COMP 102: Excursions in Computer Science
Lecture 7: Interpreting the program

Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

Class web page: www.cs.mcgill.ca/~jpineau/comp102

---

Quick Recap

- **Weeks 1-2: Hardware approach**
  - Every problem is expressed with boolean variables and operators.
  - Can implement any function using the right combination of AND, OR, NOT.
  - Hardware solutions are quick (in terms of machine running time.)
  - But this is very inflexible (need a new circuit for each program!)

- **Week 3: Software approach**
  - Always same hardware, same set of circuits (any standard computer)
  - Can implement a large variety of programs and be reprogrammed.
  - Need a layer to translate the programming language into something the computer will understand.

Today’s lecture: Machine Language
My laptop

Input

Output

Output

Input

Inside the laptop

http://commons.wikimedia.org/wiki/File:Inside_white_MacBook.jpg

1. Optical Drive
2. Hard Disk
3. RAM
4. Airport Extreme card
5. Battery
6. Processor & chipset covered by heat pipe
7. 13.3" Screen

- MagSafe power port
- Gigabit Ethernet port
- Mini DisplayPort
- Two USB 2.0 ports (up to 480 Mbps)
- Audio in/out
- Kensington lock slot
Simplifying the picture

- CPU: Performs the operations in the current instruction.
- Memory: Stores the program (sequence of instructions and data). RAM = Random Access Memory

Each cell corresponds to a bit (0 or 1)

Program instructions

Forms of memory

http://www.real-knowledge.com/memory.htm
The CPU is composed of 3 major parts:

- **ALU (Arithmetic Logic Unit)**
  - Arithmetic & Logical operations

- **Registers**
  - Storage areas for data and machine instructions operated on by ALU

- **Control unit**
  - Acts as a coordinator between the ALU and registers

**Instructions do very simple things**

- Read bits (i.e. accessing variables).
- Change the bits in a location (i.e. assigning variables).
- Move bits from 1 cell to another.
- Treat some bits as numbers to apply arithmetic operations (add, subtract, multiply, …)
- Modify which instruction is executed next -> CONTROL FLOW
- Communicate with external devices.
**Fetch-Decode-Execute Cycle**

1. **Fetch Cycle**: The Control Unit *fetches* the next instruction from memory.
2. **Decode Cycle**: The Control Unit *decodes* the instruction (figuring out what the bits represent).
3. **Execute Cycle**: The ALU *executes* the required instruction and stores results into memory.

This is the only thing the CPU does!

---

**Fetch-Decode**

1. Look at the **Program Counter** (PC) to determine the location in memory where the next instruction is stored.
2. **Retrieve** this instruction from program memory.
3. **Decode** this instruction.
4. After an instruction is fetched, **increment** the PC by the length of the instruction.
Length of instructions

• Most computer architectures assume all instructions are 32-bit long. Easy to know how many bits to fetch at a time!

• More recently, new architectures use 64-bit instructions.
  – Twice as much information per cycle = faster computing!

Problem!

Problem #1: Need to convert the program into a long stream of bits.
  – Some parts are actual memory (e.g. variables).
  – Some parts are the instructions / operations.

• Writing a program as bits directly is tedious!
• There are human-readable mnemonics for bit patterns.
Machine Instructions

MIPS32 Add Immediate Instruction

<table>
<thead>
<tr>
<th>OP Code</th>
<th>Addr 1</th>
<th>Addr 2</th>
<th>Immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>00001</td>
<td>00010</td>
<td>0000000101011110</td>
</tr>
</tbody>
</table>

Equivalent mnemonic: `addi $r1, $r2, 350`

**English translation:**

Add the contents of the register r2 and 350 and store the result in the register r1.

Machine Instructions

- Other kinds of instructions include:
  - **Transferring** data between registers or memory locations
  - **Arithmetic or logical** operations (use the ALU)
  - **Control**: test contents of a register and jump to a location

- There are binary codes for each of these (and associated mnemonics).
  
  E.g. [http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html](http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)
Execute Cycle

1. **Execute** the instruction
   - Connects the various components of the computer so that the desired operation may be carried out.

2. **Write** back the results (if any) of the execute step to some form of memory.

---

Computer Speed

- The CPU experiences high and low voltage changes, driven by the internal **clock** (vibrating quartz crystal).
  - The clock operates with a predetermined **frequency** (such as 2.4GHz).

- Each time the clock changes, the computer's processor handles the next machine instruction.

- A more accurate measurement would compare the **number of instructions per second** (MIPS: million instructions per second) as some computers use the clock ticks more efficiently than others.
  - Apple’s MacBook (2009) handles ~10,000 MIPS.
MIPS increasing

Example of an Executable

![Binary Code](http://pages.cs.wisc.edu/~larus/HP_AppA.pdf)
Back to programming languages

• We write programs in a user-friendly programming language:

```
Integer sum
Integer x
sum = 0
For x = 1 to 100
  sum = sum + x*x
End loop
Print sum
```

• How can we convert:

High-level programming language  \[\rightarrow\]  Low-level machine language

• This is a job for a compiler.

---

Compiler

• The compiler translates high-level programming language into low-level machine language:
  - Program -> Executable
  - Each programming language needs its own compiler.

• Note: The compiler is a program! (So need a compiler for the compiler…)
  - Here’s an idea: Our compiler is actually compiled by itself!
Compiler

- **Front End**: Needs to know all about the input language.
- **Back End**: Knows all about the machine itself (CPU).
- **Intermediate representation (IR)**:
  - Generated by the Front End, understood by the Back End.
  - Generic, medium-level “universal” language

Front End Parser

- **Front End** will parse the input program.
  - All computer languages come with **parsing rules**.
  - These parsing rules are the **grammar** (syntax) for the language.
  - This produces a **parse tree**, which is the Intermediate Representation.

**Example**:

```
Integer sum
Integer x
sum = 0
For x = 1 to 100
  sum = sum + x*x
End loop
Print sum
```

```plaintext
Declaration
Type integer
Variable sum
Assignment
Variable sum
Value 0
For Loop
Etc.
```
Back End

- Once the program is in an Intermediate Representation:
  - Step through the IR, considering each piece of the parse tree.
  - Figure out which machine instruction template matches each piece.
  - Use the MIPS look-up table to find the corresponding instruction in binary.

<table>
<thead>
<tr>
<th>IR</th>
<th>Assembly instruction</th>
<th>MIPS binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = y + z</td>
<td>lw, $s, [z]</td>
<td>0000 00ss ssst tttt dddd</td>
</tr>
<tr>
<td></td>
<td>lw, $t, [y]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add, $d, $s, $t</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw, $d, [x]</td>
<td></td>
</tr>
</tbody>
</table>

Now we have a program in bits, which can be executed by the CPU!

Note: Often we need to optimize the code (to make it faster):
Lots of clever techniques to improve the code.

Example

<table>
<thead>
<tr>
<th>High-level Language</th>
<th>Assembly Language</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp = y[k];</td>
<td>lw $s0, 0($2)</td>
<td>0100 1000 1100 0120 1010 1111 0101 0000 0101 1000 1100 1111 1001 0000 0101 1000 1010 1111</td>
</tr>
<tr>
<td>y[k] = y[k+1];</td>
<td>sw $s1, 0($2)</td>
<td></td>
</tr>
<tr>
<td>y[k+1] = temp;</td>
<td>sw $s0, 4($2)</td>
<td></td>
</tr>
<tr>
<td>C/Java Compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS Assembler</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

www.cise.ufl.edu/~mssz/ CompOrg/CDA-lang.html
Linking different programs

- Often different pieces of the program are built separately.

- Each piece can go through the compiler individually, to get separate executables. Need to put all this together somehow!

- **Linker**: Takes pieces of the program and puts them together into an executable.
  - Sometimes this is done as part of the compiler, sometimes separately.

---

Take-home message

- Understand the main components of the computer.

- Be familiar with the principles of Fetch-Decode-Execute-Store.

- Understand the role of machine language.

- Know the main components of the compiler (Front End, Back End, Intermediate Representation) and what purpose they serve.