University of Arizona

CSc 340

Foundations of Computer Systems

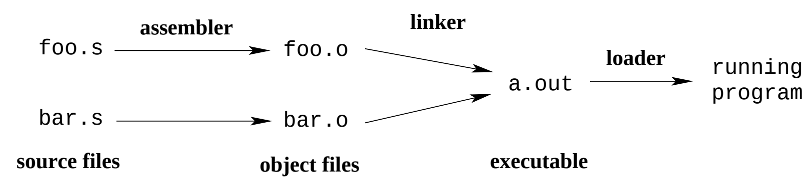
Christian Collberg

April 17, 2001

Translating Assembly Code

assembly→machine→running program

- We've learned about machine code, the binary instructions that a computer understands, and assembly code, the human-readable form of machine code.
- How is assembly code converted into machine code, and ultimately a running program? We need three programs: the assembler, the linker, and the loader.



Slide 20–1

Object file format

- An object file contains the machine code and data for a program, plus other information needed to link and load the program. The machine code and data are stored in three segments:

  text: machine code instructions

  data: initialized data (data that have an initial value)

  bss: uninitialized data (data that have no initial value)

Slide 20–2

Object file format...

- The segment names are historical. The `bss` segment holds data that is not statically initialized in the program, and is represented in the object file by the amount of memory needed to hold it.
- The other two segments contain information that must be copied from the file into memory before the program runs.
- An object file also contains a symbol table and a patch list. These will be explained below.

Slide 20–3

Assembler

The assembler converts one source (assembly code) file into one object file. It performs the following functions:

1. Translate assembly instructions into machine code instructions. This involves setting the bits properly in the machine code.
2. Implement synthetic instructions using one or more machine instructions.
3. Evaluate constant expressions. ⌜`addu $sp, $sp, 24+8`⌝ becomes ⌜ `addu $sp, $sp, 32`⌝.

Assembler

4. Organize data and text in memory. This is difficult because a source file can have multiple data and text segments: the assembler might not know the final address for a label when it is translating an instruction that uses the label.
5. Labels are a form of symbol – a symbolic name for an address.

Assembler – Algorithm

```
void assembler(file) {
  dotOFile = createFile();
  syTab = new SymbolTable();
  PC = 0;
  for line = each line in file do {
    if hasLabel(line) then
      syTab.insert(getLabel(line), PC)
    instr = translate(line, SyTab, PC);
    if instr ≠ NULL then {
      write(dotOFile, instr);
      PC += sizeOf(instr);
    }
  }
}
```


Assembler—Forward References

```
loop:   beq    $t0, 4, done
        add    $t0, $t0, 1
        b      loop
done:
```

- This program uses symbols (labels). Not knowing the value of `done`, the assembler can't translate the `bge` instruction.
- The assembler must defer translating this instruction until the address of `done` has been determined.
- There are two ways of solving the problem, both involving first figuring out the symbol addresses, then using them to generate the machine code.

Two-pass assembler

The first solution is to make two passes over the source file. This is called a two-pass assembler.

Pass 1: Compute segment sizes and starting addresses, and fill in the symbol table. The symbol table contains the name of each symbol, and its address. When the assembler sees a symbol defined (such as loop in the first instruction above) it adds it to the symbol table. The symbol addresses are left unknown. At the end of the first pass the size of the segments are known, and the symbol addresses are computed and stored in the symbol table.

Two-pass assembler...

Pass 2: Translate instructions. If an instruction uses a symbol, find the symbol in the symbol table and use its address during the translation.

- This works, but requires reading the source file twice, which can be slow.

Two-pass – Algorithm

```
void assembler(file) {
   syTab = passOne(file);
   rewind(file);
   passTwo(file, syTab);
}

SymbolTable passOne (file) {
   syTab = new SymbolTable();
   PC = 0;
   for line = each line in file do {
      if hasLabel(line) then
         syTab.insert(getLabel(line), PC)
         PC += bytesNeeded(line);
      }
   }
   return syTab;
}
```


Two-pass – Algorithm

```
void assembler(file) {
   syTab = passOne(file);
   rewind(file);
   passTwo(file, syTab);
}

void passTwo(syTab, file) {
   dotOFile = createFile();
   PC = 0;
   for line = each line in file do {
      instr = translate(line, syTab, PC);
      if instr ≠ NULL then {
         write(dotOFile, instr);
         PC += sizeOf(instr);
      }
   }
}
```

Slide 20–12

Patch List

- Another approach is to make one pass over the source file, partially translating the instructions, but make a second pass over the object file to fix (patch) those instructions that use symbols. A patch list is used to keep track of which instructions need to be patched, and which symbol(s) each uses.

Slide 20–13

Patch List...

Phase 1: Read entire source file, fill in symbol table, translate instructions (leave unknown addresses blank), and generate patch list. At the end of phase 1 the sizes and starting addresses of the segments are known, and the symbol addresses are computed and stored in the symbol table.

Phase 2: Process patch list, and patch each instruction using the address of the appropriate symbol(s).

Slide 20–14

Patch List – Algorithm

```
void assembler(file) {
   dotOFile = createFile();
   (patchList,syTab) = passOne(file, dotOFile);
   passTwo(dotOFile, patchList, syTab);
}

void passTwo(dotOFile, patchList, syTab) {
   for patch = each element on patchList do {
      applyPatch(dotOFile, patch, syTab);
   }
}
```

Slide 20–15

```
(PatchList, SymbolTable)
passOne(file, dotOFile) {
   syTab = new SymbolTable();
   PC = 0;
   patchList = new List();
   for line = each line in file do {
      if hasLabel(line) then
         syTab.insert(getLabel(line), PC)
      instr = translate(line, SyTab, PC);
      if instr ≠ NULL then {
         if incomplete(line) then {
            patch = new Patch(instr, PC);
            patchList.add(patch);
         }
         write(dotOFile, instr);
         PC += sizeOf(instr);
      }
   }
   return (patchList, syTab);
}
```

Slide 20–16

Patch List: Caveats

1. Symbol addresses aren't known until the end of phase 1. Store each symbol address in the symbol table as an offset from the beginning (base address) of its segment. During phase 2, add the offset to the segment base to get the symbol address.

Slide 20–17

2. Different instructions have different encodings, which means that patching is instruction-dependent:
   (a) ⌜`lw $t0, foo`⌝ is translated into 0x8C080000 in the first pass. The last four zeros (16 bits) should be patched with the address of foo when the patch list is processed.
   (b) ⌜`jal foo`⌝ is translated into 0x0C000000 in the first pass. The last six zeros plus two bits of the 'C' (26 bits total) should be patched with the word address of `foo`. The low 2 bits of `foo`'s address will always be zero (because instructions must be word-aligned), so leave them off.

Slide 20–18

3. Either:
   (a) Read the instruction to be patched from the object file, parse it, and patch it depending on what instruction it is, or
   (b) Store the type of instruction in the patch list to avoid re-parsing the instruction, or
   (c) Store generic patching information in the patch list (e.g. put the address into bits 0-15 of the instruction). This solution doesn't depend on the instruction encoding or instruction set – it can easily be ported to a different architecture.

Slide 20–19

Branch instructions

- Branch instructions are PC-relative, they modify the PC by a relative amount, rather than set it to a fixed address.
- This is a good thing, because it means a branch to an instruction in the same segment works even if the segment's starting address changes (because it's relative).
- Once PC-relative instructions have been patched, they don't need to be re-patched if the segment is relocated in memory.

Slide 20–20

Linker

The linker combines one or more object (`.o`) files into a single executable (`a.out`). To do this it must:

1. Combine like segments. All text segments from the object files are combined into one big text segment, and all data segments are combined into one big data segment.
2. Segment starting addresses will change, so instructions will have to be re-patched. The symbol tables and patch lists stored in the object files are used for this.

Slide 20–21

Linker

3. Resolve external references. If one object file has an undefined reference to a symbol, the linker searches the global symbols defined in the other object files to find a match and patch the instruction. Otherwise an error is reported.
4. Link in object files from specified libraries. A library is a collection of object files that are indexed by the global symbols they define. If an undefined reference is found in the index, the linker extracts the proper object file from the library and links it into the executable.

Slide 20–22

Linker. . .

When patching instructions the linker must be careful to use the correct symbol table, as several object files might define and use the same symbol. There are two types of symbols:

Private: a private symbol is defined and used in a source file. It cannot be referenced by another source file.

Global: a global symbol is defined in one source file, but may be referenced by any source file.

Slide 20–23

Linker. . .

Global definition: the definition of a global symbol. In MIPS assembly, the `.globl` directive indicates that the symbol is global. Symbols without this directive are private

External reference: a reference to a global symbol that is not defined in the current source file.

Loader

- The loader is responsible for taking an executable (`a.out`) and turning it into a running program.
- The `a.out` was linked assuming that the text segment starts at address 0, but what if there is already a program running at address 0? The new program will clearly have to start at a different address, meaning that many of its instructions have been patched with incorrect addresses and need to be patched again.

Loader...

- The loader using the symbol tables and patch lists to re-patch the instructions. Once this is done, the text and data segments are copied into memory, and the loader starts the program running at the executable's starting address.

An Example

- The following example converts an upper-case string to lower-case and prints it. The file `main.s` has two text segments and two data segments and calls the routine `print` which is defined in the file `print.s`.
- The notation `hi(symbol)` means the upper 16 bits of the symbol's address.
- `lo(symbol)` means the lower 16 bits.
- `xspim` doesn't have syntax to express this.

main.s

```
          .data
string:   .asciiz"HELLO"
          .text
main:
     subu $sp, $sp, 24
     sw   $fp, 0($sp)
     addu $fp, $sp, 24
     sw   $ra, -20($fp)
     lui  $a0, hi(string) # la $a0,string
     ori  $a0, $a0, lo(string)
     jal  tolower
     move $a0, $v0
     jal  print
     lw   $ra, -20($fp)
     lw   $fp, -24($fp)
     addu $sp, $sp, 24
     jr   $ra
```

```
        .data
        .align          2
offset: .word   0x20
        .text
tolower:
1    subu  $sp, $sp, 24
2    sw    $fp, 0($sp)
3    addu  $fp, $sp, 24
4    lui   $at, hi(offset)  # lw $t1, offset
5    lw    $t1, lo(offset)($at)
6    lbu   $t0, 0($a0)
7    beqz  $t0, done
loop:
8    addu  $t0, $t0, $t1
9    sb    $t0, 0($a0)
10   addu  $a0, $a0, 1
11   lbu   $t0, 0($a0)
12   bnez  $t0, loop
done:
13   lw    $fp, -24($fp)
14   addu  $sp, $sp, 24
15   jr    $ra
```

Slide 20–28

Assemble main.s

- First assemble `main.s` into `main.o`. The assembler processes `main.s` one line at a time, and creates a symbol table and patch list.

`main.s` Symbol Table

Symbol	**Address**	**Type**
`string`	data+0	Private
`main`	text+0	Private
`tolower`	text+52	Private
`print`	undefined	External ref
`offset`	data+8	Private
`loop`	text+80	Private
`done`	text+100	Private

Slide 20–29

Assemble main.s...

- The value of offset is `data+8` because the string `string` is stored at the beginning of the data segment. `string` has 6 characters (don't forget the 0!), but offset must be word-aligned.
- After all lines in `main.s` have been processed, the size of the data segment is known to be 12 bytes, and the text segment 112 bytes (28 instructions).

`main.s` Segment Info

Segment	**Base Addr.**	**Size**
Text	0x0	0x70 (112)
Data	0x70	0xc (12)

Slide 20–30

Assemble main.s...

`main.s` Patch List

Instruction address	**Symbol used**
text+16	`string`
text+20	`string`
text+24	`tolower`
text+32	`print`
text+64	`offset`
text+68	`offset`
text+76*	`done`
text+96*	`loop`

Slide 20–31

Assemble main.s. . .

- The instructions tagged with '*' are branch instructions that are patched with the number of instructions difference between the instruction's address and the symbol's address. E.g., to branch backwards one instruction the branch is patched with -1.
- A branch can be patched once the offset is known, even if the final addresses aren't. In the case of the `bnez` loop the instruction can be patched in the first pass because loop is at offset 80 and the branch instruction at offset 96. The difference is -16 bytes, or -4 instructions.

Applying patches

Address patched	Symbol used	Symbol address	Comments
0x10 (text+16)	`string`	0x70 (data+0)	upper 16 bits
0x14 (text+20)	`string`	0x70 (data+0)	lower 16 bits
0x18 (text+24)	`tolower`	0x34 (text+52)	bits 27-2
0x20 (text+32)	`print`	undefined	external ref.
0x40 (text+64)	`offset`	0x78 (data+8)	upper 16 bits
0x44 (text+68)	`offset`	0x78 (data+8)	lower 16 bits
0x4C (text+76)	`done`	0x64 (text+100)	+6
0x60 (text+96)	`loop`	0x50 (text+80)	-4

Applying patches. . .

Address	Before	After
0x10	0x3C040000	0x3C040000
0x14	0x34840000	0x34840070
0x18	0x0c000000	0x0C00000D
0x20	0x0c000000	0x0C000000
0x40	0x3C010000	0x3C010000
0x44	0x8C290000	0x8C290078
0x4c	0x11000000	0x11000006
0x60	0x15000000	0x1500FFFC

Assemble print.s

- Now do the same for `print.s`, containing the print subroutine.

```
        .data
newline:.asciiz"\n"
        .text
        .globlprint
print:
    subu    $sp, $sp, 24
    sw      $fp, 0($sp)
    addu    $fp, $sp, 24
    li      $v0, 4
    syscall
    lui     $a0, hi(newline)
    addu    $a0, $a0, lo(newline)
    syscall
    lw      $fp, -24($fp)
    addu    $sp, $sp, 24
    jr      $ra
```

Slide 20–36

Assemble print.s...

print.s Symbol Table		
Symbol	**Address**	**Type**
`newline`	data+0	Private
`print`	text+0	Global

print.s Segment Info		
Segment	**Base Address**	**Size**
Text	0x0	0x2C (44)
Data	0x2C	0x2 (2)

Slide 20–37

`print.s` Patch List	
Instruction address	**Symbol used**
text+20	`newline`
text+24	`newline`

Applying patches			
Address patched	**Symbol used**	**Symbol address**	**Comments**
0x14 (text+20)	`newline`	0x2C (data+0)	high 16 bits
0x18 (text+24)	`newline`	0x2C (data+0)	low 16 bits

Slide 20–38

Linking

- The assembly process produces two object (`.o`) files, `main.o` and `print.o`. Each contains a text segment with the machine instructions, data segment, symbol table, and patch list.
- The linker combines the object files into an executable. It opens both files and computes the sizes and starting addresses of the combined addresses.

Slide 20–39

Linking...

- Assume that the `main.s` text segment starts at address 0, followed by the `print.s` text segment, followed by the `main.s` data segment, and finally the `print.s` data segment:

`print.s` data segment		
Segment	**Starting Address**	**Shorthand**
main.s text	0x0	`mtext`
print.s text	0x70	`ptext`
main.s data	0x9C	`mdata`
print.s data	0xA8	`pdata`

Slide 20–40

Linking. . .

- Combining the segments has caused the addresses in three of them to change. I use the shorthand names `mtext`, `mtext`, `mdata` and `pdata` to represent starting addresses of the segments. The patch lists are used to patch the instructions.
- Note that the instruction at address 0x20 is the call to `print` that the assembler couldn't patch. The linker uses the global definition of `print` in `print.o` to resolve the reference.

Slide 20–41

Linking. . .

Patching instructions from `main.o`:		
Address patched	**Symbol used**	**Symbol address**
0x10 (mtext+16)	`string`	0x9C (mdata+0)
0x14 (mtext+20)	`string`	0x9C (mdata+0)
0x18 (mtext+24)	`tolower`	0x34 (mtext+52)
0x20 (mtext+32)	`print`	0x70 (ptext+0)
0x40 (mtext+64)	`offset`	0xA4 (mdata+8)
0x44 (mtext+68)	`offset`	0xA4 (mdata+8)
0x60 (mtext+96)	`loop`	0x50 (mtext+80)

Slide 20–42

Linking. . .

- Also note that for private symbols, such as `done`, the linker uses the definition from the symbol table for the file it is patching.

Patching instructions from `print.o`		
Address patched	**Symbol used**	**Symbol address**
0x74 (ptext+20)	`newline`	0xA8 (pdata+0)
0x78 (ptext+24)	`newline`	0xA8 (pdata+0)

Slide 20–43

Loading

- The linker assumes the text segment starts at address 0. If the program is indeed loaded (copied) into memory at address 0, it will run correctly.
- Running more than one program at a time, however, requires loading all but one of them at an address that is not zero. Suppose the program is loaded at address 1000, instead of 0. This means that the segments' starting addresses are all off by 1000.

Loading. . .

- The loader will have to use the patch lists and symbol tables to once again patch instructions to use the new starting addresses. The program will then run correctly when copied into memory at address 1000.

Readings and References

- MacCabe, pp. Chapter 10, 351–381.