Instruction Set
Architectures: Talking to the Machine
The Next Two Weeks

• Two Goals
• Prepare you for your 141 Project
  • Understand what an ISA is and what it must do.
  • Understand the design questions they raise
  • Begin to think about what makes a good ISA vs a bad one
  • See an example of designing an ISA.
• Learn to “see past your code” to the ISA
  • Be able to look at a piece of C (or Java) code and know what kinds of instructions it will produce.
  • Understand (or begin to) the compiler’s role
  • Be able to roughly estimate the performance of code based on this understanding (we will refine this skill throughout the quarter.)
In the beginning...

The Difference Engine

- Physical configuration specifies the computation
The Stored Program Computer

• The program is data
  • i.e., it is a sequence of numbers that machine interprets

• A very elegant idea
  • The same technologies can store and manipulate programs and data
  • Programs can manipulate programs.
The Stored Program Computer

- A very simple model
- Several questions
  - How are program represented?
  - How do we get algorithms out of our brains and into that representation?
  - How does the computer interpret a program?
Representing Programs

• We need some basic building blocks -- call them “instructions”

• What does “execute a program” mean?

• What instructions do we need?

• What should instructions look like?

• What data will the instructions operate on?

• How complex should an instruction be?

• How do functions work?
Program Execution

• This is the algorithm for a stored-program computer
• The Program Counter (PC) is the key

Read instruction from program storage (mem[PC])

Determine required actions and instruction size

Locate and obtain operand data

Compute result value

Deposit results in storage for later use

Determine successor instruction (i.e. compute next PC). Usually this mean PC = PC + <instruction size in bytes>
Big “A” Architecture

• The Architecture is a contract between the hardware and the software.
  • The hardware defines a set of operations, their semantics, and rules for their use.
  • The software agrees to follow these rules.
  • The hardware can implement those rules *IN ANY WAY IT CHOOSES!*
    • Directly in hardware
    • Via a software layer
    • Via a trained monkey with a pen and paper.

• This is a classic interface -- they are everywhere in computer science.
  • “Interface,” “Separation of concerns,” “API,” “Standard,”

• For your 141 project you are designing an Architecture -- not a processor.
  • (in 141L, you will design a processor)
What instructions do we need?

• Basic operations are a good choice.
  • Motivated by the programs people write.
  • Math: Add, subtract, multiply, bit-wise operations
  • Control: branches, jumps, and function calls.
  • Data access: Load and store.

• The exact set of operations depends on many, many things
  • Application domain, hardware trade-offs, performance, power, complexity requirements.
  • You will see these trade-offs first hand in the ISA project and in 141L.
Motivating Code segments

• a = b + c;
• a = b + c + d;
• a = b & c;
• a = b + 4;
• a = b - (c * (d/2) - 4);
• if (a) b = c;
• if (a == 4) b = c;
• while (a != 0) a--;
• a = 0xDEADBEEF;
• a = foo[4];
• foo[4] = a;
• a = foo.bar;
• a = a + b + c + d +... +z;
• a = foo(b); -- next class
What data will instructions operate on?

• Is specifying the instructions sufficient?
  • No! We also must what the instructions operate on.
• This is called the “Architectural State” of the machine.
  • Registers -- a few named data values that instructions can operate on
  • Memory -- a much larger array of bytes that is available for storing values.
• How big is memory? 32 bits or 64 bits of addressing.
  • 64 is the standard today for desktops and larger.
  • 32 for phones and PDAs
  • Possibly fewer for embedded processors
How do instructions access memory?

• In modern ISAs, every byte (8 bits) of data has an address.

• Arithmetic operations just operate on registers

• Memory operations access memory
  • Load -- move a piece of data from memory into a register
  • Store -- move the contents of a register into memory.
Bytes and Words

Modern machines use “byte addressable” memories

<table>
<thead>
<tr>
<th>Byte addresses</th>
<th>Word Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>data</td>
</tr>
<tr>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
<tr>
<td>Address</td>
<td>data</td>
</tr>
<tr>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>
What should instructions look like?

• They will be numbers -- i.e., strings of bits
• It is easiest if they are all the same size, say 32 bits
  • Given the address of an instruction, it will be easy to find the “next” one.
• They will have internal structure
  • Subsets of bits represent different aspects of the instruction -- which operation to perform. Which data to operate on.
  • A regular structure will make them easier to interpret
  • Most instructions in the ISA should “look” the same.
• This sets some limits
  • On the number of different instructions we can have
  • On the range of values any field of the instruction can specify
How complex should instructions be?

- More complexity
  - More different instruction types are required.
  - Increased design and verification costs
  - More complex hardware.
  - More difficult to use -- What’s the right instruction in this context?

- Less complexity
  - Programs will require more instructions -- poor code density
  - Programs can be more difficult for humans to understand
  - In the limit, decrement-and-branch-if-negative is sufficient
    - Imagine trying to decipher programs written using just one instruction.
    - It takes many, many of these instructions to emulate simple operations.

- Today, what matters most is the compiler
  - The Machine must be able to understand program
  - A program (i.e., the compiler) must be able to decide which instructions to use

- Each instruction should do about the same amount of work.
How do functions work?

- The “Stack Discipline,” “Calling convention,” or “Application binary interface (ABI)”.
- How to pass arguments
- How to keep track of function nesting
- How to manage “the stack”
Motivating Code segments

- \( a = b + c; \)
- \( a = b + c + d; \)
- \( a = b & c; \)
- \( a = b + 4; \)
- \( a = b - (c \times (d/2) - 4); \)
- \( \text{if} \ (a) \ b = c; \)
- \( \text{if} \ (a == 4) \ b = c; \)
- \( \text{while} \ (a != 0) \ a--; \)
- \( a = 0xDEADBEEF; \)
- \( a = \text{foo}[4]; \)
- \( \text{foo}[4] = a; \)
- \( a = \text{foo}.bar; \)
- \( a = a + b + c + d +... +z; \)
- \( a = \text{foo}(b); \) -- next class

- What instructions do we need?
- What should instructions look like?
- What data will the instructions operate on?
- How complex should an instruction be?

- Simplicity favors regularity
- Smaller is faster
- Make the common case fast
- Good design demands good compromises