CSE 141 Discussion

Hacking the Assembler Infrastructure
The Assembler

• So the reason the second assignment requires you to have an ISA description is simple
  – You have to translate your ISA into binary code
  – In order to do that, you need to write a program to do that for you
    • Coding yourself is too difficult/bug-ridden

• Some slides adapted from previous course iterations
From Code to Running Programs

- **C program**
  - compiled to **Assembly Language**
  - assembled to **Object (Machine Code)**

- **Library (Machine Code)**

- **Executable (Machine Code)**
  - linked to **Library (Machine Code)**
  - loaded into **Memory**

- Your Project!
From Code to Running Programs

int main()
{
    printf("hello, world\n");
    return 0;
}

ldah gp,8192(t12)
lld gp,28464(gp)
ld sp,-16(sp)
stq ra,0(sp)
stq fp,8(sp)
mov sp,fp
ld a0,-1(gp)
lda a0,32648(a0)
ldq t12,-32584(gp)
jsr ra,(t12),120006b40
ldah gp,8192(ra)
lda gp,28424(gp)
cir v0
br 120001198
mov fp,sp
ldq ra,0(sp)
ldq fp,8(sp)
ld sp,16(sp)
ret zero,(ra),0x1
nop

00 20 bb 27
30 6f bd 23
f0 ff de 23
00 00 5e b7
08 00 fe b5
0f 04 de 47
ff ff 1d 26
88 7f 10 22
b8 80 7d a7
6e 56 5b 6b
00 20 ba 27
08 6f bd 23
00 04 ff 47
00 00 e0 c3
1e 04 ef 45
00 00 5e a7
08 00 fe a5
10 00 de 23
01 80 fa 6b
1f 04 ff 47

Your Project!
Assembly Language

- Text representation of machine instructions

**Sample:**

```assembly
.data
table0:
    .word 0x000C0FFEE
    .word 0x001C0FFEE
    .word 0x002C0FFEE
    .word table1

table1:
    .word 0x00DEADBEEF
    .word 0x1DEADBEEF
    .word 0x2DEADBEEF
    .word 0x3DEADBEEF
    .word 0x4DEADBEEF, 0x5DEADBEEF, 0x6DEADBEEF
    .fill 10 0x0

.text
    la $1, table0 // load the address of a label table0 (pseudo instruction)
    lw $2, $1 // $2 <= 0x000C0FFEE
    lw $3, table0 // load the value at label table0 (pseudo instruction), $3 <= 0x000C0FFEE
    lw $4, 3(table0) // $4 <= table1
    lw $5, 1($4) // $5 <= 0x1DEADBEEF
    sw $5, table1($1) // 0x001C0FFEE is overwritten to 0x1DEADBEEF
    li $6, 0xC0FFEE // load immediate (pseudo instruction), $6 <= 0xC0FFEE
```
So where do we start?

• Fortunately, you don’t have to code EVERYTHING from scratch.
• You can find the framework at the following page:
  http://cseweb.ucsd.edu/classes/fa12/cse141/project/assembler.html
• The page will also give you a javadoc of the methods that are already implemented
Assembly Language

- Sections
  - Text Section: Instructions of the program
  - Data Section: Initialized data storage
- Keyword
  - Indicate the change of sections, how to deal with the data or ...
- Label
  - A name for an instruction/data address
- Pseudo Instruction
  - The shortcut to a sequence of instructions
Inputs and Outputs

• Input
  – Your assembly code (likely as some “.s” file)

• Output
  – Your machine code in instruction memory (as some [$prefix]_i.coe file)
  – Your initialized data memory (as some [$prefix_d.coe file]
Tips of Using Framework

- Extend the Assembler class
  - `AssembleCode()` contains a two parse algorithm
    - parse #1: scan the code and labels
    - parse #2: replace labels with offsets (or addresses) and generate code

- Implement virtual methods:
  - `processLabel`, `generateCode`, `updateProgramCounter`, initialization, `replaceInstructionLabel`, and `replaceMemoryLabel`
Why two passes?

• A label can be defined anywhere in the program, be it the .text segment or the .data segment

• A label used at instruction number $i$ could be stored at instruction number $i+10$

• This is why we have to parse twice, because labels might not so up in order (and frequently don’t)
AssembleCode() 1st Parse

- Initialization()
- getNextInputLine()
- Is it Keyword? yes → processData(): store data section into “memory”. processAdditionalKeywords()
- Is it Label? yes → processLabel(): store in a table?
- Is it an Instruction? yes → processInstruction(): parse instruction into Instruction object updateProgramCounter(): update PC, be careful about pseudo inst.

*You need to implement all the methods in red*
Function Explanations

• Initializer:
  – Mapping between asm instructions and machine code should reside here
  – You should map ISA instructions to opcodes and register names to machine code equivalents here
  – Recommend HashMap class but any data structure can be used

• processAdditionalKeywords():
  – Only if needed
Function Explanations

• Processlabel()
  – Is called when a label definition is encountered
  – Should store PC for each label defined in instructions (.text) and memory address for each label defined in .data
  – Keep data structures in mind

• updateProgramCounter()
  – Fairly self explanatory, increment every instruction
AssembleCode() 2nd Parse

- Calculate Label Address for PC
  - replaceInstructionLabel(): if there is any label within the instruction, replace it with memory address.

- Generate Code
  - generateCode(): Generate the machine code according to the instruction you feed in!

- Calculate Label Addresses for Data
  - replaceMemoryLabel(): Scan the “memory” object, replace all the labels with addresses.

- Output Data

*You need to implement all the methods in red*
Function Explanations (Part Deux)

• `generateCode()`
  – Takes an object of class Instruction
  – Generate machine code for that instruction
  – Can be implemented by a switch statement that takes opcode as input

• `replaceInstructionLabel()`
  – Replace labels used in .text section

• `replaceMemoryLabel()`
  – Replace labels used in .data section
More Help?

• As always, check out the google group for questions from previous quarters.
• Don’t hesitate to come to us in lab hours for help.
• Good luck!