Today's Topics

- Brief History of Computers
- Stored Program Model
- Fetch-Execute Cycle, registers, ALU etc
- Notion of Compilers and Interpreters
Inspired by the typewriter (!), Alan Turing (King's) created a theoretical model of a computing machine in the 1930s. He broke the machine into:

- **A tape** – infinitely long, broken up into cells, each with a symbol on them
- **A head** – that could somehow read and write the current cell
- **An action table** – a table of actions to perform for each machine state and symbol. E.g. move tape left
- **A state register** – a piece of memory that stored the current state
Universal Turing Machines

- Alan argued that a Turing machine could be made for any computable task (e.g. sqrt etc)
- But he also realised that the action table for a given turing machine could be written out as a string, which could then be written to a tape.
- So he came up with a **Universal Turing Machine**. This is a special Turing Machine that reads in the action table from the tape
  - A UTM can hence simulate any TM if the tape provides the same action table
- This was all theoretical – he used the models to prove various theories. But he had inadvertently set the scene for what we now think of as a computer!
Colossus

- 1944, Bletchley park
- Designed to break the German Lorenz SZ40/42 encryption machine
- Fed in encrypted messages via paper tape. Colossus then simulated the positions of the Lorenz wheels until it found a match with a high probability
- No internal program – programmed by setting switches and patching leads
- Highly specific use, not a general purpose computer
- Turing machine, but not universal
ENIAC

• Electronic Numerical Integrator and Computer
  • 1946, “Giant brain” to compute artillery tables for US military
  • First machine designed to be turing complete in the sense that it could be adapted to simulate other turing machines
  • But still programmed by setting switches manually...

• Next step was to read in the “action table” (aka program) from tape as well as the data
• For this we needed more general purpose memory to store the program, input data and output
Manchester Baby

- 1948 a.k.a. mark I computer
- Cunning memory based on cathode ray tube. Used the electron gun to charge the phosphor on a screen, writing dots and dashes to the tiny screen
- A light-sensitive collector plate read the screen
- But the charge would leak away within 1s so they had to develop a cycle of read-refresh
- Gave a huge 2048 bits of memory!
EDSAC

- Electronic Delay Storage Automatic Calculator
- First practical stored-program computer, built here by Maurice Wilkes et al.

- Memory came in the form of a mercury delay line

- Used immediately for research here.
- Although they did have to invent programming....
1965-70 Integrated Circuits

- Semiconductors could replace traditional electronics components → use a slice of semiconductor and 'etch' on a circuit
- End up with an Integrated Circuit (IC) a.k.a a microchip
- Much easier to pack components on an IC, and didn't suffer from the reliability issues of the soldering iron

**Moore's Law:** the number of transistors on an IC will double every two years
The Rise of Intel

- Intel started in 1968 manufacturing ICs, producing ICs with a particular target of memory (RAM, see later).
- 1969 – commissioned to make 12 custom chips for a calculator (one for keyboard scanning, one for display control, etc).
- Not enough resource so instead proposed a single general-purpose logic chip that could do all the tasks.
- 1971 - Managed to buy the rights and sold the chip commercially as the first **microprocessor**, the Intel 4004.
1971 - Microprocessor Age

- The 4004 kick-started an industry and lots of competitors emerged
- Intel very savvy and began an “intel inside” branding assault with products like the 386
- Marketing to consumers, not system builders any more
The CPU in more Detail
- Recall: Turing's universal machine reads in an action table (=program) of instructions, which it then applies to a tape (=data). Two options for our program storage in a modern machine.
So where do you store your programs and data?

<table>
<thead>
<tr>
<th>Von-Neumann</th>
<th>Harvard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Same</strong> memory for programs and data</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>- Have to decide in advance how much to allocate to each</td>
</tr>
<tr>
<td>+ Programs can modify themselves, giving great flexibility</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</td>
<td></td>
</tr>
<tr>
<td>- Can't get instructions and data simultaneously (therefore slower)</td>
<td>+ Can fetch instructions and data simultaneously</td>
</tr>
</tbody>
</table>
Simple Model of Memory

- 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.
Simple Model of a CPU

Registers

PC
X
Y
Z

ALU

MAU

CPU

Instruction Buffer
Program counter

Functional units
Memory Access unit
Arithmetic Logic unit

Registers
A Simple Command Set

- 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 K
  - **SIJ**: Store register I in memory address J
Fetch-Execute Cycle II

Memory

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

CPU

Registers

PC  2
X  63
Y  12
Z

ALU

MAU

IB  L7Y
Fetch-Execute Cycle III

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

Registers
- PC: 3
- X: 63
- Y: 12
- Z: 78

CPU

IB
- AXYZ

MAU

ALU
Fetch-Execute Cycle IV

Memory

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

Registers

- PC: 4
- X: 63
- Y: 12
- Z: 75

CPU

- IB

- MAU

- ALU
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**
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:

```
000000000001000100001100000100000
```

```
Register 1  Register 3  Addition
```

```
OP type  Register 2  Shift amount (N/A)
```

```
000000000001000100001100000100000
```

```
OP type  Register 2  Shift amount (N/A)
```

```
000000000001000100001100000100000
```

```
OP type  Register 2  Shift amount (N/A)
```
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>