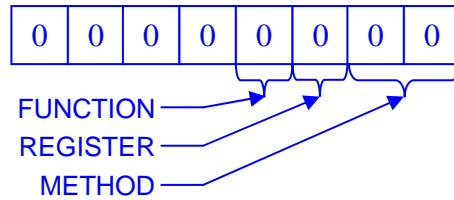


1. What opcode will blank memory initialized to 0x00 look like to the processor?

In binary, the opcode 0x00 is simply all zeros:



A function of 0 represents a store function, the register of 0 represents the accumulator ACC, and the method of 00 indicates that the operand is an address. This means that the opcode 0x00 is:

STOR ACC, [address]

Assuming all of memory is cleared, then the operand will also be 0x0000. Therefore, the final function will be:

STOR ACC, [0x0000]

2. Of the 256 possible opcodes we can get from an 8-bit opcode, how many are not being used in our instruction set, i.e., how many instructions could we add for future expansions of our processor?

I enumerated all of the 256 possible opcodes and identified the operation for each one. It identified one branch opcode (0x17), special opcodes 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, and 0x1f, all of the opcodes beginning with "001", and all of the opcodes beginning with a binary "01" as unused. I counted these and came up with 103 undefined opcodes.

I also took into account that some operations did not make sense. For example, it shouldn't be possible to store a value into a constant, opcodes 0x01 and 0x05. I also discounted the opcodes that I told you not to worry about, i.e., the opcodes that had operands of three or more bytes. These totaled 18.

Therefore, I accepted answers in the range of 103 to 121.

3. What would we need to add to our simulator to be able to include the following instructions: compare ACC with a constant, PUSH to or PULL from the stack, and take the 2's complement of ACC?

Each of these operations would require an additional opcode assigned to that function. In the case of the 2's complement, this might be a bit of a pain because all of the opcode patterns are already used for the mathematical/logical operations. The other two could be assigned to the "special operations" set without much problem.

To compare ACC with a constant, there would need to be an additional register for the "virtual subtraction" and a set of flags to hold the result.

For PUSH and PULL, at least one additional pointer register would be needed to point to memory where the stack would be contained. It is important to note that another array would not be needed here. In other words, there would be no additional memory required, only a vector or pointer to where the stack was in existing memory. Stacks work out of the same memory in which code and data are stored.

For the 2's complement, you would simply need the code to do $ACC = -ACC$.

4. If `executeInstruction()` were divided into two parts, decode and execute, what additional global resources would be needed for your simulator?

Every time you divide an operation into stages, resources are required to pass the results of a previous stage to the next. IR for example is needed to pass the results of `fetchNextInstruction()` to `executeInstruction()`.

To divide `executeInstruction()` into two parts, we would have to pass the results of the decode, i.e., what operation is being performed, to the execute function in order to perform it. This is a minimum requirement.

5. Make suggestions for ways to further subdivide the `executeInstrucion()` function.

This question was poorly written, and therefore, I did not take points away for misunderstanding it. The question was meant as a continuation of question 4, i.e., how can we "further" divide `executeInstruction()` beyond decode and execute? One way would be to divide the execution portion into a fetch operands, execute instruction, then store operands.