Why Study Assembler Programming?

Virtually all software development is done using compilers. Why study assembler?

Assembler is still important where:

- Code size is critical
- Ultimate speed is critical
- Access to processor registers is needed

AND: To understand the nature of our tools!

When we use a compiler:

- Delphi/Java/C code goes in
- What comes out?

Peewee – a tiny CPU

We'll simulate the tiny CPU in software and show:

- How written instructions can be encoded as opcodes
- How a program can "execute" these opcodes.
- The equivalence between a Delphi program and an Peewee Assembler program

A simulated processor is known as a Virtual Machine

The Fetch-Decide-Execute Cycle

The underlying model of a stored program processor is one that repeatedly:

1. Fetches an instruction
2. Determines what it is
3. Carries out action specified by the instruction

This is known as the Fetch-Decide-Execute cycle

1. Fetch an instruction
2. Determine what it is - Decode
3. Carry out action specified by the instruction - Execute

Note that for this to work, three forms of memory necessary:

1. Someplace to store the instructions
2. Some way to tell which instruction is next
3. Some state that can be changed - an effect

PeeWee – a tiny processor

Programmers Model

The programmer's model (or view) of a processor is all those things that can be accessed and/or modified.

- One 8 bit register - the Accumulator
- 256 memory locations - each 8 bits wide
- Program Counter (PC) - 8 bits wide

Note:

If memory is represented as an array, the array index (an integer) can be the memory address.

Instruction Set - Overview

The instruction set for this toy processor is too small to be used for anything useful, but that’s the point – this processor is so small, it’s really easy to understand.

Load adr Load accumulator from memory
Copy the contents of a memory location to the accumulator register.

Add adr Add memory to accumulator

Store adr Store accumulator in memory
(accumulator is unchanged)

Jump adr Jump to a specified address
Make the next instruction to be executed be the one at the specified address

Important

With these instructions, the processor cannot execute algorithms as there’s no way to make a choice.
**Effect of each Instruction in Detail**
Each instruction has one argument - the address of the memory location that the instruction will use.

**LDA - Load the Accumulator**
Load the contents of the specified memory location into the accumulator.

LDA addr \[ \text{Effect: } \text{Acc} = \text{mem}[\text{adr}] \]

If memory location 97 contains 22, after

\[ \text{LDA 97 } \quad \text{Acc} = \text{mem}[97] \]

the accumulator will also contain 22

**STA - Store the Accumulator**

STA addr \[ \text{Effect: } \text{mem}[\text{adr}] = \text{Acc} \]

eg \[ \text{STA 97 } \quad \text{mem}[97] = \text{Acc} \]

**ADD - Add to Accumulator**

ADD addr \[ \text{Effect: } \text{Acc} = \text{Acc} + \text{mem}[\text{adr}] \]

**JMP - Jump to address**

JMP addr \[ \text{Effect: } \text{PC} = \text{adr} \]

---

**Instruction Encoding**
An instruction is encoded into a single byte using the following scheme:

\[ \text{Bit: } 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \]
\[ O \quad P \quad m \quad m \quad m \quad m \quad m \quad m \]

Where the bits O & P indicate the instruction:

<table>
<thead>
<tr>
<th>OP</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LDA</td>
</tr>
<tr>
<td>01</td>
<td>STA</td>
</tr>
<tr>
<td>10</td>
<td>JMP</td>
</tr>
<tr>
<td>11</td>
<td>ADD</td>
</tr>
</tbody>
</table>

**Memory Address Encoding**
The remaining 6 bits \( mmmm \):
- hold the memory address needed by an instruction
- only 64 memory locations can be addressed.

There is no way, using single byte instructions to address all 256 memory locations.

**Example:**

\[ \text{Bit: } 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \]
\[ 0 \quad P \quad m \quad m \quad m \quad m \quad m \quad m \]

"Add 7" -> \[ 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \]

---

**PEEWEE in Software – Pascal**

```pascal
program PEEWEE;
const load=0; store=1; jump=2; add = 3; // Opcodes
var pc, acc, instr_decode_reg, adr, instruction : byte;
mem : array[0 .. 255] of byte;
begin
  pc := 0; // On startup, execute instruction at Mem[0]
  while true do
    begin
      // Fetch the instruction from memory
      instr_decode_reg := mem[pc];
      pc:= pc+1;
      // Increment PC
      // Break byte from Memory into the address
      // (lower 6 bits) and Instruction (upper bits)
      instruction := instr_decode_reg DIV 64;
      adr := instr_decode_reg AND 63;
      case instruction of
      // Decode and Execute
      jump : pc := adr;
      load : acc := mem[adr];
      store : mem[adr] := acc;
      add : acc := acc + mem[adr];
      end;
    end;
end.
```

---

**PEEWEE in Software – C version**

```c
#define load 0 // Define opcodes as constants
#define store 1 // to make code easier to read
#define jump 2
#define add 3

// Define variables to act as memory, various registers
int mem[256],pc, acc, instr_decode_reg, adr, instr = 0;

main {
  pc = 0; // On startup, execute instruction at Mem[0]
  while (1) {
    instr_decode_reg = mem[pc++]; // Fetch
    // Break byte from Memory into address
    // (lower 6 bits) and Instruction (upper bits)
    instr = instr_decode_reg / 64;
    adr = instr_decode_reg % 64;
    switch (instr) {
      // Decode and Execute
      case jump : pc = adr; break;
      case load : acc = mem[adr]; break;
      case store : mem[adr] = acc; break;
      case add : acc = acc + mem[adr];
    }
  }
}
```
Translating into Peewee Assembler

This little program doesn't do anything sensible but enables us to show an equivalent program in two languages – Pascal and Peewee assembler.

```pascal
program demo;
label loop;
var X : byte;
begin
    loop: X = X + 3;
    goto loop;
end.
```

Arbitrarily use:
Memory location 7 for X & location 8 for constant 3

This program can be translated into the following bit patterns representing the instructions – only the bit pattern is in memory, no variable names

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Memory Contents</th>
<th>When decoded by Peewee Means:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000111</td>
<td>LDA 7</td>
</tr>
<tr>
<td>1</td>
<td>11001000</td>
<td>ADD 8</td>
</tr>
<tr>
<td>2</td>
<td>01000111</td>
<td>STA 7</td>
</tr>
<tr>
<td>3</td>
<td>10000000</td>
<td>JMP 0</td>
</tr>
<tr>
<td>7</td>
<td>?????????</td>
<td>X</td>
</tr>
<tr>
<td>8</td>
<td>00000011</td>
<td>3</td>
</tr>
</tbody>
</table>

Enhancing Peewee Instruction Set

The JMP instruction set could be changed to a:

- **JZ adr** - Jump-if-Accumulator IS Zero to adr

- **JNZ adr** - Jump-if-Accumulator is NOT Zero

Can you modify the simulator code to execute these?

Which of JZ or JNZ is more useful???

Little more Challenging:
How about a **DJNZ** -DECREMENT-AND-JUMP-IF-NOT-ZERO instruction?

- **DJNZ adr** decrements the accumulator and if the result is NOT zero, jumps to adr

Even more Challenging
Can you write a program to add up the first 10 integers using this enhanced (either JZ, JNZ or DJNZ) instruction set?

Constants
Put constant values you need (eg 0, 1 or 10) into the simulated memory before the program starts running.