Meeting 8: Strings, syscalls, CALL (compile, assemble, link, load)

Brandon Myers
University of Iowa
Where we are going

Instruction set architecture (e.g., MIPS)

Compiler
- translating source code (C or Java)
- Programs to assembly language
- And linking your code to Library code

Instruction set architecture (e.g., MIPS)

Memory system

Processor

I/O system

Datapath & Control

Digital logic

How the software talks To the hardware

How a processor runs MIPS Programs!

How switches (1 or 0) can be used to build Interesting functions: from integer arithmetic to programmable computers
Check in

on clicker, write
• one major takeaway from Tuesday’s class
• one question about project 1
Representing human language in the computer

declare string in MIPS:

```
.data
my_greeting: .asciiz "I love CS2630!"
```

array of characters:

```
'I' ' ' 'l' 'o' 'v' 'e' ' ' 'C' 'S' '2' '6' '3' '0' '!' '\'0'
```

null terminator marks the end of the string

ascii-encoded 1-byte characters (shown in decimal):

```
73 32 108 111 118 101 32 67 83 50 54 51 48 10 0
```

(shown in binary):

```
01001001 00100000 01101100 01110110 01100101 00100000 01000011 01010011 00110010 00110110 00110011 00110000 00001010 00000000
```
Peer instruction

What does the function mystery do?

(Initially called with $a0 as the address of an ascii-encoded, null-terminated string)

a) reverse the substring between index 48 and 57
b) nothing sensible because $a0 gets overwritten
c) turn all letters to upper case
d) turn all letters to lower case
e) add one to only integer characters
f) infinite recursion
System.out.println?
How do you perform I/O in MIPS?

```
.data
my_greeting: .asciiz "I love CS2630!"

.text
la $a0, my_greeting
li $v0, 4
syscall
```

<table>
<thead>
<tr>
<th>Service</th>
<th>Code in $v0</th>
<th>Arguments</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>$a0 = integer to print</td>
<td></td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>$f12 = float to print</td>
<td></td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>$f12 = double to print</td>
<td></td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>$a0 = address of null-terminated string to print</td>
<td></td>
</tr>
</tbody>
</table>
Administrivia

• remember Project 1 is due Sep 26, 11:59pm
Producing an executable file

```
lw $t0, 4($s0)
addi $t0, $t0, 10
sw $t0, 8($s0)
```

```
10001110000010000000000000000100
00100001000010000000000000001010
10101110000010000000000000000100
```
Assemble (one source file)

Assemble

```
lw $t0, 4($r0)
addi $t0, $t0, 10
sw $t0, 8($r0)
```

steps:
1. translate pseudo instructions to real instructions (TAL)
   • after this step, we know the address where each TAL instruction will be stored
2. turn labels into immediates
3. translate each instruction into binary (with holes*)

*important detail for linking multiple files
Assemble step 2: turn labels into immediates

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>with resolved addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>label1:</td>
<td>j label3</td>
<td>j 0x100004</td>
</tr>
<tr>
<td>0x00400004</td>
<td></td>
<td>ori $t0,$t0,0xFF</td>
<td>(no change)</td>
</tr>
<tr>
<td>0x00400008</td>
<td>label2:</td>
<td>beq $t0,$t2,label1</td>
<td>beq $t0,$t2,-3</td>
</tr>
<tr>
<td>0x0040000C</td>
<td></td>
<td>addiu $t1,$t1,-1</td>
<td>(no change)</td>
</tr>
<tr>
<td>0x00400010</td>
<td>label3:</td>
<td>addiu $t2,$t2,-1</td>
<td>(no change)</td>
</tr>
</tbody>
</table>
a similar problem...

• A skyscraper has many floors, connected by one elevator.
• Every floor may have either one uniquely named restaurant or no restaurants.
• Every floor may have either one advertisement for any restaurant in the skyscraper or no advertisements.

• The problem is that the advertisements only say the name of a restaurant and not what floor you can find it on.

• Starting on the bottom floor, what is an algorithm for writing on each sign the floor of the restaurant it advertises?
Producing an executable file
Combining multiple assembled files
Stretching our analogy to the linker

• A crazy civil engineer/real estate tycoon has decided to buy three skyscrapers and stack them on top of each other!

• Each individual skyscraper had its own floors--numbered from 1--and its own restaurant advertisements using those floor numbers.

• consequence: some advertisements may need to be revised: get a new floor number by adding the existing floor number to the new floor number of the (formerly) bottom floor
Linker *relocates* code
Linker also *resolves symbols*

- (example in terminal)

- object files have three kinds of symbols
  - $T = \text{symbol defined in text segment (name of a function)}$
  - $D = \text{symbol defined in data segment (name of a global variable)}$
  - $U = \text{undefined symbol (name of a function or variable that we expect to be defined in another file)}$

- the linker’s job is to make sure the combined executable file has no symbols that are still U’s
Common linking errors?

• two conflicting symbols
  • e.g., two object files define the get_name() function

• unresolved symbol
  • e.g., customer.o calls sell_product(), which is not defined anywhere
Peer instruction
A “secret” about Java...
Compiling and running C vs. Java
(shown for only a single source file, i.e. ignoring the linking step)

helloworld.c

C Compiler (gcc)

lw $t0, 4($r0)
addi $t0, $t0, 10
sw $t0, 8($r0)

MIPS Assembler

assembly code
(.s or .asm files)

10001110000010000000000000000100
001000010000100000000000000001010
1010111000000100000000000000001000

MIPS processor executes the binary

helloworld.java

Java Compiler (javac)

Java bytecode
(.class files)

1const_2
istore_1
iload_1
sipush 1000
if_icmpge
44

Java Virtual Machine (JVM) *interprets* the bytecode

The JVM itself is the executable file (which you downloaded from Oracle’s website) that runs on the processor

https://en.wikipedia.org/wiki/Java_bytecode
Another detail about Java

• *Interpreting* Java bytecode is much slower than executing machine code

• The JVM is fancy: for “hot” methods (methods that run many times), the JVM compiles & assembles them into machine code

• The JVM only knows what code is “hot” once the program has been running for awhile, so the JVM’s compiler is known as a JIT (“just-in-time”) compiler

The JVM itself is the executable file (which you downloaded from Oracle’s website) that runs on the processor