Teaching London Computing

A Level Computer Science

Topic 5: Computer Architecture and Assembly

Queen Mary University of London

King’s College London

Computing at School

Supported by

Mayor of London

Network of Excellence

Department for Education

William Marsh
School of Electronic Engineering and Computer Science
Queen Mary University of London
Aims

• Understand CPU structure in more detail
  • Fetch-execute cycle
  • Faster CPUs
• Write simple assembly code
  • Use Little Man Computer
• Understand principles of compiling
• Compare compilers and interpreters
CPU Structure

What’s in a CPU
Simple Computer

- Processor
  - CPU
- Memory
  - Data
  - Program instructions
- I/O
  - Keyboard
  - Display
  - Disk

![Diagram of a simple computer system]

- **Memory**
- **CPU**
- **I/O Interfaces**: Keyboard I/F, Disk I/F, Display I/F

Data flows from Memory to CPU and from CPU to I/O interfaces. Addresses are passed from Memory to CPU.
Memory

• Each location
  • has an address
  • hold a value

• Two interfaces
  • address – which location?
  • data – what value?
Registers (or Accumulators)

- A storage area inside the CPU
- VERY FAST
- Used for arguments and results to one calculation step
CPU Structure

- Accumulators
  - Data for calculation
- Data
  - Word to/from memory
- PC
  - Address of next instruction
- Instruction
  - For memory access
Instructions

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Address</th>
</tr>
</thead>
</table>

• Instruction
  • What to do: Opcode
  • Where: memory address

• Instructions for arithmetic
  • Add, Multiply, Subtract

• Memory instructions
  • LOAD value from memory
  • STORE value in memory
### Add Instruction

<table>
<thead>
<tr>
<th>Add</th>
<th>Address</th>
</tr>
</thead>
</table>

- One address and accumulator (ACC)
  - Value at address added to accumulator

- \( \text{ACC} = \text{ACC} + \text{Memory}[\text{Address}] \)
Fetch-Execute Cycle

How the Computer Processes Instructions
Fetch-Execute

- Each instruction cycle consists of two subcycles
- Fetch cycle
  - Load the next instruction (Opcode + address)
  - Use Program Counter
- Execute cycle
  - Control unit interprets the opcode
  - ... an operation to be executed on the data by the ALU
Fetch Instruction

1. Program counter to address register
2. Read memory at address
3. Memory data to ‘Data’
4. ‘Data’ to instruction register
5. Advance program counter
1. Decode instruction
2. Address from instruction to ‘address register’
3. Access memory
4. Data from memory to ‘data register’
5. Add (e.g.) data and accumulator value
6. Update accumulator
Summary of CPU Architecture

• Memory contains data and program
  • Program counter: address of next instruction
  • Instructions represented in binary
  • Each instruction has an ‘opcode’

• Instructions contain addresses
  • Addresses used to access data

• Computer does ‘fetch-execute’
  • ‘Execute’ depends on opcode

• Computer can be built from < 10,000 electronic switches (transistors)
Discussion

- Could we act out the way a computer works?
Making a Faster Computer
Clock

- Steps in sequence are controlled by a clock
- Clock frequency
  - 8086 – 10 MHz
  - Pentium 4 – 3 GHz
  - **However, clock speeds no longer increasing much**

- Processing power depends on
  - Clock speed
  - Clock cycles to execute an instruction
1GHz Clock

- $10^9$ cycles per second
  - 1 cycle every 1 ns
  - Light travels 1 m in approx 3 ns
  - Billion instructions per second (up to)
- Each instruction takes longer
  - … many instruction in process at once
- Memory access is slower
  - 10 of ns
Moore’s Law
Faster Computer

1. Faster clocks
   • smaller IC lines (90 → 45 → 32 → 22nm)
   • shorter time to switch
2. More Registers; Bigger registers
   • 32 → 64 bits
3. Fetch-execute pipeline
   • overlap the fetch and execute
4. Cache memory
5. Multiple cores
Intel 86 Family

- 8086 – 1978
  - 16 bits
- 80386 – 1986
  - 32 bits
- Pentium 4 – 2000
  - 32 bits $\rightarrow$ 64 bits
- Intel core i3, i5, i7
- COMPATIBLE compiled program
- More instruction
- Longer words
- More on chip
  - floating point
  - cache memory
- Instructions in parallel
  - pipeline
  - superscalar
  - multi-core
Little Man Computer

A Simulator for a Simple CPU
Advantages of LMC

- Very simple
- Number of implementations
  - Java applet
  - Excel
  - Flash, .net
  - MacOS

- Some incompatibilities
- Feature: uses decimal not binary
**LMC CPU Structure**

- Visible registers shown in red

[Diagram showing the CPU structure with labeled components: Accumulator, ALU, MEM Data, Instruction, Program Counter, Mem Address, Control Unit, Memory, data, address]
Address of memory access

Data to/from memory
Feature Comparison

• Step and Animation
  • Most allow single stepping – see fetch-execute
  • Animation – e.g. flash version

• Can program using mnemonics
  • LDA, STA, ADD …
  • Supported in e.g. applet
  • Not supported in Excel, Flash versions

• Programming versus demonstration
LMC Instructions
# Instructions

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Address</th>
<th>Code</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>HLT</td>
<td>000</td>
<td>HLT</td>
<td>Halt</td>
</tr>
<tr>
<td>1xx</td>
<td>ADD</td>
<td>1xx</td>
<td>ADD</td>
<td>Add: acc + memory → acc</td>
</tr>
<tr>
<td>2xx</td>
<td>SUB</td>
<td>2xx</td>
<td>SUB</td>
<td>Subtract: acc – memory → acc</td>
</tr>
<tr>
<td>3xx</td>
<td>STA</td>
<td>3xx</td>
<td>STA</td>
<td>Store: acc → memory</td>
</tr>
<tr>
<td>5xx</td>
<td>LDA</td>
<td>5xx</td>
<td>LDA</td>
<td>Load: memory → acc</td>
</tr>
<tr>
<td>6xx</td>
<td>BR</td>
<td>6xx</td>
<td>BR</td>
<td>Branch always</td>
</tr>
<tr>
<td>7xx</td>
<td>BRZ</td>
<td>7xx</td>
<td>BRZ</td>
<td>Branch is acc zero</td>
</tr>
<tr>
<td>8xx</td>
<td>BRP</td>
<td>8xx</td>
<td>BRP</td>
<td>Branch if acc &gt; 0</td>
</tr>
<tr>
<td>901</td>
<td>IN</td>
<td></td>
<td>IN</td>
<td>Input</td>
</tr>
<tr>
<td>902</td>
<td>OUT</td>
<td></td>
<td>OUT</td>
<td>Output</td>
</tr>
</tbody>
</table>
Add and Subtract Instruction

- One address and accumulator (ACC)
  - Value at address combined with accumulator value
  - Accumulator changed

- Add: ACC ← ACC + Memory[Address]
- Subtract: ACC ← ACC – Memory[Address]
Load and Store Instruction

<table>
<thead>
<tr>
<th>LDA</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA</td>
<td>Address</td>
</tr>
</tbody>
</table>

• Move data between memory and accumulator (ACC)

• **Load:** ACC ← Memory[Address]
• **Store:** Memory[Address] ← ACC
## Branch Instructions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>Address</td>
</tr>
</tbody>
</table>

- Changes program counter
- May depend on accumulator (ACC) value

- **BR**: PC ← Address
- **BRZ**: if ACC == 0 then PC ← Address
- **BRP**: if ACC > 0 then PC ← Address
LMC Example
Compiling a Simple Program

• Compiler translates high level program to low level

• E.g. \( x = y + z \)
Running the Simple Program

PC

IR

LDA

ACC

17

0

LDA y
ADD z
STA x
HLT

x

y

17

z

9
Running the Simple Program

PC

IR
ADD

ACC
16

LDA y
ADD z
STA x
HLT

x
y 17
z 9
Running the Simple Program

PC
IR
ACC

LDA y
ADD z
STA x
HLT

x  26
y  17
z  9
Running the Simple Program

PC

IR

ACC

LDA y
ADD z
STA x
HLT

x 26
y 17
z 9
Practice Exercises

• Try the first three exercises on the practical sheet
Compilers and Interpreters
Compiler

- Compiler translates high level program to low level

source code

\[ x = y + z \]

assembly code

LDA y
ADD z
STA x
HLT

object code

11010101
10010111
01110100
10000000

- Compiled languages
  - Statically typed
  - Close to machine
  - Examples: C, C++, (Java)
  - Compiler for each CPU
Inside a Compiler

- Parser: checks that the source code matches the grammar
- Type checker: checks types of variables and expressions
- Code generator: generates (optimised) code
**Principle of an Interpreter**

- One machine can emulate another
- ‘micro-coded’ CPU
- Intel CPUs and RISC inside

\[ x = y + z \]

Code (C) in language L

CPU-L

Rep of C

CPU-X

Interpreter in X for L

01110100
10010111
11010101
Java and .NET Language

- Java is compiled to ‘byte code’
  - `x = y + z`

- Byte code for Java ‘virtual machine’ (JVM)
  - One compiler
  - Libraries
  - Other languages

- The JVM is emulated on real computers
  - ‘JIT’ compiler
How Does an Interpreter Work?

• Example: LMC emulator

```python
def readMem(memory):
    global mdr
    mdr = memory[mar]

def execute(memory, opcode, arg):
    global acc, mar, mdr, pc
    if opcode == ADD:
        mar = arg
        readMem(memory)
        acc = acc + mdr
    elif opcode == SUB:
        mar = arg
        readMem(memory)
        acc = acc - mdr
    ...

def fetch(memory):
    global pc, mar
    mar = pc
    pc = pc + 1
    readMem(memory)
```

State of the LMC: registers and memory

```
acc = 0
mdr = 0
mar = 0
pc = 0
memory = [504,105,306, 0, 11, 17,...]
```
Summary

• A CPU executes Fetch-Execute
  • Fetch: next instructions from memory
  • Execute: data to/from memory and accumulator
  • Opcode determines execute
• One machine can emulator another
• Program execution
  • Compiler: translates
  • Interpreter: parses then interprets
  • Blend: Java or .NET and byte code