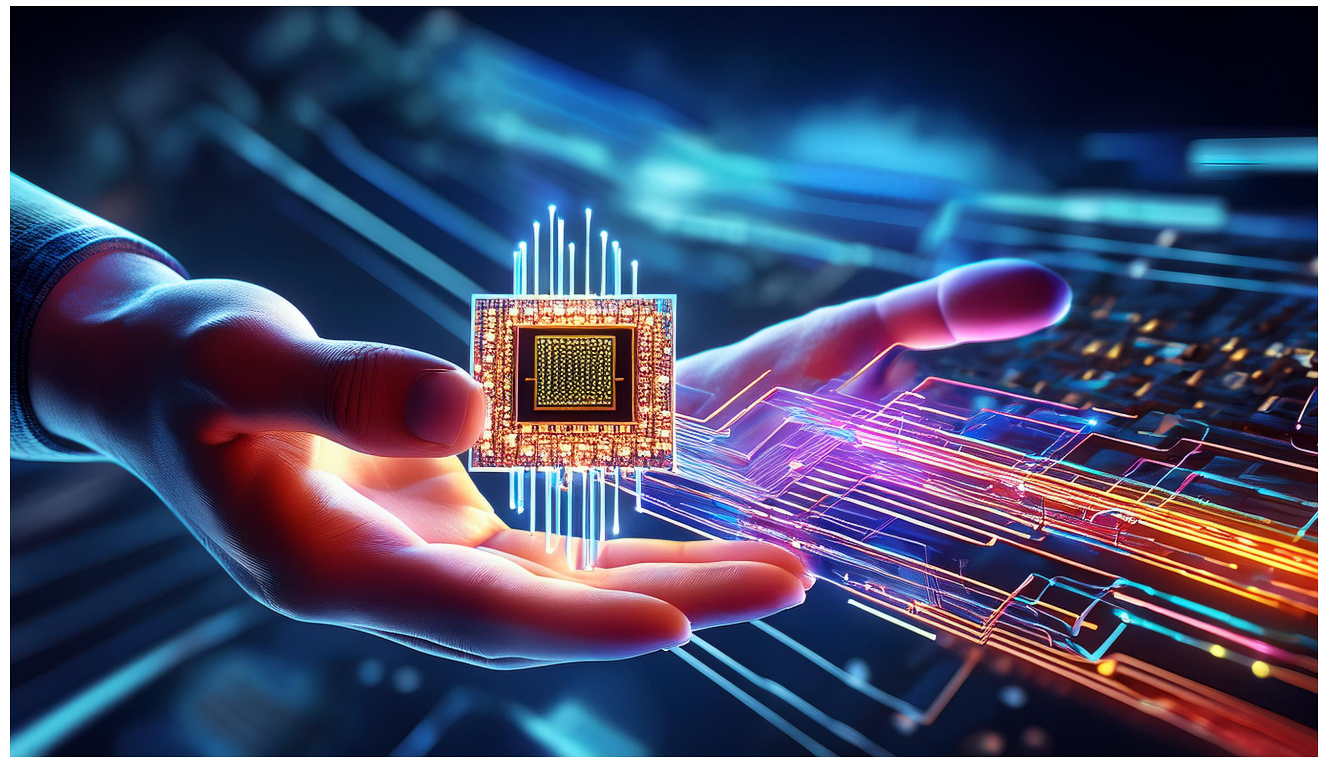
22.11.24 18:55

CSCI 250
Introduction to Computer Organisation
Lecture 5: Compiler Basics II



Jetic Gū

2024 Fall Semester (S3)

Overview

- Architecture: von Neumann
- Textbook: CO: 4.5
- Core Ideas:

  1. Intro to ARM Assembly

P1
Intro ASM

Introduction to Assembly Language

P1
Intro ASM

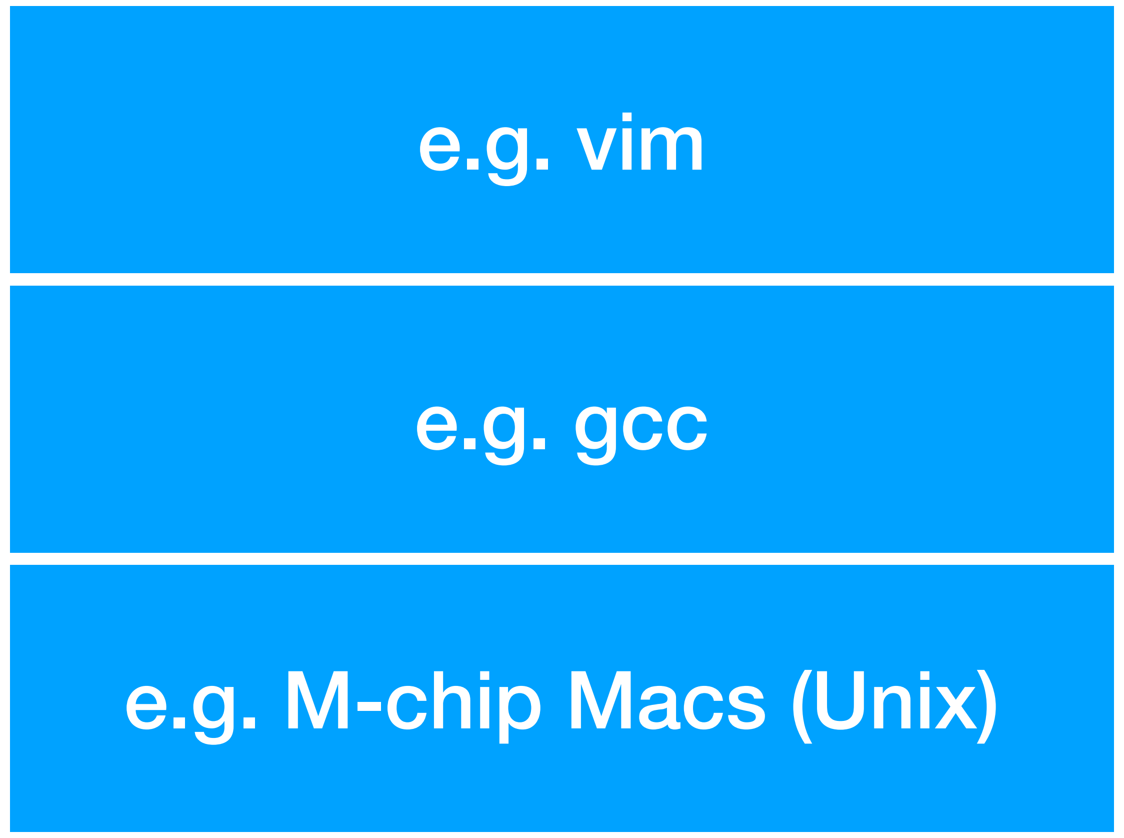
What are the properties of ASM?

- Assembly Languages are
  - The fasted programming language out there, equivalent to machine code (but readable)
  - Processor / Platform specific
- Knowledge of basic assembly language (those included in Lab 5 only) will be included in the final exam

Review

Development Environment

- You need
  - A text editor
  - An assembler
  - An appropriate computer



e.g. vim

e.g. gcc

e.g. M-chip Macs (Unix)

Technical

Caution!

- ARM64
  - Is a little different from the ARM16 we are using, but mostly because it’s 64bit. The 16bit thumb instructions are mostly the same.
  - Apple’s ARM64 assembly and Android’s ARM64 assembly differs a little, we will point out all differences, but all demos will be performed on Apple’s ARM64 for now.
  - Easy way to get into ARM assembly programming: Macs (Unix), or Raspberry Pi (Linux)

P1
Intro ASM

System Calls

- What are system calls?
    - System calls are Operating System provided functions, including the manipulation of `stdio` etc.
    - e.g. every function in `cstdio` (C++) and `stdio.h` (C) are system calls
    - Android uses Linux, which has slightly different system calls than Unix

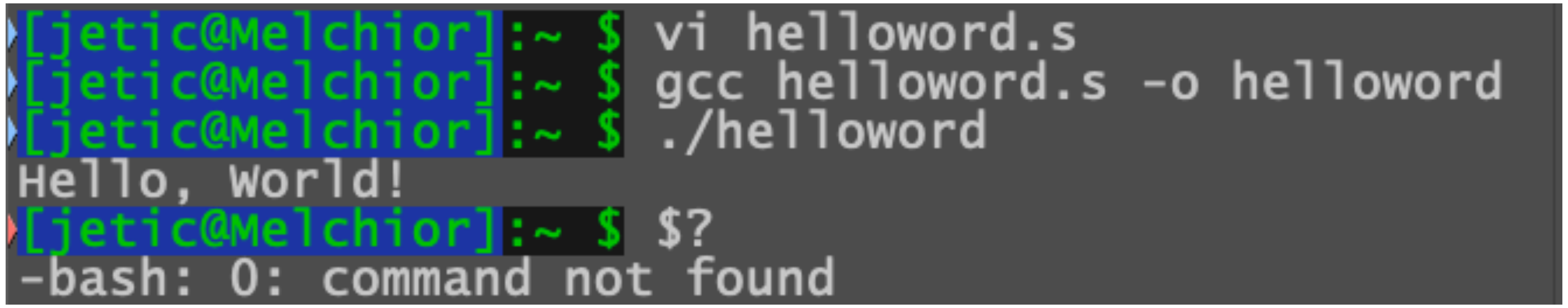
Technical

Hello World

```
.global _main
.align 2

_main:
    adr   x0, msg
    bl    _puts      // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit      // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

How to Compile?



```
[jetic@Melchior]:~ $ vi helloword.s
[jetic@Melchior]:~ $ gcc helloword.s -o helloword
[jetic@Melchior]:~ $ ./helloword
Hello, World!
[jetic@Melchior]:~ $ $?
-bash: 0: command not found
```

- I use `vim` to edit all my code, assembly code should have suffix `.s`
- `gcc` can be used as assembler, in this case it performs assembling and linking together
  - **assembling**: translate assembly code to binary
  - **linking**: system call codes are linked to the binary as well

Technical

Hello World

```
.global _main
.align 2

_main:
    adr   x0, msg
    bl    _puts      // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit      // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

Technical

Hello World

```
.global _main
.align 2

_main:
     adr    x0, msg
     bl     _puts        // syscall, write x0 to stdout
     mov    x0, #0
     b      _exit        // syscall, exit the programme
msg:
     .ascii  "Hello, World!"
```

assembler directive

Technical

P1
Intro ASM

Hello World

```
.global _main
.align 2

_main:
    adr   x0, msg
    bl    _puts      // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit      // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

assembler directive

Technical

Hello World

```
    bal _main
.align 2

_main:
    adr   x0, msg
    bl    _puts       // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit       // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

main function

Hello World

```
.global _main
.align 2

 main:
    adr    x0, msg
    bl     _puts          // syscall, write x0 to stdout
    mov    x0, #0
    b       exit          // syscall, exit the programme
msg:
    .ascii   "Hello, World!"
```

load address of string

actual string

P1
Intro ASM

Hello World

```
  bal _main
.align 2

 main:
    adr   x0, msg
    bl    _puts       // syscall, write x0 to stdout
    mov   x0, #0
    b      exit       // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

load address of string

actual string

Technical

P1
Intro ASM

Hello World

```
 obal _main
.align 2

 main:
    adr   x0, msg
    bl    _puts        // syscall, write x0 to stdout
    mov   x0, #0
    b      exit        // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

load address of string

actual string

Technical

Hello World

```
.globl _main
.align 2

_main:
    adr     x0, msg
    bl      _puts        // syscall, write x0 to stdout
    mov     x0, #0
    b       _exit        // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

system call

Technical

P1
Intro ASM

Hello World

```
.global _main
.align 2

_main:
    adr   x0, msg
    bl    _puts          // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit          // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

change x0 value

Technical

P1
Intro ASM

Hello World

```
    bal _main
.align 2

_main:
    adr   x0, msg
    bl    _puts       // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit       // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

change x0 value

Technical

P1
Intro ASM

Hello World

```
obal _main
.align 2

_main:
    adr   x0, msg
    bl    _puts        // syscall, write x0 to stdout
    mov   x0, #0
    b     _exit        // syscall, exit the programme
msg:
    .ascii  "Hello, World!"
```

change x0 value

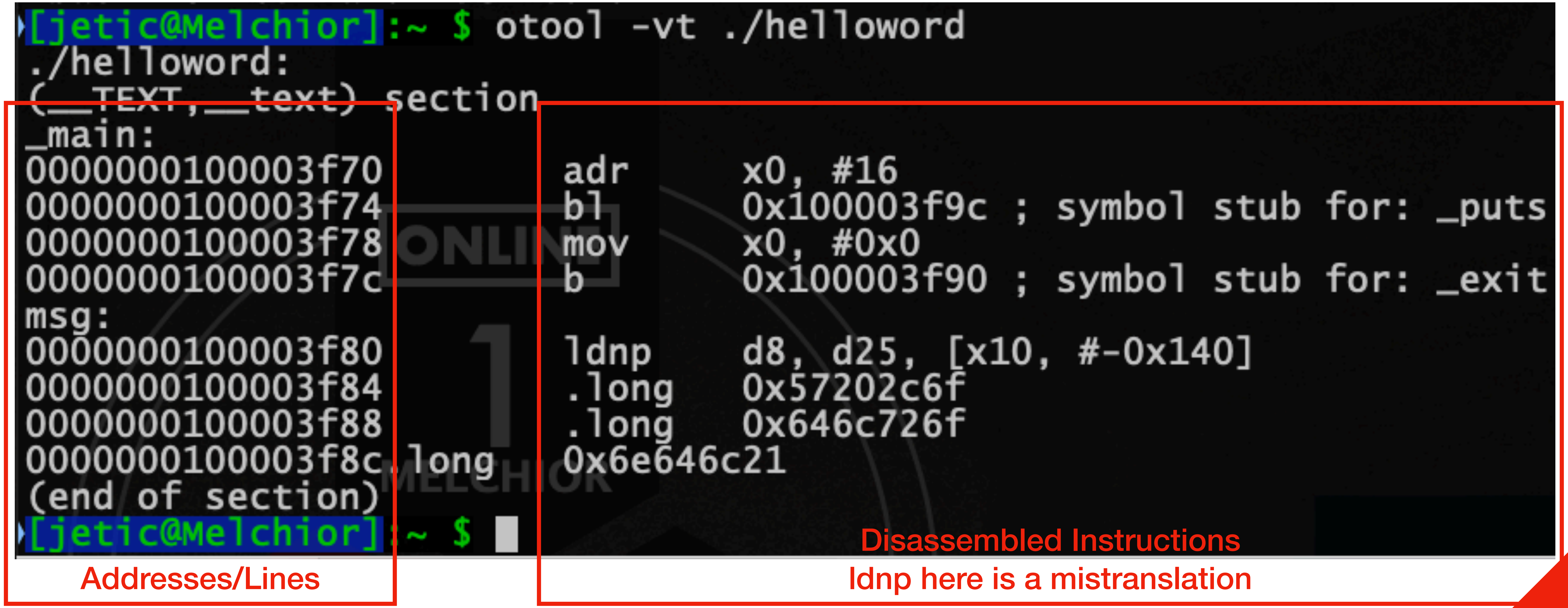
Technical

P1
Intro ASM

I want to look at my programme

- We can do that using a disassembler, like `otool`
  Note: `otool` doesn't handle variable values that well

- `otool -vt ./helloworld`



```
[jetic@Melchior]:~ $ otool -vt ./helloword
./helloword:
(__TEXT,__text) section
_main:
0000000100003f70        adr     x0, #16
0000000100003f74        bl      0x100003f9c ; symbol stub for: _puts
0000000100003f78        mov     x0, #0x0
0000000100003f7c        b       0x100003f90 ; symbol stub for: _exit
msg:
0000000100003f80        ldnp    d8, d25, [x10, #-0x140]
0000000100003f84        .long   0x57202c6f
0000000100003f88        .long   0x646c726f
0000000100003f8c.long   0x6e646c21
(end of section)
[jetic@Melchior]:~ $ 
```

Addresses/Lines

Disassembled Instructions
ldnp here is a mistranslation

Concept