C + Assembly

Systems Programming

JOHANNES KEPLER
UNIVERSITÄT LINZ

Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

Why?

- Why should we do this?
  - Efficiency: We can do better than the C compiler!
    - Make sure you really do… compiler optimization is GOOD
  - Necessity: Use existing code/libraries
    - “Extended” linking: When linking alone is insufficient and we have to create small wrappers. Extremely rare!
    - Normally everything follows the C calling convention and all we have to do is write an appropriate header file (if it does not yet exist) and link the library!
    - Important if the library does not follow the C calling convention
  - Necessity: We need to specify CPU/register/… properties we cannot specify/implement in C
    - If you are developing an operating system → go ahead!
    - Otherwise: Think again – why are we developing in C?
      - And why are we not writing a whole library in assembler and link it?
    - Example: Stop all maskable interrupts – CLI instruction (“CLear Interrupts”)
      - Note: Can only be executed by OS kernel (= in ring 0)!

Why?

- Other use cases:
  - Atomic instructions: Use of a prefix to ensure atomic operation
    - Ensure correctness on multi-core CPUs for multi-threading programs
    - In C11 some C functions were introduced for this!
  - Prevent compiler reordering
    - Ensure correctness on multi-core CPUs for multi-threading programs
  - Memory barrier: All writes before, all reads after
    - Ensure correctness on multi-core CPUs for multi-threading programs
  - Arithmetic instructions: E.g. using SIMD instructions
  - Hash/Cryptography: Employing special hardware-implemented crypto-support instructions
    - An example of efficiency → Typically several implementations, depending on the processor capabilities

Integrating assembly code in C

- Requires a special "function": `_ _asm_ _ ();`
  - Background: GCC does not "compile" anything. It "just" produces assembler code, which is then compiled and linked
  - So we skip the "outer" step and produce output directly!
- Difficulty: How to pass data from one to the other?
  Three elements exist:
  - Output: Something the assembler produces & is later needed in C
  - Input: Something created in C & required in assembler code
  - Clobbered: Registers/Flags/Memory "destroyed" (= changed) during executing the assembler code
    - The C compiler can use this register before, but may not expect it to contain the same value afterwards
    - They must be "saved" before
      - Value stored somewhere, e.g. on stack, or swapped around so that this register is not needed anymore

JKU | INSTITUTE OF NETWORKS AND SECURITY

Example

```
static void cpuid(int code, uint32_t *a, uint32_t *b, uint32_t *d) {
__asm__ volatile (
        "cpuid\n" Code
        : "=a"(*a), "=b"(*b), "=d"(*d) Output
        : "0"(code) Input
        : "ecx" ); Clobbered
}
```

- We define a function with 4 parameters: 1 input, 3 output
- This function consists of assembler code: here single instruction only
  - Could contain normal C code before or after assembler too
- The code is the string
  - Attention: Will be copied into the output "as-is". Separate lines must therefore be delimited by "\n"!
- Clobbered (last line): The register ECX (actually some output of the CPUID instruction too!) will be changed
  - Common: "cc" → Flags will be changed, "memory" → Memory
- Attention: CPUID uses 32 bit registers even in 64 Bit code/mode!

Input and Output

- Input + Output are limited to 30 parameters
- Any valid C variable in scope is acceptable
  - Output: non-constant, l-value!
- Clobbered registers will never be used for input or output
- Data types are not checked. The compiler merely selects a register/memory of suitable size.
  - Whether the register is valid for the instructions executed or not, cannot be checked. You might have to explicitly specify it then!
- Input values may never be modified, unless they are simultaneously an output too! The compiler assumes unchanged value (and you cannot "clobber" them)!
- If the code after the __asm__ does not use the output variables (e.g. only used because inputs are modified), you must use "volatile" or it might be optimized away!

Output

- Output: The first section of “:”
  - Must always be present, but may be empty
  - Start with “=“ to signal this is an “assignment”. This is a constraint.
  - “a”: Will be stored in the “A” register. Depending on the size of the value this will be AL, AX, EAX, or RAX
  - “(*a)”: The name of the C parameter to assign to it
    - Here is additionally specified that we want the value (“*a”), not the address (“a”)
  - Modifiers can be used to further specify registers/properties
    - Attention: These are NOT standardized and depend on actual CPU/assembler used!
  - They can be referenced in the assembler code by their index…
    - First output parameter is “%0”, second “%1”…
  - …or by their name…
    - Here: “%%eax”. “%%” because this is a special character in C. Printed in the assembler code is “%” alone!
  - … or by a symbolic name (not shown here)

Input

- Input: The second section of “:”
  - `: "0" (code)`
  - The variable “code” (somewhere in the current scope, here a function parameter) will be assigned the same register as the first (=0) input operand
    - Here this will be EAX!
  - This will be operand number 3 (0=EAX, 1=EBX, 2=EDX; outputs)
    - Note: We explicitly specify b and d to be EBX and EDX, as this is where the CPUID instruction returns the data we want
  - Constraints can again be added as with outputs

I386 constraints

- Constraint modifiers (selection):
  - “=“: Overwrite an existing value
    - Can be any value before, except when tied to an input operand
  - “+”: Value will be read and written
  - “&”: Will be modified before input is read → NOT used for input register
- Constraints (selection):
  - “r”: Register
  - “m”: memory, “o”: Memory with an offsettable address
  - “i”: Immediate (=constant)
  - “<“, “>”, Memory operand with auto de-/increment addressing
    - Such registers do not exist on i386 → not possible!
  - “X”: Anything
  - “0”...”9”: The same as operand 0 – 9
  - b: BL, BX, EBX, RBX, c: CL-RCX, d: D??, S: ESI, D: EDI…
  - More than one constraint: The compiler may select

Pointer example

```
void clearArray(int32_t *ptr, uint64_t length) {
      __asm__ (
              "cmpq $0,%2\n"    /* Compare argument 2 (=length) with 0 */
              "jle end\n"       /* If the len is <=0 we do nothing */
              "start:\n"        /* Label definition (jump target) */
              "decq %2\n"       /* First element is at index length-1 */
              "movl $0,(%1,%2,4)\n"     /* Clear element. */
/* Take care: %1 is an address → use appropriate addressing mode! */
              "cmpq $0,%2\n"    /* End reached? */
              "jg start\n"
              "end:\n"
      : "=r" (length)
/* Define length as output (inputs may never be changed - unless output
too. Put in any register (=argument 0; Note: address!) */
      :"r" (ptr), "0" (length)
/* ptr -> any register (=argument 1); length is the same as the output
("0" = first argument; this is argument 2) */
      : "cc"                    /* Flags register will be clobbered */
      );
}
```

Pointer example

- What does this function do?
  - Reset a complete array to “0”!
- Attention:
  - Array consists of `length` elements, each of which is 4 bytes long
  - Length of the array (count of elements, not bytes!) is 8 bytes long
- Usefulness of this function:
  - Only as an example!
  - Its is both very inefficient to implement it as assembler code (use “memset”), and regarding assembler instructions there exist simpler and faster methods to do this (e.g. “rep stosd”)

Example – Generated assembler code

- Generated code (use gcc arguments “-S -fverbose-asm”)
  - Debug information (e.g. line numbers) has been removed here!

```
.type       cpuid, @function
cpuid:
            pushq       %rbp                 Normal function header
            movq        %rsp, %rbp
            pushq       %rbx                 Save EBX (overwritten because used as output!)
            movl        %edi, -12(%rbp)      Store parameter code in red zone
            movq        %rsi, -24(%rbp)      Store parameter a in red zone
            movq        %rdx, -32(%rbp)      Store parameter b in red zone
            movq        %rcx, -40(%rbp)      Store parameter c in red zone
            movl        -12(%rbp), %eax      Put parameter code in correct register for CPUID
            cpuid                            Our actual inline assembler code!
            movl        %ebx, %esi           Move "b" temporarily into ESI
            movq        -24(%rbp), %rcx      Load address (pointer!) of "a" from stack
            movl        %eax, (%rcx)         Store value at memory location ("a")
            movq        -32(%rbp), %rax
            movl        %esi, (%rax)         Store value at memory location ("b") - ebx above
            movq        -40(%rbp), %rax
            movl        %edx, (%rax)         Store value at memory location ("d")
            popq        %rbx                 Restore EBX
            popq        %rbp                 Normal function footer
            ret
```

  - Note: EBX is a “callee safe” register (→ C Calling Convention!), so the function must save it, if it should use it (as we do here)

Pointer example – Source code

```
.globl clearArray
.type    clearArray, @function
clearArray:
        pushq    %rbp
        movq     %rsp, %rbp
        movq     %rdi, -8(%rbp)    Store pointer in red zone
        movq     %rsi, -16(%rbp)   Store length in red zone
        movq     -8(%rbp), %rdx    Move ptr in some register
        movq     -16(%rbp), %rax   Move length in some register
        cmpq $0,%rax               Note that %2 has been replaced by rax
jle end
start:
decq %rax
movl $0,(%rdx,%rax,4)
cmpq $0,%rax
jg start
end:
        movq     %rax, -16(%rbp)
        popq     %rbp
        ret
```

We terminate our lines only with „\n“
GCC inserts „\t“ for its own lines!

Assembly functions used in C

- We have a library, which has been compiled/developed according to the System V AMD64 ABI calling convention
  - Typically this will be some other programming language, but this does not matter. We will use an assembly function here!
- What is needed?
  - The library as an object file or as a “real” library
  - A matching header file so the compiler can generate correct references to the functions to be provided by the library
    - These may already be provided with the library
    - If not, we will have to write them on our own. Helpful information may be extracted from the library.

Assembly functions used in C

- Example files:
  - Assembly file: power2.s
    - The full C calling convention (example of the exponentiation function)
  - power.h: Header file for this “library”
    - int power(int base, int exp);
  - power.c: Main program calling this function
    - ```
      #include <stdio.h>
      #include <stdint.h>

      #include "power.h“

      int main(void) {
              int res;
              res=power(2, 3);
              printf("2^3 = %d\n", res);
              printf("3^2 = %d\n", power(3, 2));
              return 0;
      }
      ```
      ```

Assembly functions used in C

- Example files:
  - Makefile:

```
power2.o: power2.s
	as -o power2.o power2.s

power: power.c power2.o
	gcc -c -Wall -ansi -pedantic -g power.c -o power.o
	gcc -o power power.o power2.o
```

  - Output:

```
2^3 = 8
3^2 = 9
```

Assembly functions used in C

- Example files:
  - power2.s: Assembly code (part)

```
#PURPOSE:   This function is used to compute the value of
#           a number raised to a power.
#INPUT:     First argument - the base number
#           Second argument - the power to raise it to
#OUTPUT:    Will give the result as a return value
#NOTES:     The power must be 0 or greater
#VARIABLES:
#           %rdi - holds the base number
#           %rsi - holds the power
#           %rax - holds the current/final result
   .globl power
   .type power, @function
power:
   pushq %rbp            # Save old base pointer
   movq %rsp, %rbp       # Make stack pointer the base pointer
   movq %rdi, %rax       # Store current result
   cmpq $0, %rsi         # If the power is 0, then we return 1
```

  - power.h: `int power(int base, int exp);`

THANK YOU FOR YOUR ATTENTION!

JOHANNES KEPLER UNIVERSITÄT LINZ

INSTITUTE OF NETWORKS AND SECURITY

https://www.ins.jku.at

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)