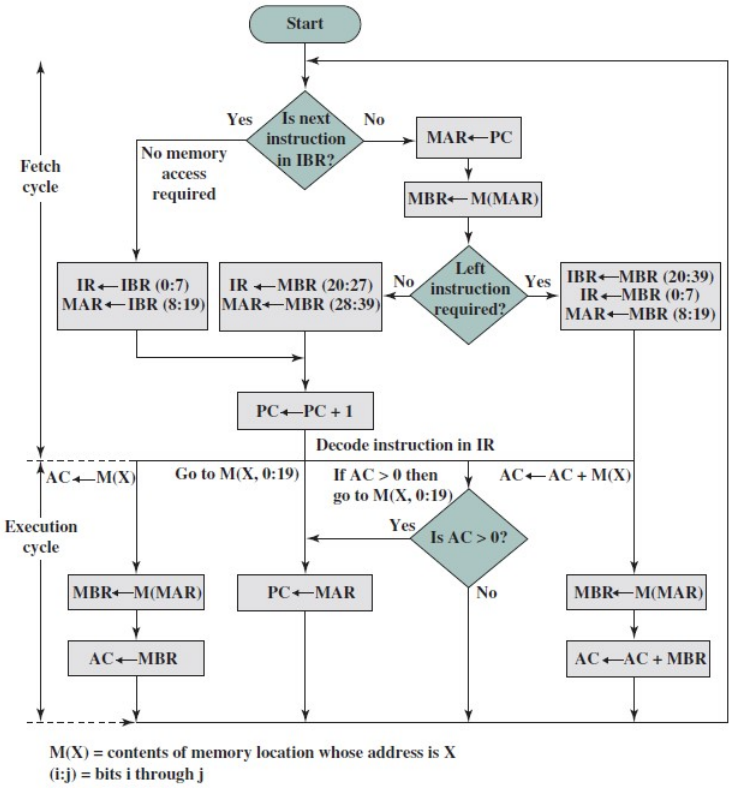
Note 01: Instruction Cycle

Instruction

We will simulate a simple instruction cycle for the IAS computer at a top level; that is, it will illustrate the changes to the registers and memory addresses of the IAS computer. Initially, the breakdown of the IAS computer is as follows:

- Its CPU consists of 7 registers:
  - Program Counter (PC): Contains the address of the next instruction word to be fetched from memory.
  - Instruction Register (IR): Contains the 8-bit opcode instruction being executed.
  - Instruction Buffer Register (IBR): Employed to hold the right-hand instruction temporarily.
  - Memory Address Register (MAR): Specifies the address in memory to be written from or read into the MBR.
  - Memory Buffer Register (MBR): Contains a word to store in memory or I/O or receive from memory or I/O.
  - Accumulator Register (AC): Employed to hold operands and results of ALU operations.
  - Multiply-Quotient Register (MQ): Employed to hold operands and results of ALU operations.
- Its words are 40 bits long.
- A number word is in twos-complement.
- An instruction word consists of two 20-bit instructions.
- An instruction is an 8-bit opcode followed by a 12-bit address.
- Its memory consists of 4096 memory locations.
- Its instruction cycle is:



Essentially, with the focus on the fetch cycle, a single cycle instance is as follows:

```
if IBR contains next instruction, then
  IR, MAR ← IBR[0:7], IBR[8:19]
else,
  MAR ← PC
  MBR ← M(MAR)
  if left instruction required, then
    IBR, IR, MAR ← MBR[20:39], MBR[0:7], MBR[8:19]
    execute instruction
  else,
    IR, MAR ← MBR[20:27], MBR[28:39]
PC ← PC + 1
execute instruction
```

where each assignment (←) represents a single simultaneous process, and the fetch cycle terminates with the execute statement.

- Its instruction set is:

Mnemonic	Opcode	Description
LMA	0A	Transfer contents from MQ to AC
LDM	09	Transfer M(X) to MQ
STA	21	Transfer contents from AC to memory location X
LDA	01	Transfer M(X) to AC
LDN	02	Transfer -M(X) to AC
ALD	03	Transfer |M(X)| to AC
ALN	04	Transfer -|M(X)| to AC
BRL	0D	Takes next instruction from left half of M(X)
BRR	0E	Takes next instruction from right half of M(X)
BPL	0F	If AC >= 0, takes next instruction from the left half of M(X)
BPR	10	If AC >= 0, takes next instruction from the right half of M(X)
ADD	05	Add M(X) to AC; put result in AC
AAD	07	Add |M(X)| to AC; put result in AC
SUB	06	Subtract M(X) from AC; put result in AC
ASB	08	Subtract |M(X)| from AC; put result in AC
MUL	0B	Multiply M(X) by MQ; put most significant bits of result in AC; least significant in MQ
DIV	0C	Divide AC by M(X); put quotient in MQ and remainder in AC
LSH	14	Multiply AC by 2
RSH	15	Divide AC by 2
STL	12	Transfer AC[28:39] to M(X)[8:19]
STR	13	Transfer AC[28:39] to M(X)[28:39]
HLT	00	Halts

We are provided the header file '`Object.h`' that contains the interface named *Object* that consists of the method:

- `toString()` - displays the memory.

Memory

We will begin by creating a memory class. For the simulation, it is necessary that:

- memory addresses have an identification
- memory addresses have a modification flag

Our memory class must have the following constraints and features:

- Be defined within the namespace *can*, in a header file named '`Memory.h`' that contains a header guard and includes the libraries *iostream*, *string*, *sstream*, *cmath*, *cctype*, and '`Object.h`'.
- Contains private fields for its data, identification, and status.
- Defines its special member functions.
- Defines a read method.
- Defines a write method.
- Defines a method that manually modifies its status.
- Overrides istream operator to read user input. It assigns the input to memory if the input is a binary string, converts to binary, then assigns the input to memory if the input is a hexadecimal string; otherwise, it assigns an 8-bit zero to memory.
- Overrides the `toString()` method so that it displays the memory in the format:

  $$id : data$$

  where *id* and *data* represent the memory's identification and content (in hexadecimal), respectively.

Conversion Functions

For the simulation, it is necessary to define:

- A function that converts a binary string to decimal.
- A function that converts a decimal to a binary string.

Simulation

Final sets are:

- A *Memory* object for each register and memory location.
- A function for each IAS instruction.
- A function to read memory content from a file.
- A function to perform the simulation. It should loop until it executes a halt instruction. It should provide a display after each assignment and execute statements from the fetch cycle.