OPCODEs

7 min

The Instruction Set Architecture defines how hardware processes binary data. Each 0 or 1 of binary data is called a *bit* and groups of these bits are put together in specific lengths that create instructions.

While the length of a specific binary instruction varies widely based on the ISA that is being used, the first few bits are always the *OPCODE* or OPeration CODE. This sequence of bits tells the processor what type of instruction it is receiving.

Every function or calculation that a processor can perform is defined by a specific OPCODE and the CU routes the remaining bits of information to the corresponding hardware that will execute the operation.

The list of all of these is included with the ISA documentation in the form of an OPCODE Table:

Mnemonic	OPCODE	Layman's Definition	Formal Definition
ADD	000001	Loads two numbers from registers and saves result into another register	rs_reg <- op_reg_1 + op_reg_2
LOAD	000010	Loads a number from a memory address location into a register	rs_reg <- mem[op_reg_1_addr]
STORE	000011	Copies data in a register to a memory address for long-term storage	mem[op_reg_1_addr] <- op_reg_2

After the OPCODE, the remaining bits in the instruction are normally referred to as the "operands". These are the pieces of data, sometimes presented as memory addresses or sometimes given directly as literals, on which the processor will operate.

The CPU will fetch the data from memory or registers, perform the function, and then return the result to the directed memory address or register.

Let's take a look at this example 32-bit instruction and identify the OPCODE:

000001 01001 10111 0001101001010110
OPCODE

Copy to Clipboard

If we use the table we had from above, the first six digits are the OPCODE (000001), telling the CPU that this is an ADD function. The remaining bits provide the CPU with instruction specifics that we will cover in the next exercise.

We will be using a Python interface to process our mock instructions for the rest of this lesson, a basic OPCODE interpreter is written in the code editor.

Instructions

1. Checkpoint 1 Passed

1.

Take a look at the Python code in the code editor, including the OPCODE table.

Guess the output of the following statement:

print(f"The '000001' OPCODE instructs the processor to {opcodeReader('000001')}.")

Copy to Clipboard

Run the code to see the answer.

Hint

Run the program to go to the next checkpoint.

2. Checkpoint 2 Passed

2.

Set the value of userInput1 to the name of the OPCODE function that is called by the following bits: 000101. Note that this value *must* be completely capitalized.

To check your answer, uncomment the statement print(userGuess1(userInput1)) and run the code.

Hint

Make sure to change the value of userInput1 from '' to 'ANSWER' and to delete the # in front of the line that looks like this: # print(userGuess1(userInput1))

3. Checkpoint 3 Passed

3.

Based on the given bits, set the values of userInput2, userInput3, and userInput4 to the correct OPCODES:

- userInput2 is called by the bits 000011
- userInput3 is called by the bits 000010
- userInput4 is called by the bits 000110

To test your answer, uncomment print(userGuess2(userInput2, userInput3, userInput4)) and run the program.

Hint

Make sure to use the operation name, in all CAPS, from the OPCODE Table.

script.py

```
from background import opcodeReader
from background import userGuess1
from background import userGuess2

# OPCODE Table:
# +-----------|--------+
# |   Name    | OPCODE |
# |-----------|--------|
# | ADD       | 000001 |
# | SUBTRACT  | 000010 |
# | MULTIPLY  | 000011 |
# | DIVIDE    | 000100 |
# | LOAD      | 000101 |
# | STORE     | 000110 |
# +--------------------+

# Checkpoint 1
print(f"The '000001' OPCODE instructs the processor to {opcodeReader('000001')}.")

# Checkpoint 2
userInput1 = 'LOAD'
print()
print(userGuess1(userInput1))


# Checkpoint 3
userInput2 = 'MULTIPLY'
userInput3 = 'SUBTRACT'
userInput4 = 'STORE'
print()
print(userGuess2(userInput2, userInput3, userInput4))
```