Computer Architecture

Building programs with Assembly and C functions

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2023/2024

Building programs with Assembly and C functions

- Often time-critical routines are written in Assembly, and the rest of the software is written in C, thus a "mixed Assembly and C" project must be created
- When you intend to mix Assembly source files and ANSI-C source files in a single application, the following issues are important:
    - Accessing Assembly variables in an ANSI-C source file
    - Accessing ANSI-C variables in an Assembly source file
    - Invoking an Assembly function in an ANSI-C source file
    - Parameter passing scheme
    - Return value

Building programs with Assembly and C functions

- We will (for now) write functions in Assembly that receive no parameters and access global variables (either declared in C or Assembly)
- Our C programs will call our Assembly functions as if they were native C functions
- To make our Assembly functions return:
  - up to a 64-bit value, leave that return value in the `%rax` register (or parts of it)
  - a 128-bit value, leave the return value in the `%rdx:%rax` registers

Important note

GCC running on x86-64 supports 128-bit signed and unsigned integer values via data types `__int128_t` and `__uint128_t`, respectively

Sharing variables between Assembly and C (1/5)

- To share global variables that are declared in Assembly, between C and Assembly, use the `extern` C keyword
- It declares to the compiler that a variable is defined (the memory is reserved) in another source file (in our case, in the Assembly source file(s))

Important note

The `extern` C Keyword can be used in different ways to share variables between C and Assembly. The following is a recommended practice, that avoids common problems

- On the C source:
  1. Declare the *functions and variables* implemented in Assembly and used in C in a separate `.h` file (often called `asm.h`). Declare those Assembly variables using the `extern` C keyword (functions are `extern` by default):

Listing 1: asm.h

```
int asm_function();
extern int asm_integer;
```

  2. Use the keyword `#include` to include the previous `.h` file in the C source files (`.c` files) that use the Assembly functions or variables, and use the Assembly functions/variables like native C functions/variables:

Listing 2: main.c

```
#include "asm.h"
...
int main() {
  ...
  asm_integer=10;
  asm_function();
  ...
}
```

Sharing variables between Assembly and C (3/5)

- On the Assembly source:
  3. Declare the variables and functions used by the C sources and define them as visible using the `.global` directive

Listing 3: asm.s

```
.section .data
asm_integer:            # variable declaration
    .int 5
.global asm_integer     # define variable as global

.section .text
.global asm_function    # define function as global
asm_function:           # start of the function
    ...
    movl $0, %eax       # reaching here, will return 0
                        # (rax will not be changed until ret)
    ret
```

Sharing variables between Assembly and C (4/5)

1. To share global variables that are declared in C, between C and Assembly, simply declare them as global variables in the C source file

Listing 4: main.c

```c
#include <stdio.h>
#include "asm.h"

int op1=0, op2=0;

int main(void) {
  int res;
  ...
  res = sum();
  ...
  return 0;
}
```

2. On the Assembly source, define them as visible using the `.global` directive
3. Acess them with %rip-relative addressing

Listing 5: asm.s

```
.section .data
   .global op1
   .global op2

.section .text
   .global sum
   sum:
      ...
      movl op1(%rip), %ecx    # copy the value of op1 to ecx
      movl op2(%rip), %eax    # copy the value of op2 to eax
      ...
      ret
```

The x86-64 application binary interface

- The x86-64 application binary interface (ABI) describes the conventions for x86-64 code running on Linux systems
- This includes rules about how function arguments are placed, where return values go, what registers functions may use, how they may allocate local variables, and so forth
- Calling conventions constrain both *callers* and *callees*. A caller is a function that calls another function; a callee is a function that was called (the currently-executing function is a callee)
- We will discuss several details during the semester, but for now you only have to consider the return value and which registers can be freely used by a function

Important note on writing Assembly functions

- Until we detail the use of the stack *DO NOT* use any of these *callee saved* registers in your functions: %rbx, %rbp, %r12, %r13, %r14, %r15
- This is particularly important if you call your functions from other programmer's Assembly or C code (e.g., unit tests)

Example: Sum two variables - C source

Listing 6: main.c

```c
#include <stdio.h>
#include "asm.h" // defines op1, op2 and sum_op1_op2(void)

int main(void) {
    long res=0;
    printf("Value of op1?:");
    scanf("%d",&op1);
    printf("Value of op1?:");
    scanf("%hd",&op2);

    /* res = op1 + op2; */
    res = sum_op1_op2();

    printf("%ld = %d + %d\n", res, op1, op2);
    return 0;
}
```

Listing 7: asm.h

```c
long sum_op1_op2(void);
extern int op1;
extern short op2;
```

Example: Sum two variables - Assembly source

Listing 8: asm.s

```
.section .data
  op1:                # declare op1, op2
        .int 0
  op2:
        .short 0
  .global op1, op2    # define op1, op2 as globals

.section .text

  .global sum_op1_op2 # define global function long sum_op1_op2(void)

sum_op1_op2:
    movl   op1(%rip), %ecx  # place op1 in ecx
    movslq %ecx, %rcx       # sign extend to quad word
    movw   op2(%rip), %ax   # place op2 in rax
    movswq %ax, %rax        # sign extend to quad word
    addq   %rcx, %rax       # add rcx to rax, result is in rax
                            # and will be our return value
    ret                     # return to the caller function
```

Example: Sum two variables - Makefile

Listing 9: Makefile

```
main: main.o asm.o
	gcc main.o asm.o -z noexecstack -o main

main.o: main.c asm.h
	gcc -g -Wall -Wextra -fanalyzer -c main.c -o main.o

asm.o: asm.s
	gcc -g -Wall -Wextra -fanalyzer -c asm.s -o asm.o

run: main
	./main

clean:
	rm *.o main
```

Important note

The linker flag `-z noexecstack` is needed in recent versions of gcc to silence an executable stack warning

Practice

- Write a C program that calls `increment()`, a function implemented in Assembly
- Function `increment()` increments the value of the global integer variable `g_number` and returns the this value (after the increment)
- The C program should assign a test value to `g_number`, call `increment()` and then print both `g_number` and the value returned by the function
- Write a `Makefile` to compile your program