GCC and Assembly language

- during the construction of an operating system kernel, microkernel, or embedded system it is vital to be able to access some of the microprocessor attribute unavailable in a high level language

- for example the operating system might need to:
  - modify a processes, stack pointer (`%rsp`)
  - turn interrupts on and off
  - manipulate the virtual memory directory processor register

GCC and Assembly language

- one could use an assembly language source file
  - define many functions which: `get`, `set` registers

- this is inefficient, as it requires a `call`, `ret` to set a register
  - cause cache misses and introduce a 3 instruction overhead

Consider an example (dangeous)

- suppose we wanted to get and set the value of the stack pointer: `rsp`
- we might initially start to write an assembly file: `foo.S` which sets and gets the stack pointer

foo.S

```
.globl foo_setsp
    # void setsp (void *p)
    #
    # move the parameter, p, into $rsp
    #
foo_setsp:
    pushq %rbp
    movq  %rsp, %rbp

    movq %rdi, %rsp

    leave
    ret
    #
    # void *getsp (void)
    #
foo_getsp:
    movq %rsp, %rax
    ret
```

Now write some C code: bar.c

```
extern void foo_setsp (void *p);

void someFunc (void)
{
  void *old = foo_getsp();
  foo_setsp((void *)0x1234);
}
```

Compile and link the code

```
$ as -o foo.o foo.S
$ gcc -c bar.c
$ gcc foo.o bar.o
```

- what are the problems with this code?
  - hint examine what happens to the stack pointer

Problems

- the stack pointer is modified during the `call` and `ret` instructions
  - this might possibly be ok, if we were to use `setsp` as part of a context switch
  - but it is certainly dangeous and inefficient

- we also have to know how `gcc` will pass the parameter, `p`

- `gcc` *might* alter its parameter passing mechanism if we choose a different optimization compiler setting
  - certainly `gcc` cannot inline our assembly code if we use `.c` and `.S` files

Writing the example the correct way

- a better technique is to use the inline assembler code feature of `gcc`
- this allows `gcc` to understand enough of what operands we are passing in to the assembler sequence
  - it also allows `gcc` to optimize its generated code around our hand crafted assembler code
- our code is also inlined, so we will not experience the side effects of the stack pointer being adjusted

GCC Assembler interface

- you can specify which operands are inputs to an instruction
  - which operands are outputs
  - which registers are trashed

- abridged syntax (see ⟨`http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Extended-Asm.html#Extended-Asm`⟩ for full details)

Abridged ebnf for modern GNU C Asm

```
asmStatement := "asm" [ "volatile" ] "(" instructionstring
                [ ":" outputs [ ":" inputs [ ":" trashed ]]]) =:

outputs := "[" symbolicname "]" operandconstraint "(" expression ")" =:
inputs := "[" symbolicname "]" operandconstraint "(" expression ")" =:
trashed := { registername } =:

operandconstraint := { "=" | "r" | "g" | "m" | "f" } =:

symbolicname := string =:
instructionstring := string =:
registername := string =:
```

- ebnf
  - `a := b =:` means production `a` is defined by rule `b`
  - `[ a ]` means `a` is optional
  - `a | b` means either `a` or `b` may be present
  - `"a"` means literal character `a`

Correct implementation of someFunc

```
void someFunc (void)
{
  void *old;

  asm volatile ("movq %%rsp, %[dest]"
                : [dest] "=rm" (old));
  asm volatile ("movq %[src], %%rsp"
                :: [src] "rm" (0x1234));
}
```

Output of gcc -O0 -S

```
someFunc:
  pushq  %rbp
  movq  %rsp, %rbp
#APP
  movq %rsp, -8(%rbp)
#NO_APP
  movl $4660, %eax
#APP
  movq %eax, %rsp
#NO_APP
  leave
  ret
```

Output of gcc -O3 -S

```
someFunc:
#APP
  movq %rsp, %rax
#NO_APP
  movl $4660, %eax
#APP
  movq %eax, %rsp
#NO_APP
  ret
```

- now lets change the C code to use the "g" specifier

Further improvement

```
void someFunc (void)
{
  void *old;

  asm volatile ("movq %%rsp, %[dest]"
                : [dest] "=rm" (old));
  asm volatile ("movq %[src], %%rsp"
                :: [src] "rmg" (0x1234));
}
```

Output of gcc -O3 -S

```
someFunc:
#APP
  movq %rsp, %rax
  movq $4660, %rsp
#NO_APP
  ret
```

- choose specifiers which legally map onto legitimate assembly instructions
  - `gcc` does **not** check these, however the assembler will check them

Simple example

```
#include <stdio.h>

main (void)
{
   int result = 1;
   int input  = 2;

   /* result += input */
   asm volatile ("addl %[src],%[dest]"
        : [dest] "=rm" (result)
        : [src] "r" (input), "rm" (result));
   printf("value of dest = %d\n", result);
}
```

```
$ gcc -O0 -S testasm.c
```

Assembler output from GCC -O0

```
main:
  pushq  %rbp
  movq   %rsp, %rbp
  subq   $16, %rsp
  movl   $1, -8(%rbp)
  movl   $2, -4(%rbp)
  movl   -4(%rbp), %eax
#APP
# 9 "testasm.c" 1
  addl %eax,-8(%rbp)
# 0 "" 2
#NO_APP
  movl   $.LC0, %eax
  movl   -8(%rbp), %edx
  movl   %edx, %esi
  movq   %rax, %rdi
  movl   $0, %eax
  call   printf
```

C code explained

```
asm volatile ("addl %[src],%[dest]"
     : [dest] "=rm" (result)
     : [src] "r" (input), "rm" (result));
```

- addl performs `dest += src`
- `dest` can either be a register or memory operand and is both an output and input
  - `=` means output
  - `r` means register
  - `m` means memory
- `src` must be in a register, and is treated as an input

Assembler output from GCC -O3

- since we have told `gcc` the specifications of our assembly instruction we can ask `gcc` to perform aggressive optimizations!
  - and still preserve meaning

-
```
$ gcc -O3 -S testasm.c
```

Assembler output from GCC -O3

```
main:
  movl  $1, %esi
  movl  $2, %eax
#APP
# 9 "testasm.c" 1
  addl %eax,%esi
# 0 "" 2
#NO_APP
  movl  $.LC0, %edi
  xorl  %eax, %eax
  jmp   printf
```

- notice how `gcc` has chosen to use register operands for the `addl` instruction

Use of GCC Assembler syntax in LuK

- used heavily in `mod/PortIO.mod` and `mod/SYSTEM.mod`

- both modules written in Modula-2, not C
  - however GNU Modula-2 uses exactly the same assembly language interface as C
  - only that it uses uppercase keywords: `ASM`, `VOLATILE`

PortIO.mod

- consider procedure function `Out8`

```
PROCEDURE Out8 (Port: CARDINAL; Value: BYTE) ;
BEGIN
   ASM VOLATILE('outb %al,$0x80') ;  (* linux idea for slowing fast machines down *)
   ASM VOLATILE('movl %[port], %%edx ; movb %[Value], %%al ; outb %%al,%%dx'
                :: [port] "rm" (Port), [Value] "rm" (Value) : "eax", "edx") ;
   ASM VOLATILE('outb %al,$0x80')    (* linux idea for slowing fast machines down *)
END Out8 ;
```

C equivalent

```
void PortIO_Out8 (unsigned int Port, unsigned char Value)
{
  asm volatile("outb %al,$0x80");   /* linux idea for slowing fast machines down */
  asm volatile("movl %[port], %%edx ; movb %[Value], %%al ; outb %%al,%%dx"
                :: [port] "rm" (Port), [Value] "rm" (Value) : "eax", "edx") ;
  asm volatile("outb %al,$0x80");    /* linux idea for slowing fast machines down */
}
```

Result of gcc -m32 -O0

```
PortIO_Out8:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $4, %esp
  movl  12(%ebp), %eax
  movb  %al, -4(%ebp)
#APP
# 4 "cportio.c" 1
  outb  %al,$0x80
# 0 "" 2
# 5 "cportio.c" 1
  movl  8(%ebp), %edx ; movb -4(%ebp), %al ; outb %al,%dx
# 0 "" 2
# 7 "cportio.c" 1
  outb  %al,$0x80
# 0 "" 2
#NO_APP
  leave
  ret
```

Result of gcc -O3

```
PortIO_Out8:
  pushl  %ebp
  movl   %esp, %ebp
#APP
# 4 "cportio.c" 1
  outb  %al,$0x80
# 0 "" 2
# 5 "cportio.c" 1
  movl  8(%ebp), %edx ; movb 12(%ebp), %al ; outb %al,%dx
# 0 "" 2
# 7 "cportio.c" 1
  outb  %al,$0x80
# 0 "" 2
#NO_APP
  popl  %ebp
  ret
```

Conclusion

- when writing operating systems and microkernels you need to use assembly language occasionally
- it is likely that speed matters at the position where assembly language is deployed
- some operations must be inlined (stack related operations)
  - thus use gcc inline assembly mechanism
- very powerful and allows different optimization settings to co-exist with your code
  - it is widely used in the Linux kernel

Further reading

- examine LuK source code module for `SYSTEM.mod` and in particular the procedure function `TRANSFER`