ISA & CSE 141 Project

Vineet Kumar
vkumar@cs
From Code to Running Programs

C program

- compiler
- assembler

Assembly Language

Object (Machine Code)

- linker

Library (Machine Code)

Executable (Machine Code)

- loader

Memory

Your Project!
From Code to Running Programs

int main()
{
    printf(“hello, world\n”);
    return 0;
}
Design Issues of ISA

• Perform basics functions
  • Arithmetic/Logic instructions
  • Branches (Control the program flow)
  • Memory access?
  • How to deal with function calls?

• # of instructions to represent a program?

• Simplicity?
  • Hardware implementation -> efficiency!
Classification of ISAs

- \( C = A + B \) (Assume \( A, B, C \) in memory)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>REG-MEM</th>
<th>REG-REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1, A</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R3, R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R3, C</td>
<td>Add R3, R1, B</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store R3, C</td>
<td>R2</td>
</tr>
</tbody>
</table>


Project Overview

• Design YOUR OWN Instruction Set Architecture
  • Capable of implementing the benchmarks
• Develop an assembler
• Develop a simulator
Your Own ISA

- Instruction width (machine code width): 17 bit
- Data and address widths (word size): 34 bit
- General purpose enough to be able to run provided benchmarks and more general programs
Your Own ISA (cont.)

- MUST include the following instructions
  - \texttt{in} \ [dest] \ [channel[3:0]]
    read a 34-bit data from the specified channel and save at [dest]
  - \texttt{out} \ [src] \ [channel[3:0]]
    write a 34-bit data from [src] to the specified channel
  - \texttt{halt}
    Stop execution and return control to the simulator's command prompt.

- You may need to leave opcode space for expansion in case you leave out an instruction you need!
Assembly Language

- Text representation of machine instructions

Sample:

```assembly
.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, 1($1)     // 0x001COFFEE is overwritten to 0x1DEADBEEF
li $6, 0xC0FFEE  // load immediate (pseudo instruction), $6 <= 0xC0FFEE
.data
table0:
.word 0x000C0FFEE
.word 0x001C0FFEE
.word 0x002C0FFEE
.word table1
.table1:
.word 0x0DEADBEEF
.word 0x1DEADBEEF
.word 0x2DEADBEEF
.word 0x3DEADBEEF
.fill 10 0x0
```
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
Assembler

- Translate assembly code to machine code
  - From add $s0, $a1, $t7 to 0x00AF8020
- Develop an assembler
  - You may use the java-based
  Assembler framework
    - Accept your assembly code as input
    - Output
      - [$prefix]_i.coe: machine code
      - [$prefix]_d.coe: initial data memory
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
AssembleCode() 1st Parse

- Initialization()

- getNextInputLine()

- Is it Keyword?
  - yes: processData(): store data section into “memory”.
  - no: processAdditionalKeywords()

- Is it Label?
  - yes: processLabel(): store in a table?
  - no: processInstruction(): parse instruction into Instruction object

- Is it an Instruction?
  - yes: updateProgramCounter(): update PC, be careful about pseudo inst.

- EOF?

*You need to implement all the methods in red*
AssembleCode() 2nd Parse

- Calculate Label Address for PC
- Generate Code
- Calculate Label Addresses for Data
- Output Data

replaceInstructionLabel(): if there is any label within the instruction, replace it with memory address.
generateCode(): Generate the machine code according to the instruction you feed in!
replaceMemoryLabel(): Scan the “memory” object, replace all the labels with addresses.

*You need to implement all the methods in red
Example

- A very simple instruction set architecture
- Load $rs, offset($rt)
- Add $rs, $rt, $rd
- Store $rs offset($rt)
Simulator

• Simulate the behavior of a processor running your own ISA
  • Input: the assembled machine code
  • Output: what the code supposed to do
Simulator

• Develop a simulator
  • You may use the provided simulator framework

• Tip:
  • Starting by modify the switch statement in the "execute" method in the provided ISASimulator class
Benchmarks

• Programs used to validate the functionality and performance of a computer system

• Fibonacci numbers
  • Function calls

• SuperGarbage
  • A virtual machine
  • Be careful of branch instructions.