ANSWERS

A10.7 Programming languages and translators

Operating systems, software and translators · A level · OCR H446 1.2.2, AQA 7517 4.6.2.1, Eduqas A500QS 1.8 · about 45 min

What this lesson is about

Machine code, assembly and imperative high-level languages, the Little Man Computer, and choosing an assembler, compiler, interpreter or bytecode.

- Classifying languages
- Assembly language in the exam
- Translators

Questions

6 marks in all

1. Which describes assembly language? [1 mark]

- ☐ **A** A low-level language of mnemonics, where each instruction translates to one machine code instruction
- ☐ **B** A high-level language translated by a compiler
- ☐ **C** Binary codes executed directly by the processor
- ☐ **D** A language that runs on any processor without translation

> **Answer:** A. Machine code is the binary; assembly is its readable form, one line for one instruction.

2. Which are advantages of writing in assembly language rather than a high-level language? [1 mark]

Tick every answer that is true.

- ☐ **A** Direct control of the processor's registers and memory
- ☐ **B** The code can be hand-tuned to be very small and fast
- ☐ **C** The program can run on any processor
- ☐ **D** The code is quicker to write and easier to read

> **Answer:** A, B. Assembly is tied to one processor family, and slower to write and harder to read.

3. In the Little Man Computer, what does the instruction BRZ 12 do? [1 mark]

- ☐ **A** Branches to address 12 if the accumulator is zero
- ☐ **B** Branches to address 12 always
- ☐ **C** Stores zero in address 12
- ☐ **D** Branches to address 12 if the accumulator is positive or zero

> **Answer:** A. BRA branches always and BRP branches when the accumulator is zero or positive.

4. Why do most assemblers make two passes over the source code? [1 mark]

- ☐ **A** A label can be used before the line that defines it, so all addresses are found first
- ☐ **B** The first pass removes comments and the second checks spelling
- ☐ **C** Each pass translates half of the program
- ☐ **D** The second pass optimises the code the first pass produced

> **Answer:** A. Pass one builds the symbol table; pass two uses it to fill in addresses.

5. A company is releasing a game to the public and does not want players to see how it works. Which translator should it use, and why? [1 mark]

- ☐ **A** A compiler, because it produces standalone object code, so the source code is not distributed
- ☐ **B** An interpreter, because it stops at the first error
- ☐ **C** An assembler, because games are written in assembly
- ☐ **D** An interpreter, because it runs faster

> **Answer:** A. Interpreted programs need the source code on every user's machine.

6. Why do some compilers produce bytecode rather than machine code? [1 mark]

- ☐ **A** The bytecode can run on any computer with the right virtual machine, so the program is portable
- ☐ **B** Bytecode runs faster than machine code
- ☐ **C** Bytecode does not need to be translated or interpreted
- ☐ **D** Bytecode is easier for people to read than source code

> **Answer:** A. Only the virtual machine has to be written for each kind of hardware.

The task: an LMC assembler

Write a two-pass assembler for the LMC program in `source`, a list of strings, one line per address starting at address 0. Each line is one of: a mnemonic alone (`"OUT"`); a mnemonic and an operand (`"BRZ end"`); a label and a mnemonic (`"end HLT"`); or a label, a mnemonic and an operand (`"one DAT 1"`). A word is a label if it is not a key in `OPCODES`. An operand is either a label or a whole number. - **Pass 1**: for each line that has a label, store the label and its address in a dictionary, and print `<label> = <address>`. - **Pass 2**: for each line, the machine code is the opcode from `OPCODES` plus the operand's value (the label's address, or the number, or 0 if there is no operand). Print `<address>: <code>`, with the code as three digits, so 0 prints as `000` and 1 as `001`. The robot stays still.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

OPCODES = {"ADD": 100, "SUB": 200, "STA": 300, "LDA": 500, "BRA": 600, "BRZ": 700,
           "BRP": 800, "INP": 901, "OUT": 902, "HLT": 0, "DAT": 0}

source = [
    "INP",
    "STA count",
    "loop LDA count",
    "OUT",
    "BRZ end",
    "SUB one",
    "STA count",
    "BRA loop",
    "end HLT",
    "count DAT",
    "one DAT 1",
]
```

The hint students can ask for: The first pass only needs to notice labels and remember which address each is on. The second pass looks up the opcode for each mnemonic and adds the operand, which is either a number or a label to look up. A word is a label if it is not a mnemonic.

A solution

```
from bugbot import *
connect()

OPCODES = {"ADD": 100, "SUB": 200, "STA": 300, "LDA": 500, "BRA": 600, "BRZ": 700,
           "BRP": 800, "INP": 901, "OUT": 902, "HLT": 0, "DAT": 0}

source = [
    "INP",
    "STA count",
    "loop LDA count",
    "OUT",
    "BRZ end",
    "SUB one",
    "STA count",
    "BRA loop",
    "end HLT",
    "count DAT",
    "one DAT 1",
```

```
]

def split_line(line):
    parts = line.split()
    label = None
    if parts[0] not in OPCODES:
        label = parts.pop(0)
    mnemonic = parts[0]
    operand = parts[1] if len(parts) > 1 else None
    return label, mnemonic, operand

symbols = {}
for address, line in enumerate(source):
    label, mnemonic, operand = split_line(line)
    if label is not None:
        symbols[label] = address
        print(f"{label} = {address}")

for address, line in enumerate(source):
    label, mnemonic, operand = split_line(line)
    if operand is None:
        value = 0
    elif operand in symbols:
        value = symbols[operand]
    else:
        value = int(operand)
    code = OPCODES[mnemonic] + value
    print(f"{address}: {code:03d}")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.