after disassemble debug, the static variable b will be store in the address 0x200b5c., the static variable c will be store in the address 0x200b5a. The address of static variable a is 0x200b44. The static variable e will be store in the address 0x200b4e. The address of static variable d is 0x200b40.

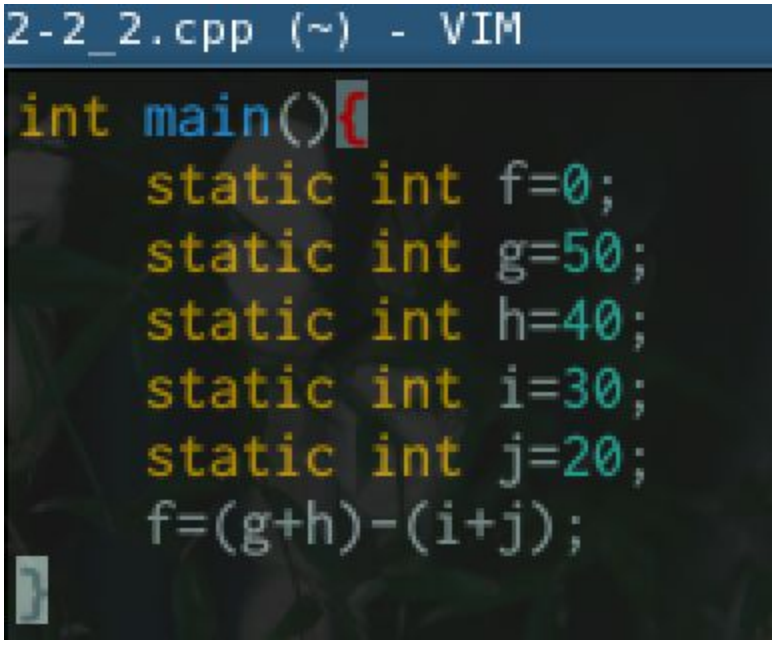
```
+(gdb) break main
Breakpoint 1 at 0x4004ca: file 2-2_1.cpp, line 7.
+(gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at 2-2_1.cpp:7
7               a=b+c;
+(gdb) next 2
9       }
+(gdb) disassemble
Dump of assembler code for function main():
   0x00000000004004c6 <+0>:     push   %rbp
   0x00000000004004c7 <+1>:     mov    %rsp,%rbp
   0x00000000004004ca <+4>:     mov    0x200b5c(%rip),%edx        # 0x60102c <_ZZ4mainE1b>
   0x00000000004004d0 <+10>:    mov    0x200b5a(%rip),%eax        # 0x601030 <_ZZ4mainE1c>
   0x00000000004004d6 <+16>:    add    %edx,%eax
   0x00000000004004d8 <+18>:    mov    %eax,0x200b4a(%rip)        # 0x601028 <_ZZ4mainE1a>
   0x00000000004004de <+24>:    mov    0x200b44(%rip),%edx        # 0x601028 <_ZZ4mainE1a>
   0x00000000004004e4 <+30>:    mov    0x200b4e(%rip),%eax        # 0x601038 <_ZZ4mainE1e>
   0x00000000004004ea <+36>:    sub    %eax,%edx
   0x00000000004004ec <+38>:    mov    %edx,%eax
   0x00000000004004ee <+40>:    mov    %eax,0x200b40(%rip)        # 0x601034 <_ZZ4mainE1d>
=> 0x00000000004004f4 <+46>:    mov    $0x0,%eax
   0x00000000004004f9 <+51>:    pop    %rbp
   0x00000000004004fa <+52>:    retq
End of assembler dump.
+(gdb)
```

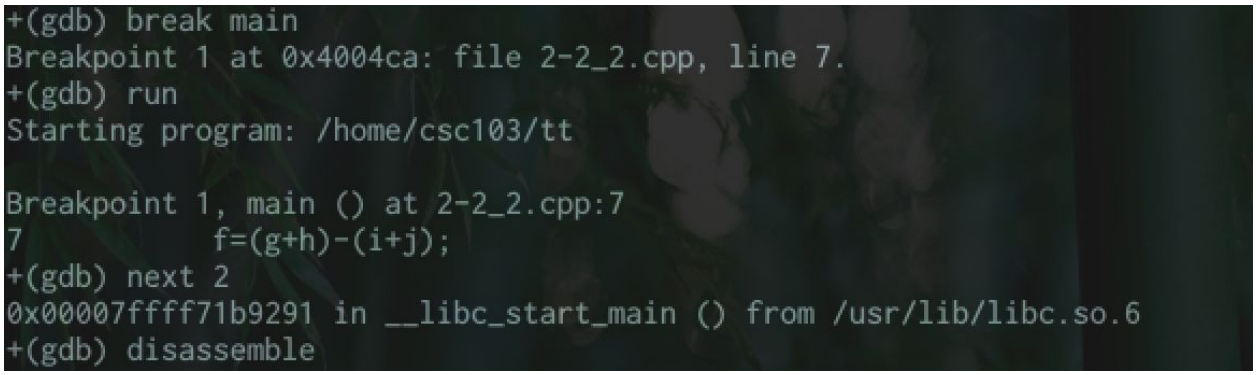
In the register part, $pc is program counter, the current address in hex is 0x4004f4.

```
+(gdb) display/i$pc
1: x/i $pc
=> 0x4004f4 <main()+46>:        mov    $0x0,%eax
+(gdb)
```

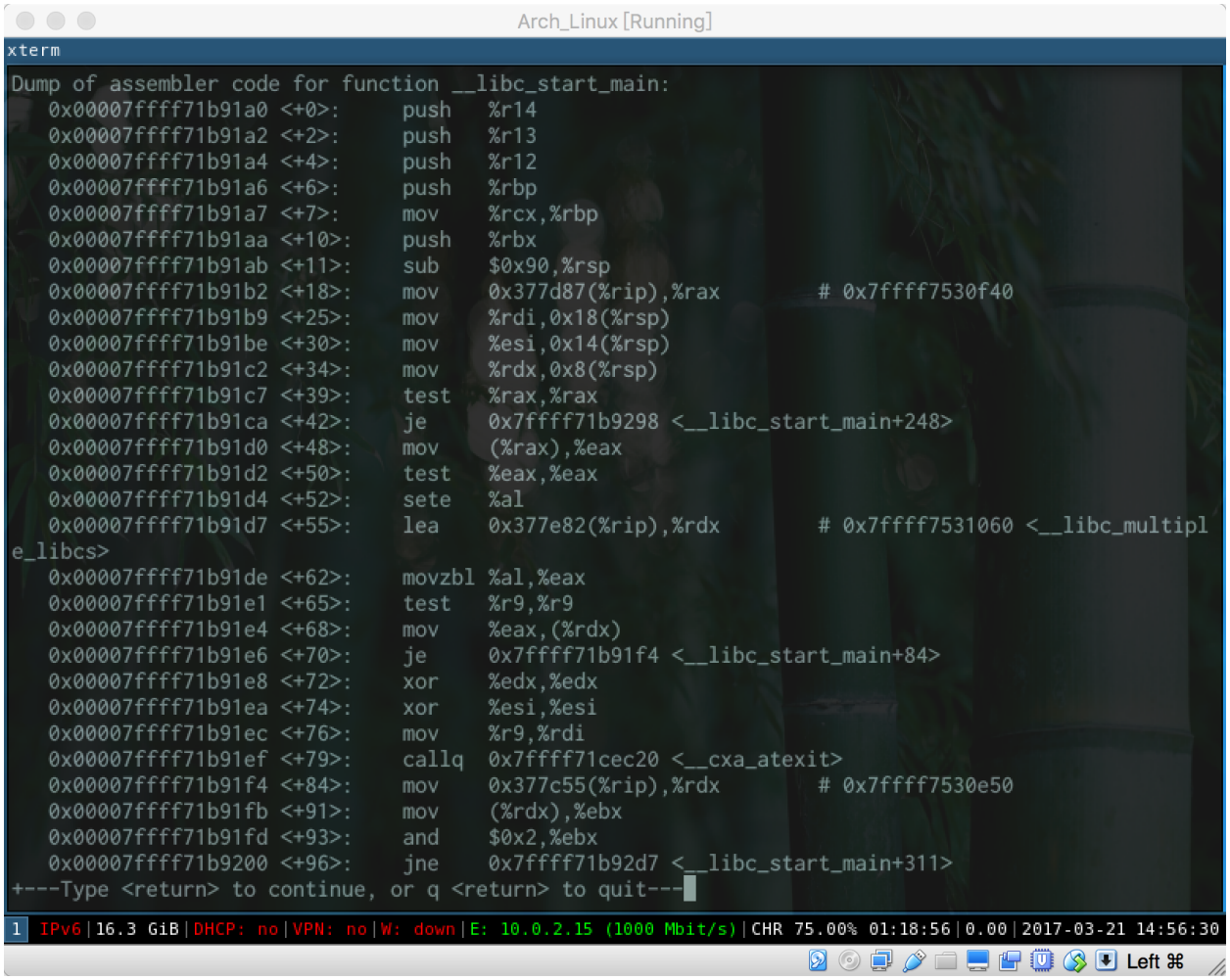
The below code is my 2-2_2.cpp. I assign values from 0,50,40,30,20 to static variables f-i.



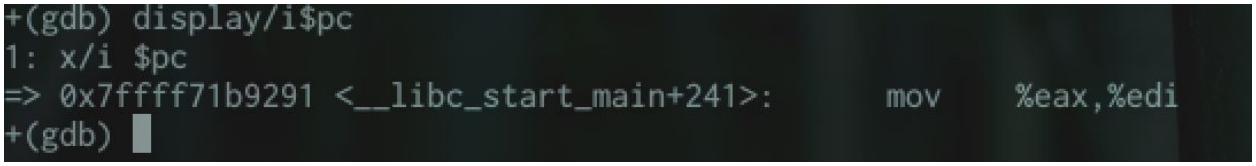
The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca.



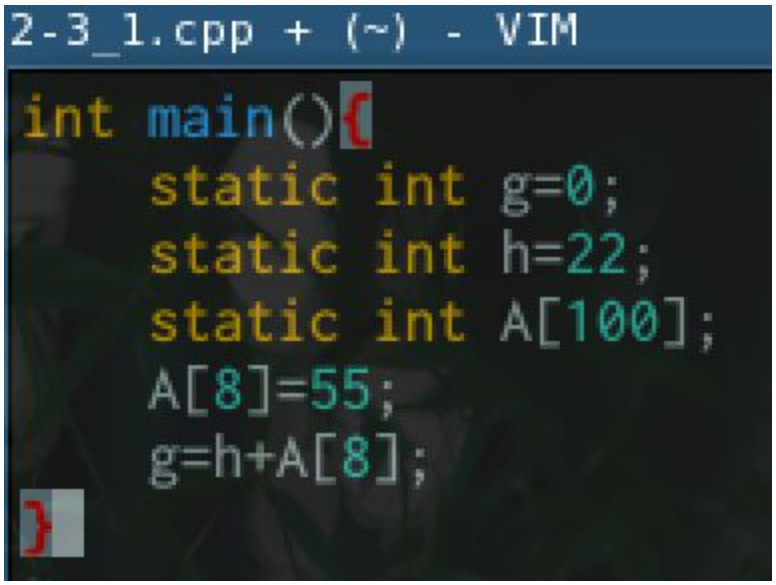
Then after disassemble debug, we have two parts to do. One is g+h, other one is i+j which the static variable g will be store in the address 0x377d87., the static variable h will be store in the address 0x377d18. the static variable i will be store in the address 0x377d14., the static variable j will be store in the address 0x377d04.



In the register part, $pc is program counter, the current address is 0x7ffff71b9291.



The below code is my 2-3_1.cpp. I assign values from 0,22 to static variables g,h. Then declare array A[100].



The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca. Then after disassemble debug, the static variable h will be store in the address 0x200b48., the array A[50] will be store in the address 0x200bcc. The address of static variable g is 0x200b78. The value g will also represent by $eax.

```
+(gdb) break main
Breakpoint 1 at 0x4004ca: file 2-3_1.cpp, line 5.
+(gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at 2-3_1.cpp:5
5               A[8]=55;
+(gdb) next 2
7       }
+(gdb) disassemble
Dump of assembler code for function main():
   0x00000000004004c6 <+0>:     push   %rbp
   0x00000000004004c7 <+1>:     mov    %rsp,%rbp
   0x00000000004004ca <+4>:     movl   $0x37,0x200bcc(%rip)        # 0x6010a0 <_ZZ4mainE1A+3
   0x00000000004004d4 <+14>:    mov    0x200bc6(%rip),%edx         # 0x6010a0 <_ZZ4mainE1A+32
   0x00000000004004da <+20>:    mov    0x200b48(%rip),%eax         # 0x601028 <_ZZ4mainE1h>
   0x00000000004004e0 <+26>:    add    %edx,%eax
   0x00000000004004e2 <+28>:    mov    %eax,0x200b78(%rip)         # 0x601060 <_ZZ4mainE1g>
=> 0x00000000004004e8 <+34>:    mov    $0x0,%eax
   0x00000000004004ed <+39>:    pop    %rbp
   0x00000000004004ee <+40>:    retq
End of assembler dump.
```

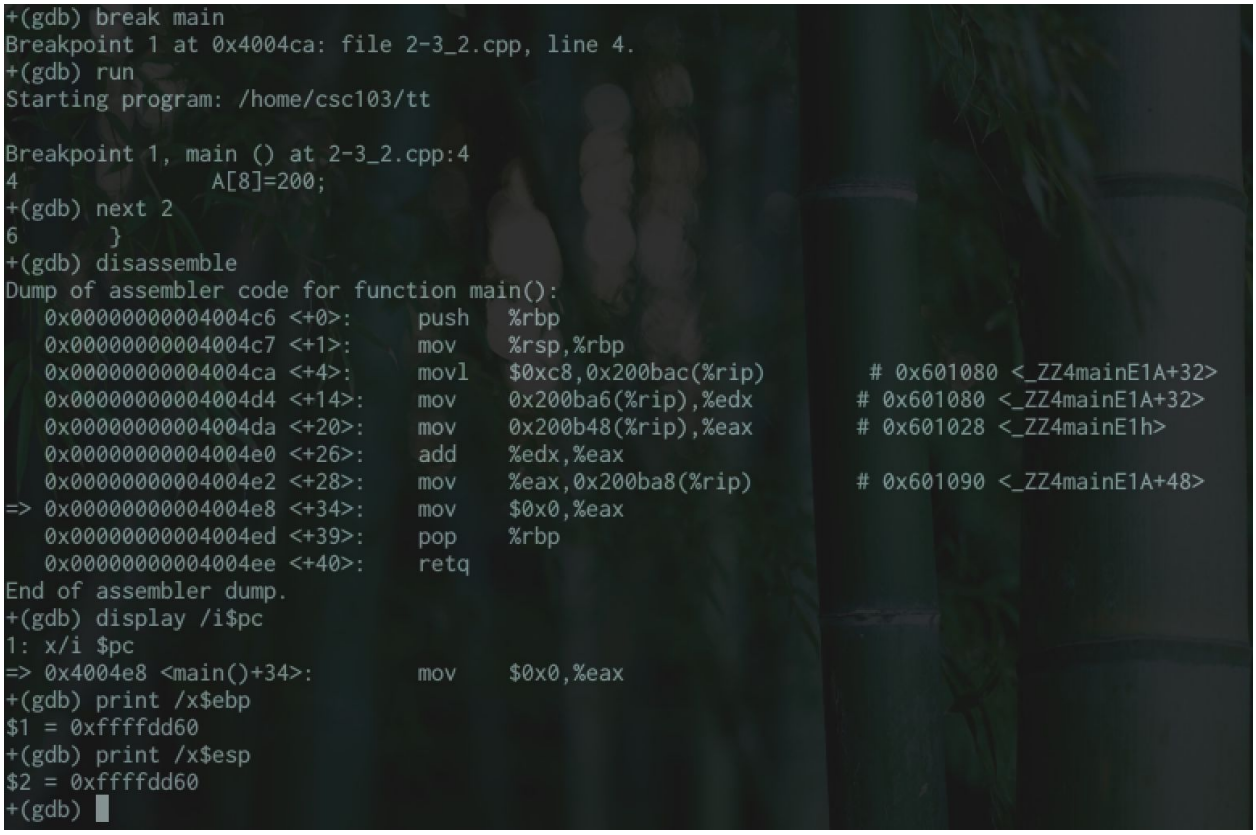
In the register part, $pc is program counter, the current address is 0x4004ea.

```
+(gdb) display /i$pc
1: x/i $pc
=> 0x4004e8 <main()+34>:        mov    $0x0,%eax
+(gdb)
```

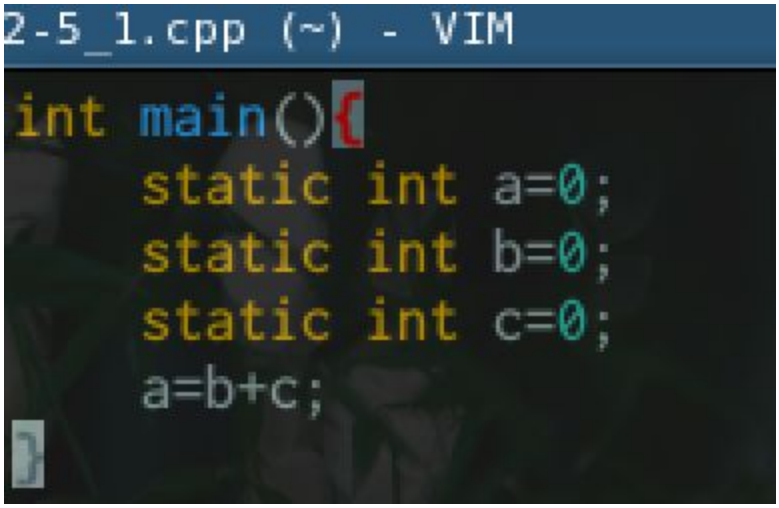
The below code is my 2-3_2.cpp. I assign value 25 to static variable h. Then declare array A[100].

```
2-3_2.cpp + (~) - VIM
int main(){
    static int h=25;
    static int A[100];
    A[8]=200;
    A[12]=h+A[8];
}
```

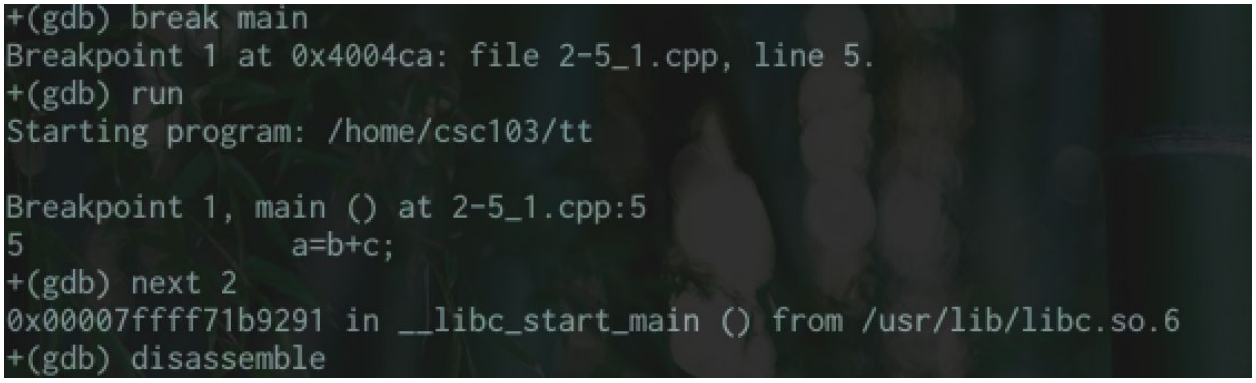
The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca. Then after disassemble debug, the static variable h will be store in the address 0x200ba6, the array A[8] will be store in the address 0x200b48, there is a value as 200. The address of static variable A[12] is 0x200ba8. The value will also represent by $eax. In the register part, $pc is program counter, the current address is 0x4004e8. In hex represent way, $1 which is h and it's address is 0xffffdd60. $2 which is A[12] and it's address is 0xffffdd60.



The below code is my 2-5_1.cpp. I assign value 0 to static variables a-c. Then declare the sum of b and c is a.



The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca.



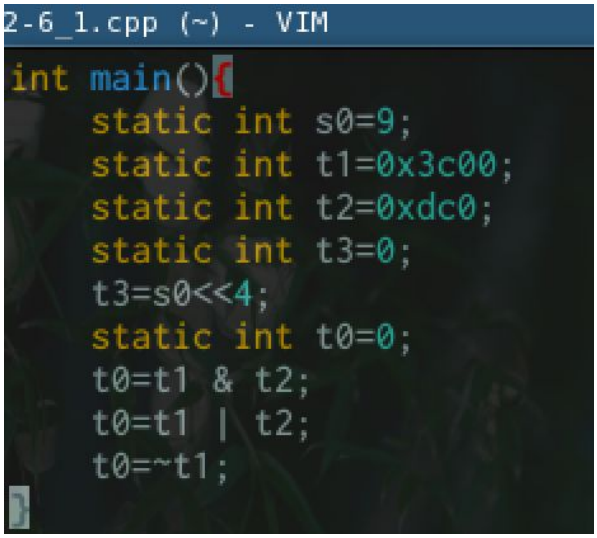
Then after disassemble debug, the static variable b will be store in the address 0x377d87, the static variable a is 0x377e82. The value will also represent by $eax.

```
Dump of assembler code for function __libc_start_main:
   0x00007ffff71b91a0 <+0>:     push   %r14
   0x00007ffff71b91a2 <+2>:     push   %r13
   0x00007ffff71b91a4 <+4>:     push   %r12
   0x00007ffff71b91a6 <+6>:     push   %rbp
   0x00007ffff71b91a7 <+7>:     mov    %rcx,%rbp
   0x00007ffff71b91aa <+10>:    push   %rbx
   0x00007ffff71b91ab <+11>:    sub    $0x90,%rsp
   0x00007ffff71b91b2 <+18>:    mov    0x377d87(%rip),%rax        # 0x7ffff7530f40
   0x00007ffff71b91b9 <+25>:    mov    %rdi,0x18(%rsp)
   0x00007ffff71b91be <+30>:    mov    %esi,0x14(%rsp)
   0x00007ffff71b91c2 <+34>:    mov    %rdx,0x8(%rsp)
   0x00007ffff71b91c7 <+39>:    test   %rax,%rax
   0x00007ffff71b91ca <+42>:    je     0x7ffff71b9298 <__libc_start_main+248>
   0x00007ffff71b91d0 <+48>:    mov    (%rax),%eax
   0x00007ffff71b91d2 <+50>:    test   %eax,%eax
   0x00007ffff71b91d4 <+52>:    sete   %al
   0x00007ffff71b91d7 <+55>:    lea    0x377e82(%rip),%rdx        # 0x7ffff7531060 <__libc_multipl
e_libcs>
   0x00007ffff71b91de <+62>:    movzbl %al,%eax
   0x00007ffff71b91e1 <+65>:    test   %r9,%r9
   0x00007ffff71b91e4 <+68>:    mov    %eax,(%rdx)
   0x00007ffff71b91e6 <+70>:    je     0x7ffff71b91f4 <__libc_start_main+84>
   0x00007ffff71b91e8 <+72>:    xor    %edx,%edx
   0x00007ffff71b91ea <+74>:    xor    %esi,%esi
   0x00007ffff71b91ec <+76>:    mov    %r9,%rdi
   0x00007ffff71b91ef <+79>:    callq  0x7ffff71cec20 <__cxa_atexit>
   0x00007ffff71b91f4 <+84>:    mov    0x377c55(%rip),%rdx        # 0x7ffff7530e50
   0x00007ffff71b91fb <+91>:    mov    (%rdx),%ebx
   0x00007ffff71b91fd <+93>:    and    $0x2,%ebx
   0x00007ffff71b9200 <+96>:    jne    0x7ffff71b92d7 <__libc_start_main+311>
+---Type <return> to continue, or q <return> to quit---
```

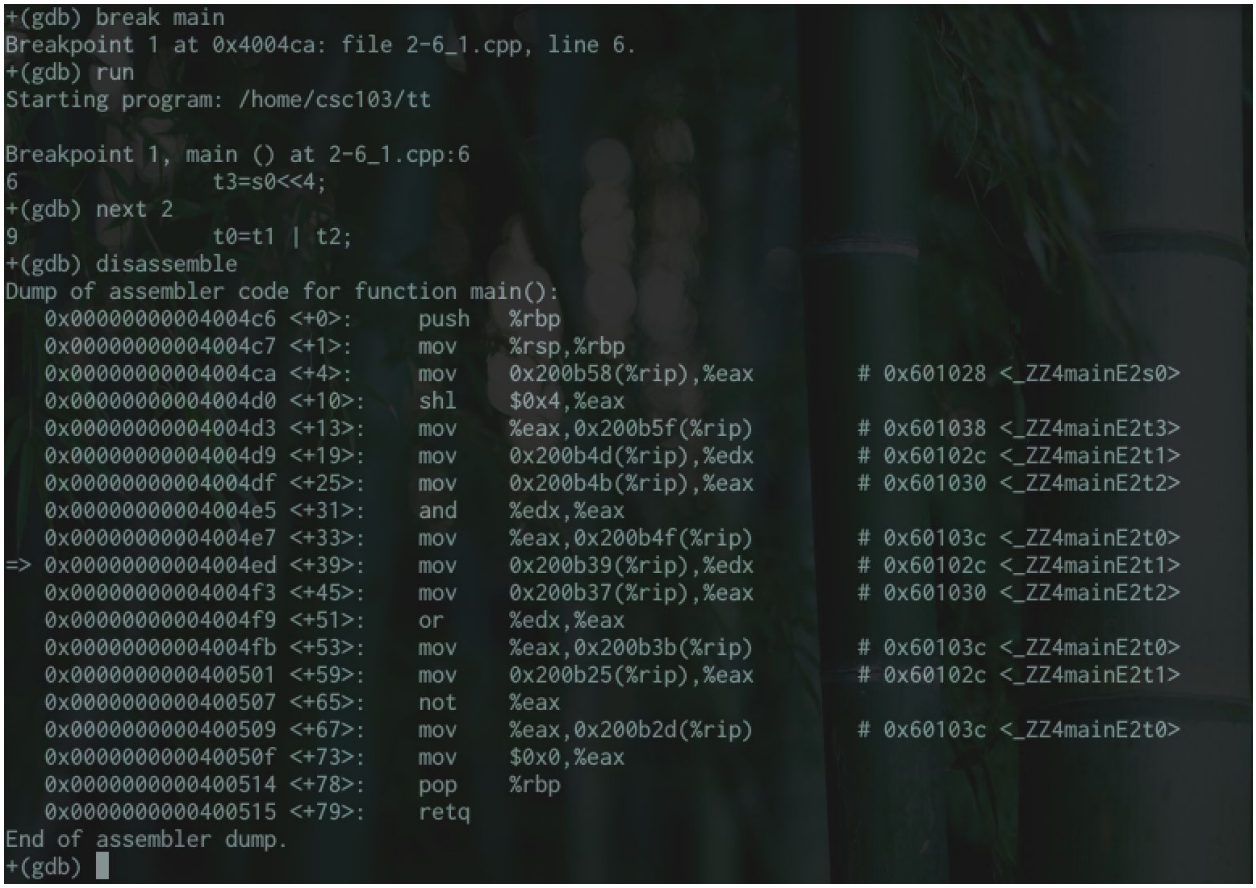
In the register part, $pc is program counter, the current address is 0x7ffff71b9291. In hex represent way, $1 which is b and it's address is0x4004f0 . $2 which is c and it's address is 0xffffdd70.

```
+(gdb) display /i$pc
1: x/i $pc
=> 0x7ffff71b9291 <__libc_start_main+241>:      mov    %eax,%edi
+(gdb) print /x$ebp
$1 = 0x4004f0
+(gdb) print /x$esp
$2 = 0xffffdd70
+(gdb)
```

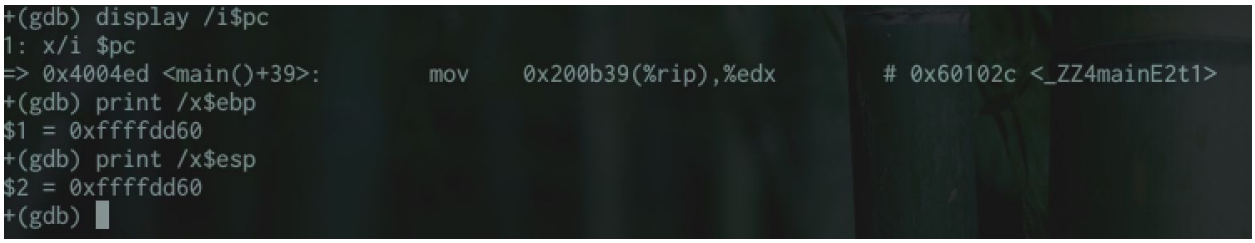
The below code is my 2-6_1.cpp.



The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca. In this debugging, that process will do and, or and not operations. In AND operation, the address is 0x200b4f. In OR operations, the address is 0x200b3b. In NOT operation, the address is 0x200b2d.



In the register part, $pc is program counter, the current address is 0x4004ed. In hex represent way, $1 which is t1 and it's address is0xffffdd60 . $2 which is t2 and it's address is 0xffffdd60.



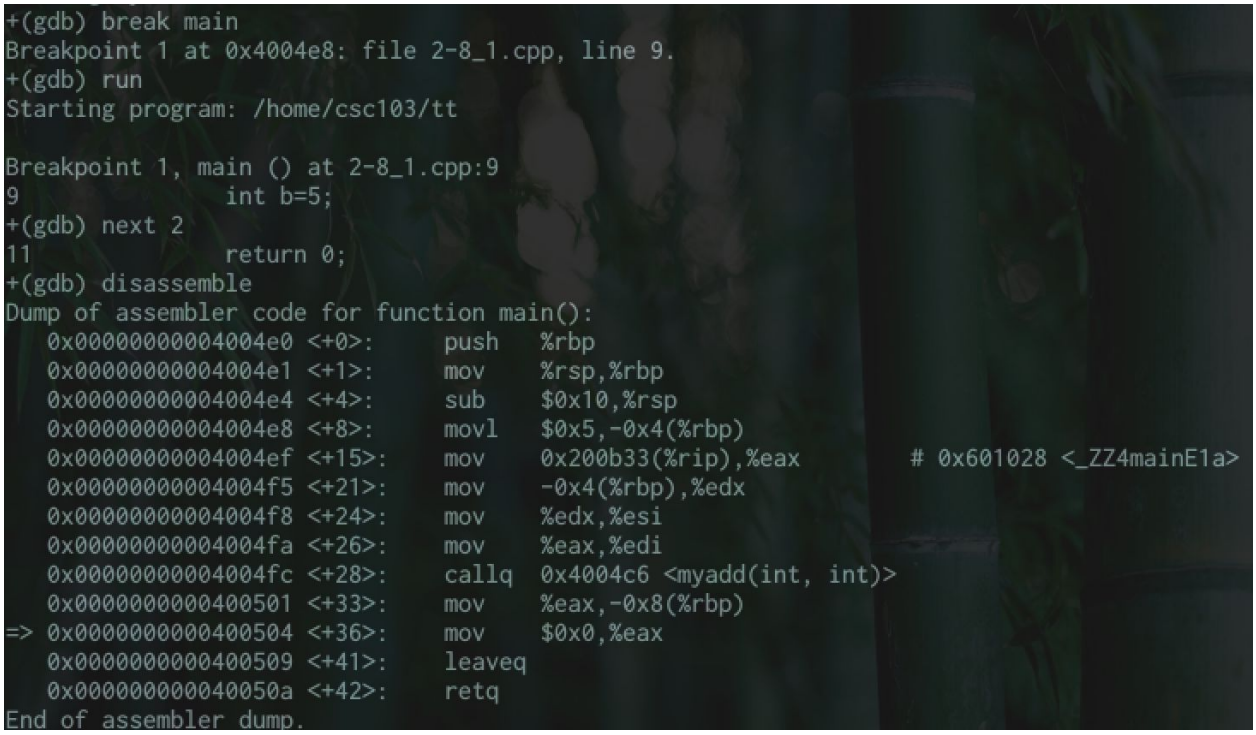
The below code is my 2-8_1.cpp.

2-8_1.cpp + (~) - VIM

```
int myadd(int x, int y){
    int temp;
    temp=x+y;
    return temp;
}

int main(){
    static int a=2;
    int b=5;
    int c=myadd(a,b);
    return 0;
}
```

The below is my debugging process. First this code will breakpoint 1 at the address 0x4004ca. In this debugging, that process will create one function and input this function in the main function. 0x4004c6 is the final address of c after using myadd function.



In the register part, $pc is program counter, the current address is 0x400504. In hex represent way, $3 which is a and it's address is0xffffdd60 . $2 which is b and it's address is 0xffffdd50.

❖5. Conclusions

In the conclusion, I learned how to debugging in MIPS, MS Visual Studio and GDB in Linux. Then we called this take home test as Comparison ISA. We should to compare the similarity and difference of all threes. That cause each debug system, whatever the Windows X32 bit or Linux X64 bit will debugging or compiling each function. I spent more than 10 hours to finish all the sections. In the suggestion, it is better to do the efficiently work in less time.