Computer Labs: Mixed C and Assembly Programming

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

October 27, 2011

Assembly Programming: Why?

Some things can be done only in assembly For example:

- Input/Output operations
- Issue the return from interrupt call

Basically, execute machine instructions that are not used for general programming.

Sometimes, assembly is better You have total control on the instructions executed:

- Good for performance (depends on the compiler)
- Good for timing (only for simple architectures)

Assembly Programming: Why Not?

Coding Performance

- Programming in assembly requires a lot more effort from the programmer

Robustness

- The number of bugs in a program is roughly proportional to the number of lines of code

Code Portability

- Even Linux device drivers use some C kernel functions for I/O

Assembly Programming in LCOM

- No “standard” Minix 3 device driver has assembly code
- All lab assignments could be implemented in C only
- However, assembly programming is fairly common in embedded systems
  - Usually, used together with C.

Mixing C and Assembly

Inline Assembly The assembly code fragments are embedded in C source code.

Example GCC

```
asm( "hlt" );
```

Convenient to optimize a small code fragment.

Linked Assembly Assembly code and C code are written in separate files.

- The assembly files are assembled separately to object code
- The executable is built by linking the object code with that generated by the C compiler

Easier to maintain, especially if the code is supposed to run in computers with different machine code.

GNU Assembler (Gas)

- Is the assembler used to generate object code from the output of the GNU C compiler
  - Actually, it is a family of assemblers, as `gcc` supports several computer architectures.
- GCC supports both
  - Inline assembly
  - Linked assembly
- Minix 3 build system includes the `gas2ack` tool
  - Translates from GNU assembly to the language accepted by its default assembler, the ACK assembler
  - Requires linked assembly

GNU's Assembler Conventions (AT&T Syntax)

- Register names are preceded by a %, e.g `%eax`
- Immediate operands are prefixed with a `$`, e.g. `$8`
- The size of the operands is specified by appending the character `b, w, l` (byte, word, long) as appropriate to the instruction mnemonic, e.g. `movb`
- In two operand instructions the order is: source, destination

  `movb $8, %ah`
  - Intel's convention is: destination, source
- Memory references must be enclosed in parenthesis `()`:

  `displacement(base reg., offset reg., scalar multiplier)`

  instead of:

  `[base reg. + displacement + offset reg. * scalar multiplie`
  - Either or both of the numeric parameters, and either of the register parameters may be ommitted. E.g.

    `movl %ecx, 8(,%eax,4)`
- Actually, GAS also supports the "Intel syntax", but apparently `gas2ack` does not

GAS Key Syntatic Elements (1/2)

Comments C style: `/* */`

- Also `#`, for IA-32: comment till the end of the line

Symbol "one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters `'_.$'`"

- "No symbol may begin with a digit."
- " Case is significant."

Statement

- "begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is."
  - "The key symbol determines the syntax of the rest of the statement."
  - "If the symbol begins with a dot '.' then the statement is an assembler directive"
  - "If the symbol begins with a letter the statement is an assembly language instruction"
- "ends at a newline character or line separator character. (The line separator is usually ';"'

GAS Key Syntatic Elements (2/2)

Constants "A constant ... is a value known by inspection, without knowing any context"

Character Constants **Chars** just like C chars, e.g. `'0'`, `\n`

Strings just like C strings, e.g. `"Hello, World!"`

Numbers

Integers May be in binary, octal, decimal or hexadecimal.

- Depending on their prefix: `0b` (or `0B`), `0`, no-prefix, `0x` (or `0X`)
- Negative number use the prefix operator `-`

Flonums represents a floating point number

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                 # - pi, a flonum
```

GAS: Symbols

- Are used by programmers to name things

  Label "represents the current value of the active location counter"
  - A symbol followed by a colon `:`
  - Can be used as:
    - The name of a function
    - The name of a variable
    - The name of a constant/literal

  Dot `'.'` "refers the current address that `as` is assembling into"
- Can be assigned an arbitrary value

GAS Expressions

Def: “specifies an address or numeric value.”

Integer Exprs

Operators Essentially, C operators: arithmetic, shift, bitwise boolean, comparison, logic boolean

Arguments Can be symbols, numbers or subexpressions, which are delimited by `'('` and `')'`

GAS Sections

Def: “a section is a range of addresses, with no gaps; all data “in” those addresses is treated the same for some particular purpose. For example there may be a “read only” section. “

- They are used to ensure that the linker keeps related “entities” together

- An object file generated by `as` has at least 3 sections, any of which may be empty:

  text code (program) section
  data initialized data section
  bss uninitialized data section

  - Space can be allocated in the bss
  - No initial value can be assigned to it.
    - The run time may initialize it to 0, when the program starts running

(Some) GAS Directives/Pseudo Ops (1/4)

Section specification specifies the section the assembly code will be assembled into

- `.text` code (program) section
- `.data` initialized data section
- `.bss` uninitialized data section
- `.section <section_name>` for defining an arbitrarily named section. Not clear this is supported by Minix 3.

Symbol related

- `.global/.globl` makes symbol visible to linker
- `.extern` not needed: GAS "treats all undefined symbols as external"
- `.bss` uninitialized data section
- `.section <section_name>` for defining an arbitrarily named section. Not clear this is supported by Minix 3.

(Some) GAS Directives/Pseudo Ops (2/4)

Data definiton ... in the `.data` section

`.ascii/.asciz` ASCII strings (/zero terminated)

`.byte` byte

`.hword/.short` 16-bit number

`.int/.long` 4 bytes (depends on architecture)

`.double` floating point (FP) number (depends on configuration)

`.float/.single` FP number (depends on configuration)

IMPORTANT IA-32 architecture is **little endian**

```
prompt_str:
    .ascii "Enter Your Name: "
var:
    .int 40
array:
    .byte  89, 10, 67, 1, 4, 27, 12,
           34, 86, 3
```

(Some) GAS Directives/Pseudo Ops (3/4)

Space Allocation ... in the `.bss` section

- It makes no sense to define data in the uninitialized section

`.lcomm` "Reserve length (an absolute expression) bytes for a local common denoted by symbol."

`.comm` Also reserves space, but with a twist. You can check the documentation.

```
.bss
   # Reserve 32 bytes of memory
   .lcomm  buff, 32
```

(Some) GAS Directives/Pseudo Ops (4/4)

`.equ/.set` "Sets the value of a symbol to expression. I.e. defines a symbolic constant

```
prompt_str:
    .ascii "Enter Your Name: "
pstr_end:
    .set STR_SIZE, pstr_end - prompt_str
```

Note Could have used `.`, i.e. the dot symbol, rather than defining the `pstr_end` symbol.

`.rept/.endr` Repeat the sequence of lines in the "reptition block"

```
.rept     3
.long     0
.endr
```

GAS, GCC and Include Files (1/2)

- GAS does not include a pre-processor
- It is possible to take advantage of GCC's pre-processor:
  - Invoke `gas` via `gcc`
    - The name of the file should have the suffix `.s`
  - In Minix 3, the default C compiler also behaves like `gcc`
    - `ENTRY` is a macro defined in `<machine/asm.h>`

GAS, GCC and Include Files (2/2)

```
/* void set_timer2_freq(); */
/*     using an initialized global variable for the frequency
#include <machine/asm.h>
#include "i8254.h"

.global _freq
.data
_freq:
        .short 0

ENTRY(set_timer2_freq)
movw _freq, %cx /* read the frequency from the global variable
movb $(SEL_T2 | LSB_MSB | SQR_WAVE | BIN_MODE), %al /* configu
outb $TIMER_CTRL
movl $((TIMER_FREQ) & 0x0000FFFF), %eax /* compute the divisor
movl $((TIMER_FREQ >>16) & 0x0000FFFF), %edx
div %cx
movb %cl,%al /* load LSB */
outb $TIMER_2
movb %ch,%al /* load MSB */
outb $TIMER_2
ret
```

Further Reading

- Dr. Paul Carter, *PC Assembly Language*
  - Section 1.3: Assembly Language
  - Section 1.4: Creating a Program
- OSdev.org: Inline Assembly
- GAS Syntax Chapter of the x86 Assemby Wikibook
- Ram Narayan. “Linux assemblers: A comparison of GAS and NASM, IBM DeveloperWorks, 17 Oct. 2007
- “An Introduction to the GNU Assembler
- “Using as, the official documentation from GNU