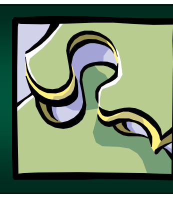
Programs

Part 3

1



Compilers, Assemblers & Linkers

Programs, Coding, and Nerds… oh my!

2

Compilers & Assemblers

- When you hit "compile" or "run" (e.g. in your Java IDE), many actions take place *"behind the scenes"*
- You are usually only aware of the work that the parser does

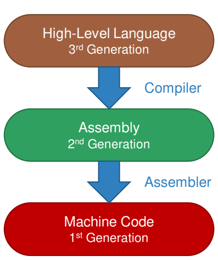
3

Development Process

1. Write program in high-level language
2. Compile program into assembly
3. Assemble program into objects
4. Link multiple objects programs into one executable
5. Load executable into memory
6. Execute it

4

From Abstract to Machine



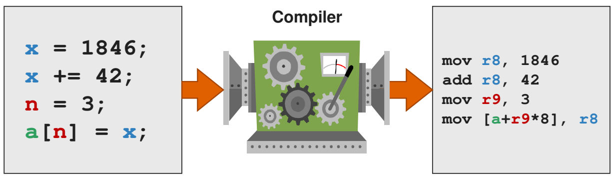
5

Compiler

- Convert programs from high-level languages (such as C or C++) into assembly language
- Some create machine-code directly…
- *Interpreters*, however…
  - never compile code
  - Instead, they run parts of their own program

6

Compilers: 3rd → 2nd Generation



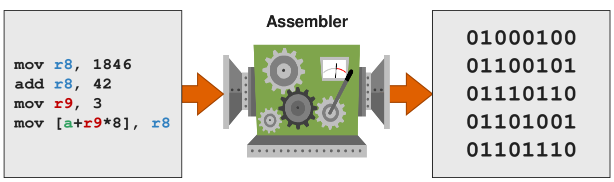
7

Assembler

- Converts assembly into the binary representation used by the processor
- Often the result is an *object file*
  - usually not executable - yet
  - contains computer instructions and information on how to "link" into other executable units
  - file may include: relocation data, unresolved labels, debugging data

8

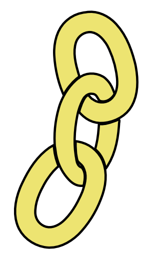
Assembler: 2nd → 1st Generation



9

Linkers

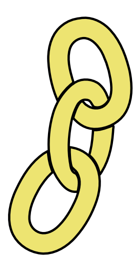
- Often, parts of a program are created separately
- Happens *more often than you think* – almost always
- Different parts of a program are called *objects*
- A *linker* joins them into a single file



10

What a Linker Does

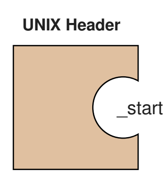
- Connects labels (identifiers) - used in one object - to the object that defines it
- So, one object can call another object
- A linker will show an error if there are label conflicts or missing labels



11

Linking your program

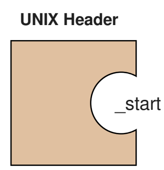
- UNIX header is defined by crt1.o and crti.o
- They are supplied behind the scenes, *so you don't need to worry about them*



12

Linking your program

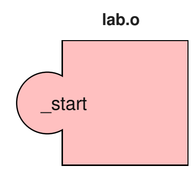
- It references a subroutine called _start
- But… it is not defined in the header
- It is used to start your program (main in Java)



13

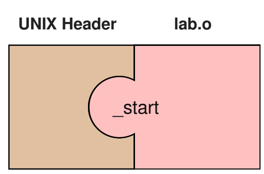
Linking your program

- Your program supplies this subroutine
- The linker connects the two, so the header calls your subroutine



14

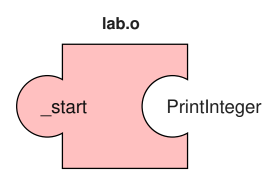
Linking to the UNIX Header



15

You will use my library

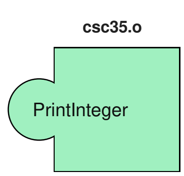
- To make labs easier, you will use my library
- Your program will reference its subroutines



16

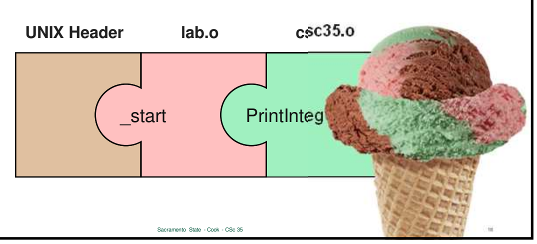
You will use my library

- Once the object file "csc35.o" is linked, the program is complete

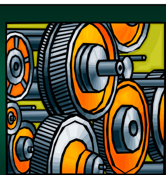


17

You will use my library



18



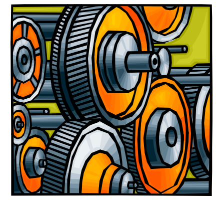
Assembly Basics

The beautiful language of the computer

19

Assembly Language

- *Assembly* allows you to write machine language programs using easy-to-read text
- Assembly programs is based on a specific processor architecture
- So, it won't "port"



20

Assembly Benefits

1. Consistent way of writing instructions
2. Automatically counts bytes and allocates buffers
3. *Labels* are used to keep track of **addresses** which prevents common machine-language mistakes

21

1. Consistent Instructions

- Assembly combines related machine instructions into a single notation *(and name)* called a *mnemonic*
- For example, the following machine-language actions are different, but related:
  - register → memory
  - register → register
  - constant → register

22

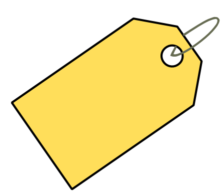
2. Count and Allocate Buffers

- Assembly automatically counts bytes and allocates buffers
- Miscounts (when done by hand) can be very problematic – and can lead to hard to find errors

23

3. Labels & Addresses

- Assembly uses *labels* to store **addresses**
- Used to keep track of locations in your programs
  - data
  - subroutines (functions)
  - …and much more



24

Battle of the Syntax

- The basic concept of assembly's notation and syntax hasn't changed
- However, there are two major competing notations
- They are *just* different enough to make it confusing for students and programmers *(who are used to the other notation)*

25

Battle of the Syntax

- AT&T Syntax
  - dominate on UNIX / Linux systems
  - registers prefixed by %, values with $
  - receiving register is last
- Intel Syntax
  - *actually created by Microsoft*
  - dominate on DOS / Windows systems
  - neither registers or values have a prefix
  - receiving register is first

26

AT&T Example

```
# Just a simple add

mov $42, %rbx       #rbx = 42
mov value, %rax     #rax = value
add %rbx, %rax      #rax += rbx
```


27

Intel Example

```
# Just a simple add

mov rbx, 42         #rbx = 42
mov rax, value      #rax = value
add rax, rbx        #rax += rbx
```


28

Assembly Program Structure

How these little beasties are organized

29

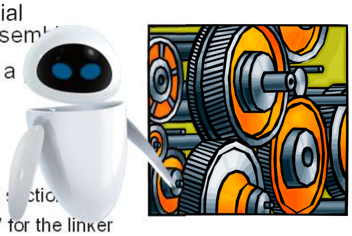
Assembly Programs

- Assembly programs are divided into two sections
- *data section* allocate the bytes to store your constants, variables, etc…
- *text section* contains the instructions that will make up your program

30

Directives

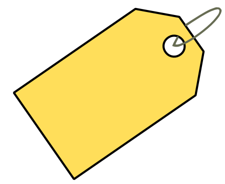
- A *directive* is a special command for the assem
- Notation: starts with a
- What they do:
  - allocate space
  - define constants
  - start the text or data
  - make labels "global" for the linker



31

Labels

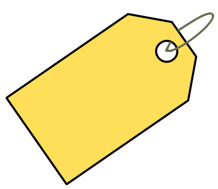
- You can define *labels* by following an identifier with a colon
- As the assembler is reading your program, it is generating machine code instructions and storage



32

Labels

- When the assembler sees a label declaration, it will save the current address (at that point) into a table
- Anytime you use a label, it is replaced by that address
- Labels are addresses



33

Hello World – Using csc35.o

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"

.text
.global _start

_start:
    lea   rax, message
    call  PrintString

    call  Exit
```

34

Hello World – Using csc35.o

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"

.text
.global _start

_start:
    lea   rax, message
    call  PrintString

    call  Exit
```

Data Section

35

Data Section

Use Intel format

No prefix characters

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"
```

Start data section

36

Data Section

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"
```

Create a label called 'message'. It will store an address.

Allocate the bytes required to store text

37

Hello World – x86, Linux

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"

.text
.global _start

_start:
    lea   rax, message
    call  PrintString

    call  Exit
```

Text / Code Section

38

Text / Code Section

```
.text
.global _start

_start:
    lea   rax, message
    call  PrintString

    call  Exit
```

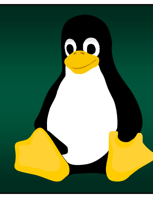
Start text section

Make label visible to the linker. Header will call _start

Loads the Effective Address 'message' into rbx

Call the library subroutine (it needs an address in rbx)

39



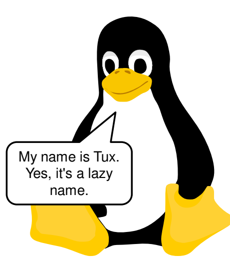
Basics of UNIX

Feel the pow-wah of the dark side

40

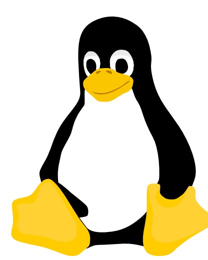
Basics UNIX

- UNIX was developed at AT&T's Bell Labs in 1969
- Design goals:
  - operating system for mainframes
  - stable and powerful
  - but not exactly easy to use – GUI hadn't been invented yet



41

Basics UNIX



- There are versions of UNIX with a nice graphical user interface
- A good example is all the various versions of Linux
- However, all you need is a command line interface

42

Command Line Interface

- Command line interface is text-only
- But, you can perform all the same functions you can with a graphical user interface
- This is how computer scientists have traditionally used computers

```
>gcc hello.c
>ls
a.out hello.c

>a.out
Hello world!
```


43

Command Line Interface

- Each command starts with a name followed by zero or more arguments
- Using these, you have the same abilities that you do in Windows/Mac

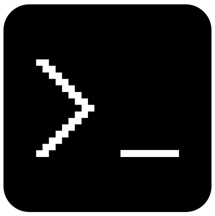
Spaces separate name & arguments

name ***argument1 argument2*** …

44

`ls` Command

- Short for *List*
- Lists all the files in the current directory
- It has arguments that control how the list will look
- Notation:
  - directory names have a slash suffix
  - programs have an asterisk suffix



45

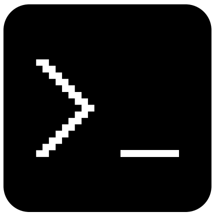
`ls` Command

```
> ls
a.out*  csc35/  html/  mail/
test.asm
```


46

`ll` Command

- Short for *List Long*
- This command is a shortcut notation for **`ls -l`**
- Besides the filename, its size, access rights, etc… are displayed



47

`ll` Command

```
> ll

-rwx------  1 cookd othcsc  4650 Sep 10 17:44 a.out*
drwx------  2 cookd othcsc  4096 Sep  5 17:49 csc35/
drwxrwxrwx 10 cookd othcsc  4096 Sep  6 11:04 html/
drwxrwxrwx  2 cookd othcsc  4096 Jun 20 17:58 mail/
-rw-------  1 cookd othcsc    74 Sep 10 17:44 test.asm
```


48

rm Command

- Short for *Remove*
- It essentially deletes a file
- Be careful…
  - files don't go into a "recycle bin"
  - they are gone forever!
- It can also delete multiple files using patterns

49

rm Command

```
> ls
a.out*  html/  mail/  test.asm

> rm a.out

> ls
html/  mail/  test.asm
```


50

nano Application

- Nano is the UNIX text editor (well, the best one – that is)
- It is very similar to Windows Notepad – but can be used on a terminal
- You will use this to write your programs

51

nano Application

- Nano will open and edit the filename provided
- If the file doesn't exist, it will create it

```
nano filename
```


52

as Command

- This is the GNU assembler
- It will take an assembly program and convert it into an object
- You will be alerted of any syntax errors or unrecognized mnemonics (typos)

53

as Command

- The **–o** specifies the next name listed is the output file
- So, the second is the output file (object)
- The third is your input (assembly)

```
as –o lab.o lab.asm
```


54

as Command

- <u>Be very careful</u> – anything after –o will be destroyed
- There is no "undo" in UNIX!
- Check the two extensions for "o" <u>then</u> "asm"

```
as –o lab.o lab.asm
```


55

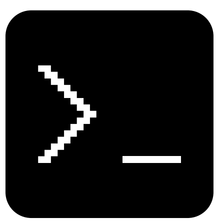
as Command

```
> ls
lab.asm

> as –o lab.o lab.asm

> ls
lab.asm  lab.o
```


56

ld Command

- This is the GNU linker
- It will take one (or more) objects and link them into an executable
- You will be alerted of any unresolved labels



57

ld Command

- The **–o** specifies the next name is the output
- The second is the <u>output</u> file (executable)
- The third is your input objects (1 or more)

```
ld –o a.out csc35.o lab.o
```


58

ld Command

- <u>Be very careful</u> – if you list your input file (an object) first, it will be destroyed
- I will provide the "csc35.o" file

```
ld –o a.out csc35.o lab.o
```


59

ld Command

```
> ls
lab.o  csc35.o

> ld –o a.out lab.o csc35.o

> ls
lab.o  csc35.o  a.out*
```


60

alpine Application

- Alpine is an e-mail application
- Has an easy-to-use interface similar to Nano
- You will use this software to submit your work

61

alpine Application

- To run Alpine, just type its name at the command line
- There are no arguments
- You will have to login (again)

```
alpine
```


62

pwd Command

- Short for *Print Working Directory*
- It displays the path your current directory (the one you are looking at).
- Slashes separate the directory names

63

pwd Command

```
> pwd

/gaia/class/student/cookd
```


64

cd Command

- Short for *Change Directory*
- Allows you to change your current working directory
- If you specify a folder name, you will move into it
- If you use .. (two dots), you will go to the parent folder

65

cd Command

```
> cd csc35

> cd ..
```

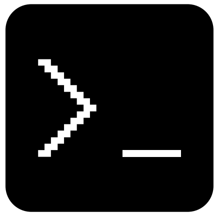
Move into csc35 folder

Return to parent folder

66

`mkdir` Command

- Short for *Make Directory*
- Essentially the same as making a new subfolder in Windows or Mac-OS
- You may want to create one to store your CSc 35 work

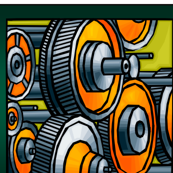


67

`mkdir` Command

```
> ls
a.out*  html/  mail/  test.asm

> mkdir csc35

> ls
a.out*  csc35/  html/  mail/  test.asm
```


68



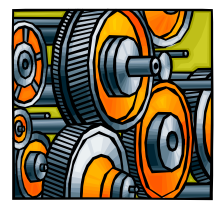
Machine Language

The raw bytes of your program

69

Machine Language

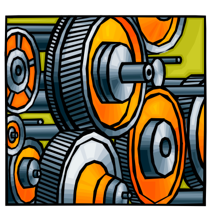
- The instructions, that are *actually* executed on the processor, are just bytes
- In this raw binary form, instructions are stored in *Machine Language (aka Machine Code)*



70

Machine Language

- Each instruction is *encoded* (stored) is in a compact binary form
- Easy for the processor to interpret and execute
- Some instructions may take more bytes than others – not all are equal in complexity



71

Instruction Encoding

- Each instruction must contain <u>everything</u> the processor needs to know to do something
- Think of them as functions in Java: they need a name and arguments to work

72

Instruction Encoding

- For example: if you want it to add 2 things…
- The instruction needs:
  - something to tell the processor to add
  - something to identify the two "things"
  - destination to save the result

73

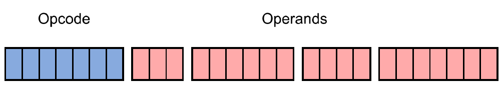
Operation Codes

- Each instruction has a <u>unique</u> *operation code (Opcode)*
- This is a value that specifies the <u>exact</u> operation to be performed by the processor
- Assemblers use friendly names called *mnemonics*

74

Typical Instruction Format

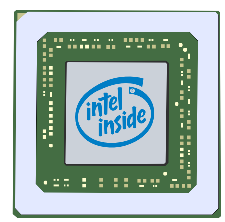
- The opcode is, typically, followed by various *operands* – what data is to be used
- These can be register codes, addressing data, literal values, etc…



75

Intel x64 Encoding

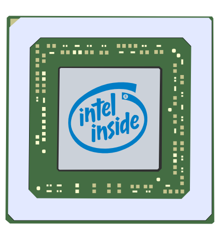
- The Encoding of the Intel x64 Processor is complex
- …and it is very, very difficult to encode
- So, we will practice encoding using a different processor



76

Intel x64 Encoding

- *But… don't worry…*
- We will cover the Intel x64 encoding later in the semester



77

Herky 6000 Processor

- The Herky 6000 is a simple processor that *mirrors the behavior* of the Intel x64
- … but is very easy to encode
- We will practice on it

78

Herky 6000 Specs

- Each instruction is 24-bit (3 byte)
- 16 general purpose registers (we can use Intel names)
- All the major addressing modes

79

Herky 6000 Specs

- Most instruction fields line up cleanly on each nibble
- So, each hex digit is a field
- With a bit of practice, you can read the machine code.

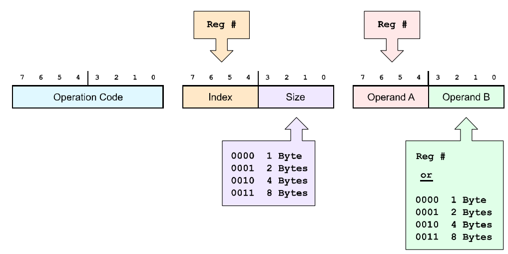
80

Herky 6000 Instruction Format

- First Byte → Opcode
  - unique for every instruction
  - you have to look these up
- Second Byte → Addressing
- Third Byte → Operands
  - Operand B contain register #
  - … or a immediate byte count

81

Herky 6000 Instruction Format



82

Very Basic Herky Modes

Mode	Shorthand Notation
Register Unary	`reg`
Immediate Unary	`imm`
Register, Register	`reg, reg`
Register, Immediate	`reg, imm`

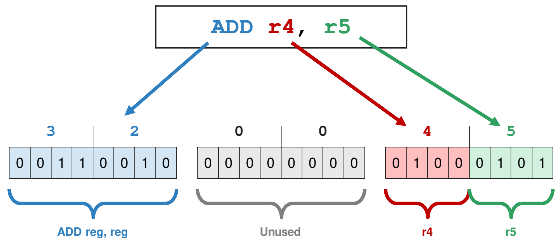
83

Herky Equivalent Registers

Intel	Herky	Intel	Herky
rax	r0	r8	r8
rbx	r1	r9	r9
rcx	r2	r10	r10
rdx	r3	r11	r11
rsi	r4	r12	r12
rdi	r5	r13	r13
	r6	r14	r14
	r7	r15	r15

84

Herky Machine Code Example



85

Herky 6000 Specs

- Sometimes an instruction needs to store an immediate
- But, how many bytes is it?
- The Herky Processor the second operand to store the byte count

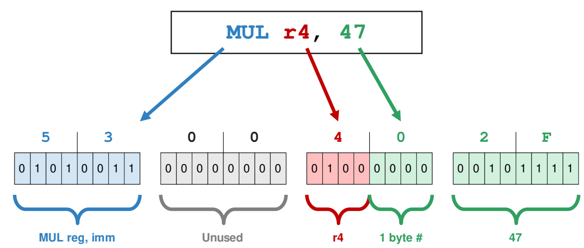
86

Immediate Byte Size

Code	Value	Size of Immediate (2^n)
0000	0	1 byte (8-bit)
0001	1	2 bytes (16-bit)
0010	2	4 bytes (32-bit)
0011	3	8 bytes (64-bit)
0100	4	16 bytes (128-bit)

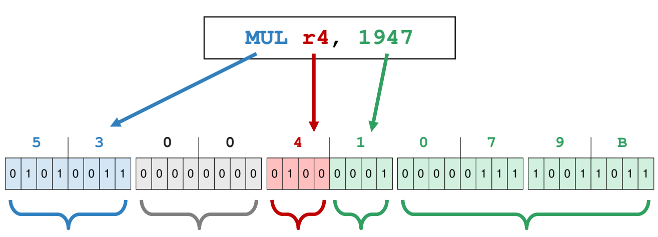
87

Herky Immediate Example



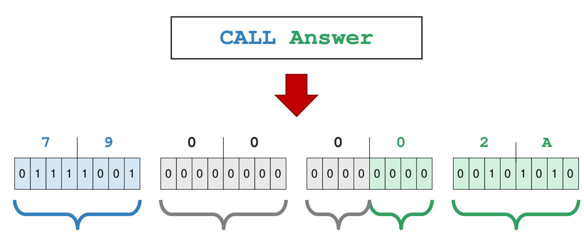
88

Herky 2-byte Immediate Example



89

Herky Call Example (Answer = 42)



90

Encoding Example

LDR reg, imm → 1011 0011

```
mov   rax, 47        B3 00 00 2F
mov   rcx, 1900      B3 00 21 07 6C
add   rax, rcx       32 00 02
```

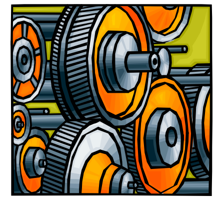
076C = 1900. Two bytes are needed

ADD reg, reg → 0011 0100

91

How Assemblers Work

- Assemblers count bytes as data and instructions are created
- These numbers of often saved and used later by the linker and the program itself



92

Starting at 0000

Current address

```
0000   mov rax, 47      B3 00 00 2F
0004   mov rcx, 1900    B3 00 21 07 6C
0009   add rax, rcx     32 00 02
```

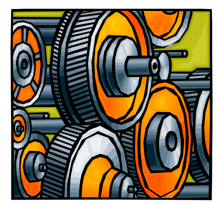
Uses 0000 to 0003

Uses 0004 to 0008

93

How Assemblers Work

- Labels are assigned (whenever defined) to the current byte count
- When referenced later, their addresses are used
- Labels do **not** generate bytes



94

Starts at 2000. PrintString = 079B

```
2000   woof:                  woof = 2000
2000      .ascii "Dog\0"      44 6F 67 00
2004   meow:                  meow = 2004
2004      .ascii "Kitty\0"    4B 69 74 74 79 00
200A   _start:                _start = 200A
200A      lea  rax, meow      E5 00 01 20 04
200F      call PrintString    79 00 01 07 9B
```

Created 4 bytes

Address for "meow" is inserted,

95

Resulting Machine Code

Labels aren't present in executable programs

```
2000   44 6F 67 00
2004   43 61 74 74 79 00
200A   E5 00 01 20 04
200F   79 00 01 07 9B
```

Nor this column

These nice visual line breaks aren't present either

96

A More Accurate View

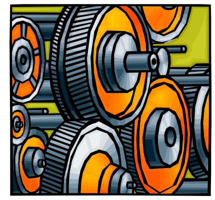
```
44 6F 67 00 43 61 74 74 79 00 E5 00 01 20 04 79 00 01 07 9B
```

Just a series of bytes

97

The Result

- Programs are just a long array of bytes
- Some bytes contain data and others are part of instructions
- This is what a program *truly* is… just a series of bytes



98