Warm-Up Problem

- Define, using our shorthand, code(term$_1$ $\rightarrow$ term$_2$ STAR factor)

CS 241 Lecture 17

Code Generation Continued, Loading
With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

lvalue thoughts

- There is no single, perfect solution to the lvalue problem.
- Consider these three, or anything else that makes sense to you.
- Consider *code ordering* (not doing things in a different order than they're written), *simplicity* (not overcomplicating your generation), and *modularity*.

Some Complex code Cases

- What should we generate for
  stmt → PRINTLN LPAREN expr RPAREN SEMI
  ?
- You had to write MIPS code to do this in A2!
- But, that would be a lot of code to generate every time println is used…

Code Reuse

- A compiler mixes the code it outputs with code from a *runtime environment*.

Definition

A *runtime environment* is the execution environment provided to an application or software by the operating system to assist programs in their execution. Such an environment may include things such as procedures, libraries, environment variables and so on.

Runtime Environment

- For example, the standard C library, providing functions such as malloc, free, printf and memcpy, is part of the execution environment of C programs. On Unix and Unix-like systems including Linux, it is provided in libc.so. Archaic, nonstandard operating systems have other conventions; for instance, on Windows, it is provided by msvcrt.dll.
- So, how will we print? Make sure a print function is available in the runtime environment, instead of making the compiler do it!

MERL files

- We will provide you with some MERL files to help you.
  - This stands for MIPS Executable Relocatable Linkable. It's a format for object files (.o files), which is where our runtime environment will come from. The .so in libc.so stands for "shared object": it's just a shared object file!
  - MERL is based on the original Unix format; modern Unix uses ELF, which you'll see again in CS350.

MERL files

- Briefly, when we compile files, almost always what is output is not pure machine code (i.e., it isn’t just binary associated with your code). There is often some additional header information as well. That header + machine code = an object file.
- These files can help us store additional information needed by the loader and linker, which will ultimately let us get our runtime environment.

What This Means For Us Now

- For us, we will provide you with a print.merl file that you will use to link with your assembled output:

```
./wlp4gen < source.wlp4ti > source.asm
cs241.linkasm < source. asm > source. Merl
cs241.merl source.merl print.merl > exec.mips
```

- Then finally, we can call `mips.twoints exec.mips` or `mips.array exec.mips`

A Note on “Real” Compilers

- gcc and clang aren’t compilers! They’re *compiler drivers*.
- A compiler driver is a program that calls other programs to compile, assemble, and link code. The compiler driver isn’t sophisticated, but is very platform-specific, since it needs to know what (shared) object files constitute the runtime environment.

A Note on “Real” Compilers

- In the case of gcc:
  - The compiler: cc1 (cc1plus for g++)
  - The assembler: as or gas
  - The linker: ld or gold
  - The runtime environment: libc.so on most systems (+ libstdc++.so for C++)
- You can see all these steps if you run your compiler with -v

Using print

- To use print, we need to add `.import print` to the beginning of our file.
- After this, we can use `print` in our MIPS code. It will print the contents of $1. Be mindful that you may need to save $1 depending on what you want to print.
- Note also that `print` will overwrite the data in $31. We will need to save and restore it.

Code for println

```
code(println(expr);)= push($1)
                     + code(expr)
                     + add $1, $3, $0
                     + push($31)
                     + lis $5 + .word print
                     + jalr $5 + pop($31)
                     + pop($1)
```

Note: You might be okay with overwriting $1.

Administrata

- We’re going to do loading now, which means we just skipped from the middle of Module 8 to Module 10.
- We’re going out of order because A7 and A8 include codegen *and* loading.
- Modules 9 and 10 are independent from each other, so you can read them out of order.

Administrata P2

- We'll finish loading so that you're prepared for A7 and A8, then do the last of codegen, then optimization, then linking.
- That means we'll do part of 8, then part of 10, then the rest of 8, then 8b, then the rest of 10, then 9. Doing things in order is for suckers.
- Note that 8b is optional. You're recommended to learn some of it because it will help you understand codegen more completely. Optimization itself is not tested.

Loaders

Let’s write Baby’s First Operating System

```
operating system ver. 1.0
repeat:
  p <- next program to run
  read the program into memory at address 0x0
  jalr $0
  beq $0, $0, repeat
```

Issues

- The operating system is a program that needs to be in memory. Where should it be?
- What if they conflict?
  - Could choose different addresses at assembly time, but how do we make sure they can never conflict?
  - A more flexible option is to make sure that code can be *loaded anywhere*. That way, we just take any free space!

Loader’s New role:

- Loader’s job:
  - Take a program P as input
  - Find a location α in memory for P
  - Copy P to memory, starting at α
  - Return α to the OS.

Baby's First OS v2

```
operating system ver. 2.0
repeat:
    p  <-- choose program to run
    $3 <-- loader(p)
    jalr $3
    beq $0, $0, repeat
```

Code for `loader`

```
loader ver. 2.0
$\alpha$ <-- findFreeRAM(N)
for (i=0; i<codeLength; ++i) {
    mem[$\alpha$+4i] = file[i];
}
$30 <-- $\alpha$ + N
return $\alpha$ to OS
```

Problems

- How did we assemble .word label?
- It compiles to an address; the address of that label…
- assuming that the program was loaded at 0!
- Loader needs to fix this!

Problems

- `.word id`: need to add alpha to id
- `.word constant`: do not relocate!
- `beq, bne` (whether they use labels or not!): do not relocate! Why?
- Seems easy enough, but there's a problem…

Machine Code

Recall that we translate assembly code into machine code (bits). Given:

`0x00000018`

Was this line of code a `.word 0x18` or a `.word id`? You can't know!

Thus, we need a way for our loader to know what does and what doesn't need to be relocated. We need to remember which .words were labels!

Object Code

- Usually, the output of assemblers isn't pure machine code; it is object code!
- We've seen this: these are our MERL files (MIPS Executable Relocatable Linkable)
- In this file, we need the code, but also the location of any .word ids (i.e., words that need to be relocated). We'll need more information on top of that soon...

```
Assembly             MERL In
                     Assembly
                     beq $0, $0, 2
                     .word endModule
                     .word endCode
lis $3               lis $3
.word 0xabc          .word 0xabc
lis $1               lis $1
.word A              reloc1: .word A
jr $1                jr $1
B:                   B:
jr $31               jr $31
A:                   A:
beq $0, $0, B        beq $0, $0, B
.word B              reloc2: .word B
                     endCode:
                     .word 1
                     .word reloc1
                     .word 1
                     .word reloc2
                     endModule:
```

Loading

Requires two passes:

- Pass 1: Load the code from the file into the selected location in memory
- Pass 2: Using the *relocation table*, update any memory addresses that have *relocation entries*. (We'll see other kind of entries later...)

Caveat

- Note: Even with this, it is possible to write code that only works at address 0:
  ```
  lis $2
  .word 12
  jr $2
  jr $31
  ```
- You should never encode address as anything other than labels, so that your loader can update the references! (That is, never hard-code addresses)

```
read_word()                  //skip the first word in the MERL file
endMod ← read_word()         // second word is the address of end of MERL file
codeSize ← read_word() - 12  // Use the third word to compute size of the code section
α ← findFreeRAM(codeSize)    // find starting address α

for(int i=0; i<codeSize; i+= 4)  // load the actual program (not the header and footer)
  MEM[α + i] ← read_word()       // starting at α

i ← codeSize + 12      //start of relocation table
while (i < endMod)
  format ← read_word()
  if (format == 1)     // indicates relocation entry
    rel ← read_word() // address to be relocated: this is relative to start of header
                       // not start of code, location to update is MEM[α + rel - 12]
    MEM[α + rel - 12] += α - 12  // go forward by α but also back by 12
                                 // since we did not load the header
  else
    ERROR // unknown format type
  i += 8  // update to next entry in MERL footer
```

```
read_word()                          //skip the first wor
endMod ← read_word()                 // second word is th
codeSize ← read_word() - 12          // Use the third wor
$\alpha$ ← findFreeRAM(codeSize)     // find starting add

for(int i=0; i<codeSize; i+= 4)  // load the actu
  MEM[$\alpha$ + i] ← read_word()    // starting at $\alpha$

i ← codeSize + 12        //start of relocation tabl
while (i < endMod)
  format ← read_word()
  if (format == 1)       // indicates relocation en
    rel ← read_word()    // address to be relocated
                         // not start of code, loca
    MEM[$\alpha$ + rel - 12] += $\alpha$ - 12  // go forward by
                                 // since we did
  else
    ERROR // unknown format type
  i += 8  // update to next entry in MERL footer
```

Assembly	MERL In Assembly
	beq $0, $0, 2
	.word endModule
	.word endCode
lis $3	lis $3
.word 0xabc	.word 0xabc
lis $1	lis $1
.word A	reloc1: .word A
jr $1	jr $1
B:	B:
jr $31	jr $31
A:	A:
beq $0, $0, B	beq $0, $0, B
.word B	reloc2: .word B
	endCode:
	.word 1
	.word reloc1
	.word 1
	.word reloc2
	endModule: