

Take Home Test 1:

Comparison ISA

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

Contents

1.Objective	3
2.Part I Analysis in MIPS	4
3.Part II Analysis in x86 Intel on Windows 32-bit	25
4.Part III Analysis in LINUX, gdb/gcc 64 bit	36
5. Conclusions	49

❖ 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.

```
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

2-2_1.asm

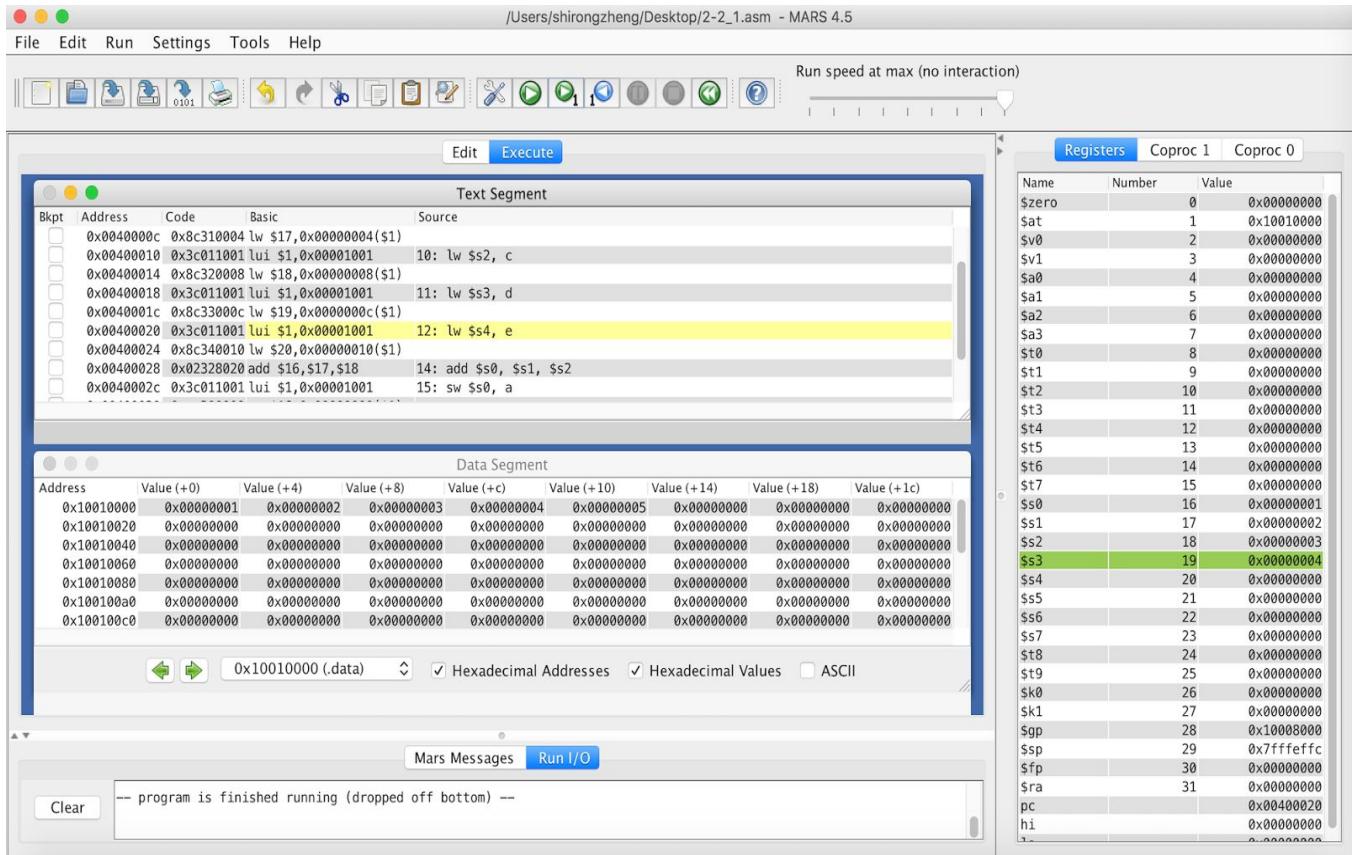
The below is my screenshot after I compile my 2-2_1.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate.

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).

The screenshot shows the MARS 4.5 assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. The title bar indicates the file is /Users/shirongzheng/Desktop/2-2_1.asm - MARS 4.5. The main window is divided into several sections:

- Text Segment:** A table showing assembly instructions. The first instruction is highlighted in yellow: `8: lw $s0, a`. Other visible instructions include `lw $s1, b`, `lw $s2, c`, `lw $s3, d`, and `lw $s4, e`.
- Data Segment:** A table showing memory starting at address 0x10010000. The first row shows the address 0x10010000 and its corresponding value 0x00000001.
- Registers:** A table showing the state of various registers. The \$zero register is 0, and the \$s0 register is 16.
- Run Control:** Buttons for Edit, Execute, Run speed at max (no interaction), and a step control.
- Mars Messages:** A text area indicating "Assemble: operation completed successfully."

In registers, All values from \$s0-\$s4 are 0X00000000 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 0x7fffffc, 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.

The screenshot shows the MARS 4.5 assembly debugger interface. The assembly window displays the following code:

```

Text Segment
Bkpt Address Code Basic Source
0x00400018 0x8c011001 lui $1,0x00001001 11: lw $s3, d
0x0040001c 0x8c33000c lw $19,0x0000000c($1)
0x00400020 0x3c011001 lui $1,0x00001001 12: lw $s4, e
0x00400024 0x8c340010 lw $20,0x00000010($1)
0x00400028 0x02328020 add $16,$17,$18 14: add $s0, $s1, $s2
0x0040002c 0x3c011001 lui $1,0x00001001 15: sw $s0, a
0x00400030 0xac300000 sw $16,0x00000000($1)
0x00400034 0x02149822 sub $19,$16,$20 17: sub $s3, $s0, $s4
0x00400038 0x3c011001 lui $1,0x00001001 18: sw $s3, d

```

The Registers window shows the following values:

Name	Number	Value
\$at	1	0x10010000
\$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	0x00000005
\$s1	17	0x00000002
\$s2	18	0x00000003
\$s3	19	0x00000004
\$s4	20	0x00000005
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7fffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040002c
hi		0x00000000
lo		0x00000000

The Data Segment window shows the memory dump:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

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.

The screenshot shows the MARS 4.5 assembly debugger interface. The assembly window displays the following code:

```

Text Segment
Bkpt Address Code Basic Source
0x00400014 0x8c320008 lw $18,0x00000008($1)
0x00400018 0x3c011001 lui $1,0x00001001 11: lw $s3, d
0x0040001c 0x8c33000c lw $19,0x0000000c($1)
0x00400020 0x3c011001 lui $1,0x00001001 12: lw $s4, e
0x00400024 0x8c340010 lw $20,0x00000010($1)
0x00400028 0x02328020 add $16,$17,$18 14: add $s0, $s1, $s2
0x0040002c 0x3c011001 lui $1,0x00001001 15: sw $s0, a
0x00400030 0xac300000 sw $16,0x00000000($1) 16: sw $s0, a
0x00400034 0x02149822 sub $19,$16,$20 17: sub $s3, $s0, $s4

```

The Registers window shows the following values:

Name	Number	Value
\$at	1	0x10010000
\$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	0x00000005
\$s1	17	0x00000002
\$s2	18	0x00000003
\$s3	19	0x00000004
\$s4	20	0x00000005
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7fffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040002c
hi		0x00000000
lo		0x00000000

The Data Segment window shows the memory dump:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x00400000	0x3c011001	0x8c300000	0x3c011001	0x8c310004	0x3c011001
0x00400020	0x3c011001	0x8c340010	0x02328020	0x3c011001	0xac300000

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 screenshot shows the MARS 4.5 assembly debugger interface. The Text Segment pane displays assembly code with several instructions highlighted in yellow. The Data Segment pane shows memory starting at address 0x10010000. The Registers pane on the right lists the state of all 32 registers. The message window at the bottom indicates the program has finished running.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$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	0x00000005
\$s1	17	0x00000002
\$s2	18	0x00000003
\$s3	19	0x00000000
\$s4	20	0x00000005
\$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	0x7ffffefc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400040
hi		0x00000000
lo		0x00000000

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 “ $t_0 = g + h$ ”. In other Add part, “sub \$t1, \$s3, \$s4” means “ $t_1 = i + j$ ”. In the final Subtraction part, “sub \$s0, \$t0, \$t1 means “ $f = t_0 - t_1$ ”. Then store word(sw) of each operation. The lw means load word.

```
1 .data
2 f: .word 0
3 g: .word 50
4 h: .word 40
5 i: .word 30
6 j: .word 20
7 .text
8 lw $s0, f
9 lw $s1, g
10 lw $s2, h
11 lw $s3, i
12 lw $s4, j
13 # t0 = g+h
14 add $t0, $s1, $s2
15 # t1 = i+j
16 add $t1, $s3, $s4
17 # f = t0 - t1
18 sub $s0, $t0, $t1
19 sw $s0, f
```

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).

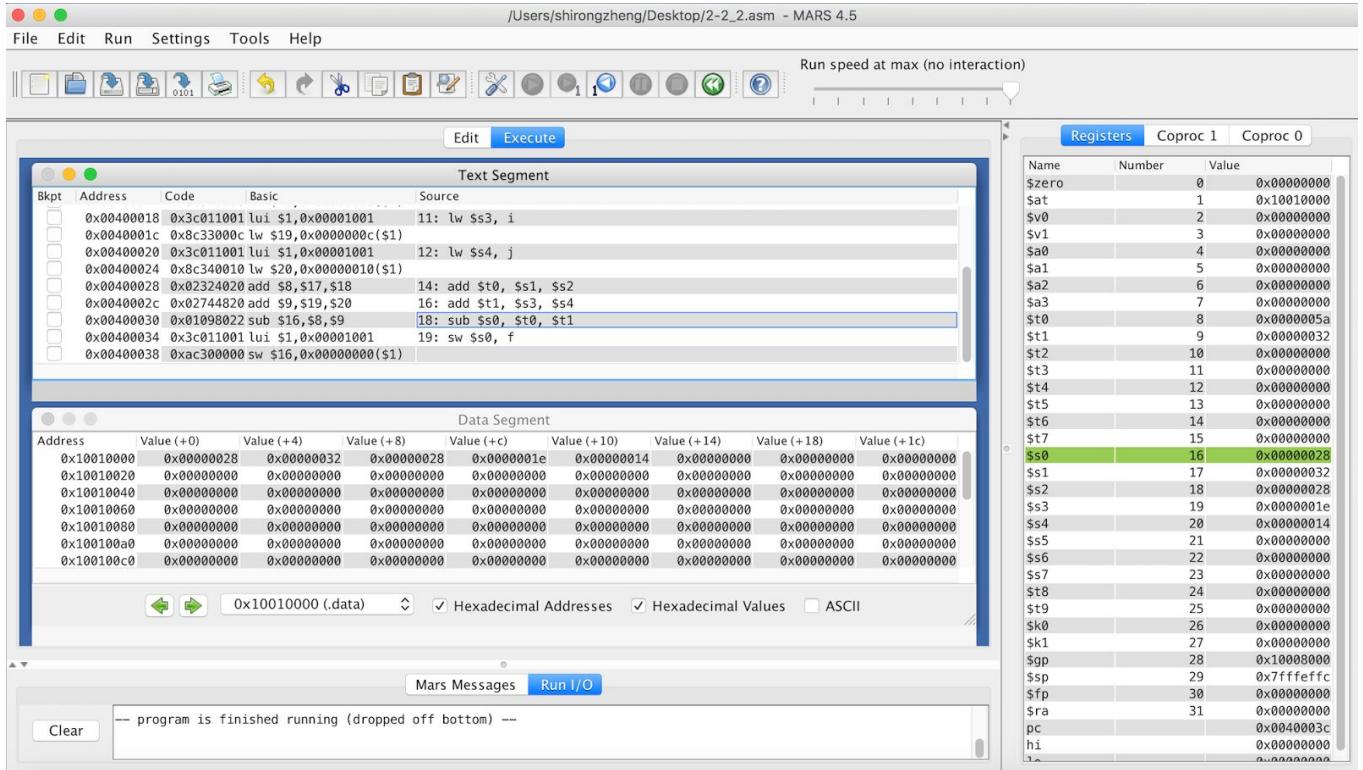
The screenshot shows the MARS 4.5 assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. A toolbar with various icons is located above the main window. The main area is divided into several sections:

- Text Segment:** A table showing assembly instructions. The instruction at address 0x00400000 is highlighted in yellow: `lui $1,0x00001001`. The source column shows the assembly code with line numbers 8, 9, 10, 11, and 12 corresponding to the addresses.
- Data Segment:** A table showing memory values. The first row shows the address 0x10010000 and its corresponding value 0x00000000. Subsequent rows show values for addresses 0x10010020 through 0x100100c0, all initialized to 0x00000000.
- Registers:** A table showing the state of various registers. The columns are Name, Number, and Value. Values are mostly 0x00000000, except for \$gp at 0x10008000 and \$sp at 0x7fffffc.
- Mars Messages:** A status bar at the bottom indicating "Assemble: operation completed successfully."

In registers, All values from \$s0-\$s4 are 0X00000000 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, 100 to 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

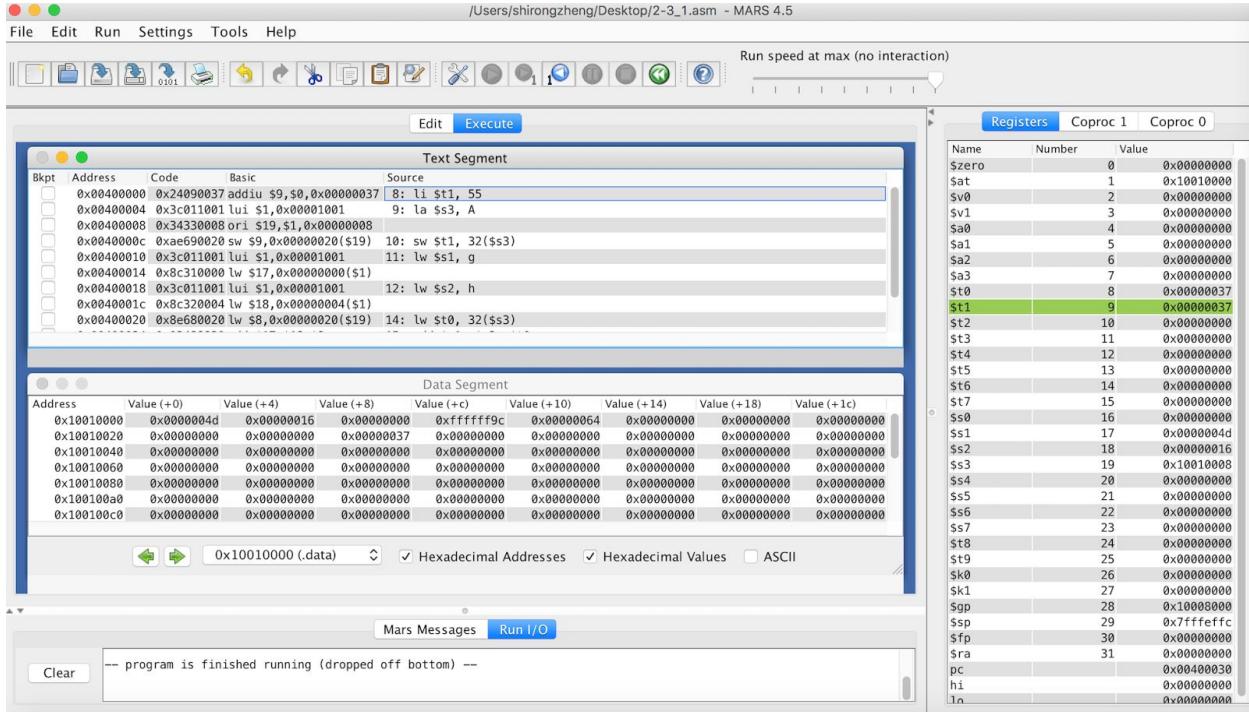
1 .data
2 g: .word 0
3 h: .word 22
4 A: .word 0-100
5 size: .word 100
6 .text
7 #just to set A[8] to 55
8 li $t1, 55
9 la $s3, A
10 sw $t1, 32($s3)
11 lw $s1, g
12 lw $s2, h
13 #loading the value of A[8] into t0
14 lw $t0, 32($s3)
15 add $s1, $s2, $t0
16 sw $s1, g
```

2-3_1.asm

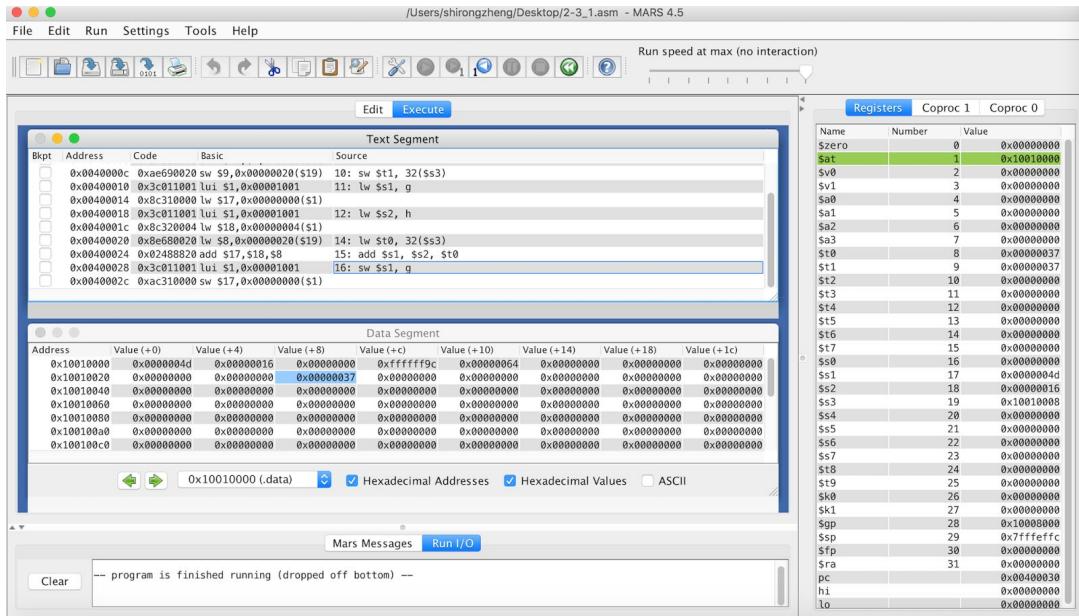
The below is my screenshot after I compile my 2-3_1.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate. The li means load immediate. That set the array A[8] to 55.

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 0X00000000 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 0x7fffffc, 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 0xfffffff9c. That we could say the address of going from 0x1001000 to 0x1001000c). In registers, All values are 0X00000000 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 screenshot shows the MARS 4.5 assembly debugger interface. The assembly window displays the following code:

```

Text Segment
Bkpt Address Code Basic Source
0x00400000 0x3c011001 lui $1,0x00001001 6: lw $s2, h
0x00400004 0x8c320000 lw $18,0x00000000($1)
0x00400008 0x24090c8 addiu $9,$0,0x000000c8 8: li $t1, 200
0x0040000c 0x3cd11001 lui $1,0x00001001 9: la $s3, A
0x00400010 0x34330004 ori $19,$1,0x00000004
0x00400014 0xae690020 sw $9,0x00000020($19) 10: sw $t1, 32($s3)
0x00400018 0x8e680020 lw $8,0x00000020($19) 12: lw $t0, 32($s3)
0x0040001c 0x02484020 add $8,$18,$8 13: add $t0, $s2, $t0
0x00400020 0xae680030 sw $8,0x00000030($19) 14: sw $t0, 48($s3)

```

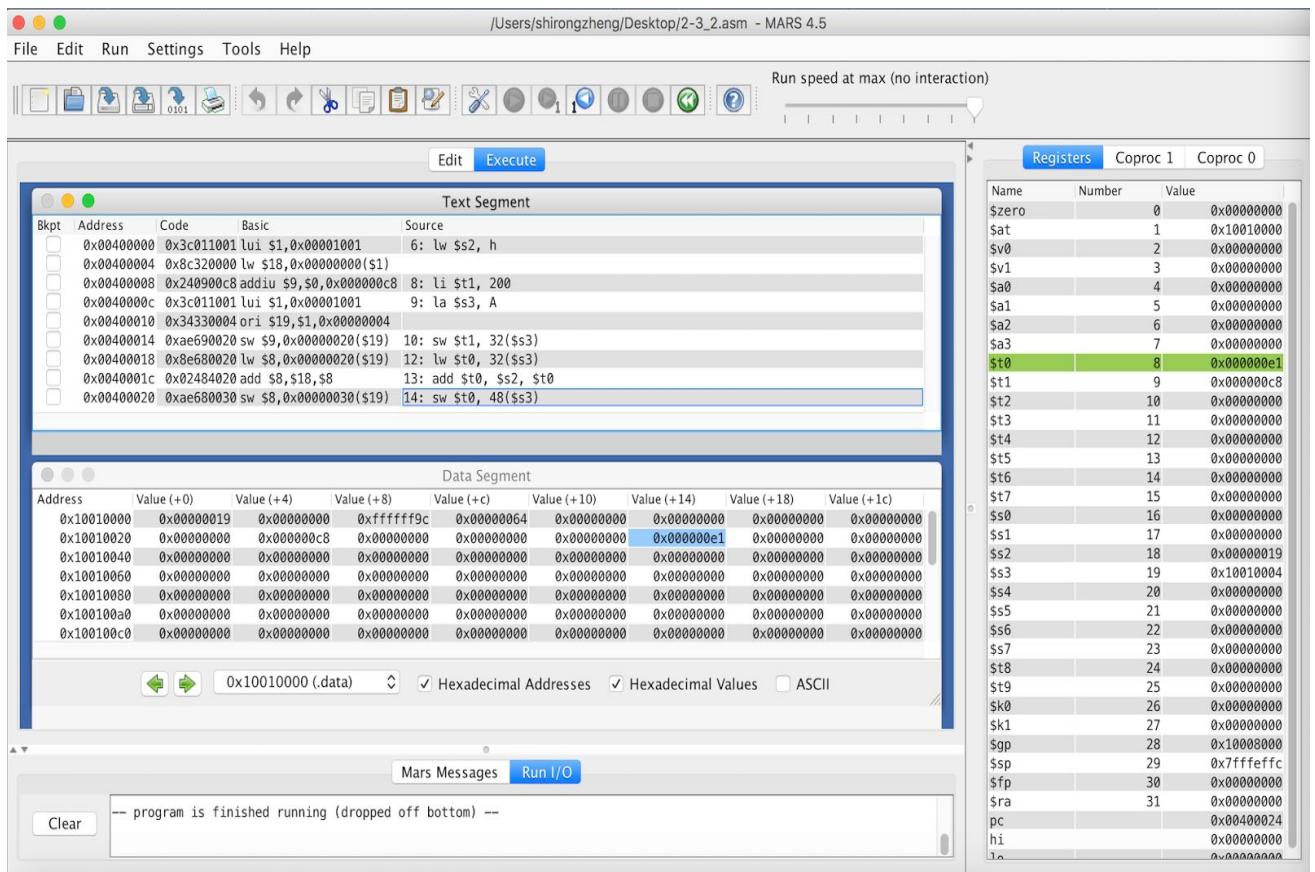
The Data Segment window shows memory starting at address 0x10010000:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000019	0x00000000	0xfffffff9c	0x00000064	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x000000c8	0x00000000	0x00000000	0x00000000	0x000000e1	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

The Registers window shows the following register values:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000001
\$t1	9	0x000000c8
\$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	0x00000019
\$s3	19	0x10010004
\$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		0x00400024
hi		0x00000000
lo		0x00000000

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.

The screenshot shows a text editor window with the title "2-5_2.asm". The code is as follows:

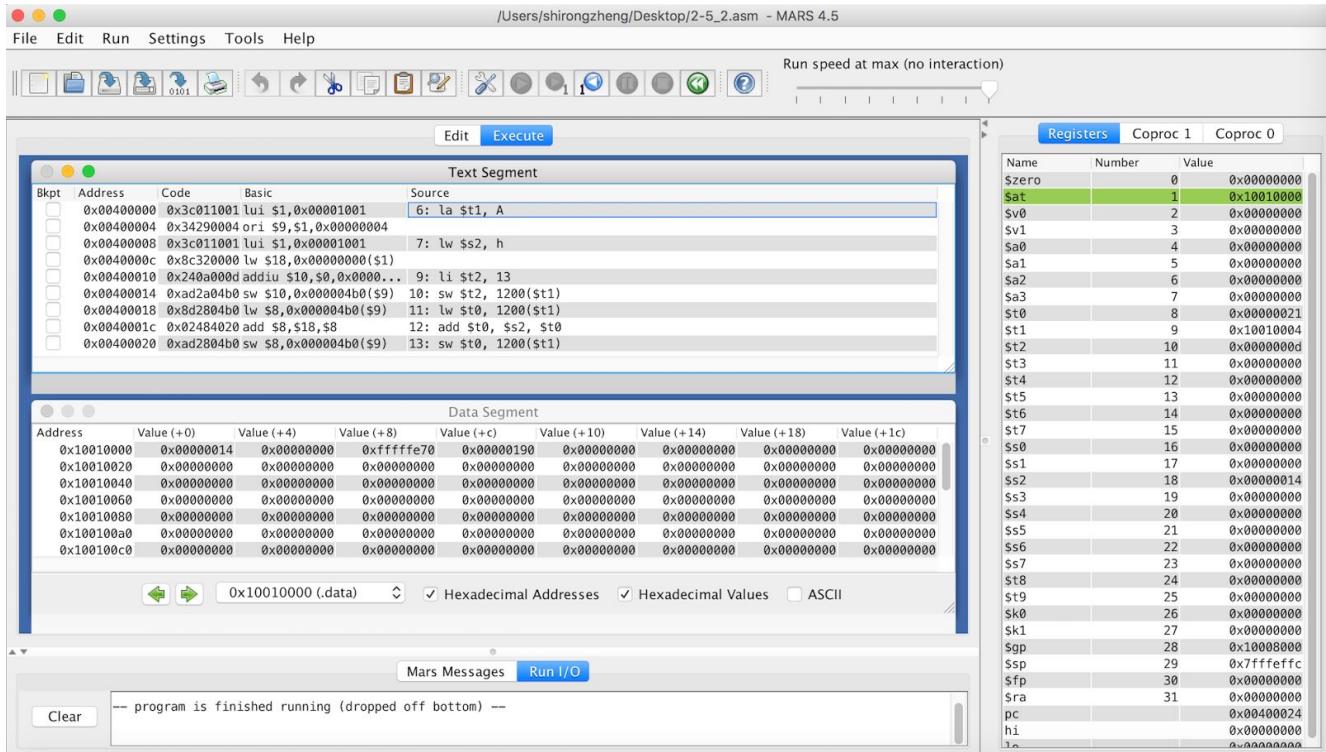
```
1 .data
2 h: .word 20
3 A: .word 0-400
4 size: .word 400
5 .text
6 la $t1, A
7 lw $s2, h
8 #initializing A[300] to 13
9 li $t2, 13
10 sw $t2, 1200($t1)
11 lw $t0, 1200($t1)
12 add $t0, $s2, $t0
13 sw $t0, 1200($t1)
```

2-5_2.asm

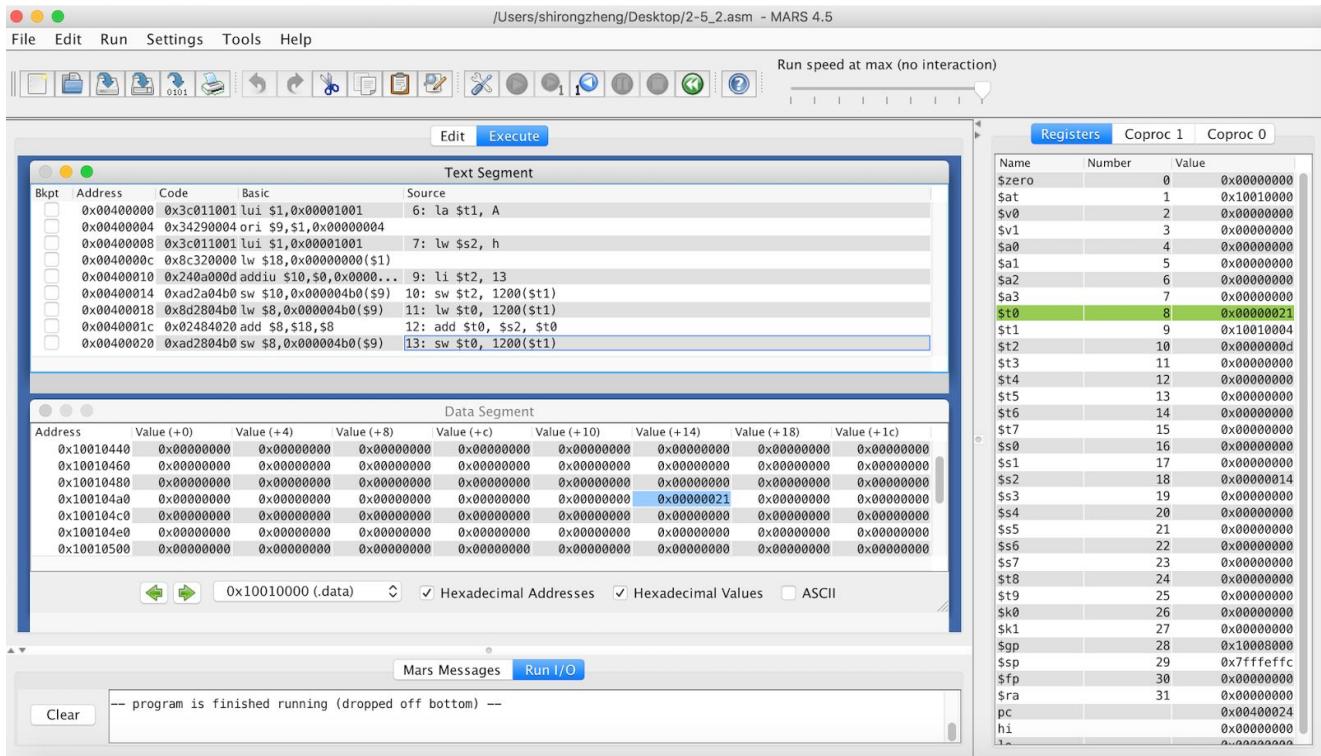
The below is my screenshot after I compile my 2-5_2.asm code. The lw means load word when I assign the address for letter a-e. The lui means load upper immediate. The li means load immediate. That set the array A[300] to 13.

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 0X00000000 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

```

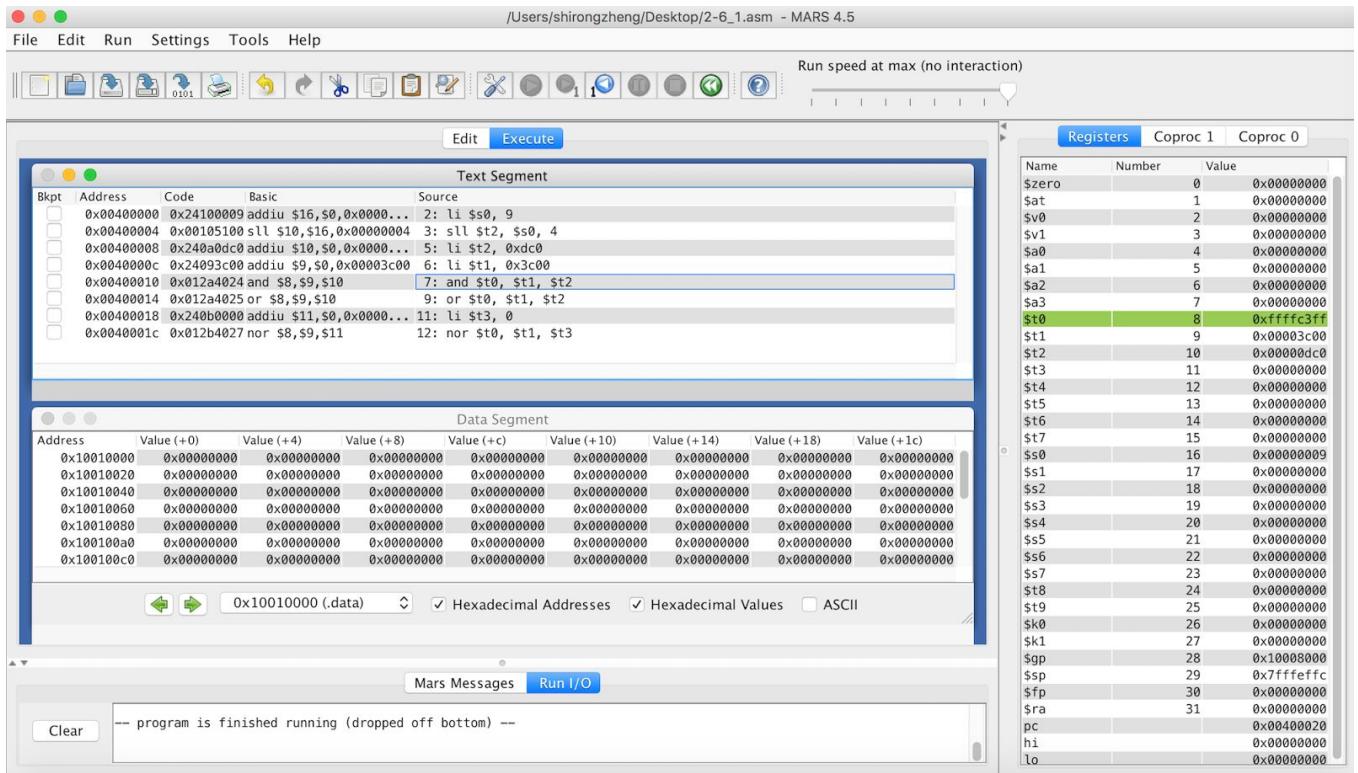
2-6_1.asm

In this code, we don't assign any value to that, because there are not initial static variables. In registers, All values are 0X00000000 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.

The screenshot shows the MARS 4.5 assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. The toolbar below has various icons for file operations and simulation controls. The main window is divided into several panes:

- Text Segment:** Shows assembly code with columns for Bkpt, Address, Code, Basic, and Source. The source code includes instructions like addiu, sll, li, and and.
- Data Segment:** Shows memory dump tables for addresses 0x10010000 to 0x100100c0, displaying values for columns Value (+0), Value (+4), Value (+8), Value (+c), Value (+10), Value (+14), Value (+18), and Value (+1c).
- Registers:** A table showing register names, numbers, and values. Key values include \$gp at 0x10008000, \$sp at 0x7fffeffc, and \$t0 at 0xfffff3cff.
- Mars Messages:** A log window showing the message "-- program is finished running (dropped off bottom) --".

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 0xfffff3cff. All the answers could verify in the register window.

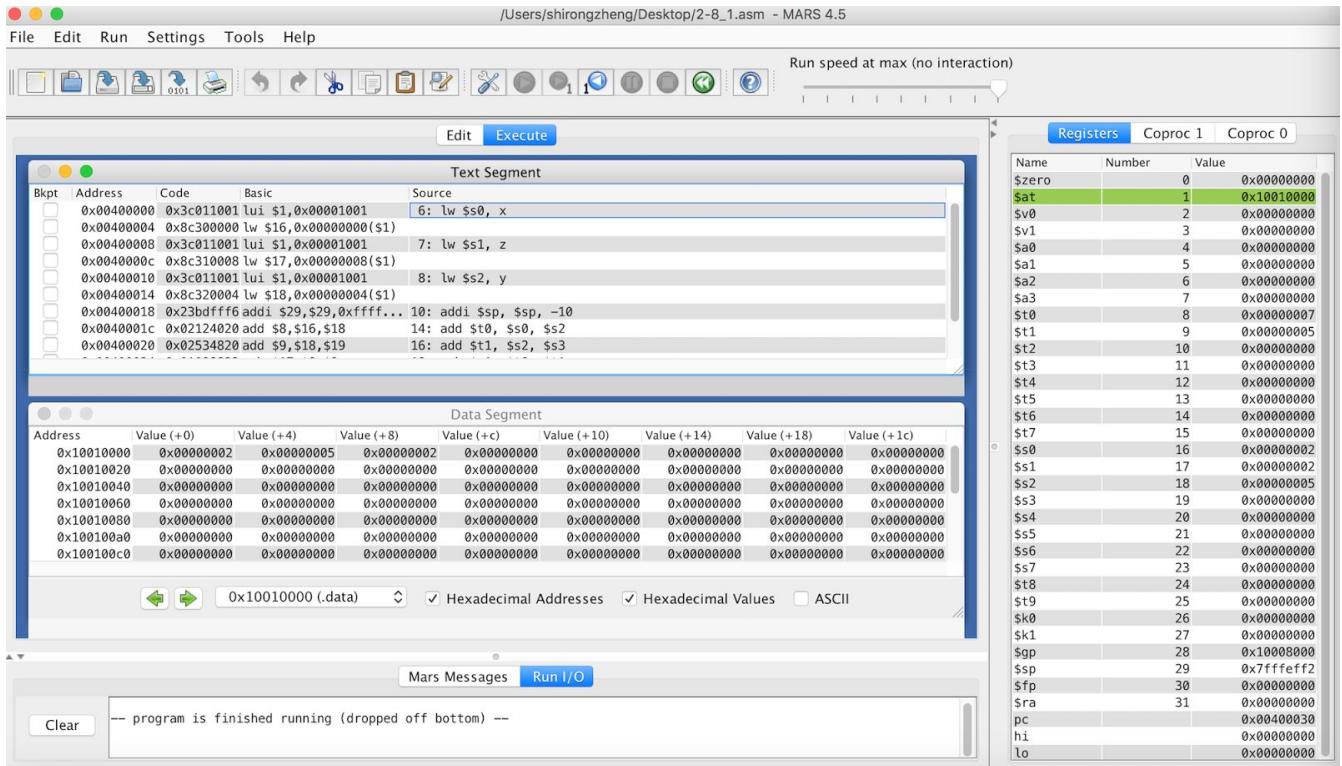


The below is my 2-8_2.asm code. The lw mean load address. The addi in myadd function is $\$sp = \$sp + (-10)$. Then math function, $z = (0+x) - (0+b)$.

```
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 0X00000000 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 0x7fffff2, 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.

The screenshot shows the MARS 4.5 assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. The title bar indicates the file is "/Users/shirongzheng/Desktop/2-8_1.asm - MARS 4.5".

Text Segment Window:

Bkpt	Address	Code	Basic	Source
	0x0040000c	0x8c310008 lw \$17, 0x00000008(\$1)		
	0x00400010	0x3c011001 lui \$1, 0x00001001	8: lw \$s2, y	
	0x00400014	0x8c320004 lw \$18, 0x00000004(\$1)		
	0x00400018	0x23bdffff addi \$29,\$29,0xffff...	10: addi \$sp, \$sp, -10	
	0x0040001c	0x02124020 add \$8,\$16,\$18	14: add \$t0, \$s0, \$s2	
	0x00400020	0x02534820 add \$9,\$18,\$19	16: add \$t1, \$s2, \$s3	
	0x00400024	0x01098822 sub \$17,\$8,\$9	18: sub \$t1, \$t0, \$t1	
	0x00400028	0x3c011001 lui \$1, 0x00001001	19: sw \$s1, z	
	0x0040002c	0xac310008 sw \$17, 0x00000008(\$1)		

Data Segment Window:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0x00000005	0x00000002	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

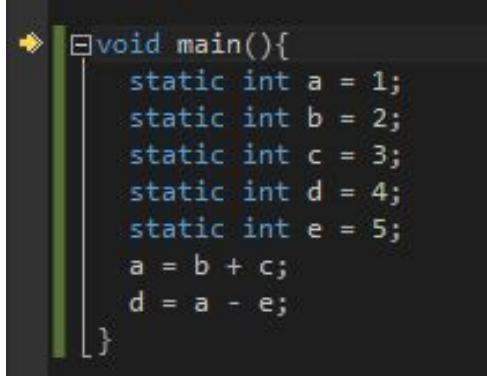
Registers Window:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000007
\$t1	9	0x00000005
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000002
\$s1	17	0x00000002
\$s2	18	0x00000005
\$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	0x10000000
\$sp	29	0x7ffffeff2
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400030
hi		0x00000000
lo		0x00000000

◆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.

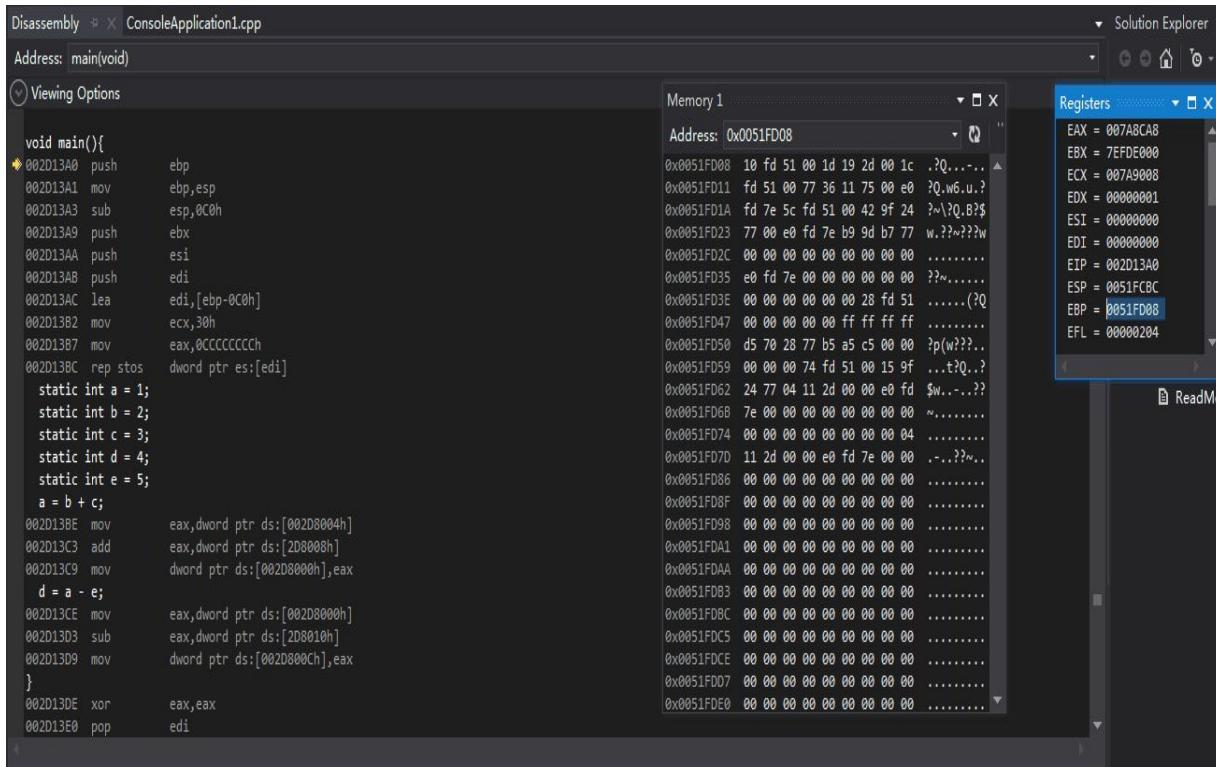
The below is my 2-2_1.cpp file. I will assign some constant values to my static variables.



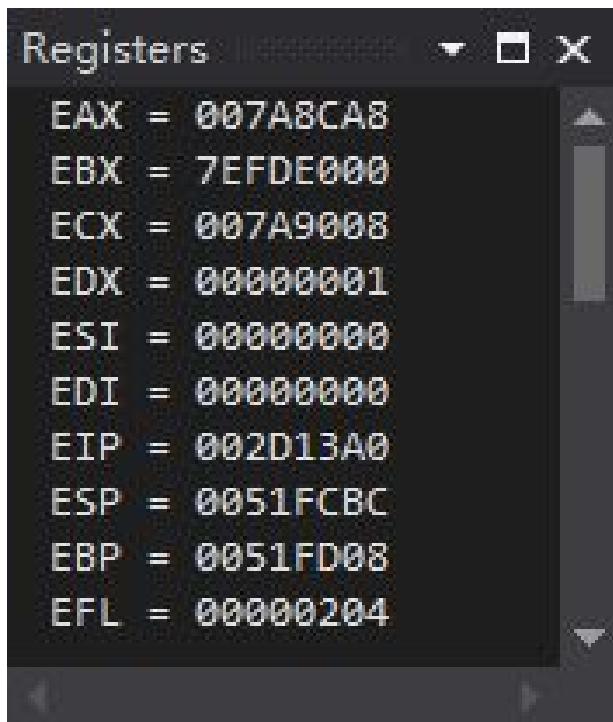
```
void main(){
    static int a = 1;
    static int b = 2;
    static int c = 3;
    static int d = 4;
    static int e = 5;
    a = b + c;
    d = a - e;
}
```

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.

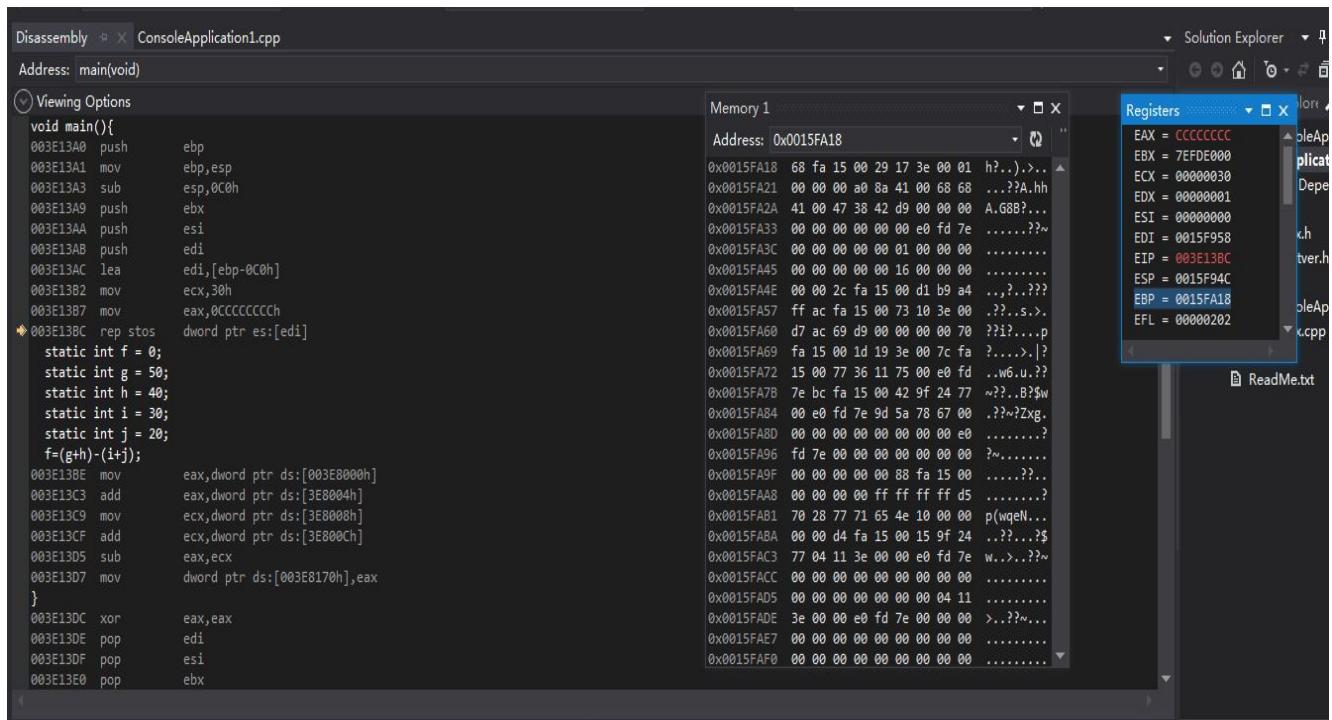
002D13DE	xor	eax,eax
002D13E0	pop	edi
002D13E1	pop	esi
002D13E2	pop	ebx
002D13E3	mov	esp,ebp
002D13E5	pop	ebp
002D13E6	ret	

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

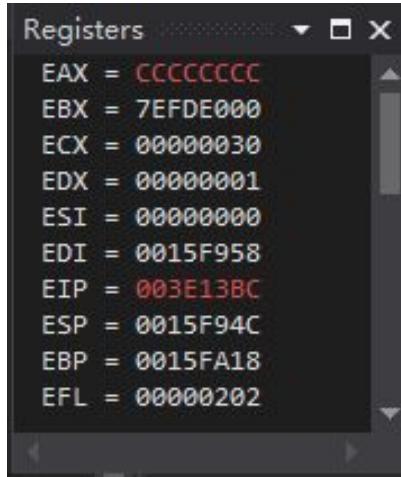
```
void main(){
    static int f = 0;
    static int g = 50;
    static int h = 40;
    static int i = 30;
    static int j = 20;
    f=(g+h)-(i+j);
}
```

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.

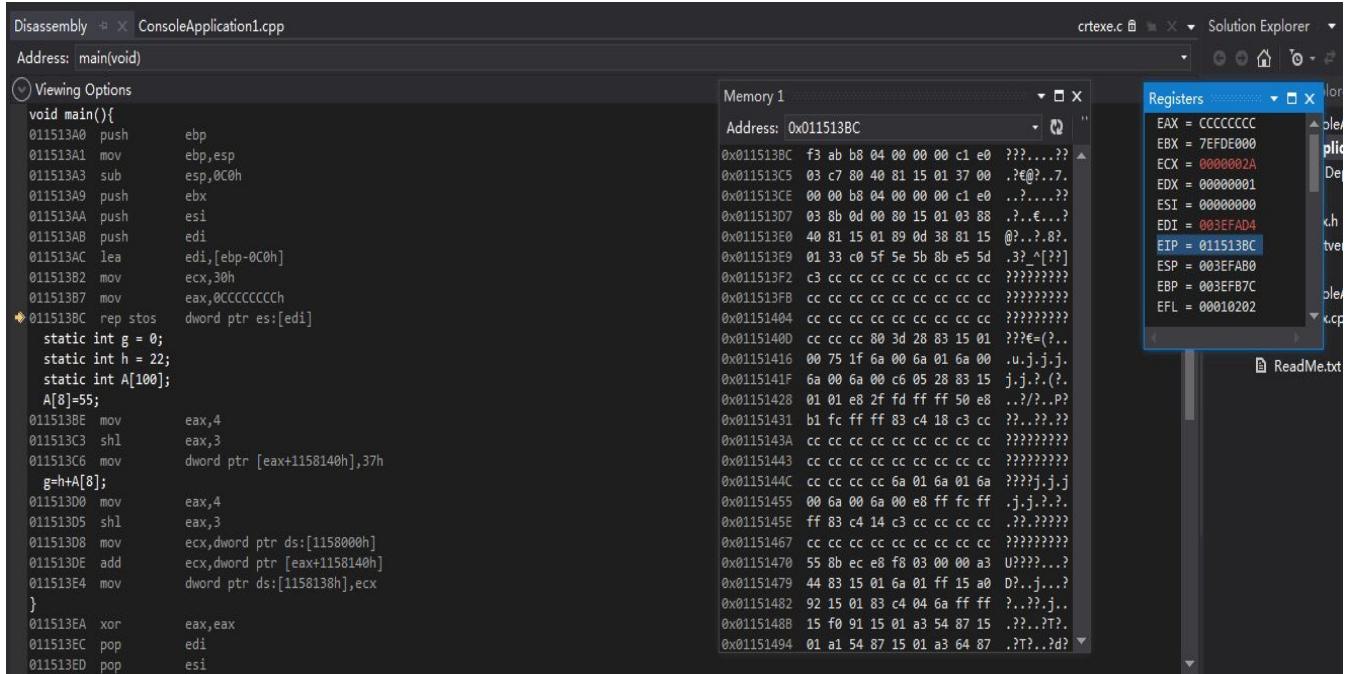
```
003E13A0    push      ebp
003E13A1    mov       ebp,esp
003E13A3    sub       esp,0C0h
003E13A9    push      ebx
003E13AA    push      esi
003E13AB    push      edi
003E13AC    lea       edi,[ebp-0C0h]
003E13B2    mov       ecx,30h
003E13B7    mov       eax,0CCCCCCCCh
003E13BC    rep stos   dword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

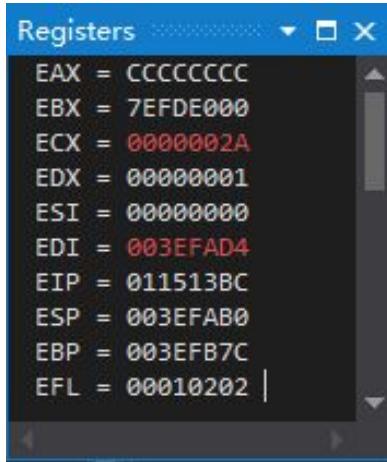
```
void main(){
    static int g = 0;
    static int h = 22;
    static int A[100];
    A[8]=55;
    g=h+A[8];
}
```

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.

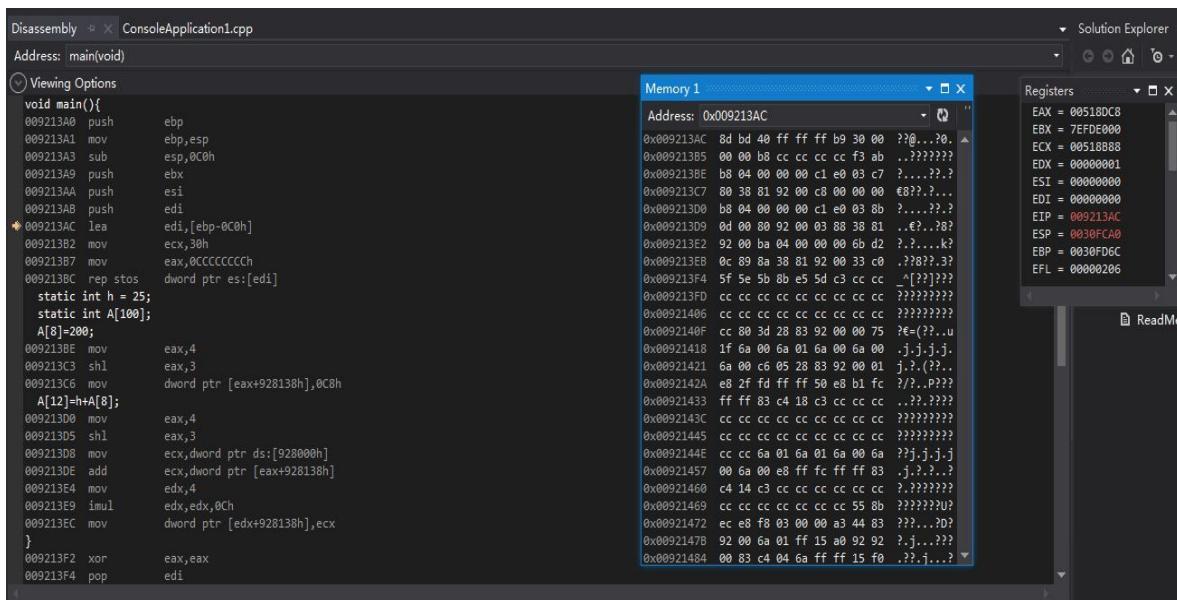


The below is my 2-3_2.cpp file. I will assign constant value 25 to my static variable h and also declare the array A[100].

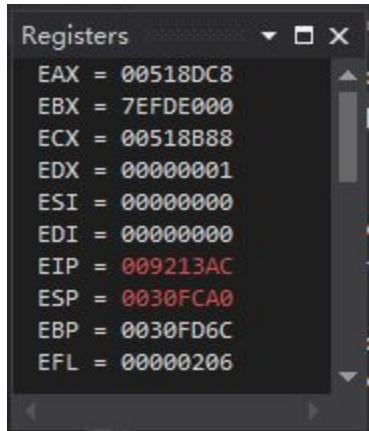
```
void main(){
    static int h = 25;
    static int A[100];
    A[8]=200;
    A[12]=h+A[8];
}
```

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".



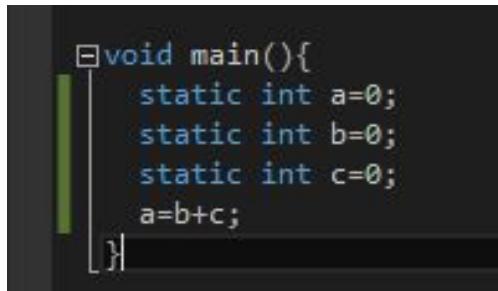
The below is my register window of this code. EFL=00000206.



A screenshot of a debugger's Registers window. The window title is "Registers". It lists the following register values:

Register	Value
EAX	00518DC8
EBX	7EFDE000
ECX	00518B88
EDX	00000001
ESI	00000000
EDI	00000000
EIP	009213AC
ESP	0030FCA0
EBP	0030FD6C
EFL	00000206

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

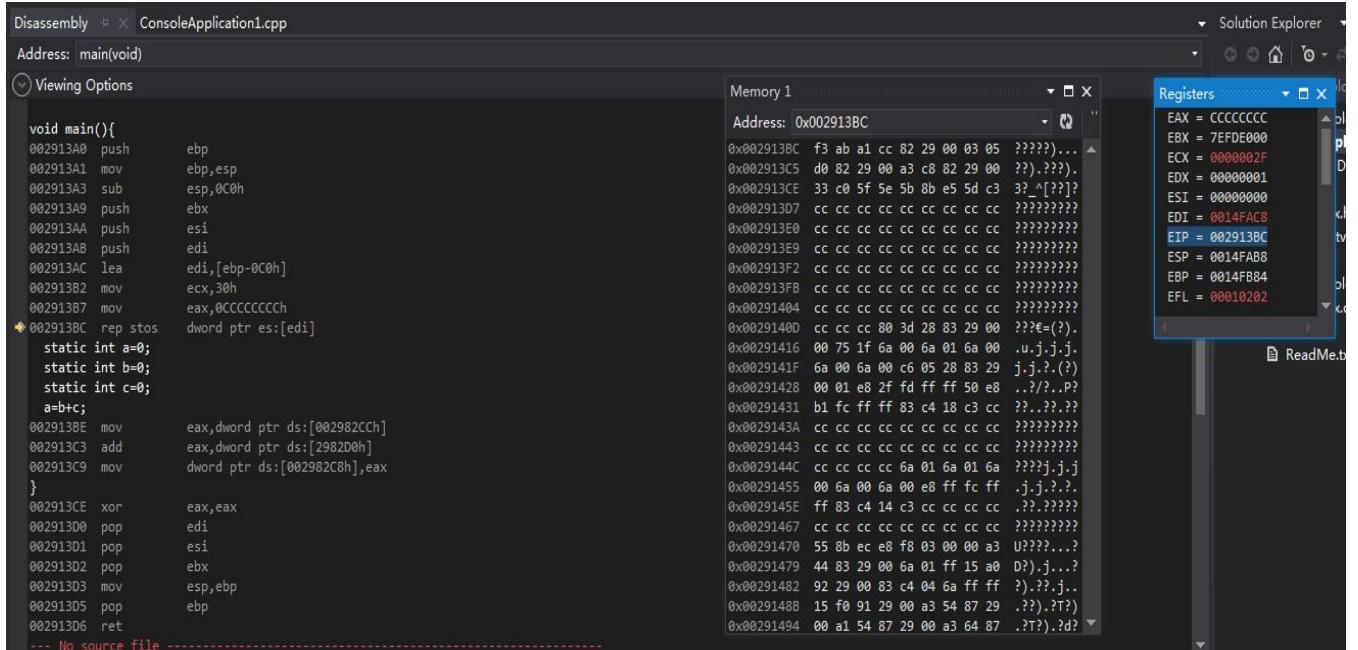


```
void main(){
    static int a=0;
    static int b=0;
    static int c=0;
    a=b+c;
}
```

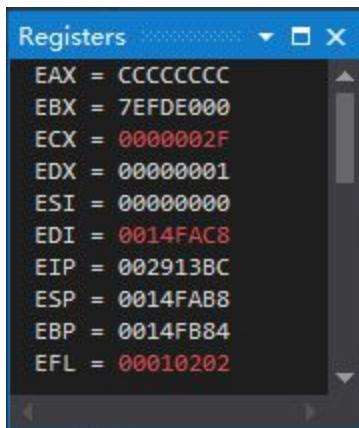
2-5_1.cpp

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

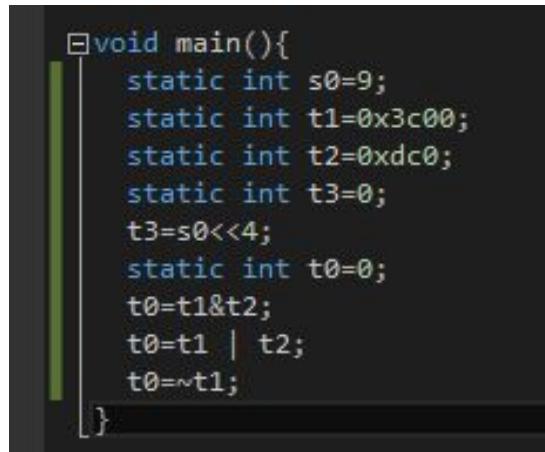
the target operand is 32 bits", so this will bitwise-AND the 32-bit value at the address computed by taking the contents of the ebp register and subtracting four with 0.



In the register, the EAX=CCCCCC, 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 screenshot shows the Microsoft Visual Studio IDE interface. The left pane displays the assembly code for the `main` function:

```

void main(){
    push    ebp
    mov     ebp,esp
    sub    esp,0C0h
    push    ebx
    push    esi
    push    edi
    lea     edi,[ebp-0C0h]
    mov     ecx,30h
    mov     eax,0CCCCCCCCh
    rep stos dword ptr es:[edi]
    static int s0=0;
    static int t1=0x3C00;
    static int t2=0xdC0;
    static int t3=0;
    t3=s0<4;
    mov     eax,dword ptr ds:[0019802Ch]
    shl     eax,4
    mov     dword ptr ds:[00198028h],eax
    static int t0=0;
    t0=t1&t2;
    mov     eax,dword ptr ds:[00198030h]
    and     eax,dword ptr ds:[198034h]
    mov     dword ptr ds:[0019802Ch],eax
    t0=t1 | t2;
    mov     eax,dword ptr ds:[00198030h]
    or     eax,dword ptr ds:[198034h]

```

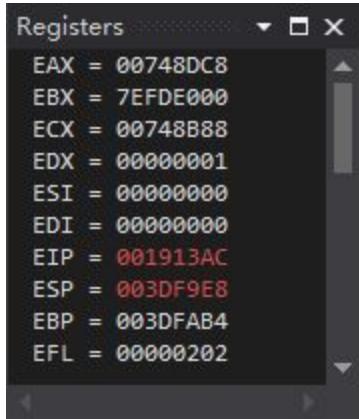
The middle pane shows the memory dump for address **0x001913AC**:

Address	Value	Content
0x001913AC	8d bd 40 ff ff fb b9 30 00	??@...?0.
0x001913B5	00 00 b8 cc cc cc cc f3 ab	..???????
0x001913BE	a1 2c 80 19 00 c1 e0 04 a3	?,\$..???
0x001913C7	c8 82 19 00 a1 30 80 19 00	??.?0E..
0x001913D0	23 05 34 80 19 00 a3 cc 82	#,.4E..???
0x001913D9	19 00 a1 30 80 19 00 b6 05	.?0E....
0x001913E2	34 80 19 00 a3 cc 82 19 00	4E..???
0x001913EB	a1 30 80 19 00 f7 d0 a3 cc	?0E..???
0x001913F4	82 19 00 33 c0 5f 5e 5b 8b	?..?3_?[?]
0x001913FD	e5 5d c3 cc cc cc cc cc	?]????????
0x00191406	cc cc cc cc cc cc cc cc	???????????
0x0019140F	cc 80 3d 28 83 19 00 00 75	?€=?...u
0x00191418	1f 6a 00 6a 01 6a 00 6a 00	.j..j..j..j.
0x00191421	6a 00 c6 05 28 83 19 00 01	j.?.(?..
0x0019142A	e8 2f fd ff 50 e8 b1 fc	?/.P???
0x00191433	ff ff 83 c4 18 c3 cc cc cc	..???.???
0x0019143C	cc cc cc cc cc cc cc	???????????
0x00191445	cc cc cc cc cc cc cc cc	???????????
0x0019144E	cc cc 6a 01 6a 01 6a 00 6a	??j..j..j..j
0x00191457	00 6a 00 e8 ff fc ff ff 83	.j..?..?..?
0x00191460	c4 14 c3 cc cc cc cc cc	????????
0x00191469	cc cc cc cc cc cc cc 55 8b	??????U?
0x00191472	e8 f8 03 00 00 a3 44 83	??.?.?D?
0x0019147B	19 00 6a 01 ff 15 a0 92 19	.j..?..?
0x00191484	00 83 c4 04 6a ff ff 15 f0	.??.j..?.

The right pane shows the Registers window:

Register	Value
EAX	00748DC8
EBX	7EFDE000
ECX	00748B88
EDX	00000001
ESI	00000000
EDI	00000000
EIP	001913AC
ESP	003DF9E8
EBP	003DFAB4
EFL	00000202

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.

```
2-2_1.cpp (~) - VIM

int 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;
}
```

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

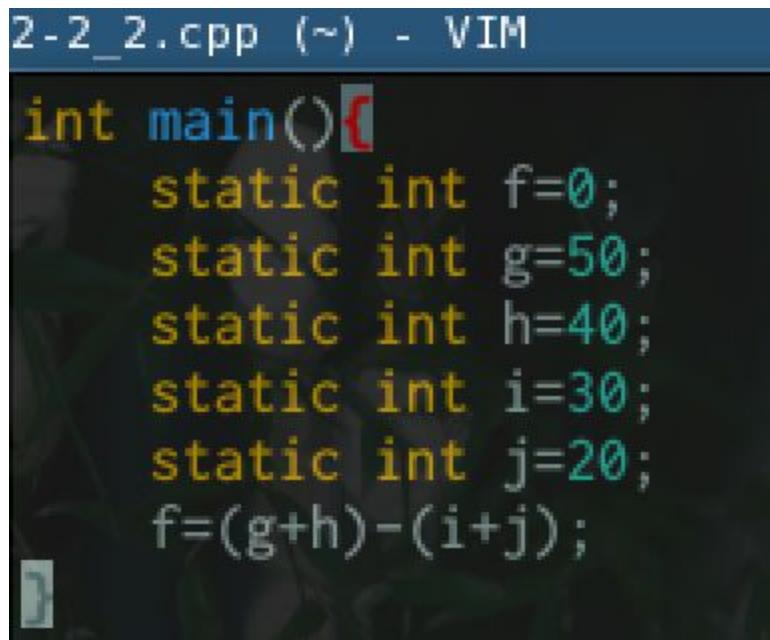
Breakpoint 1, main () at 2-2_1.cpp:7
7          a=b+c;
+(gdb) next 2
9      }
+(gdb) disassemble
Dump of assembler code for function main():
0x00000000004004c6 <+0>:    push   %rbp
0x00000000004004c7 <+1>:    mov    %rsp,%rbp
0x00000000004004ca <+4>:    mov    0x200b5c(%rip),%edx      # 0x60102c <_ZZ4mainE1b>
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.



```
2-2_2.cpp (~) - VIM
int 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);
```

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

```
+(gdb) break main
Breakpoint 1 at 0x4004ca: file 2-2_2.cpp, line 7.
+(gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at 2-2_2.cpp:7
7          f=(g+h)-(i+j);
+(gdb) next 2
0x00007ffff71b9291 in __libc_start_main () from /usr/lib/libc.so.6
+(gdb) disassemble
```

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.

Arch_Linux [Running]

```
xterm
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 0x7ffff71cecc20 <_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---
```

1 IPv6 | 16.3 GiB | DHCP: no | VPN: no | W: down | E: 10.0.2.15 (1000 Mbit/s) | CHR 75.00% 01:18:56 | 0.00 | 2017-03-21 14:56:30

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

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

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].

```
2-3_1.cpp + (~) - VIM
int main(){
    static int g=0;
    static int h=22;
    static int A[100];
    A[8]=55;
    g=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 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 0xfffffd60. \$2 which is A[12] and it's address is 0xfffffd60.

```
+ (gdb) break main
Breakpoint 1 at 0x4004ca: file 2-3_2.cpp, line 4.
+ (gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at 2-3_2.cpp:4
4           A[8]=200;
+ (gdb) next 2
6
+ (gdb) disassemble
Dump of assembler code for function main():
0x00000000004004c6 <+0>:    push   %rbp
0x00000000004004c7 <+1>:    mov    %rsp,%rbp
0x00000000004004ca <+4>:    movl   $0xc8,0x200bac(%rip)      # 0x601080 <_ZZ4mainE1A+32>
0x00000000004004d4 <+14>:   mov    0x200ba6(%rip),%edx      # 0x601080 <_ZZ4mainE1A+32>
0x00000000004004da <+20>:   mov    0x200b48(%rip),%eax      # 0x601028 <_ZZ4mainE1h>
0x00000000004004e0 <+26>:   add    %edx,%eax
0x00000000004004e2 <+28>:   mov    %eax,0x200ba8(%rip)      # 0x601090 <_ZZ4mainE1A+48>
=> 0x00000000004004e8 <+34>:  mov    $0x0,%eax
0x00000000004004ed <+39>:  pop    %rbp
0x00000000004004ee <+40>:  retq
End of assembler dump.
+ (gdb) display /i$pc
1: x/i $pc
=> 0x4004e8 <main()>+34:     mov    $0x0,%eax
+ (gdb) print /x$ebp
$1 = 0xfffffd60
+ (gdb) print /x$esp
$2 = 0xfffffd60
+ (gdb) 
```

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.

```
2-5_1.cpp (~) - VIM
int main(){
    static int a=0;
    static int b=0;
    static int c=0;
    a=b+c;
```

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

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

Breakpoint 1, main () at 2-5_1.cpp:5
5           a=b+c;
+(gdb) next 2
0x00007ffff71b9291 in __libc_start_main () from /usr/lib/libc.so.6
+(gdb) disassemble
```

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 is 0x4004f0 . \$2 which is c and it's address is 0xfffffd70.

```

+(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 = 0xfffffd70
+(gdb) ■

```

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

```
2-6_1.cpp (~) - VIM
int 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;
```

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.

```

+(gdb) break main
Breakpoint 1 at 0x4004ca: file 2-6_1.cpp, line 6.
+(gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at 2-6_1.cpp:6
6          t3=s0<<4;
+(gdb) next 2
9          t0=t1 | t2;
+(gdb) disassemble
Dump of assembler code for function main():
0x00000000004004c6 <+0>:    push   %rbp
0x00000000004004c7 <+1>:    mov    %rsp,%rbp
0x00000000004004ca <+4>:    mov    0x200b58(%rip),%eax      # 0x601028 <_ZZ4mainE2s0>
0x00000000004004d0 <+10>:   shl    $0x4,%eax
0x00000000004004d3 <+13>:   mov    %eax,0x200b5f(%rip)      # 0x601038 <_ZZ4mainE2t3>
0x00000000004004d9 <+19>:   mov    0x200b4d(%rip),%edx      # 0x60102c <_ZZ4mainE2t1>
0x00000000004004df <+25>:   mov    0x200b4b(%rip),%eax      # 0x601030 <_ZZ4mainE2t2>
0x00000000004004e5 <+31>:   and    %edx,%eax
0x00000000004004e7 <+33>:   mov    %eax,0x200b4f(%rip)      # 0x60103c <_ZZ4mainE2t0>
=> 0x00000000004004ed <+39>:  mov    0x200b39(%rip),%edx      # 0x60102c <_ZZ4mainE2t1>
0x00000000004004f3 <+45>:   mov    0x200b37(%rip),%eax      # 0x601030 <_ZZ4mainE2t2>
0x00000000004004f9 <+51>:   or     %edx,%eax
0x00000000004004fb <+53>:   mov    %eax,0x200b3b(%rip)      # 0x60103c <_ZZ4mainE2t0>
0x0000000000400501 <+59>:   mov    0x200b25(%rip),%eax      # 0x60102c <_ZZ4mainE2t1>
0x0000000000400507 <+65>:   not    %eax
0x0000000000400509 <+67>:   mov    %eax,0x200b2d(%rip)      # 0x60103c <_ZZ4mainE2t0>
0x000000000040050f <+73>:   mov    $0x0,%eax
0x0000000000400514 <+78>:   pop    %rbp
0x0000000000400515 <+79>:   retq
End of assembler dump.
+(gdb) 
```

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 is 0xfffffd60 . \$2 which is t2 and it's address is 0xfffffd60.

```

+(gdb) display /i$pc
1: x/i $pc
=> 0x4004ed <main()+39>:      mov    0x200b39(%rip),%edx      # 0x60102c <_ZZ4mainE2t1>
+(gdb) print /x$ebp
$1 = 0xfffffd60
+(gdb) print /x$esp
$2 = 0xfffffd60
+(gdb) 
```

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.

```

+(gdb) break main
Breakpoint 1 at 0x4004e8: file 2-8_1.cpp, line 9.
+(gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at 2-8_1.cpp:9
9          int b=5;
+(gdb) next 2
11         return 0;
+(gdb) disassemble
Dump of assembler code for function main():
0x00000000004004e0 <+0>:    push   %rbp
0x00000000004004e1 <+1>:    mov    %rsp,%rbp
0x00000000004004e4 <+4>:    sub    $0x10,%rsp
0x00000000004004e8 <+8>:    movl   $0x5,-0x4(%rbp)
0x00000000004004ef <+15>:   mov    0x200b33(%rip),%eax      # 0x601028 <_ZZ4mainE1a>
0x00000000004004f5 <+21>:   mov    -0x4(%rbp),%edx
0x00000000004004f8 <+24>:   mov    %edx,%esi
0x00000000004004fa <+26>:   mov    %eax,%edi
0x00000000004004fc <+28>:   callq  0x4004c6 <myadd(int, int)>
0x0000000000400501 <+33>:   mov    %eax,-0x8(%rbp)
=> 0x0000000000400504 <+36>:  mov    $0x0,%eax
0x0000000000400509 <+41>:   leaveq 
0x000000000040050a <+42>:   retq  
End of assembler dump.

```

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 is 0xfffffd60 . \$2 which is b and it's address is 0xfffffd50.

```

+(gdb) display /i$pc
3: x/i $pc
=> 0x400504 <main() + 36>:      mov    $0x0,%eax
+(gdb) print /x$ebp
$3 = 0xfffffd60
+(gdb) print /x$esp
$4 = 0xfffffd50

```

❖ 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.