Weifan Lin

Csc342 Section G

03/01/2015

Homework

Title:

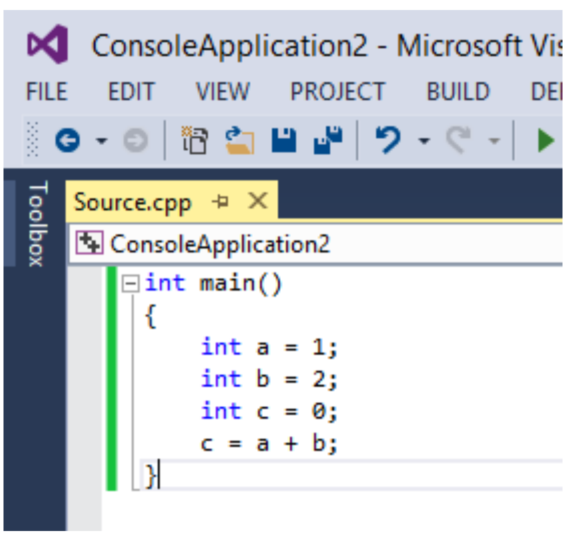
Comparison Instruction Set Architecture

Objective:

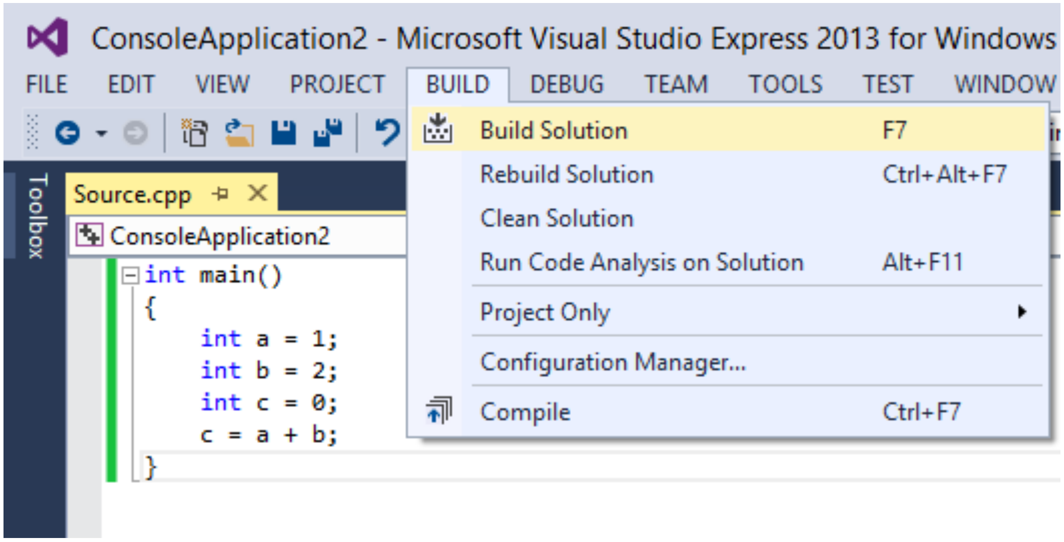
The goal to understand the instruction set of a computer by debugging a piece of code in c language on three different platforms: MS Debugger, GDB, and MIPS on MARS.

MS Debugger:

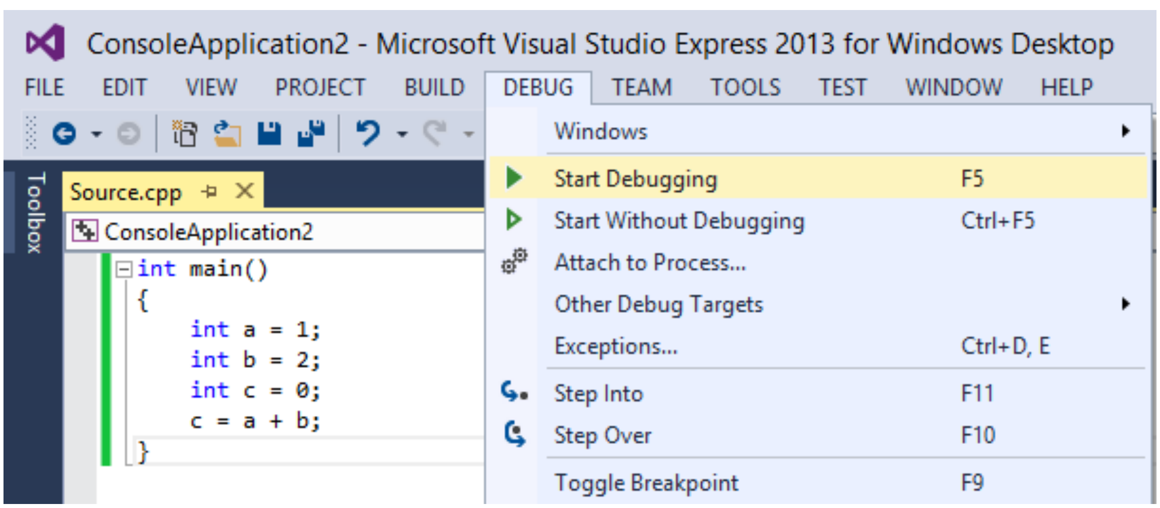
To begin we write the following code in Microsoft Visual Studio



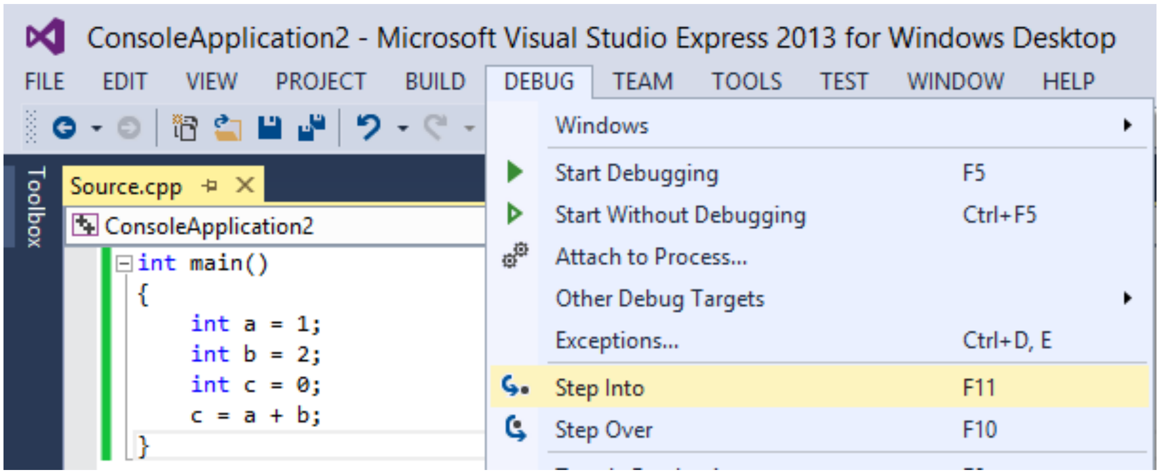
And build it as we click on “Build Solution” under BUILD menu.



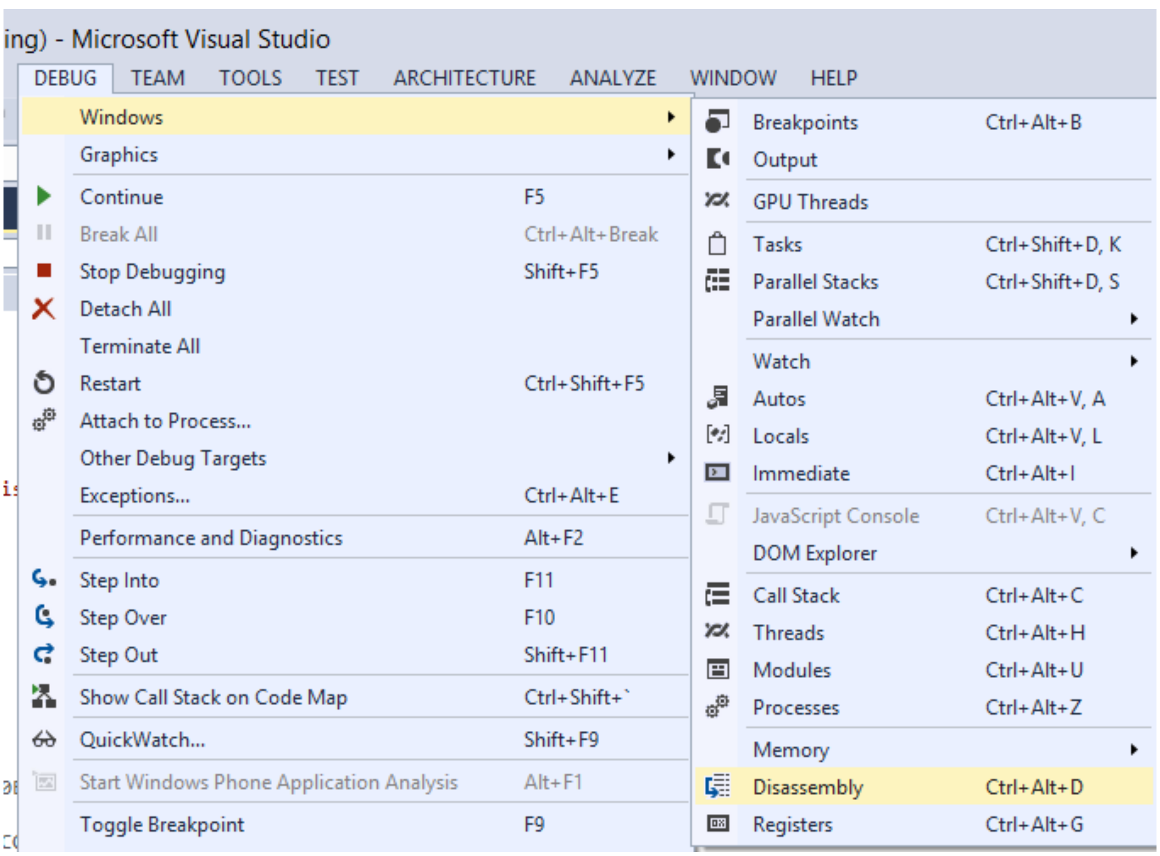
Then we go to DEBUG menu and start debugging.



After finish debugging, click “Step Into”, this allows us to execute code one statement at a time.

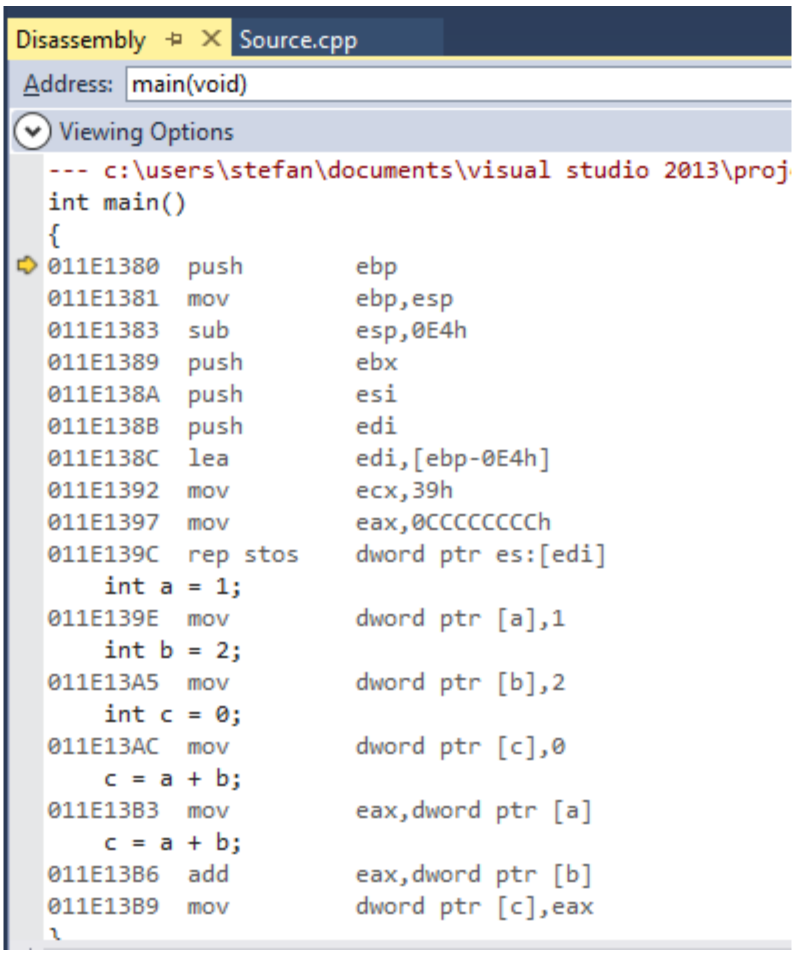


Now we can go to “Windows” under DEBUG menu to view Disassembly and Registers.



The Disassembly window shows the assembly code and memory address where each instruction is located. The Registers window displays register contents and change of register values as our code executes.

Disassembly window:



As we can see there is a yellow arrow points to “011E1380 push ebp”, this indicates which instruction that we are excuted. We can press F10 to step over to next instruction, and the register values will change as well.

“011E1380   push   ebp”

“012F52A0” is a address in hexadecimal refer to an instruction, in this instruction “push” means to save the value of current register, and “ebp” is base pointer register.

“011E1381   mov   ebp,esp”

esp is stack pointer, and this instruction copys register from ebp to esp and ebp now points to the top of the stack.

“011E1383   sub   esp,0E4h”

In this instruction is to allocate space for local variables. 0E4h is hexadecimal has value of 228 in decimal, sub means to subtract 228 bytes for local variables.

“011E1389   push   ebx”

“011E138A   push   esi”

“011E138B   push   edi”

These three instructions are to save processor registers used for temporaries. ebx is base pointer, esi is source index register, and edi is destination index register.

“int a = 1;”

“011E139E mov dword ptr [a],1”

“int b = 2;”

“011E13A5 mov dword ptr [b],2”

“int c = 0;”

“011E13AC mov dword ptr [c],0”

“c = a + b;”

“011E13B3 mov eax,dword ptr [a]”

“011E13B6 add eax,dword ptr [b]”

“011E13B9 mov dword ptr [c],eax”

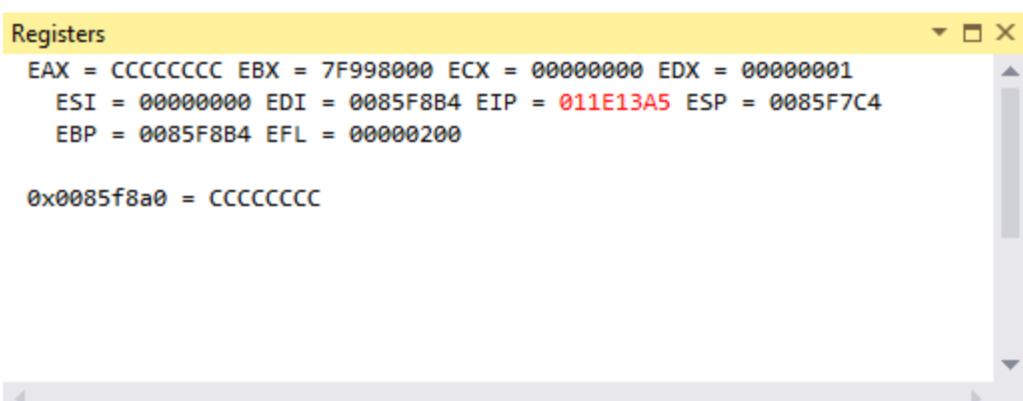
After these instructions, now the local variables are located on the stacks between ebp and esp.

Now we continue press F10 until the yellow arrow points to

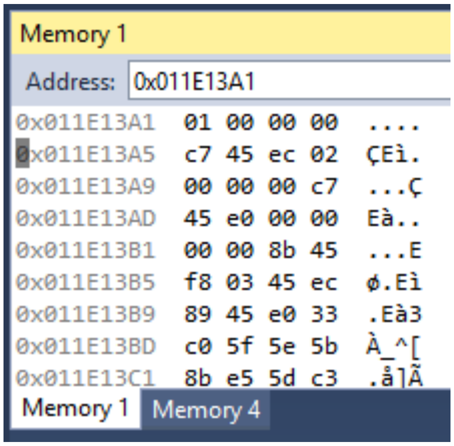
“int b = 2;”

“011E13A5 mov dword ptr [b],2”

and we look at the register window:

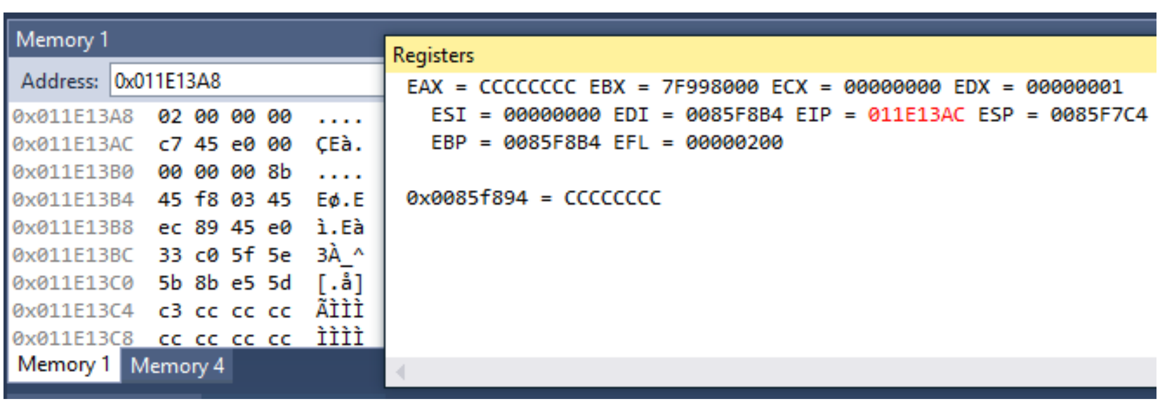


EIP (instruction pointer) address is 011E13A5. We copy it and paste it into memory window:

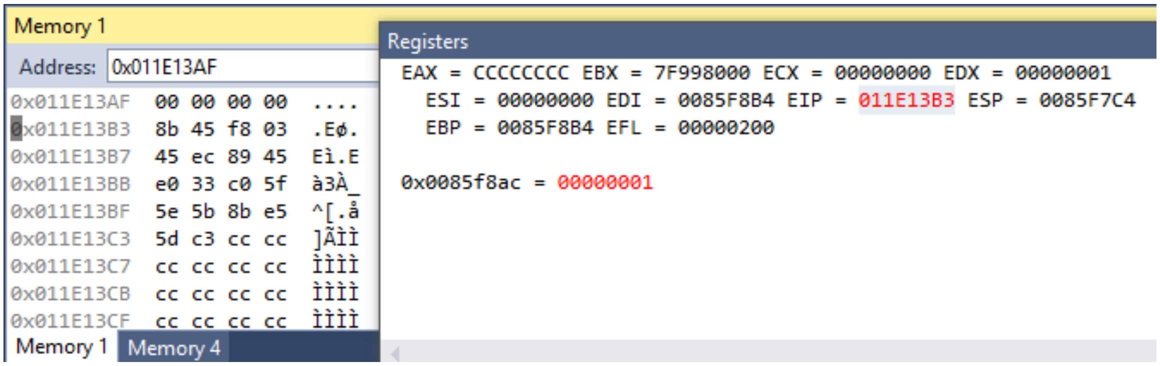


and we scroll up a little we can see the address 011E13A1 which is the address of variable “a” that has hexadecimal value of “01000000”, “01” are the least 2 significant bits and “a” has the deciaml value of 1.

If we press F10 one more time and we do the same thing we can find the variable “b” that has decimal value of 2:



and the same thing finding the variable “c”:



GDB:

We have this following program code in c from our book (p65), and we name it hw342.c

```c
int main()
{
    int a, b, c, d, e;
    a = b + c;
    d = a - e;
    return 0;
}
```

Now we can compile it with debugging symbols and no optimizations and then run GDB

```
stefan@Ubuntu:~/Desktop$ CFLAGS="-g -O0" make hw342
cc -g -O0     hw342.c    -o hw342
stefan@Ubuntu:~/Desktop$ gdb hw342
```

Inside GDB, we will break on main and run until we get to the return statement. We put the number 2 after next to specify that we want to run next twice

```
(gdb) break main
Breakpoint 1 at 0x4004f1: file hw342.c, line 4.
(gdb) run
Starting program: /home/stefan/Desktop/hw342

Breakpoint 1, main () at hw342.c:4
4               a = b + c;
(gdb) next 2
6               return 0;
```

Now as we type in disassemble command it shows the assembly instructions for the current function

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     mov    -0x10(%rbp),%eax
   0x00000000004004f4 <+7>:     mov    -0x14(%rbp),%edx
   0x00000000004004f7 <+10>:    add    %edx,%eax
   0x00000000004004f9 <+12>:    mov    %eax,-0xc(%rbp)
   0x00000000004004fc <+15>:    mov    -0x8(%rbp),%eax
   0x00000000004004ff <+18>:    mov    -0xc(%rbp),%edx
   0x0000000000400502 <+21>:    sub    %eax,%edx
   0x0000000000400504 <+23>:    mov    %edx,%eax
   0x0000000000400506 <+25>:    mov    %eax,-0x4(%rbp)
=> 0x0000000000400509 <+28>:    mov    $0x0,%eax
   0x000000000040050e <+33>:    pop    %rbp
   0x000000000040050f <+34>:    retq
End of assembler dump.
```

```
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
```

The first two instructions are called the function prologue or preamble. We first push the old base pointer onto the stack to save it for later. Then we copy the value of the stack pointer the base pointer. After this, %rbp points to the base of main’s stack frame.

```
0x00000000004004f1 <+4>:     mov    -0x10(%rbp),%eax
```

%rbp is called the base register, and -0x10 is the displacement. This is equivalent to %rbp + -0x10. Because the stack grows downwards, subtracting 10 from the base of the current stack frame moves us into the current frame itself, where local variables are stored. And this instruction moves b into %eax.

```
0x00000000004004f4 <+7>:     mov    -0x14(%rbp),%edx
```

This is the same thing as last instruction, subtracting 14 form the base of the current stack frame and moves c into %edx.

```
0x00000000004004f7 <+10>:    add    %edx,%eax
0x00000000004004f9 <+12>:    mov    %eax,-0xc(%rbp)
```

This instruction clearly is the add the value in %edx to %eax, and store the result in %rbp -0xc. So it is add c to b and stores the result a in %rbp.

```
0x00000000004004fc <+15>:    mov    -0x8(%rbp),%eax
0x00000000004004ff <+18>:    mov    -0xc(%rbp),%edx
0x0000000000400502 <+21>:    sub    %eax,%edx
0x0000000000400504 <+23>:    mov    %edx,%eax
0x0000000000400506 <+25>:    mov    %eax,-0x4(%rbp)
```

Now move a into %eax and move e into %edx, and sub e from a and move the result %edx to %eax as value d. And lastly move d to %rbp.

Lets check if it’s correct. As we type in “x &d” to find the memory address of d, we get 0xffffdf20. And to verify, we type in “x $rbp -4” we get the same address.

```
(gdb) x &d
0x7fffffffde3c: 0xffffdf20
(gdb) x $rbp -4
0x7fffffffde3c: 0xffffdf20
```

```
=> 0x0000000000400509 <+28>:    mov    $0x0,%eax
   0x000000000040050e <+33>:    pop    %rbp
   0x000000000040050f <+34>:    retq
```

Lastly, we move value 0 to %eax, because this the our return value at the end. And we pop the old base pointer off the stack and store it back in %rbp and then retq jumps back to our return address, which is also stored in the stack frame.