Making plain binary files using a C compiler (i386+)

Cornelis Frank

April 10, 2000

I wrote this article because there isn't much information on the Internet concerning this topic and I needed this for the EduOS project.

So if you blow up your computer because of my bad "English" that's your problem not mine.

1 Which tools do you need?

- An i386 PC or higher.
- A Linux distribution like Red Hat or Slackware.
- GNU GCC compiler. This C compiler usually comes with Linux. To check if you're having GCC type the following at the prompt:

  ```
  gcc --version
  ```

  This should give an output like:

  ```
  2.7.2.3
  ```

  The number probably will not match the above one, but that doesn't really matter.
- The `binutils` for Linux.
- NASM Version 0.97 or higher. The Netwide Assembler, NASM, is an 80x86 assembler designed for portability and modularity. It supports a range of object file formats, including Linux 'a.out' and ELF, NetBSD/FreeBSD, COFF, Microsoft 16-bit OBJ and Win32. It will also output plain binary files. Its syntax is designed to be simple and easy to understand, similar to Intel's but less complex. It supports Pentium, P6 and MMX opcodes, and has macro capability.
  Normally you don't have NASM on your system. Download it from:
  `http://sunsite.unc.edu/pub/Linux/devel/lang/assemblers/`
- A text editor like `pico` or `emacs`.

1.1 Installing The Netwide Assembler

Assuming that `nasm-0.97.tar.gz` is in the current directory type:

```
gunzip nasm-0.97.tar.gz
tar -vxf nasm-0.97.tar
```

This will create a directory called `nasm-0.97`. Go to that directory. Next we will compile this assembler by typing:

```
./configure
make
```

This will create the executables `nasm` and `ndisasm`. You can copy these files to you `/usr/bin` directory to make them easily accessible. Now you can remove the `nasm-0.97` directory from your system. I personally compiled the NASM successfully under Red Hat 5.1 and Slackware 3.1, so this shouldn't give big troubles.

2 Making a first binary file using C

Create a file called `test.c` using your text editor. Put herein:

```
int main () {
}
```

Compile this by typing:

```
gcc -c test.c
ld -o test -Ttext 0x0 -e main test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

This creates our binary file called `test.bin`. We can view this binary file using `ndisasm`. Do this by typing:

```
ndisasm -b 32 test.bin
```

This will give the following output:

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  C9                leave
00000004  C3                ret
```

We get three columns. The first one contains the memory addresses of the instructions. The second column contains the byte code of the instructions and the last column contains the instruction itself.

2.1 Dissection of test.bin

The code we get just seems to set up a basic framework for a function. The register `ebp` is being saved for later use concerning function parameter handling. As you can notice the code is 32 bit. GNU GCC only can create 32 bit code. So if you would like to run this code you first need to set up a 32 bit environment like Linux does. Here fore you need to go to protected mode.
You can also create directly a binary file using `ld`. Here fore compile `test.c` like this:

```
gcc -c test.c
ld test.o -o test.bin -Ttext 0x0 -e main -oformat binary
```

This will produce exactly the same binary code as the previous method.

3 Program using a local variable

Next we will take a look on how GCC handles the reservation of a local variable. Here fore we will create a new `test.c` which contains:

```
int main () {
  int i; /* declaration of an int */
  i = 0x12345678; /* hexadecimal */
}
```

Compile this by typing:

```
gcc -c test.c
ld -o test -Ttext 0x0 -e main test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

After we compiled we get the next binary file:

```
00000000  55              push ebp
00000001  89E5            mov ebp,esp
00000003  83EC04          sub esp,byte +0x4
00000006  C745FC78563412  mov dword [ebp-0x4],0x12345678
0000000D  C9              leave
0000000E  C3              ret
```

3.1 Dissection of test.bin

The first two and last two instructions are the same as in the previous example. There are only two new instructions added between the old ones. The first one decreases `esp` with 4. This is the way GCC reserves an `int`, which is four bytes in size, on the stack. The following instruction immediately demonstrates us the usage of the `ebp` register. This register remains unchanged in the function and is only used to refer to the local variables on the stack. The place on the stack were these local variables are stored is usually called the local stack frame. In this context the `ebp` register is called the frame pointer.
The next instruction fills the on the stack reserved `int` up with the value `0x12345678`. Also notice the reversed order in which the processor stores data. In the second column, line four, we see `...78563412`. This phenomena is called *backwards storage*[1].
Note that you also can create directly a binary file using `ld` as shown before. So compile with:

```
gcc -c test.c
ld -o test.bin -Ttext 0x0 -e main -oformat binary test.o
```

This gives us the same binary file as before.

3.2 Direct assignment

When we change:

```
int i;
i = 0x12345678;
```

[1]See also: Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 1.4.1. Bit and Byte Order

into,

```
int i = 0x12345678;
```

we get exactly the same binary file. This is very important to notice as it is *not* so when we use global variables.

4 Program using a global variable

Next we will take a look on how GCC handles global variables. This will be done using the next `test.c` program.

```
int i; /* declaration of global variable */
int main () {
  i = 0x12345678;
}
```

Compile this by typing:

```
gcc -c test.c
ld -o test -Ttext 0x0 -e main test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

This leads us to the following binary code:

```
00000000  55                  push ebp
00000001  89E5                mov ebp,esp
00000003  C705101000007856    mov dword [0x1010],0x12345678
         -3412
0000000D  C9                  leave
0000000E  C3                  ret
```

4.1 Dissection of test.bin

The instruction in the middle of the code will write our value we assigned to somewhere in the memory, in our case to address `0x1010`. This is because by default the linker `ld` page-aligns the data segment. We can turn this off by using the parameter `-N` with the linker `ld`. This gives us as binary file:

```
00000000  55                  push ebp
00000001  89E5                mov ebp,esp
00000003  C705100000007856    mov dword [0x10],0x12345678
         -3412
0000000D  C9                  leave
0000000E  C3                  ret
```

As we can see now, the data is stored right after the code. We can also specify the data segment ourself. Compile here fore the program `test.c` with:

```
gcc -c test.c
ld -o test -Ttext 0x0 -Tdata 0x1234 -e main -N test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

This will give us as binary file:

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  C705341200007856  mov dword [0x1234],0x12345678
         -3412
0000000D  C9                leave
0000000E  C3                ret
```

Now the global variable is being stored at our gives address `0x1234`. Thus, if we use the parameter `-Tdata` with `ld`, we can specify the location of the data segment ourself. Otherwise the data segment is located right after the code. By storing the variable somewhere in the data memory it remains accessible even outside the `main` function. This is why they call `int i` a *global variable*.
We can also create directly the binary file using `ld` with the parameter `-oformat binary`.

4.2 Direct assignment

Some of my experiments point out that direct assigned global variables can be handled as normal global variables or can be stored as data directly after the code in the binary file. `ld` handles the global variables as data when there are already data constants used.
Take a look at the following program:

```
const int c = 0x12345678;
int main () {
}
```

Compile this with:

```
gcc -c test.c
ld -o test.bin -Ttext 0x0 -e main -N -oformat binary test.o
```

This gives as binary file:

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  C9                leave
00000004  C3                ret
00000005  0000              add [eax],al
00000007  007856            add [eax+0x56],bh
0000000A  3412              xor al,0x12
```

We can see that there are some extra bytes at the end of our binary file. This is a read-only data section aligned on 4 bytes which contains our global constant.

4.2.1 Usage of objdump

With `objdump` we can get even more information.

```
objdump --disassemble-all test.o
```

This gives us the next screen dump:

```
test.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:      55                pushl  %ebp
   1:      89 e5             movl   %esp,%ebp
   3:      c9                leave
   4:      c3                ret
Disassembly of section .data:
Disassembly of section .rodata:

00000000 <c>:
   0:      78 56             js     58 <main+0x58>
   2:      34 12             xorb   $0x12,%al
```

We can clearly see the read-only data section containing our global constant `c`. Now take a look at the next program:

```
int i = 0x12345678;
const int c = 0x12346578;
int main () {
}
```

When we compile this program and do an `objdump` on this we get:

```
test.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                   pushl  %ebp
   1:   89 e5                movl   %esp,%ebp
   3:   c9                   leave
   4:   c3                   ret
Disassembly of section .data:

00000000 <i>:
   0:   78 56                js     58 <main+0x58>
   2:   34 12                xorb   $0x12,%al
Disassembly of section .rodata:

00000000 <c>:
   0:   78 56                js     58 <main+0x58>
   2:   34 12                xorb   $0x12,%al
```

We can see our `int i` in the data section and our constant `c` in the read-only data section. So when `ld` has to use global constants it automatically uses the data section to store global variables.

5 Pointers

Now let's see how GCC handles pointers to variables. Therefore we will use the following program.

```
int main () {
  int i;
  int *p; /* a pointer to an integer */
  p = &i; /* let pointer p points to integer i */
  *p = 0x12345678; /* makes i = 0x12345678 */
}
```

This program results in the following binary code:

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  8D55FC            lea edx,[ebp-0x4]
00000009  8955F8            mov [ebp-0x8],edx
0000000C  8B45F8            mov eax,[ebp-0x8]
0000000F  C70078563412      mov dword [eax],0x12345678
00000015  C9                leave
00000016  C3                ret
```

5.1 Dissection of test.bin

Again the first two and last two instructions are the same as usual. Next we've got:

```
sub     esp,byte +0x8
```

This instruction will reserve 8 bytes on the stack for local variables. Seems like a pointer is being stored using 4 bytes. At this point the stack looks like in figure 1. As you can see the `lea` instruction

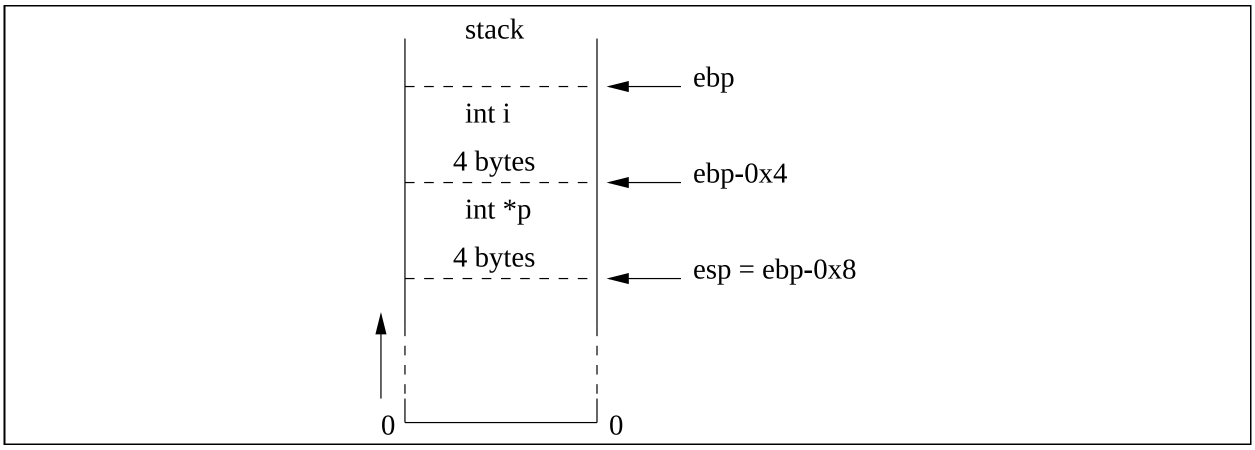

Figure 1: The stack

will load the effective address of `int i`. Next this value is being stored in `int *p`. After this the value of `int *p` is being used as a pointer to a `dword` wherein the value `0x12345678` is being stored.

6 Calling a function

Now let's take a look on how GCC handles function calls. Take a look at the next example:

```
void f (); /* function prototype */

int main () {
  f (); /* function call */
}

void f () { /* function definition */
}
```

This will give us as binary code:

```
  00000000  55                 push ebp
  00000001  89E5               mov ebp,esp
  00000003  E804000000         call 0xc
  00000008  C9                 leave
  00000009  C3                 ret
  0000000A  89F6               mov esi,esi
  0000000C  55                 push ebp
  0000000D  89E5               mov ebp,esp
  0000000F  C9                 leave
  00000010  C3                 ret
```

6.1 Dissection of test.bin

In the function `main` we can see clearly a `call` to the empty function `f` at address `0xC`. This empty function has the same basic structure as the function `main`. This means that there is no structural difference between the entry function and any other function. When you link using `ld` and you add `-M >mem.txt` to the `ld` parameters you will get a text file wherein you find usefull documentation on how everything is linked and stored into the memory. In the file `mem.txt` you'll find somewhere two lines like these:

```
  Address of section .text set to 0x0
  Address of section .data set to 0x1234
```

This means that the binary code starts at address `0x0` and the data area where the global variables are being stored starts at address `0x1234`. You'll also find something like:

```
  .text           0x00000000       0x11
   *(.text)
   .text          0x00000000       0x11 test.o
                  0x0000000c                f
                  0x00000000                main
```

The first column contains the name of the section. In our case it is a `.text` section. The second column contains the origin of the sections. The third column contains the length of the sections and the last column contains some extra information like the name of functions and used object files. We can see clearly now that the function `f` starts at offset `0xC` and that the function `main` is the entry point of the binary file. And the length `0x11` of the program is also correct since the last instruction (ret) is at address `0x10` and takes 1 byte.

6.2 Usage of objdump

`objdump` can be used to display information from object files. This information is useful to examin the internal structure of the object files. Use `objdump` by typing:

```
objdump --disassemble-all test.o
```

This will give the following output to the screen:

```
test.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:       55                pushl  %ebp
   1:       89 e5             movl   %esp,%ebp
   3:       e8 04 00 00 00    call   c <f>
   8:       c9                leave
   9:       c3                ret
   a:       89 f6             movl   %esi,%esi

0000000c <f>:
   c:       55                pushl  %ebp
   d:       89 e5             movl   %esp,%ebp
   f:       c9                leave
  10:       c3                ret
Disassembly of section .data:
```

Again this is very usefull when you want to study the binary code that GCC creates. Notice that they are not using the Intel syntax for displaying the instructions. They use instruction representations like `pushl` and `movl`. The `l` at the end of the instructions indicates that the instructions perform operations on 32-bit (`long`) operands. An other important difference contrary to Intels syntax is that the order of the operands is reversed. Next example shows us the two different notations for the instruction that moves the data from register `EBX` to register `EAX`.

```
MOV     EAX,EBX             ; Intel syntax
movl    %ebx,%eax           ; 'GNU' syntax
```

As for Intel the first operand is the destination and the second operand is the source.

7 Return codes

You probably noticed that I always use `int main ()` as my function definition, but I never actually return an `int`. So, let us try it.

```
int main () {
  return 0x12345678;
}
```

This program gives the following binary code:

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  B878563412        mov eax,0x12345678
00000008  EB02              jmp short 0xc
0000000A  89F6              mov esi,esi
0000000C  C9                leave
0000000D  C3                ret
```

7.1 Dissection of test.bin

As you can see values are being returned using the register `eax`. Because it is a register we do not need to explicitly fill the register with a return value, so we can also return *nothing* instead. There is an other advantage to it. Because the return code is stored in a register, we also do not need to explicitly read the return code. We use this all the time when we call the `ANSI C` function `printf` to print something on the screen. We always use:

```
printf (...);
```

While `printf` actually returns an `int` to the caller. Of course the compiler can't use this method if the type of the return parameter is bigger than 4 bytes. In the next paragraph we will demonstrate a situation inwhich this occures.

7.2 Returning data structures

Consider next program,

```
typedef struct {
  int a,b,c,d;
  int i [10];
} MyDef;

MyDef MyFunc (); /* function prototype */

int main () { /* entry point */
  MyDef d;
  d = MyFunc ();
}

MyDef MyFunc () { /* a local function */
  MyDef d;
  return d;
}
```

This program let us generate next binary code.

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC38            sub esp,byte +0x38
00000006  8D45C8            lea eax,[ebp-0x38]
00000009  50                push eax
```

```
0000000A  E805000000         call 0x14
0000000F  83C404             add esp,byte +0x4
00000012  C9                 leave
00000013  C3                 ret
00000014  55                 push ebp
00000015  89E5               mov ebp,esp
00000017  83EC38             sub esp,byte +0x38
0000001A  57                 push edi
0000001B  56                 push esi
0000001C  8B4508             mov eax,[ebp+0x8]
0000001F  89C7               mov edi,eax
00000021  8D75C8             lea esi,[ebp-0x38]
00000024  FC                 cld
00000025  B90E000000         mov ecx,0xe
0000002A  F3A5               rep movsd
0000002C  EB02               jmp short 0x30
0000002E  89F6               mov esi,esi
00000030  89C0               mov eax,eax
00000032  8D65C0             lea esp,[ebp-0x40]
00000035  5E                 pop esi
00000036  5F                 pop edi
00000037  C9                 leave
00000038  C3                 ret
```

Dissection of test.bin

At address `0x3` of the function `main` we see that the compiler reserves `0x38` bytes on the stack. This is the size of the structure `MyDef`. At address `0x6` to `0x9` we see the solution to “the problem”. Since `MyDef` is bigger than 4 bytes, the compiler passes a pointer to `d` to the function `MyFunc` at address `0x14`. This function can then use that pointer to fill up `d` with data. Please notice that a parameter is being passed to the function `MyFunc` while this function actual doesn’t have any parameters at all in its C function declaration. To fill the data structure, `MyFunc` uses a 32 bit data movement instruction:

```
0000002A  F3A5               rep movsd
```

7.3 Returning data structures II

Of course we can now ask ourselfs the question: Which pointer will be given to the function `MyFunc` if we don’t want to store the returned data structure? Consider therefore next program.

```
typedef struct {
  int a,b,c,d;
  int i [10];
} MyDef;

MyDef MyFunc (); /* function prototype */

int main () { /* entry point */
```

```
  MyFunc ();
}

MyDef MyFunc () { /* a local function */
  MyDef d;
  return d;
}
```

The produced binary code,

```
00000000  55                  push ebp
00000001  89E5                mov ebp,esp
00000003  83EC38              sub esp,byte +0x38
00000006  8D45C8              lea eax,[ebp-0x38]
00000009  50                  push eax
0000000A  E805000000          call 0x14
0000000F  83C404              add esp,byte +0x4
00000012  C9                  leave
00000013  C3                  ret
00000014  55                  push ebp
00000015  89E5                mov ebp,esp
00000017  83EC38              sub esp,byte +0x38
0000001A  57                  push edi
0000001B  56                  push esi
0000001C  8B4508              mov eax,[ebp+0x8]
0000001F  89C7                mov edi,eax
00000021  8D75C8              lea esi,[ebp-0x38]
00000024  FC                  cld
00000025  B90E000000          mov ecx,0xe
0000002A  F3A5                rep movsd
0000002C  EB02                jmp short 0x30
0000002E  89F6                mov esi,esi
00000030  89C0                mov eax,eax
00000032  8D65C0              lea esp,[ebp-0x40]
00000035  5E                  pop esi
00000036  5F                  pop edi
00000037  C9                  leave
00000038  C3                  ret
```

Dissection

This code shows us that — although there aren't any local variables in the entry function `main` at address `0x0` — the function reserves some place on the stack for a variable of exactly `0x38` bytes in size. Then a pointer to this data structure is being passed to the function `MyFunc` at address `0x14`, just as in the previous example. Also notice that the function `MyFunc` hasn't change internally.

8 Passing function parameters

In this section we will take a look on how function parameters are passed to functions. Let's take a look at the example:

```
char res; /* global variable */

char f (char a, char b); /* function prototype */

int main () { /* entry point */
  res = f (0x12, 0x23); /* function call */
}

char f (char a, char b) { /* function definition */
  return a + b; /* return code */
}
```

This will generate as binary code:

```
00000000  55                  push ebp
00000001  89E5                mov ebp,esp
00000003  6A23                push byte +0x23
00000005  6A12                push byte +0x12
00000007  E810000000          call 0x1c
0000000C  83C408              add esp,byte +0x8
0000000F  88C0                mov al,al
00000011  880534120000        mov [0x1234],al
00000017  C9                  leave
00000018  C3                  ret
00000019  8D7600              lea esi,[esi+0x0]
0000001C  55                  push ebp
0000001D  89E5                mov ebp,esp
0000001F  83EC04              sub esp,byte +0x4
00000022  53                  push ebx
00000023  8B5508              mov edx,[ebp+0x8]
00000026  8B4D0C              mov ecx,[ebp+0xc]
00000029  8855FF              mov [ebp-0x1],dl
0000002C  884DFE              mov [ebp-0x2],cl
0000002F  8A45FF              mov al,[ebp-0x1]
00000032  0245FE              add al,[ebp-0x2]
00000035  0FBED8              movsx ebx,al
00000038  89D8                mov eax,ebx
0000003A  EB00                jmp short 0x3c
0000003C  8B5DF8              mov ebx,[ebp-0x8]
0000003F  C9                  leave
00000040  C3                  ret
```

8.1 C calling convention

The first thing we notice is that the parameters are pushed onto the stack in reversed order. This is the C calling convention. The C calling convention in 32-bit programs is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The *caller* pushes the function's parameters on the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last).
- The *caller* then executes a near `CALL` instruction to pass control to the *callee*.
- The *callee* receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `ESP` in `EBP` so as to be able to use `EBP` as a base pointer to find its parameters on the stack. However, the *caller* was probably doing this too, so part of the calling convention states that `EBP` must be preserved by any C function. Hence the *callee*, if it is going to set up `EBP` as a frame pointer, must push the previous value first.
- The *callee* may then access its parameters relative to `EBP`. The doubleword at `[EBP]` holds the previous value of `EBP` as it was pushed; the next doubleword, at `[EBP+4]`, holds the return address, pushed implicitly by `CALL`. The parameters start after that, at `[EBP+8]`. The leftmost parameter of the function, since it was pushed last, is accessible at this offset from `EBP`; the others follow, at successively greater offsets. Thus, in a function such as `printf` which takes a variable number of parameters, the pushing of the parameters in reverse order means that the function knows where to find its first parameter, which tells it the number and type of the remaining ones.
- The *callee* may also wish to decrease `ESP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `EBP`.
- The *callee*, if it wishes to return a value to the *caller*, should leave the value in `AL`, `AX` or `EAX` depending on the size of the value. Floating-point results are typically returned in `ST0`.
- Once the *callee* has finished processing, it restores `ESP` from `EBP` if it had allocated local stack space, then pops the previous value of `EBP`, and returns via `RET` (equivalently, `RETN`).
- When the *caller* regains control from the *callee*, the function parameters are still on the stack, so it typically adds an immediate constant to `ESP` to remove them (instead of executing a number of slow `POP` instructions). Thus, if a function is accidentally called with the wrong number of parameters due to a prototype mismatch, the stack will still be returned to a sensible state since the *caller*, which *knows* how many parameters it pushed, does the removing.

8.2 Dissection

So after the two bytes are pushed onto the stack there is a `call` to the function `f` at address `0x1c`. This function first descreases `esp` with 4 bytes for local use. Next the function makes local copies of it's function parameters. After that `a + b` is being calculated and returned in register `eax`.

9 32-bit stack alignment

Please notice that — even when the two parameters were pushed onto the stack as bytes — the function reads then from the stack as if they were dwords! It seems as if the processor pushes bytes in 32-bit mode as dword. This is because the stack is aligned onto 32-bit[2]. This is very important to know when you have to write a 32-bit function in assembler following the C calling convention yourself.

[2]See also: Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 4.2.2. Stack Alignment

10 Other statements

Of course we also could look on how GCC handles `for` loops, `while` loops, `if-else` statements and `case` constructions, but this doesn't really matter when you want to write them yourself. And if you don't want to write them yourself it also doesn't matter since you don't have to bother about it.

11 Conversions between fundamental data types

In this part we will have a closer look at how the C compiler converts the fundamental data types. These data types are:

- signed char and unsigned char (1 byte)
- signed short and unsigned short (2 bytes)
- signed int and unsigned int (4 bytes)

First we will have a look on how the computer handles signed data types.

11.1 Two's complement

The two's complement representation of signed integers is used in the Intel architecture **IA-32**. The two's complement representation of a nonnegative integer n is the bit string obtained by writing n in base 2. If we take the bitwise complement of the bit string and add 1 to it, we obtain the two's complement representation of $-n$. A machine that uses the two's complement representation as its binary representation in memory for integral values is called a *two's complement machine*. Notice that in the two's complement representation 0 and -0 are being represented by the same binary string containing all zeros. Example:

$$\begin{aligned}(0)_{10} &= (00000000)_2 \\ (-0)_{10} &= \overline{(00000000)_2} + 1 \\ &= (11111111)_2 + 1 \\ &= (00000000)_2 \\ &= (0)_{10}\end{aligned}$$

Wherein $(\ldots)_x$ stands for a number represented in base x. Notice also that negative numbers are characterized by having the high bit on. Of course you don't have to do the conversion to a negative version of a certain number yourself. The `IA-32` architecture has a specific instruction for this, called `NEG`. Table 1 shows us the two's complement representation of a char. The advantage

	Range						
unsigned	128	...	255	0	1	...	127
signed	-128	...	-1	0	1	...	127

Table 1: The two's complement of a char

of the two's complement notation is that you can calculate with negative numbers the same way as with positive numbers.

11.2 Assignments

Here we will take a look at some C assignments and there result in assembly. The used C program is displayed below

```
main () {
  unsigned int i = 251;
}
```

When we compile this to a plain binary file we get

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC04            sub esp,byte +0x4
00000006  C745FCFB000000    mov dword [ebp-0x4],0xfb
0000000D  C9                leave
0000000E  C3                ret
```

When we replace the used assignment with

```
unsigned int i = -5;
```

we get next instruction at address `0x6`

```
00000006  C745FCFBFFFFFF    mov dword [ebp-0x4],0xfffffffb
```

Now lets take a look at the signed integers. The statement

```
int i = 251;
```

results in

```
00000006  C745FCFB000000    mov dword [ebp-0x4],0xfb
```

An the statements which uses a negative integer

```
int i = -5;
```

results in

```
00000006  C745FCFBFFFFFF    mov dword [ebp-0x4],0xfffffffb
```

Seems like signed and unsigned assignments are treated the same way.

11.3 Conversion of signed char to signed int

Here for we will study next little program:

```
main () {
  char c = -5;
  int i;
  i = c;
}
```

When we generate a binary file we get

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  C645FFFB          mov byte [ebp-0x1],0xfb
0000000A  0FBE45FF          movsx eax,byte [ebp-0x1]
0000000E  8945F8            mov [ebp-0x8],eax
00000011  C9                leave
00000012  C3                ret
```

Dissection

First we see at address `0x3` the reservation of 8 bytes onto the stack for the local variables `c` and `i`. The compiler takes 8 bytes to make it possible to align the integer `i`. Next we see that the char `c` at `[ebp-0x1]` is being filled with `0xfb`, which of course represents -5. (`0xfb` = 251, 251 - 256 = -5) Notice also that the compiler uses `[ebp-0x1]` instead of `[ebp-0x4]`. This because of the *little endian* representation. The next instruction `movsx` does the actual conversion from a signed char to a signed integer. `MOVSX` sign-extends its source (second) operand to the length of its destination (first) operand, and copies the result into the destination operand[3]. The last instruction (before leave) then writes the signed integer stored in `eax` to int `i`.

11.4 Conversion of signed int to signed char

Lets see at the opposite conversion.

```
main () {
  char c;
  int i = -5;
  c = i;
}
```

Notice that the statement `c = i` only make sense when the value in `i` is between -128 and 127. Because it has to be in the range of the signed char. Compilation results into next binary file

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  C745F8FBFFFFFF    mov dword [ebp-0x8],0xfffffffb
0000000D  8A45F8            mov al,[ebp-0x8]
00000010  8845FF            mov [ebp-0x1],al
00000013  C9                leave
00000014  C3                ret
```

Dissection

`0xfffffffb` is indeed -5. When we only look at the less significant byte `0xfb` and we move this to a signed char, we also get -5. So for the conversion from a signed int to a signed char we can use a simple `mov` instruction.

[3]See also: Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 6.3.2.1. Type Conversion Instructions

11.5 Conversion of unsigned char to unsigned int

Take a look at the C program

```
main () {
  unsigned char c = 5;
  unsigned int i;
  i = c;
}
```

This will generate the binary file

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  C645FF05          mov byte [ebp-0x1],0x5
0000000A  0FB645FF          movzx eax,byte [ebp-0x1]
0000000E  8945F8            mov [ebp-0x8],eax
00000011  C9                leave
00000012  C3                ret
```

Dissection

We get the same binary file as for the conversion from signed char to signed int except for the instruction at address `0xA`. Here we have the instruction `movzx`. `MOVZX` zero-extends its source (second) operand to the length of its destination (first) operand, and copies the result into the destination operand.

11.6 Conversion of unsigned int to unsigned char

Here fore we did use the file

```
main () {
  unsigned char c;
  unsigned int i = 251;
  c = i;
}
```

Please notice again that the integer value is restricted from 0 to 255. This because an unsigned char can't handle any bigger numbers. The accompanying binary file

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  C745F8FB000000    mov dword [ebp-0x8],0xfb
0000000D  8A45F8            mov al,[ebp-0x8]
00000010  8845FF            mov [ebp-0x1],al
00000013  C9                leave
00000014  C3                ret
```

Dissection

The actual conversion instruction, the `mov` instruction at address `0xD`, is the same as for the conversion from signed integers to signed chars.

11.7 Conversion of signed int to unsigned int

The file

```
main () {
  int i = -5;
  unsigned int u;
  u = i;
}
```

The binary

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  C745FCFBFFFFFF    mov dword [ebp-0x4],0xfffffffb
0000000D  8B45FC            mov eax,[ebp-0x4]
00000010  8945F8            mov [ebp-0x8],eax
00000013  C9                leave
00000014  C3                ret
```

Dissection

There is no specific conversion between signed and unsigned integers. The only difference is when you perform operations on the integers. Signed integers will have to use instructions like `idiv`, `imul` where unsigned integers will use the unsigned versions of there instructions being `div`, `mul`.

12 Basic environment for GCC compiled code

Because I can't find any official documentation on this subject I tried to figure it out for myself. Here's what I've got:

- 32-bit mode, so protected mode with enabled 32 bit code flag in GDT or LDT table.
- Segment registers CS, DS, ES, FS, GS and SS have to point to the same memory area. (aliases)
- Because un-initialised global variables are stored "right" after the code you have to keep a little area free. This area is called the BSS section. Notice that initialised global variables are stored in the DATA section in the binary file itself right after the code section. Variables declared with `const` are stored in the RODATA (read-only) section which is also part of the binary file itself.
- Make sure the stack can't overwrite the code and global variables.

In the Intel documentation[2] they refer to this as **Basic Flat Model**[4]. Don't misunderstand this. We don't have to use the **Basic Flat Model**. As long as the C compiled binary has his `CS`, `DS` and `SS` pointing to the same memory area (using aliases) everything will work. So we can use the full multisegment protected paging model as long as every C compiled binary has his *local* basic flat memory model[5].

13 Extern access to global variables

In this section we will take a look on how to access global C variables *not* from within the C program. This is usefull when you load the C program with another program (written in assembly) which has to initialize some global variables of the C program. Of course we could pass the variables using the C program's stack, but then these variables are always stored on the stack which was not the intention. We could also make a global variable table somewhere in the memory at a fixed point — so the C program has its address as a constant — but then we have to use stupid pointers to that table. So here is how we will do it. In the file `test.c` comes:

```
int myVar = 5;
int main () {
}
```

We compile this C program using:

```
gcc -c test.c
ld -Map memmap.txt -Ttext 0x0 -e main -oformat binary -N \
  -o test.bin test.o
ndisasm -b 32 test
```

This gives us,

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  C9                leave
00000004  C3                ret
00000005  0000              add [eax],al
00000007  00                db 0x00
00000008  05                db 0x05
00000009  0000              add [eax],al
0000000B  00                db 0x00
```

As you can see the variable `myVar` is stored at location `0x8`. Now we have to get that address from `ld` using its memory map file `memmap.txt` which we did create using the parameter `-Map`. Herefore we use the command:

```
cat memmap.txt | grep myVar | grep -v '\.o' | \
  sed 's/ *//' | cut -d' ' -f1
```

This gives us our address of the variable `myVar` in module `test.o`.

[4]See also: Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 3.3. Memory Organization

[5]See also: Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, Chapter 3: Protected-mode memory management

```
0x00000008
```

When we put this value in an environment variable (UNIX) `MYVAR`, we can use this to tell `nasm` where to look for the global C variable `myVar`. Example:

```
nasm -f bin -d MYVAR_ADDR=$MYVAR -o init.bin init.asm
```

In `init.asm` the code which uses this directive could look like:

```
...
mov     ax,CProgramSelector
mov     es,ax
mov     eax,[TheValueThatMyVarShouldContain]
mov     [es:MYVAR_ADDR],eax
...
```

13.1 The size of the BSS section

When the C program is a kernel it has to know how big its BSS section is for its memory management. This size can also be extracted from the file `memmap.txt`. Herefore we use:

```
cat memmap.txt | grep '\.bss ' | grep -v '\.o' | sed 's/.*0x/0x/'
```

For our example `test.c` this gives us:

```
0x0
```

We can pass this value like the way we did it for the global variables.

13.2 Global static variables

In C there is no way to access `static` variables directly. This is just because they are declared as being `static`. This rule also applies to the described external access method. When a global variable is declared as `static` there is no address of this variable in the memory map file generated by the linker `ld`. So we can't determine the address of this variable. The keyword `static` provides us with a great protection mechanism.

14 Implementation of ANSI C stdarg.h on IA-32

This header file provides the programmer with a portable means of writing functions such as `printf` that have a variable number of arguments. The header file contains one `typedef` and three macros[6]. How these are implemented is system-dependent, but on the IA-32 a possible implementation is:

```
#ifndef STDARG_H
#define STDARG_H

typedef char* va_list;

#define va_rounded_size(type) \
(((sizeof (type) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
```

[6]Source: A Book on C, fourth edition, A.10. Variable Arguments

```
#define va_start(ap, v) \
((void) (ap = (va_list) &v + va_rounded_size (v)))

#define va_arg(ap, type) \
(ap += va_rounded_size (type), *((type *)(ap - va_rounded_size (type))))

#define va_end(ap) ((void) (ap = 0))

#endif
```

In the macro `va_start`, the variable `v` is the last argument that is declared in the header to your variable argument function definition. This variable cannot be of storage class `register`, and it cannot be an array type or a type such as `char` that is widened by automatic conversions. The macro `va_start` initializes the argument pointer `ap`. The macro `va_arg` accesses the next argument in the list. The macro `va_end` performs any cleanup that may be required before function exit.
In the given implementation we're using a macro `va_rounded_size`. This macro is needed since the IA-32 aligns the stack — which is used to pass us the variables of a function — on 32-bit boundaries, indicated by the statement `sizeof (int)`. The macro `va_start` will let the argument

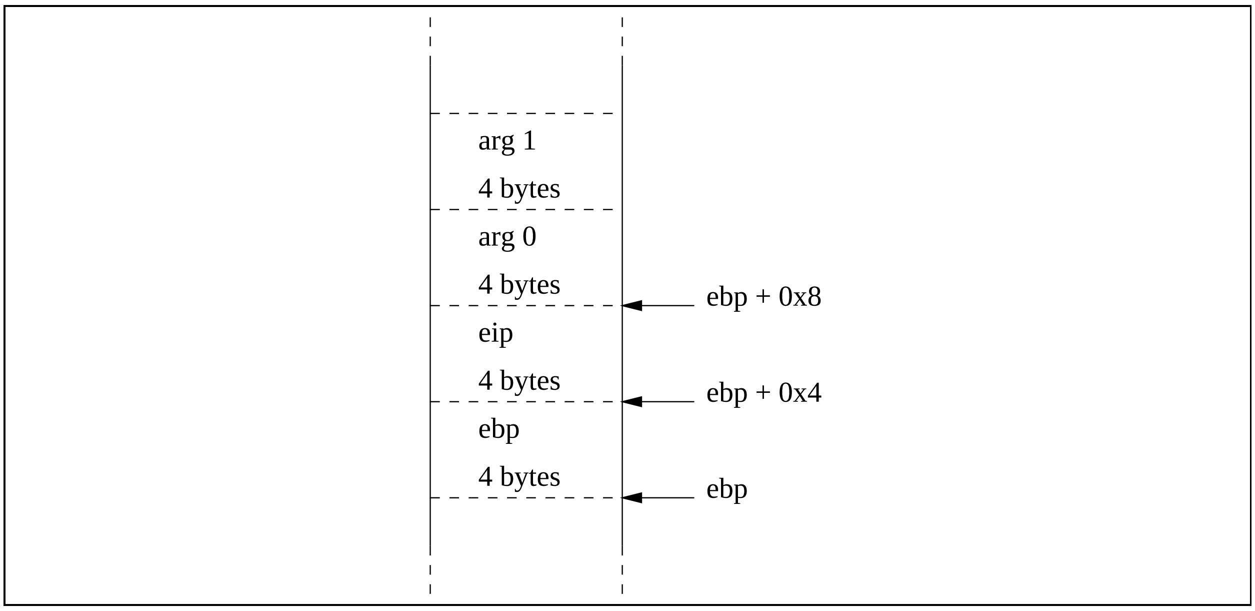

Figure 2: The arguments on the IA-32 stack

pointer `ap` point to the variable after the given (first) variable `v`. This macro doesn't return anything (indicated by the leading `(void)`).
The macro `va_arg` first increases the argument pointer `ap` by the size of the given type `type`. After that it returns the next (actually the previous argument since the argument pointer `ap` first did increase) argument on the stack of type `type`. At first sight this way of handling seems very weird but its the only way since we have to put the variable we want to return at the end of a macro definition, after the last comma.
Finally macro `va_end` will reset the argument pointer `ap` without returning anything.

References

[1] A Book on C
Programming in C, fourth edition
Addison-Wesley — ISBN 0-201-18399-4

[2] Intel Architecture Software Developer's Manual
Volume 1: Basic Architecture
Order Number: 243190
Volume 2: Instruction Set Reference Manual
Order Number: 243191
Volume 3: System Programming Guide
Order Number: 243192

[3] NASM documentation
http://www.cryogen.com/Nasm

[4] Manual Pages
gcc, ld, objcopy, objdump