

Embedding Assembly Code in C Programs

תרגול 7
שילוב קוד אסמבלי בקוד C.

History

- In the early days of computing, most programs were written in assembly code.
 - This becomes unmanageable for programs of significant complexity.
 - Since assembly code does not provide any form of type checking, it is very easy to make basic mistakes, such as using a pointer as an integer.
 - Even worse, writing in assembly code locks the entire program into a particular class of machine.
- Early compilers for higher-level programming languages did not generate very efficient code and did not provide access to the low-level object representations.
- Programs requiring maximum performance or requiring access to object representations were still often written in assembly code.

Nowadays

- Compilers have largely removed performance optimization as a reason for writing in assembly code.
- Code generated by a high-quality compiler is generally as good or even better than what can be achieved manually.
- The C language has largely eliminated machine access as a reason for writing in assembly code.
 - Unions.
 - Pointer arithmetic.
 - Ability to operate on bit-level data representations.
- This provides a sufficient access to the machine for most programmers.
 - E.g., almost every part of a modern operating system such as Linux is written in C.

Why embedding?

- There are times when writing in assembly code is the only option:
 - (Especially) When implementing an operating system.
- Examples:
 - There are a number of special registers storing process state information that the operating system must access.
 - There are either special instructions or special memory locations for performing input and output operations.
 - Even for application programmers, there are some machine features, such as the values of the condition codes, that cannot be accessed directly in C.

The challenge:

- To integrate code consisting mainly of C with a small amount written in assembly language.
- Methods:
 - Writing the assembly function in a different file using the same conventions for argument passing and register usage as are followed by the C compiler, and combine all the code using the Linker.

E.g., file *p1.c* contains *C code* and file *p2.s* contains *assembly code*. The command:

```
unix> gcc -o p p1.c p2.s
```

will cause file *p1.c* to be compiled, file *p2.s* to be assembled, and the resulting object code to be linked to form an executable program *p*.
 - Using Inline Assembly.

Inline Assembly (`asm` directive)

- Inline assembly allows the user to insert assembly code directly into the code sequence generated by the compiler.
- Features are provided to specify instruction operands and to indicate to the compiler which registers are being overwritten by the assembly instructions.
- The resulting code is, of course, highly machine-dependent, since different types of machines do not have compatible machine instructions.
- The `asm` directive is also specific to GCC. Therefore, incompatibility with many other compilers.

Basic form of inline assembly

- The basic form of inline assembly is to write code that looks like a procedure call:

```
asm( code-string );
```

- *code-string* is an assembly code sequence given as a quoted string.
- The compiler will insert this string verbatim into the assembly code being generated, and hence the compiler-supplied and the user supplied assembly will be combined.
- Important: The compiler **doesn't** check the string for errors, and so the first indication of a problem might be an error report from the assembler.

An example:

- Consider functions with the following prototypes:

```
int ok_smul(long x, long y, long *dest);
```

```
int ok_umul(unsigned long x, unsigned long y,  
unsigned long *dest);
```

- Each is supposed to compute the product of arguments x and y and store the result in the memory location specified by argument dest. As return values, they should return 0 when the multiplication overflows and 1 when it does not.
- We have separate functions for signed and unsigned multiplication, since they overflow under different circumstances.

An example: - first try

- The strategy here is to exploit the fact that register %eax is used to store the return value.
- **Assuming** the compiler uses this register for variable result, the first line will set the register to 0. The inline assembly will insert code that sets the low-order byte of this register appropriately, and the register will be used as the return value.


```
1  /* First attempt. Does not work */
2  int ok_smul_1(long x, long y, long* dest)
3  {
4      int result = 0;
5      *dest = x*y;
6      asm("setae %al");
7      return result;
8  }
```

An example: - first try (2)

■ The generated assembly code:

```
1 ok_smul_1:
2   imulq    %rsi,%rdi    # compute x*y
3   movq     %rdi, (%rdx) # store at dest
4   setae    %al          # set low-order byte of %rax
5   movl     $0,%eax
6   ret
```

generated by asm

- 
- Unfortunately, GCC has its own ideas of code generation.
 - Instead of setting register `%eax` to 0 at the beginning of the function, the generated code does so at the very end, and so the function always returns 0.
 - The fundamental problem is that the compiler has no way to know what the programmer's intentions are, and how the assembly statement should interact with the rest of the generated code.

An example: - second try

- A less than ideal code, but working code:

```
1  /* Second attempt. Works in limited contexts */
2  int dummy = 0;
3
4  int ok_smul_2(long x, long y, long* dest)
5  {
6      int result;
7
8      *dest = x*y;
9      result = dummy;
10     asm("setae %al");
11     return result;
12 }
```

- Same strategy as before, but this time the code reads a global variable `dummy` to initialize `result` to 0.
- Compilers are typically more conservative about generating code involving global variables, and therefore less likely to rearrange the ordering of the computations.
- Even the above code depends on quirks of the compiler to get proper behavior. In fact, it only works when compiled with optimization enabled. Otherwise it stores `result` on the stack and retrieves its value just before returning.

Extended form of `asm`

- The extended version of the `asm` allows the programmer to specify which program values are to be used as operands to an assembly code sequence and which registers are overwritten by the assembly code.
- With this information the compiler can generate code that will correctly set up the required source values, execute the assembly instructions, and make use of the computed results.
- It will also have information it requires about register usage so that important program values are not overwritten by the assembly code instructions.

Extended form syntax

- The general syntax of an extended assembly sequence is as follows:

```
asm( code-string [ : output-list [ : input-list  
    [ : overwrite-list ] ] ] );
```

- The square brackets denote optional arguments.
 - *code-string* – the assembly code sequence.
 - *output-list* – results generated by the assembly code.
 - *input-list* – source values for the assembly code.
 - *overwrite-list* – registers that are overwritten by the assembly code.
- These lists are separated by the colon (':') character. As the square brackets show, we only include lists up to the last nonempty list.

Extended form syntax (2)

- The syntax for the code string is reminiscent of that for the format string in a printf statement.
- Input and output operands are denoted by references to output and input lists.
- Within the assembly code, operands are given names, written as `%[name]` (since gcc 3.1).
- Register names such as “%eax” must be written with an extra ‘%’ symbol, e.g., “%%eax.”
- Insert return characters between multiple instructions: `\n\t`

The Output List

- A comma-separated list
- Each elements contains three components:

[name] tag (expr)

- name – name of the operand
- tag – the output constraint:

Constraint	Meaning
"=r"	Update value stored in a register
"+r"	Read and update value stored in a register
"=m"	Update value stored in memory
"+m"	Read and update value stored in memory
"=rm"	Update value stored in a register or in memory
"+rm"	Read and update value stored in a register or in memory

- expr – the C expression indicating the destination for the instruction's result (any assignable value)

The Input List

- Has the same format as the output list
- The input constraints are of the form:
 - r – operand is read from register
 - m – operand is read from memory
 - rm – operand is read from either register or memory

An example: - third try

```
1  /* Uses the extended assembly statement to get reliable code */
2  int ok_smul_3(long x, long y, long* dest)
3  {
4      int result;
5
6      *dest = x*y;
7      asm("setae %%bl          # set low order byte\n\t"
8          "movzbl %%bl,%[val] # zero extend to be result\n\t"
9          : [val] "=r" (result) /* output */
10         :          /* no inputs */
11         : "%bl"      /* overwrites */
12         );
13     return result;
14 }
```

An example: - third try (2)

```
1 ok_smul_3:
2     pushq    %rbx           # save %rbx
3     imulq    %rsi,%rdi      # compute x*y
4     movq     %rdi, (%rdx)   # store at dest
5     setae    %bl            # set low-order byte
6     movzbl   %bl,%eax       # zero extend %eax
7     popq     %rbx           # restore %rbx
8     ret
```

generated by asm

An example: - third try (3)

- We have two code strings:
 - One for the `setae` instruction
 - One to zero-extend the low-order byte to form the result
- `%bl` is the destination of `setae` and the source of `movzbl`
- We have comments, lines end `\n\t`
- The name `val` indicates the final value generated by the code
 - The program variable `result`

- The code shown above works regardless of the compilation flags.
- As this example illustrates, it may take a little creative thinking to write assembly code that will allow the operands to be described in the required form.
 - E.g., there are no direct ways to specify a program value to use as the destination operand for the `setae` instruction, since the operand must be a single byte.
 - Instead, we write a code sequence based on a specific register and then use an extra data movement instruction to copy the resulting value to some part of the program state.
- Gcc generates code to save and restore `%rbx`
- However, no need to specify a specific register as destination for “`setae`”

An example - `umul`

- We can't use the same code since GCC use `imull` (signed multiply) instruction for both signed and unsigned multiplication.
- This generates the correct value for either product, but it sets the carry flag according to the rules for signed multiplication.
- Therefore, we must include an assembly code that performs unsigned multiplication using the `mull` instruction.

An example - umul (2)

```
1 int ok_umul(unsigned long x, unsigned long y, unsigned long* dest)
2 {
3     unsigned char bresult;
4
5     asm("movq %[x],%%rax    # get x\n\t"
6         "mulq %[y]          # unsigned long multiply by y\n\t"
7         "movq %%rax,%[p]    # store low-order 8 bytes at dest\n\t"
8         "setae %[b]         # set result\n\t"
9         : [p] "=m" (*dest), [b] "=r" (bresult) /* outputs */
10        : [x] "r" (x), [y] "r" (y)             /* inputs */
11        : "%rax", "%rdx"                       /* overwrites */
12        );
13
14     return (int) bresult;
15 }
```


An example - `umul` (3)

- The outputs:

- ☐ `p` – the product, stored in memory
- ☐ `b` – the status byte, held in a register

- The inputs:

- ☐ `x`, `y` – our local variables, stored in registers

- The overwrites:

- ☐ We overwrite the registers `%rax,%rdx`