

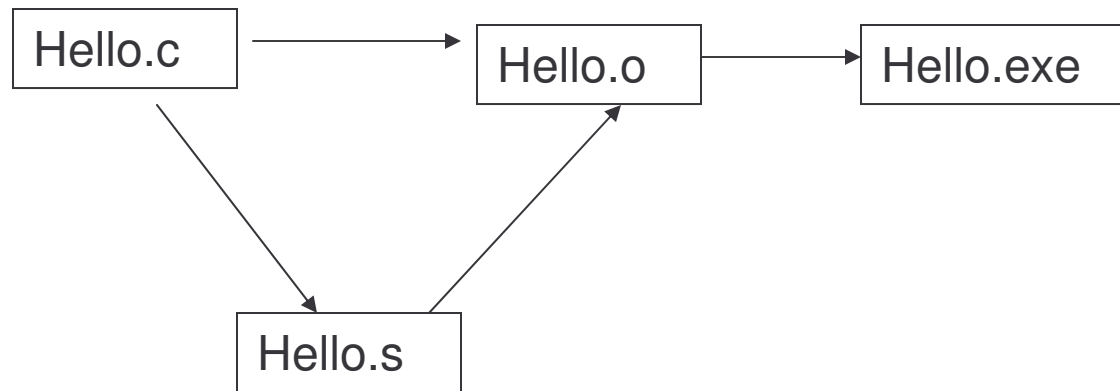
My Assembler

Matt Mags

What's an Assembler?

- An Assembler is a program that performs intermediate steps of compilation between the compiler and the linker to produce executable code.
- It also converts Assembly mnemonics directly into binary code

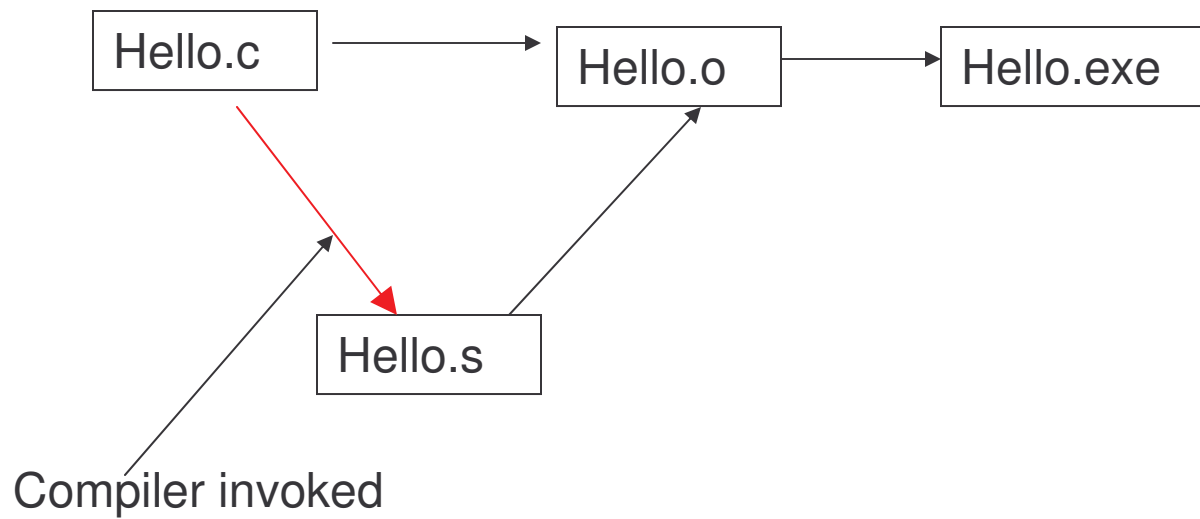
Process of Hello.c



Hello.c

```
int main()  
{  
    printf("Hello World");  
}
```

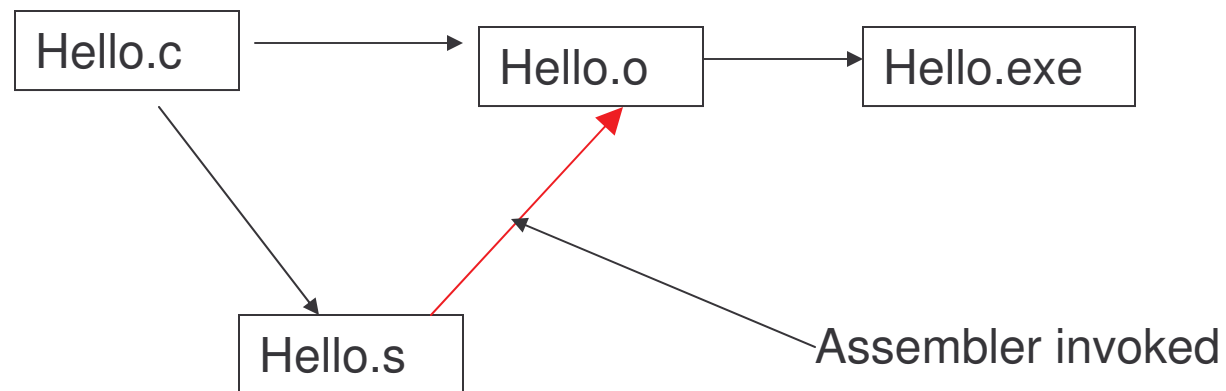
Steps in Compiling Hello.c



Hello.S

```
.file      "Hello.c"
.def      __main;      .scl  2;      .type  32;      .endif
.text
LC0:
.ascii   "Hello World\0"
.globl  _main
.def    _main;  .scl  2;      .type  32;      .endif
_main:
    pushl      %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    call __alloca
    call __main
    subl $12, %esp
    pushl      $LC0
    call _printf
    addl $16, %esp
    leave
    ret
.def    _printf;      .scl  2;      .type  32;      .endif
```

Steps in compiling Hello.c



Hello.o

```
4C 01 03 00 00 00 00 00 00 F4 00 00 00 0D 00 00 00 00 00 04 00 2E 74 65 78 74 00 00 00 00 00 00 00
00 00 00 00 40 00 00 00 8C 00 00 00 CC 00 00 00 00 00 00 00 04 00 00 00 20 00 00 60 2E 64 61 74
61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40 00 00 C0 2E 62 73 73 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 80 00 00 C0 48 65 6C 6C 6F 20 57 6F 72 6C 64 00 55 89 E5 83 EC 08 83 E4
F0 B8 00 00 00 00 89 45 FC 8B 45 FC E8 00 00 00 00 E8 00 00 00 00 83 EC 0C 68 00 00 00 00 E8 00
00 00 00 83 C4 10 C9 C3 90 90 90 90 21 00 00 00 0B 00 00 00 14 00 26 00 00 00 09 00 00 00 14 00
2E 00 00 00 03 00 00 00 06 00 33 00 00 00 0C 00 00 00 14 00 2E 66 69 6C 65 00 00 00 00 00 00 00
FE FF 00 00 67 01 48 65 6C 6C 6F 2E 63 00 00 00 00 00 00 00 00 00 00 00 5F 6D 61 69 6E 00 00 00
0C 00 00 00 01 00 20 00 02 00 2E 74 65 78 74 00 00 00 00 00 00 00 00 01 00 00 00 03 01 3C 00 00 00
04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 64 61 74 61 00 00 00 00 00 00 00 00 02 00 00 00 03 01
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 62 73 73 00 00 00 00 00 00 00 00 00 03 00
00 00 03 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 5F 5F 5F 6D 61 69 6E 00 00 00
00 00 00 00 20 00 02 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 5F 5F 61 6C 6C 6F
63 61 00 00 00 00 00 00 00 00 02 00 5F 70 72 69 6E 74 66 00 00 00 00 00 00 00 00 20 00 02 00 04 00
00 00
```

What is a .O file?

- MY ASSEMBLER makes an Object file, or non-executable byte code file called *filename.o*, I use the Microsoft Portable Executive Common Object File Format called PE-COFF.
- The linker expects an object file in either COFF or ELF format which will then be converted into a functional program
- Basically my task to produce a file acceptable to the linker and allow it to complete compilation

COFF Structure

- A COFF file has four main structures:
 - File Header
 - Sections
 - Section Header
 - Section Data
 - Symbol Table

File Header

- Holds information about size of other sections.
- Example fields:
 - Magic Number(2 bytes)
 - Offset of Symbol Table(4 bytes)
 - Length of Symbol Table(4 bytes)



4C 01 03 00 00 00 00 00 00 00 F4 00 00 00 0D 00 00
00 00 00 04 00

The diagram shows a sequence of 16 bytes in a file header. The first two bytes, '4C' and '01', are red and correspond to the 'Magic Number' field. The next four bytes, '03', '00', '00', and '00', are black and correspond to the 'Offset of Symbol Table' field. The next four bytes, 'F4', '00', '00', and '00', are green and correspond to the 'Length of Symbol Table' field. The final four bytes, '0D', '00', '00', and '00', are blue. Arrows from the list above point to these fields: a red arrow from 'Magic Number' to '4C 01', a green arrow from 'Offset of Symbol Table' to '03 00 00 00', and a blue arrow from 'Length of Symbol Table' to 'F4 00 00 00'.

Section Header

- Example Fields:
 - Name of Section (8 bytes null terminated)
 - Length of Section data (4 bytes)
 - Offset of Section data (4 bytes)

The diagram illustrates a section header structure with three fields: Name (8 bytes), Length (4 bytes), and Offset (4 bytes). The hex values are shown in a grid format, with arrows indicating the mapping from the list items to the corresponding bytes in the header.

2E	74	65	78	74	00	00	00	00	00	00	00
00	00	00	00	40	00	00	00	8C	00	00	00
CC	00	00	00	00	00	00	00	04	00	00	00
20	00	00	60								

Arrows indicate the following mappings:

- Red arrow: Name of Section (8 bytes null terminated) - points to the first 8 bytes (2E 74 65 78 74 00 00 00).
- Blue arrow: Length of Section data (4 bytes) - points to the next 4 bytes (00 00 00 00).
- Green arrow: Offset of Section data (4 bytes) - points to the final 4 bytes (8C 00 00 00).

Section Data

- Code and Data blocks to be executed
- Sections important because all offsets are Section relative
- A hexdump of a Section is literally unformatted computer memory

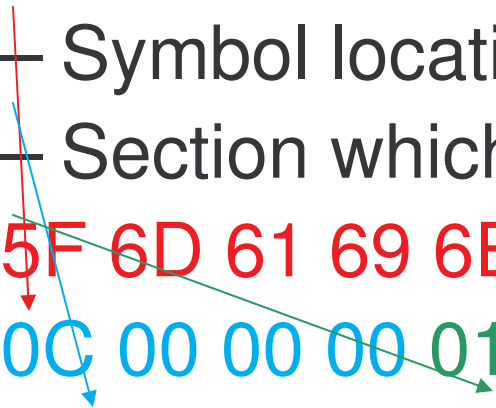
Symbol Table

- Symbols are the names of variables, functions, and constants.
- Example Fields:
 - Symbol name(8 bytes null terminated)
 - Symbol location (4 bytes)
 - Section which contains symbol (2 bytes)

5F 6D 61 69 6E 00 00 00

0C 00 00 00 01 00 20 00

02 00



How do Symbols work?

Symbols in Red

```
.file      "Hello.c"
  .def     __main; .scl      2;          .type     32;          .endif
  .text
LC0:
  .ascii  "Hello World\0"
.globl _main
  .def     _main;   .scl      2;          .type     32;          .endif
_main:
  pushl   %ebp
  movl    %esp, %ebp
  subl   $8, %esp
  andl   $-16, %esp
  movl   $0, %eax
  movl   %eax, -4(%ebp)
  movl   -4(%ebp), %eax
  call   __alloca
  call   __main
  subl   $12, %esp
  pushl  $LC0
  call   _printf
  addl   $16, %esp
  leave
  ret
  .def     _printf; .scl      2;          .type     32;          .endif
```

How do Symbols work?

- The linker accesses the Symbol table and fills in the external references.
- Let's examine the "printf" function: The assembler scans down the file and realizes that printf is not user defined.
- It notes that printf is external and completes
- It is then the linker's responsibility to find printf and add it to the file.

What the linker wants

Assembler
call printf

Linker
call printf

Assembler realizes
it is not defined in
file and lays down

Linker looks through
libraries and finds
printf function. It adds
it to file and supplies
the address

E9 00 00 00 00

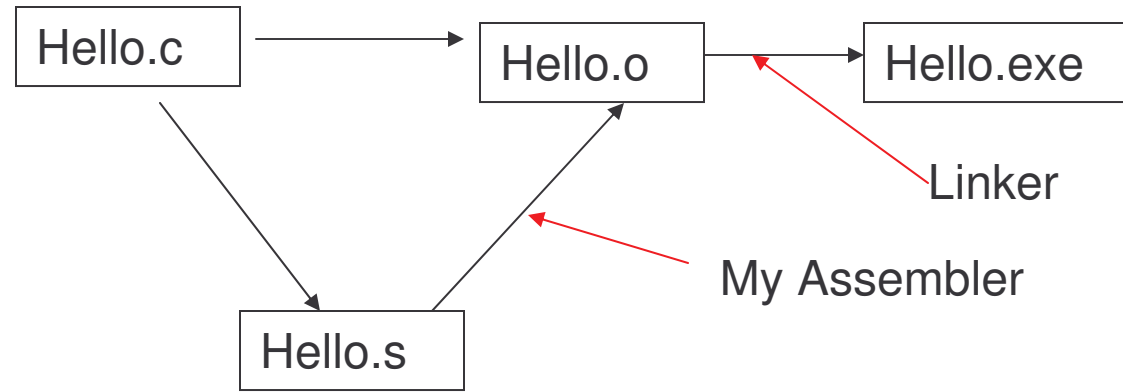
E9 54 00 32 00

Jump Instruction

Jump Address



WHAT I DID



1. I used gcc to produce a .s file or assembler file.
2. I had to be able to handle whatever instructions GCC threw at me and produce a functional COFF or .o file.
3. Target ultimately was to produce z215

Assembler Process

- Assembler's generally work on a multi-pass system:
 1. First they scan the file locating all the symbols
 2. Then they proceed down the file converting each mnemonic to binary code.

How do we Assemble Hello.S?

```
.file      "Hello.c"
.def __main;      .scl  2;      .type  32;      .endif
.text
LC0:
    .ascii "Hello World\0"
.globl _main
    .def _main;  .scl  2;      .type  32;      .endif
_main:
    pushl      %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    call __alloca
    call __main
    subl $12, %esp
    pushl      $LC0
    call _printf
    addl $16, %esp
    leave
    ret
    .def _printf;      .scl  2;      .type  32;      .endif
```

First Pass

- Because a Symbol may be used in a Section before it is declared, we need to know if it will be declared in the file or is external.
- Symbols generally have a .def entry stating whether the symbol is a variable or a function and whether it is a static entry.

Locate all Symbols

```
.file    "Hello.c"
    .def  __main; .scl    2;      .type    32;      .endif
    .text
LC0:
    .ascii "Hello World\0"
.globl _main
    .def  _main;   .scl    2;      .type    32;      .endif
_main:
    pushl %ebp
    movl  %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl  $0, %eax
    movl  %eax, -4(%ebp)
    movl  -4(%ebp), %eax
    call  __alloca
    call  __main
    subl $12, %esp
    pushl $LC0
    call  _printf
    addl $16, %esp
    leave
    ret
    .def  _printf; .scl    2;      .type    32;      .endif
```

Symbols in Red

Sections and Instructions

```
.file "Hello.c"
.def __main; .scl 2; .type 32; .endif
.text
LC0:
  .ascii "Hello World\0"
.globl _main
.def _main; .scl 2; .type 32; .endif
_main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  andl $-16, %esp
  movl $0, %eax
  movl %eax, -4(%ebp)
  movl -4(%ebp), %eax
  call __alloca
  call __main
  subl $12, %esp
  pushl $LC0
  call _printf
  addl $16, %esp
  leave
  ret
.def _printf; .scl 2; .type 32; .endif
```

Sections

- This program has one section. That tells the linker and OS that the code and data in the program must be maintained in a continuous block and if a virtual address offset must be used.

Instructions

- Converting instructions into binary is the most difficult part of the process.
- Intel instructions have variable lengths (anywhere from one byte to 12 bytes in length).
- There are several hundred instructions many of which have a dozen or more variations.

Addressing Modes

- Mov register, register
- Mov register, immediate
- Mov register, offset
- Mov register, address-index
- Mov offset, register
- Mov offset, immediate
- Mov offset, address-index
- Mov immediate, offset
-etc

Problem of numbers

- Must be able to convert numbers of type int ($2^{32}-1$) size into an array of bytes (255 each).
- Programmer creates an Integer which contains the number 600
- That is bigger than a byte, how do we convert it?

Converting Numbers

- We create an array of bytes
- We start with $x = 2^{31}$ the largest number a signed Integer can contain (2^{32} is negative).
- We iterate over bits from left to right
- Go through a loop dividing this x by 2.
- Each time our number is greater than x we set the bit we're working on to 1 else 0

History begins with Intel

- Backward compatibility
- GCC choice of instructions
- Optimization considerations

Most recent versions?

- Can we just use most recent version of instructions?
- No, could alter memory and stack
- My *filename.o* files would not be compatible with many other user programs.

Handling Instruction

- I tried two different approaches to dealing with all of GCC's instructions.
- Though both were functional neither was very satisfactory and had numerous flaws
- This is still a very real problem for professional compilers and assemblers today

First Attempt

- I began with simple pattern matching to create instructions.
- I.E. Just hard code that when you find this exact instruction, set down these exact bytes

• Example:

• sarw \$4, %ax

110011011 000001111 1100000000 000000100



First Attempt

- Pros
 - Simple to implement
 - Easy to maintain
 - Easy to debug
- Cons
 - Takes a long time to add new instructions
 - Small variations on instructions are regarded as completely different such as `mov 4, %ax` is different then `mov 5, %ax`
 - Long search time and prone to interrelated faults

How to build Instructions

- Building instructions:
- Basic idea on a two byte instruction such as `mov %ax, %bx`:
 - First byte is Opcode (i.e. tells processor to perform a mov)
 - Second byte:
 - First two bits refers to what types of operands to expect
 - Next 6 bits refer to operands, 3 bits each. There are 8 General registers (2^3).

Second Attempt

- Pros
- One function can facilitate many variations on an instruction.
- New variations can be added easily
- Low search time
- Cons
- Building design is VERY loose.
- Difficult to isolate and understand pattern in instructions
- Can be very difficult to debug

Instructions get yet more complicated

- MOVs may involve labels and addresses.
 - Mov %eax, _MyLabel (This is a Label)
 - Loads data at MyLabel into %eax
 - Mov %eax, \$Label (This is a string)
 - Loads address of Label into %eax
 - Mov %eax, \$_MyLabel (This is a Label)
 - Loads address of MyLabel into %eax

Address and Label instructions

- Each instruction has a special variation to deal with addresses and labels.
- An instruction must be decoded to see if it contains an address, then that address must be loaded
- This can be harder than it sounds since some labels are used before they are declared or are external and the address is unknown.

Solution

- Labels which are unknown must be entered after file is complete.
- Mark the location which needs to be altered, fill up space with 0's, then replace when address located.
- Program requires a relocatable entry or will crash because data often kept in separate section

Jmps

- Jmps are relative flow control
 - This means rather than say GO_TO 0x6754, say GO_FORWARD 64 bytes
 - Jmps come in variable sizes, 2 bytes or 5.
 - JMP _My_label. We have not yet found _My_label.
 - How far away is label? Can it be represented in two bytes or do we need 5?

Jmp Problems

- By the time we locate the label its jumping to, we have likely gone a good distance down the program.
- Already laid down 2 or 5 bytes of 0's.
- Can't just shift bytes, would throw off all the marks "to be filled in" later we've made.
- Solution: convert all addresses and treat all Jmps as 5 bytes.

Output

- At this point *theoretically* the file is completely scanned and ready to be formatted.
- I created a tool called COFF to print out in plain text the structure of my filename.o
- After this call the gcc linker, LD and run it on my file

Conclusion

- What I learned:
 - Aside from the challenges of constructing and maintaining a program thousands of lines long I developed a good understanding of the Compiling process and the way the OS handles programs.
- What I could do with more time:
 - My z215 program does not yet work satisfactorily and my instruction library still has holes in it. In a few more weeks I could have corrected these flaws.

The End