Computer Fundamentals: CPUs, Fetch-Execute, Compilation

Dr Robert Harle
Today's Topics

- Stored Program Models
- The Fetch-Execute Cycle, registers, ALU etc
- Machine code, assembly, higher languages
- Compilers vs. interpreters
A modern computer boils down to three fundamental things

- **Storage/memory** – giving the ability to hold state (programs & data)
- **Processing unit (CPU)** – giving the ability to manipulate state.
- **A program** – giving the ability to instruct the CPU how to manipulate state in storage
We think of memory abstractly, as being split into discrete chunks, each given a unique *address*.

- We can read or write in whole chunks.
- Modern memory is big.
A program is just a sequence of instructions. The instructions available depend on the CPU manufacturer.

We will make up some very simple instruction labels:

- **Lij**: Load value at memory address I into register J
- **Aijk**: Add register I to J and put the result in register K
- **Sij**: Store register I in memory address J
## Fetch-Execute Cycle I

### Memory Table

<table>
<thead>
<tr>
<th></th>
<th>L6X</th>
<th>L7Y</th>
<th>AXYZ</th>
<th>SZ8</th>
<th>63</th>
<th>12</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram

1. **Fetch**
   - Memory location L6X is fetched.
2. **Execute**
   - The fetched instruction is executed in the ALU.
3. **Increment PC**
   - The program counter (PC) is incremented.

### CPU Components
- **CPU**
- **IB**
- **MAU**
- **ALU**
- **Registers**
  - PC: 2 1
  - X: 6 3
  - Y: 
  - Z: 

### Registers
- **PC**
- **X**
- **Y**
- **Z**
Add Functions

- **Fx**: Jump to address x and run code from there
- **RET**: Jump back to where we left off
### Functions

<table>
<thead>
<tr>
<th></th>
<th>L9X</th>
<th>L7Y</th>
<th>F14</th>
<th>L8X</th>
<th>F14</th>
<th>49</th>
<th>6</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>

When we jump we use a special memory address (slot 10 here) to note where we came from so we can RET there.
The storage mixes together the program and the data... this is efficient but dangerous!
Storage Models

We use this model

Von-Neumann Architecture

Harvard Architecture
## Choosing an Architecture

<table>
<thead>
<tr>
<th></th>
<th>Von-Neumann</th>
<th>Harvard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Same</strong> memory for programs and data</td>
<td></td>
<td><strong>Separate</strong> memories for programs and data</td>
</tr>
<tr>
<td>+ Don't have to specify a partition so more efficient memory use</td>
<td></td>
<td>- Have to decide in advance how much to allocate to each</td>
</tr>
<tr>
<td>+ Programs can modify themselves, giving great flexibility</td>
<td></td>
<td>+ Instruction memory can be declared read only to prevent viruses etc writing new instructions</td>
</tr>
<tr>
<td>- Programs can modify themselves, leaving us open to malicious modification (viruses!)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Can't get instructions and data simultaneously (therefore slower)</td>
<td></td>
<td>+ Can fetch instructions and data simultaneously</td>
</tr>
</tbody>
</table>
Instruction Sets

- The list of instructions a CPU supports is its Instruction Set Architecture (ISA)
  - Initially all used different instructions but there is clearly an advantage to using the same instruction sets
  - Intel's x86 set is a de-facto standard for PCs
  - ARM's v6 and v7 specifications are used for lower power applications (phones etc)
Computers don't store text instructions like L6X, but rather a binary code for each instruction

Called *machine code*
What the CPU 'understands': a series of instructions that it processes using the fetch-execute technique.

E.g. to add registers 1 and 2, putting the result in register 3 using the MIPS architecture:

```
00000000000100010000110000100000
```

- **Register 1**: 000000000001
- **Register 2**: 1000100001100000
- **Register 3**: 000000010000
- **Addition**: 11000001100000
- **OP type**: 000000000001
- **Shift amount**: N/A

**Diagram:**

```
+---+---+---+
| O | E | X |
+---+---+---+
| R1| R3| A  |
+---+---+---+
```

- **Register 1**: R1
- **Register 2**: R2
- **Register 3**: R3
- **Addition**: A
- **OP type**: O
- **Shift amount**: E

- **Machine Code**
Assembly

- Essentially machine code, except we replace binary sequences with text that is easier for humans
- E.g. add registers 1 and 2, storing in 3:

```
add $s3, $s1, $s2
```

- Produces small, efficient machine code when assembled
- Almost as tedious to write as machine code
- Becoming a specialised skill...
- Ends up being architecture-specific if you want the most efficient results :-(

Levels of Abstraction for Programming

- High Level Languages
- Procedural Languages
- Assembly
- Machine Code

Compile

- Human friendly
- Geek friendly
- Computer friendly
A compiler is just a software program that converts high-level code to machine code for a particular architecture (or some intermediary).

Writing one is tricky and we require strict rules on the input (i.e. on the programming language). Unlike English, ambiguities cannot be tolerated!
Handling Architectures

Source Code (e.g. C++)

C++ Compiler for x86

Binary executable for PC (x86)

C++ Compiler for ARM

Binary executable for ARM
Interpreters

- The final binary is a compiled program that can be run on one CPU architecture.
- As computers got faster, it became apparent that we could potentially compile 'on-the-fly'. i.e. translate high-level code to machine code as we go.
- Call programs that do this **interpreters**

<table>
<thead>
<tr>
<th>Architecture agnostic – distribute the code and have a dedicated interpreter on each machine</th>
<th>Have to distribute the code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easier development loop</td>
<td>Errors only appear at runtime</td>
</tr>
<tr>
<td></td>
<td>Performance hit – always compiling</td>
</tr>
</tbody>
</table>
Software Libraries

- Sometimes we package up useful chunks of code into libraries
  - Just a grouping of functions compiled to machine code
  - You can't 'run' a library – it's just a collection of functions
  - The intention is that each library is installed once per machine and many programs use it
Library Advantages

- Modern software makes **extensive** use of libraries
  - Makes the program smaller (references library functions rather than defining them itself)
  - Established libraries are well tested so fewer bugs
  - Experts in a specific area typically write the associated libraries so performance often better
Dependency Hell

- Programs are dependent on the libraries being present
  - They can be deleted
  - Or upgraded to incompatible versions
- Libraries can depend on other libraries too
- Can find yourself in a difficult state where you need multiple versions of libraries!
So what have we Learnt?

- Computers need three things: storage, processing and programs
- Computers just do a very simple **fetch-execute** loop very fast to run through some collection of machine code instructions one at a time
- The machine code is usually generated for us via a **compilation** step that takes in more human-friendly program descriptions
- We can potentially compile as we run the program: this is an **interpreter**
- Computers have software libraries that are collections of useful function implementations that can be used by any program
So what have we Learnt?

- Computers need three things: storage, processing and programs.
- Computers just do a very simple *fetch-execute* loop very fast to run through some collection of machine code instructions one at a time.
- The machine code is usually generated for us via a *compilation* step that takes in more human-friendly program descriptions.
- We can potentially compile as we run the program: this is an *interpreter*.
- Computers have software libraries that are collections of useful function implementations that can be used by any program.