CS Basics
4) Programming in Assembly Language

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

Programming in Assembly Language

- **Development Process**
  - Assembly Language
  - Development Process Description
  - Using Make
- **First Steps**
- **Flags Register**
  - Increment and Decrement
  - Conditional Jumps
- **Signed Values**
- **Structure of an Assembly Program**
  - Strings in Assembly
- **Conclusion**

Development Process

Compiler vs. Assembler

Basic idea: text in, machine code out

- Because writing machine instructions by hand is tedious!

Compiler

- A "translator program"
- Reads source code (C, C++, Java, ...)
- Writes out *object code* files

Assembler

- Special type of translator for *assembly language*
- Characteristic: *Total control over the generated object code*
- Opposed to compilers, in general highly architecture dependent

Assembly Language

Machine instructions are written in a text file

- Readable by humans

Mnemonic[1]

- Every machine instruction has a mnemonic

Example

- Machine instruction `0x9C`
- Pushes the flags register on the stack
- Mnemonic: `PUSHF`
- Easier to remember than `0x9C`

---

[1] From Greek “mnemonika” — memory aid

Assembly Source Code

Arrangement of mnemonics

```
mov rax, 1      ; Code for sys_write
mov rdi, 1      ; File descriptor: 1 (Standard Output)
mov rsi, EatMsg ; Pass offset of the message
mov rdx, EatLen ; Pass the length of the message
syscall         ; Make kernel call
```

Mnemonic + operands = instruction

Comments

Comments start with “;”

- Content after the “;” is ignored by the assembler

Assembly is not C or Java

- Large code fragments are difficult to understand
- Code should be commented extensively

Beware of “write-only” source code

- You write some assembly code in October
- Then, you start to learn C in November
- In January, you try to figure out what you did in October…
- …mission impossible…

Only solution: properly comment your code!

Assembling

The Assembler

- Transforms assembly language source files into an *object module*

Object modules cannot be run directly

- They need to be *linked* to other object modules (code parts, libraries…)

Code in general is split over multiple files

- Easier to maintain
- Separation by functionality
- Code reuse

The Linker and Object Modules

Initially, the linker seems superfluous

- At first, you will only work with a single source file

Some parts of a program might be reused

- Routines
- Libraries of functions
- Advantage: *write and test once, reuse many times*

Object module contains

- Program code, including named procedures
- References to named procedures in other modules
- Named data objects such as numbers and strings with predefined values
- Empty named data objects, which serve as place holders
- References to data objects in other modules
- Debugging information
- Other odds and ends that help the linker create an executable file

Assembling and Linking Overview

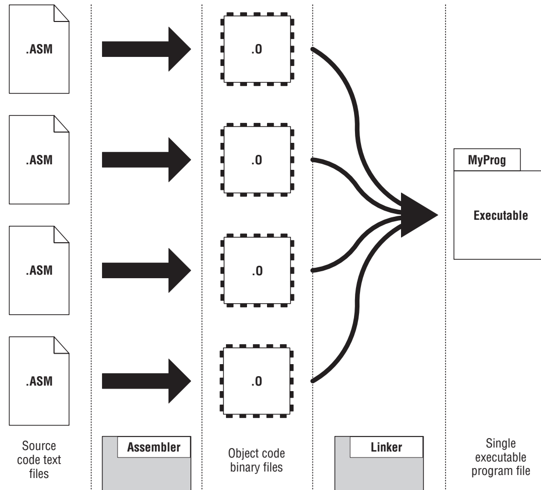

Figure 5-8: The assembler and linker

Development Process Overview

1. **Create assembly language source code**
2. **Assemble the source code into object modules**
3. **Link the object modules into an executable program file**
4. **Test and debug the program**
5. **Fix and extend the source code**
6. **. . . repeat . . .**

Development Process Overview

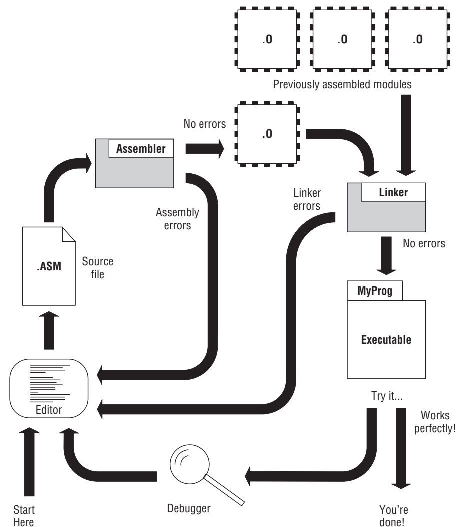

Figure 5-9: The assembly language development process

Development Process I

Create and edit assembler source code files

- Using an editor of your choice (vim, emacs, gEdit, kate, ...)
- Save the file with ending `.asm`
- Properly comment it!

Assemble the file

- Generates the corresponding object module (ending: `.o`)
- Assembly errors might occur

Back to the editor

- Fix the errors using the messages given by the assembler
- Try assembling again

A note regarding *warning messages*

- Do not ignore any warning messages!
- Even if the code is correct, they may provide important hints for improving it

Development Process I, Example

Assembling the file `eatsyscall.asm`

```
$ cd asm-4-programming/examples/eatsyscall
$ ls
eatsyscall.asm  Makefile
$ nasm -f elf64 -g -F dwarf eatsyscall.asm -o eatsyscall.o
$ ls
eatsyscall.asm  eatsyscall.o Makefile
```

- `nasm` invokes the assembler
- `-f elf64` instructs it to generate the object module in the “elf” format (64-bit)
- `-g` requests *debug information* to be included in the object module
- `-F dwarf` specifies the format for the debug information (“dwarf”, to be used with gdb/ddd)
- `eatsyscall.asm` is the source file
- `-o eatsyscall.o` specifies the output file (optional)

Development Process II

Link the object modules

- Input: one or more object modules (`.o` files)
- Output: an executable program file
- Linker: `ld` (like "load")

Linker Errors

- References to external functions are not correct
- In general, the error lies in our code
- Try to read and understand the messages

Starting and testing the executable file might...

- ...work as expected (seldom)
- ...produce errors or just simply crash
- ...lead to unwanted behavior (bugs!)

Development Process II, Example

Link the file `eatsyscall.o`

```
$ ls
eatsyscall.asm  eatsyscall.o Makefile
$ ld -o eatsyscall eatsyscall.o
$ ls
eatsyscall  eatsyscall.asm  eatsyscall.o  Makefile
```

- `ld` invokes the linker
- `-o eatsyscall` instructs it to write the executable file to `eatsyscall`
  If this option is not given, the output will be called `a.out`
- `eatsyscall.o` specifies the input file

Test the file

```
$ ./eatsyscall
Eat at Joe's!
```

Development Process III

Debugging using a debugger

- Designed to help locate and identify bugs

Execute machine instructions

- One at a time, step by step
- Inspect registers or memory while program is running

Development Process III, Example

Invoke the gdb debugger

- Run `gdb EXECUTABLE` in a shell
- Use `(gdb) break LABEL` to set a breakpoint
- Use `(gdb) run` to start the execution
- Use CTRL-x a to view the source while debugging
- Use `(gdb) n` to execute the next instruction
- Use `(gdb) print $REG` to inspect a register

Use a front-end for gdb

- `gdb` can be complicated to use
- `kdbg` and `ddd` provide graphical front-ends for gdb
- Invocation: `kdbg EXECUTABLE` or `ddd EXECUTABLE`
- Comfortably inspect registers and memory
- Many other front-ends for `gdb` exist as well...

See also the provided gdb cheat sheet for additional information!

Make Overview

`make` helps structuring the compilation of files

- Widespread usage in the C world (C, C++, asm)
- Might also be used for other tasks! (i.e. these slides are built using `make`)

Controlled using `Makefiles`

- Written by the developer or a tool
- Lists files to be compiled/linked and dependencies among them
- Intelligent: Recompiles files only if needed

“Similar” to ant, maven, graddle for Java

Make Dependencies

Object files depend on source files

- If the source file has changed, the object file must be recompiled
- For this, `make` compares the modification dates of the `.o` and the `.asm`, `.c` or `.cpp` files

Executable files depend on object files

- The linker will have to re-link the files if one of them has changed.

Dependencies are specified using rules

- For each file generated, they specify the file(s) it depends on
- Special commands can be specified for generation

Make Example I

`make` already includes many *implicit rules* for building executables and other files.[2] Using them shortens the contents of `Makefiles`:

- **An executable depending on a single object file**

  ```
  eatsyscall:  eatsyscall.o
  ```

- **An executable depending on multiple object files**

  ```
  linkbase:  linkbase.o linkparse.o linkfile.o
  ```

- **Explicit rules may also be used instead**

  ```
  eatsyscall:  eatsyscall.o
  	ld -o eatsyscall eatsyscall.o
  ```

Important: Indentation must be done using `<TAB>` characters (not spaces!)

[2] Hint: Use “`make -p`” to display them!

Make Example II

To assemble a `.asm`-file into an executable file requires *two steps*

- First assemble the `.asm` file into a `.o` file
- Then link the `.o` file into an executable binary
- The order of the rules is not the order of the execution of the steps.

Example

```
eatsyscall: eatsyscall.o
    ld -o eatsyscall eatsyscall.o
eatsyscall.o: eatsyscall.asm
   nasm -f elf64 -g -F dwarf eatsyscall.asm
```

First Steps

Assembly Language Sandbox I

For studying the assembly language, we will use a “sandbox”

- Minimal program, no functionality
- Useful for playing around and trying out things
- May be changed and re-used as much as we want

Solution

- A program “skeleton” with no instructions, which we extend
- The only thing required: a defined *entry point*

Assembly Language Sandbox II

- **sandbox.asm**

```
SECTION .data

SECTION .bss

SECTION .text

global _start

_start:
    ; Put your experiments in here...
    nop
```

- *Note: This program does not exit properly, it can just be used for testing inside a debugger.*

The `mov` Instruction

***Copies(!)* data from one location to another**

- Syntax: `mov destination, source`

Is one of the most frequently used instructions Can be used to copy data

- From register to register
- From a register to memory
- From memory to a register
- *But NOT from memory to memory!*

Example

```
mov rax, 42
```

Immediate Data (Immediate Addressing)

The data is part of the machine instruction itself

- It is not stored in a register or in memory
- The opcode implies the instruction length
- Examples (opcodes in comments):

```
mov rax, 42       ; B8 2A 00 00 00
mov rbx, 'Hello'  ; 48 BB 48 65 6C 6C 6F 00 00 00
mov rcx, 0xABCD   ; B9 CD AB 00 00
```

Beware of register size!

- The following causes the value to be only partially stored:

```
mov cl, 0x67EF ; warning: byte data exceeds bounds
```

Register Addressing

Register addressing

- Simply name the register to work with:

```
mov bl, ch        ; 8-bit
add dx, ax        ; 16-bit
mov ebp, esi      ; 32-bit
add ecx, edx      ; 32-bit
add rax, rbx      ; 64-bit
```

`add` instruction

- Adds the source to the destination

Example

Test the following instructions in a debugger

- Set a breakpoint on the first instruction
- Single-step through all instructions
- Inspect registers after each step
- See like register halves (8-bit) are used for accessing BX and CX

```
mov ax, 0x67FE
mov bx, ax
mov cl, bh
mov ch, bl
```

Copying Data from/to Memory

To access data in memory, registers are used

```
mov rbx, [Address] ; copy 64 bits from mem to rbx

mov rbx, Address   ; copy memory address (!) to rbx
mov ax, [rbx]      ; copy 16 bits
mov eax, [rbx+3]   ; copy 32 bits with offset 3
mov rcx, 5
mov rax, [rbx+rcx] ; copy 64 bits with offset 5
```

The []-operator accesses memory at the given address

- `[Address]` refers to memory address "`Address`"
- `[rbx]` refers to memory whose address is stored in `rbx`
- `[rbx+3]` memory at address in `rbx` plus 3
- `[rbx+rcx]` memory whose address is the sum of the values in `rbx` and `rcx`

Register size specifies amount of data copied!

Addresses vs. Labels

In general, *labels* (see Slide 50) are used as addresses

```
SECTION .data
  EatMsg: db "Eat at Joe's"

SECTION .text
  ...
  mov rcx, EatMsg   ; copy the address
  mov rdx, [EatMsg] ; copy the first 64 bits
                    ; of the message
```

- `EatMsg` is the label for the address of the string
- `[EatMsg]` is the content (i.e. first 8 bytes!) of the string

Flags Register

Flags Register I

The RFLAGS register is a 64-bit register

- Contains multiple 1-bit *flags*
- 17 flags are defined
- In the following, we present only the most useful ones

CF: Carry Flag

- Used in *unsigned* arithmetic operations
- Set if the result “carries out” a bit from the operand

OF: Overflow Flag

- Set when the result of an arithmetic operation on a signed integer quantity becomes too large
- Is generally used as “carry” flag in *signed* arithmetic

Flags Register II

SF: Sign Flag

- Set when the result of an operation forces the operand to become negative
- I.e. if the most significant bit becomes `1` during the operation

ZF: Zero Flag

- Set when the result of an operation becomes zero

PF: Parity Flag

- Indicates the parity *only in the least significant byte* of an operation result
- Parity: Number of bits set to `1`; if even, `PF` is `1`.

Flags Register III

IF: Interrupt enable flag

- Can be set by the programmer (`cli` and `sti`)
- Requires enough privileges to set (i.e. OS only in general)
- Specifies if *maskable interrupts* will be handled or not
- ...if IF=0, interrupts will be ignored
- ...if IF=1, interrupts may occur and will be handled

TF: Trap Flag

- Used by debuggers to enable single step mode
- If set, the CPU only executes a single instruction and stops immediately after it

Increment and Decrement

`inc` instruction

- Adds `1` to the given operand

`dec` instruction

- Subtracts `1` from the given operand

Example

```
mov eax, 0xFFFFFFFF
inc eax

mov ebx, 0x2D
dec ebx
```

- `eax` becomes `0`
- `ebx` becomes `0x2C`
- `inc` sets PF (parity), AF (auxiliary) and ZF (zero)
- `dec` clears all flags (except IF)

Conditional Jumps

For most of the flags, there is an associated *conditional jump instruction*

- E.g. for ZF: `jnz` (“jump if not zero”)
- If ZF is cleared (=0), execution jumps to specified destination
- If ZF is set (=1), instruction flow continues normally

Example: First loop

```
  ...
  mov rax, 5
loop1:
  dec rax
  jnz loop1
  ...
```

As long as `rax` is not zero, jump to `loop1`

Example: kangaroo.asm I

```
SECTION .data                     ; Section containing initialized data

    Snippet:    db "KANGAROO"

SECTION .text                     ; Section containing code

global _start                     ; Linker needs this to find the entry point!

_start:
    mov rbx, Snippet
    mov rax, 8

loop:
    add byte [rbx], 32
    inc rbx
    dec rax
    jnz loop

exit:
    mov rax, 60                   ; Code for exit syscall
    mov rdi, 0                    ; Return a code of zero
    syscall                       ; Make kernel call
```

Example: `kangaroo.asm` II

The string “KANGAROO” is stored in memory

- Can be addressed using the label “`Snippet`”

`eax` serves as counter

- Initialized to the length of the string
- Decremented on each character modification

`ebx` points to a character in the string

- Initialized to the address of `Snippet`
- Incremented on each character modification
- Points to the next character to modify

String modification

- By adding 32 to a character, it is converted to lowercase

→ **Run the program in the debugger and inspect memory!**

Signed Values

Signed Values I

- **Remember: x86 architecture uses two's complement for signed values**
  - -42 is stored as 256-42 (=0x100-42) in a 8-bit register
  - …in 32 bits: 0x100000000-42
  - …in 64 bits: 0x10000000000000000-42
- **Let us try the following code**
  Note: "`jmp`" is unconditional (always jumps)

```
  mov eax, 0
loop2:
  dec eax
  jmp loop2
```

- `eax` **becomes**

```
0xFFFFFFFF (-1)
0xFFFFFFFE (-2)
0xFFFFFFFD (-3) ...
```

Signed Values II

Another Example

- Incrementing the largest positive number leads to a negative number
- Changes sign flag (SF)

```
mov eax, 0x7FFFFFFF
inc eax      ; sets PF AF SF OF
```

`neg` (Negate) Instruction

`neg` transforms a positive into a negative number

Example

```
mov rax, 42
neg rax      ; sets CF AF SF
add rax, 42
```

Copying Signed Numbers (`movsx`)

When using `mov`

- The sign of the copied value is not preserved
- Depending on size, the destination value loses its sign

Solution: `movsx` — move with Sign eXtension

```
mov ax, -42
mov bx, ax       ; ok, bx is -42
mov ebx, eax     ; wrong, ebx is 65494
movsx ebx, ax    ; ok, ebx is -42
```

Multiplication and Division

`mul` and `div`: unsigned operations

- `mul` multiplies two operands
- The result is larger than the individual operands
- If the operands are 64 bit registers, the result may require up to 128 bits…
- …Solution: use two registers for storing the result

`imul` and `idiv`: signed operations

- Same functionality for signed numbers

`mul`: Implicit Operands

- **`mul` has an *implicit* operand!**
  - Depending on size, one factor is always in `al`, `ax`, `eax` or `rax`
  - Second factor lies in a register specified by the programmer
- **Result (product) is stored in two registers**
  - For 8 bit multiplication (using `al`), `ax` is used
  - For larger multiplications, `dx`, `edx` and `rdx` are used to store the “higher-order” parts of the result (sets `CF` and `OF`)

Examples

```
mov rax, 0xFFFFFFFFFFFFFFFF  ; 2^64-1 (!)
mov rbx, 2
mul rbx         ; result stored in rax and rdx!

mov rax, 56
mov rbx, -67
imul rbx        ; signed multiplication
```

Structure of an Assembly Program

Structure of an Assembly Program

Initial comment block

- Author, date, version, description…

`.data` Section

- Contains all *initialized data*
- Values must be specified before assembling
- Increases executable file size

`.bss` Section

- Contains placeholders for *uninitialized data*
- Space will be reserved in memory
- Does not increase executable file size

`.text` Section

- Contains the actual code (opcodes)
- GNU/Linux: Requires a *global* label `_start` (see Slide 50)

Example: eatsyscall.asm

```
;  Executable name : eatsyscall   ;  Version          : 1.0
;  Created date    : 1/7/2009     ;  Last update      : 2/18/2009
;  Author          : Jeff Duntemann
;  Description     : A simple program in assembly for Linux, using NASM,
;    demonstrating the use of Linux syscalls to display text.
;
;  Build using these commands:
;    nasm -f elf64 -g -F dwarf eatsyscall.asm
;    ld -o eatsyscall eatsyscall.o
SECTION .data                      ; Section containing initialized data
    EatMsg: db "Eat at Joe's!",10
    EatLen: equ $-EatMsg           ; Compute the length of the string

SECTION .bss                       ; Section containing uninitialized data

SECTION .text                      ; Section containing code

global _start                      ; Linker needs this to find the entry point!

_start:
    mov rax, 1                     ; Code for sys_write call
    mov rdi, 1                     ; Specify File Descriptor 1: Standard Output
    mov rsi, EatMsg                ; Pass offset of the message
    mov rdx, EatLen                ; Pass the length of the message
    syscall                        ; Make system call

    mov rax, 60                    ; Code for exit syscall
    mov rdi, 0                     ; Return a code of zero
    syscall                        ; Make system call
```

Labels

Label Definition

- Must begin with a letter or underscore
- Is case sensitive
- Definition of label name should be followed by a colon (`:`)

Labels are used for addressing memory and as targets for jumps

- A label is a textual name of a value or a memory address

```
global _start
```

- Is required in every GNU/Linux program
- Defines the entry point used by the program loader to find the first instruction to execute

Initialized Data

Data Definition

- In the `.data` section
- Using data definition directives

Examples

```
MyByte:   db 0x07                 ; single byte
MyWord:   dw 0xFFFF               ; 16 bits
MyDouble: dd 0xB8000000           ; 32 bits
MyQuad:   dq 0xB800000011000000   ; 64 bits

EatMsg:   db "Eat at Joe's!",10   ; multiple bytes
```

What is a String in Assembly?

A string is just an array of bytes in memory

- Each byte may be interpreted as a character (beware of the encoding!)
- The size of the string is *not* stored anywhere (responsibility of the programmer!)
- There is no end-of-string delimiter (like e.g. 0x00 in C)

Examples

```
SECTION .bss ; Uninitialized data
  Buff:     resb 16  ; 16 byte buffer

SECTION .data    ; Initialised data
  HexStr:   db "00 01 02 FD FE FF",10
  Digits:   db "0123456789ABCDEF"
```

Problem: Actual length of a string?

Calculate String Length I

In general, the length of the string is required

Possible solution: hard-code it

```
EatMsg:    db "Eat at Joe's!"
EatLen:    equ 13
```

- But if the string changes, length must be adjusted as well
- This is tedious and error-prone!

Btw: `equ` stands for "equate"

- Associates a label with a constant value

```
FieldWidth:    equ 10

; The following two lines are equal!
mov rax, 10
mov rax, FieldWidth
```

Calculate String Length II

Better: Use the "here"-token: `$`

- Represents the current location while NASM is assembling the file
- `$` and `EatMsg` are both locations (i.e. addresses in memory)
- Can be used to calculate the length of the string

```
EatMsg:    db "Eat at Joe's!"
EatLen:    equ $-EatMsg
```

String Concatenation

Strings can be concatenated using a comma (“,”)

Examples

- Add “End of Line” (EOL, 10 / 0x0A) to the end of the string

```
EatMsg:   db "Eat at Joe's!",10
```

- Multi-line string

```
TwoLine:  db  "Eat at Joe's ...",10,"... tonight!",10
```

When displayed, produces two lines

```
Eat at Joe's ...
... tonight!
```

Example: stringcopy.asm

```
SECTION .data                       ; Section containing initialized data
    InitString:      db "Eat at Joe's!",10
    InitLen:         equ $-InitString
    TargetString:    db "             ",10
    TargetLen:       equ $-TargetString

SECTION .text                       ; Section containing code
global _start                       ; Linker needs this to find the entry point!
_start:
    mov r8, InitLen                 ; Loop counter = length of string
    mov r9, InitString              ; Position in source string
    mov r10, TargetString           ; Position in target string

loop:
    mov al, byte [r9]               ; Copy single byte from source to AL
    mov byte [r10], al              ; Copy single byte from AL to target
    inc r9                          ; Increment source position
    inc r10                         ; Increment target position
    dec r8                          ; Decrement loop counter
    jnz loop                        ; Loop as long as counter is > 0

output:
    mov rax, 1                      ; Code for sys_write call
    mov rdi, 1                      ; Specify File Descriptor 1: Standard Output
    mov rsi, TargetString           ; Pass offset of the message
    mov rdx, TargetLen              ; Pass the length of the message
    syscall                         ; Make kernel call
```

Conclusion

Conclusion

Assembling

- First assemble `.asm` into `.o`
- Link all `.o` files into a single executable
- Typically automated using `make`

Structure of a program

- Initialization of data in memory
- Placeholders to reserve space in memory
- Instructions