Lecture 16

COP3402 FALL 2015 – DR. MATTHEW GERBER – 11/16/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

Assemblers In Operation

Back to the Tiny ISA

Opcode	Hex	Mnemonic	Result
0001	1	LOAD <x>	A ← Mem[x]
0010	2	ADD <x>	A ← A + Mem[x]
0011	3	STORE <x>	Mem[x] ← A
0100	4	SUB <x>	A ← A – Mem[x]
0101	5	IN <Device_#>	A ← Specified Device Input
0110	6	OUT <Device_#>	Specified Device Output ← A
0111	7	HALT	Stop the machine
1000	8	JMP <x>	PC ← x
1001	9	SKIPZ	If Z = 1 skip next instruction
1010	A	SKIPG	If G = 1 skip next instruction
1011	B	SKIPN	If L = 1 skip next instruction

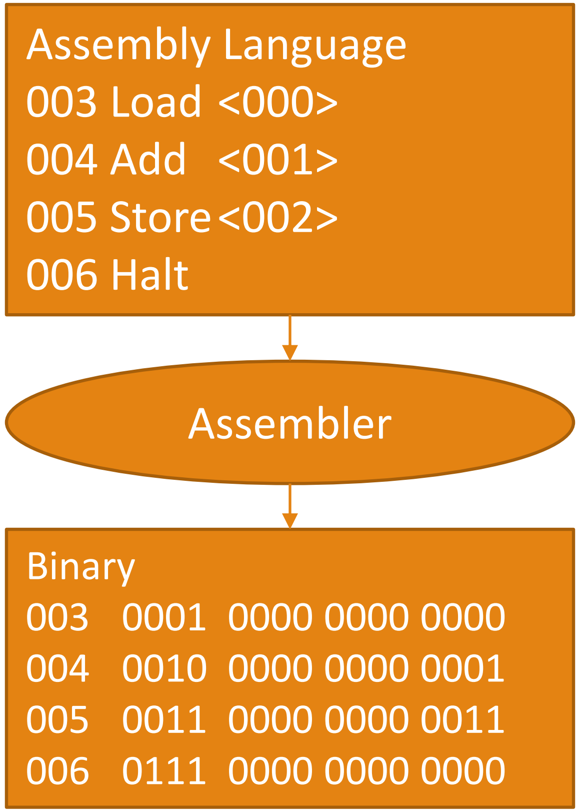
One-Address Architecture

Again, the Tiny Computer is a *one-address* architecture.

- Instructions consist of 16 bits
- 4 bits for the opcode, 12 bits for the address

OP	ADDRESS
0001	**0000 0001 0001**

Opcodes to Binary



The assembler's *main* job is translating opcodes to binary code.

- 1 LOAD
- 2 ADD
- 3 STORE
- 4 SUB
- 5 IN
- 6 OUT
- 7 HALT
- 8 JMP
- 9 SKIPZ
- A SKIPG
- B SKIPN

However, that's not its *only* job…

Programming and Labels

In any programming language, we assign locations to variables.

- Consider the statement:

$$C := X + Y$$

Assume:

- X is assigned to address 1
- Y is assigned to address 0
- C is assigned to address 2

Then…

Before and After

Memory

000 1245

001 1755

002 0000

003 Load <000>

004 Add <001>

005 Store <002>

006 Halt

Memory

000 1245

001 1755

002 3000

003 Load <000>

004 Add <001>

005 Store <002>

006 Halt

Assemblers and Directives

Assembly language isn't any different. Labels are simpler, but we still need them.

We accomplish this as one of four *pseudo-operations*, or assembler *directives*. Directives *do not generate code* – instead, they are instructions to the *assembler* about the code *to* be generated.

- **.begin** tells the assembler where the program starts
- **.data** reserve a memory location
- **.end** tells the assembler where the program ends
- ..and **labels** give names to memory locations.

Let's see how this looks.

Assembly Example 1

Label	opcode	address
start	.begin	
	in	x005
	store	a
	in	x005
	store	b
	load	a
	sub	TWO
	add	b
	out	x009
	halt	
a	.data	0
b	.data	0
TWO	.data	2
	.end	start

Here we see a program that uses all three directives, and three labels.

- By convention, labels to be used as variables are lower case, while labels to be used as constants are upper case
- There is *no* enforcement of constants in assembly language
  - (Or of pretty much anything else)
- The effect of the program is to:
  - Input two numbers from device 5
  - Subtract two from the first number
  - Add the second number
  - Output the result to device 9
  - Halt
- Note that we see separate code and data sections; this is common

Assembly Example 2

	Label	opcode	address
01		; This is	
02		; a comment	
03	start	.begin	x200
04	here	LOAD	sum
05		ADD	a
06		STORE	sum
07		LOAD	b
08		SUB	one
09		STORE	b
0A		SKIPZ	
0B		JMP	here
0C		LOAD	sum
0D		HALT	
0E	sum	.data	x000
0F	a	.data	x005
10	b	.data	x003
11	one	.data	x001
12		.end	start

Here's another one, that's computing a hardcoded 5 x 3.

- You've seen these programs (or ones very much like them) before
- We've already thought about their translation to bytecode
- In other words, we already know what the assembler does
- The new question is, **how?**

Actually, it's not a very new question.

The Two-Pass Assembler

Structurally, **an assembler is just a really simple compiler.**

Since scope and variable types aren't issues, we can divide the two passes very simply.

- Pass 1 builds the symbol table
- Pass 2 generates object code

This is the real reason we don't have you write an assembler in this course

- The two passes of an assembler are just much easier versions of what you are already doing

Assembler Pass 1

	Label	opcode	address	
01		; This is		
02		; a comment		
03	start	.begin	x200	
04	here	LOAD	sum	(x200)
05		ADD	a	(x201)
06		STORE	sum	(x202)
07		LOAD	b	(x203)
08		SUB	one	(x204)
09		STORE	b	(x205)
0A		SKIPZ		(x206)
0B		JMP	here	(x207)
0C		LOAD	sum	(x208)
0D		HALT		(x209)
0E	sum	.data	x000	(x20A)
0F	a	.data	x005	(x20B)
10	b	.data	x003	(x20C)
11	one	.data	x001	(x20D)
12		.end	start	

In pass one the assembler goes through the program line by line to build the symbol table.

- The symbol table is really simple: one entry per label, each one an address
- Note that they're relative to the address we specified in our **.begin** directive

Symbol	Address
sum	x20A
a	x20B
b	x20C
one	x20D

Opcode and Symbol Tables

Opcode	Mnemonic
0001	LOAD
0010	ADD
0011	STOR
0100	SUB
0101	IN
0110	OUT
0111	HALT
1000	JMP
1001	SKIPZ
1010	SKIPG
1011	SKIPN

Symbol	Address
sum	x20A
a	x20B
b	x20C
one	x20D

Now, using both the symbol table and the opcode table, the assembler translates the program to object code.

We assume that the program can be loaded anywhere in memory, so we use relative addressing.

Specifically, we use *PC-relative* addressing!

- (Note: There are much, much better ways of doing relative addressing)

Assembler Pass 2

	Label	Opcode	Address			Object Code
01		; This is				
02		; a comment				
03	start	.begin	x200			
04	here	LOAD	sum		x200	0001 0000 0000 1001 (9 is the offset)
05		ADD	a		x201	0010 0000 0000 1001
06		STORE	sum		x202	0011 0000 0000 0111 (7 is the offset)
07		LOAD	b	o	x203	0001 0000 0000 1000
08		SUB	one	f	x204	0100 0000 0000 1000
09		STORE	b	f	x205	0011 0000 0000 0110
0A		SKIPZ		s	x206	1001 0000 0000 0000
0B		JMP	here	e	x207	1000 1111 1111 1000 (-7 – one's complement numbering)
0C		LOAD	sum	t	x208	0001 0000 0000 0001
0D		HALT		s	x209	0111 0000 0000 0000
0E	sum	.data	x000		x20A	0000 0000 0000 0000
0F	a	.data	x005		x20B	0000 0000 0000 0101
10	b	.data	x003		x20C	0000 0000 0000 0011
11	one	.data	x001		x20D	0000 0000 0000 0001
12		.end	start		x20E	

When looking at the offsets, keep in mind that the PC is always pointing at the *next* instruction.

Object Code

A typical object code file has several sections:

- Header section: Size of code, name source file, size of data
- Text section: Binary Object code
  - Yes, it's called the "text" section even though it's in binary
- Data section: Binary data
- Relocation information section: Addresses to be edited by the linker
- Symbol table section: Global and imported symbols
  - Object files generally have to include a symbol table, because otherwise they can't be usefully linked with other object files
  - This means that object files intended for the same environment need to agree on a lot of things like parameter passing
- Debugging section (optional): Source file and line number information, description of data structures

Object Code for the Example

Program name: start
Starting address text: x200
Length of text in bytes: x014
Starting address data: x20A
Length of data in bytes: x008

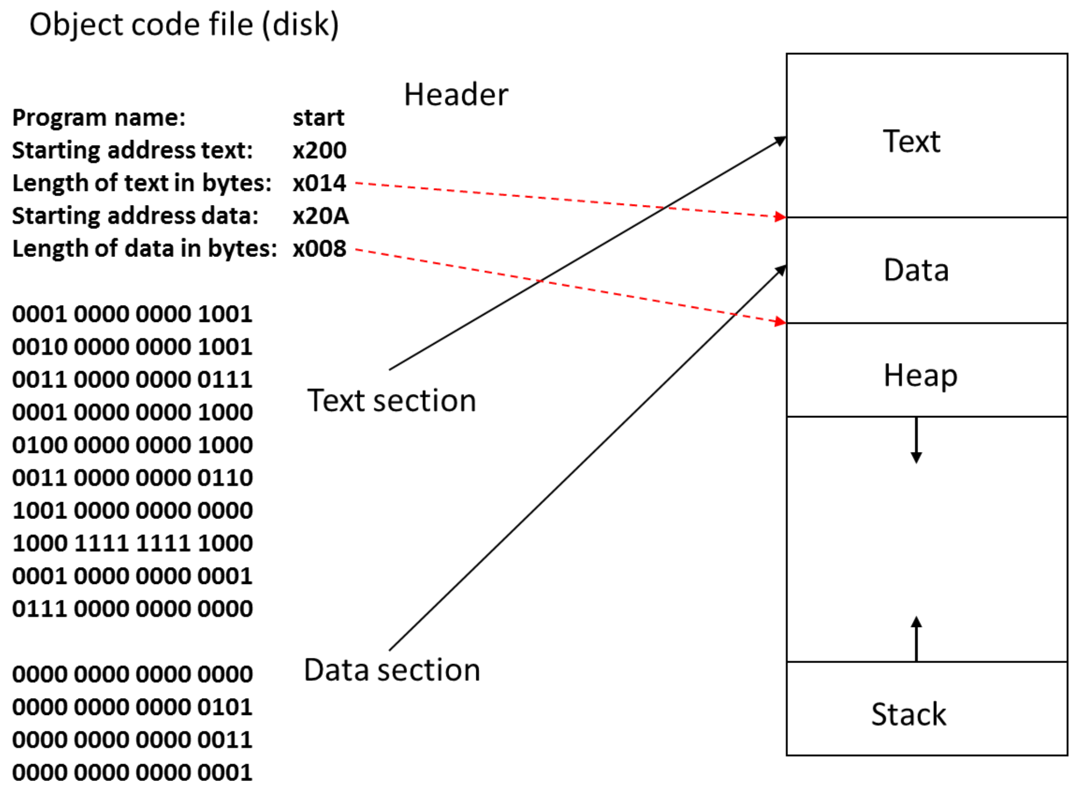
0001 0000 0000 1001
0010 0000 0000 1001
0011 0000 0000 0111
0001 0000 0000 1000
0100 0000 0000 1000
0011 0000 0000 0110
1001 0000 0000 0000
1000 1111 1111 1000
0001 0000 0000 0001
0111 0000 0000 0000

0000 0000 0000 0000
0000 0000 0000 0101
0000 0000 0000 0011
0000 0000 0000 0001

Here's what the object code *might* look like for our example program.

- We see here the header, code and data sections

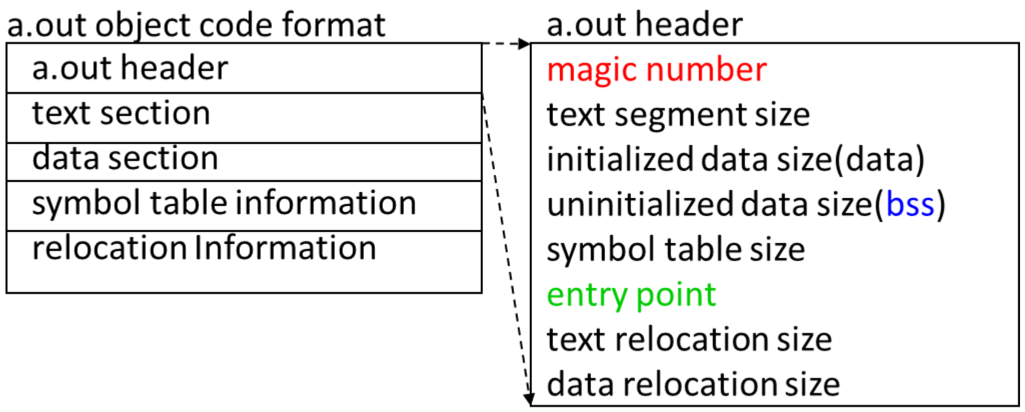
Loading Object Code



Here's what a *very simplified* version of loading that code into memory might look like.

- Our program doesn't *use* a heap or stack, but that doesn't mean it doesn't get access to one
- The length of the text and data segments are used to set their boundaries when the program is loaded
- We're skipping the whole link step here, but we want you to get the simple version of this idea

UNIX a.out format (that’s “assembler output”)



Here’s what a real (albeit old) executable file looks like!

- This is an executable file format instead of an object code file format – not *quite* the same thing
- You may notice it still looks really, really, really familiar

Next Time:
Loaders