CEC 470 Project

Two-Stage Instuction Decoder

Due: Friday, Dec 5, 2019 before noon.
What to submit: You will be submitting a C source file containing two C functions, ***FetchNextInstruction()***, and ***InstructionExecute ( )***. You also need to include the executable file. Submit your answer to the Questions as pdf files. Please upload all the files in one zip file on Canvas.

Introduction

This assignment is to design a simple processor based on the simplified 2-step instruction cycle. You will be creating two functions in software, one to simulate the fetch next instruction cycle, ***fetchNextInstruction()***, and one to simulate the execute instruction cycle, ***executeInstruction()***. Neither of these functions will have parameters nor will they return values. They will operation on global data meant to simulate the registers and memory of the processor.
This simulated machine consists of four registers that will be represented in your software with four global variables.

- PC -- Program counter (16 bit) used to hold the address of the next instruction to execute. It is initialized to zero.

- IR -- Instruction register (8 bit) used to hold the current instruction being executed.

- MAR -- Memory Address Register (16 bit) used to hold an address being used as a pointer, i.e., an indirect reference to data in memory.

- ACC -- Accumulator (8 bit) used to operate on data.

Memory will be simulated with an array ***memory[65536]***.
The function fetchNextInstruction() will perform the following steps:

- Use the program counter (PC) as an index to retrieve an instruction from the array memory[].

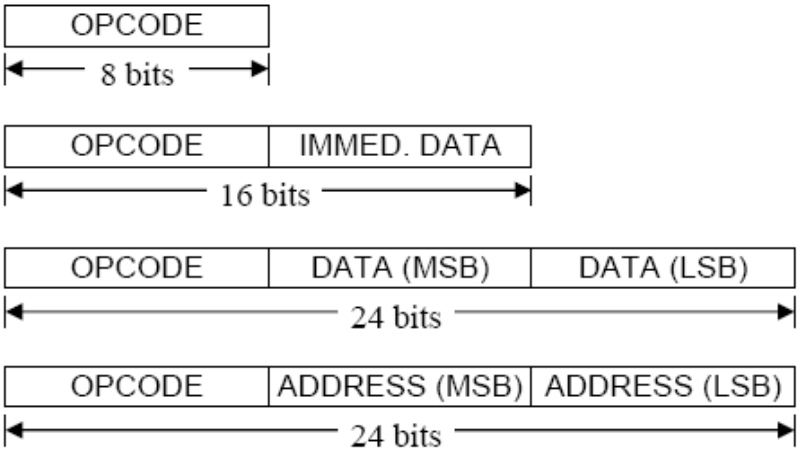
- Store the retrieved instruction to the instruction register (IR).
- Determine from the instruction how much to increment PC so that it points to the next instruction in memory[] and perform the increment.

The function executeInstruction() will examine IR to determine which operation to perform and it will perform it on the data contained in the registers and in the array memory[].

Instruction (Opcode) Set

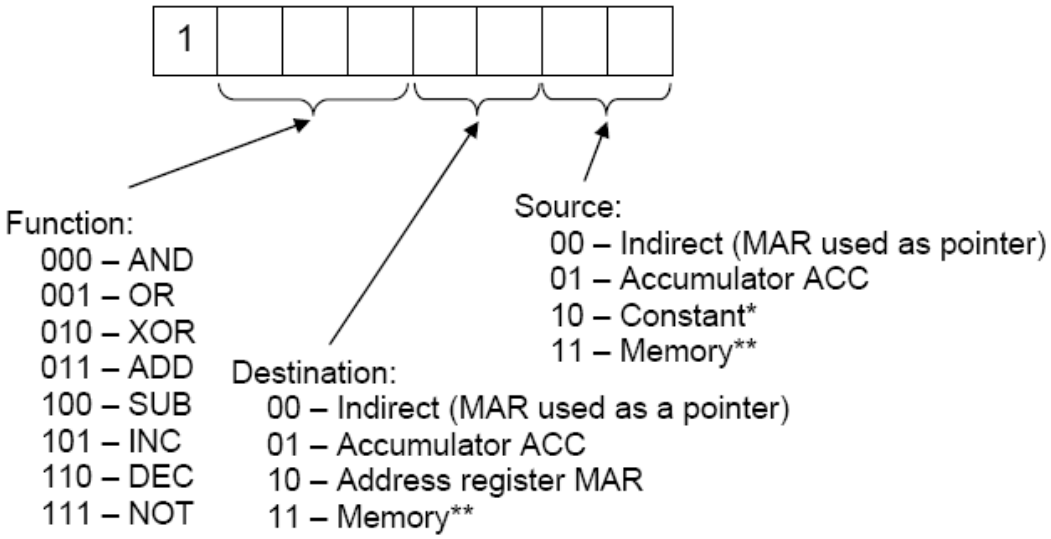
The following is a break down of the opcodes of the simple machine for which you are designing a simulator. The data (operands) for each instruction will follow the opcode immediately in memory. It is important to note that when using a16-bit operand (data or memory address), the byte immediately following the opcode is the most significant byte of the address and the byte

following that is the least significant. This is referred to as "Big Endian". The opcode for this machine will always be 8 bits. This means that the size of an instruction along with its data could be 1 byte (an opcode that doesn't need an operand), 2 bytes (an opcode that uses 8-bit immediate data, i.e., a constant), or 3 bytes (an opcode that uses an address as its operand or 16-bit immediate data).



Mathematical or Logical Operations:

If the most significant bit of the opcode is 1, then the opcode represents a mathematical or logical function. The opcode may be followed by 0, 1, or 2 bytes of operands.

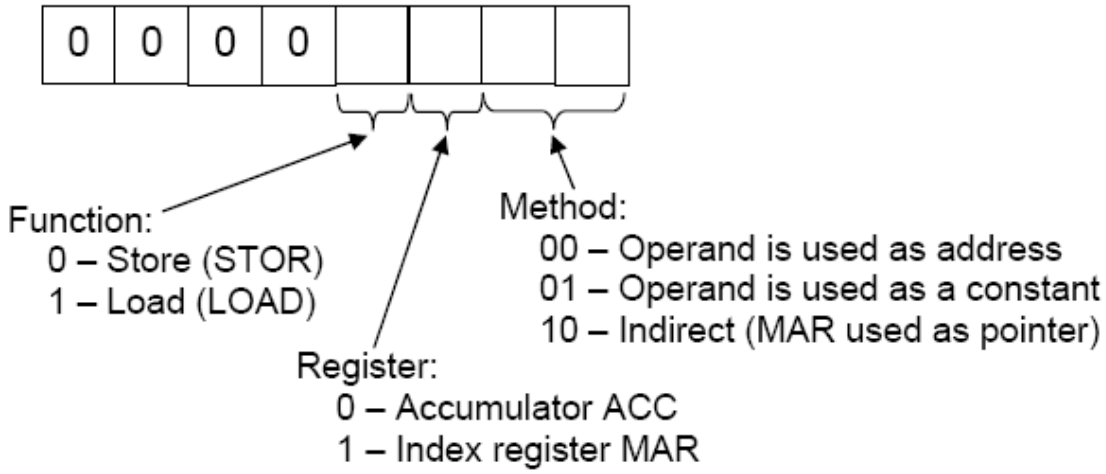


* -- The constant is passed to the processor as an 8- or 16-bit operand following the opcode.
** -- The memory address is passed to the processor as a 16-bit operand following the opcode

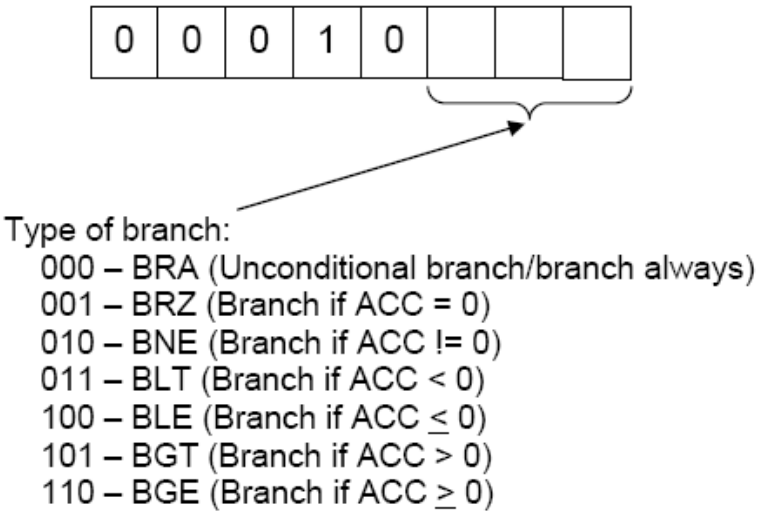
Memory Operations:

If the most significant four bits are 0000, then the opcode represents a memory or data transfer operation. When the “Method” bit is equal to 00, the opcode is followed by a 16-bit operand that serves as the memory address. When the “Method” bit is equal to 01, the opcode is followed by

an operand that serves as a constant (8 bits for ACC and 16 bits for MAR). Otherwise, the value contained in index register
MAR is used as the memory address.



Branches/Jumps:
If the most significant five bits are 00010, then the opcode represents an unconditional or conditional branch or jump. The opcode is ***always*** followed by a 16-bit operand that serves as the memory address.



Special Opcodes:
There are two additional opcodes included in the instruction set that represent special operations not included in the list above. None of the special opcodes require operands.

- 00011000_2 – NOP (No operation. fetchNextInstruction() should simply increment PC by one to the next instruction)
- 00011001_2 – Halt (Stops processor)

Sample Framework

The code below represents a basic framework for the simulator. It includes the prototypes for

fetchNextInstruction() and executeInstruction() along with the global definitions for each of the machine's registers. Note that the numeric types char and int differ across different platforms, so be sure to perform a bit-wise AND after every mathematical operation to clear the unused bits, i.e., AND ACC with 0xff and MAR with 0xffff to clear upper bits. The loop below continuously calls fetchNextInstruction() and executeInstruction() until the PC register points at a HALT instruction.

```
#define HALT_OPCODE 0x19

void fetchNextInstruction(void);
void executeInstruction(void);

unsigned char memory[65536];
unsigned char ACC=0;
unsigned char IR=0;
unsigned int MAR=0;
unsigned int PC=0;

int _tmain(int argc, _TCHAR* argv[])
{

// Execution loop.  Continue fetching and executing
// until PC points to a HALT instruction
   while(memory[PC]!=HALT_OPCODE)
   {
        fetchNextInstruction();
        executeInstruction();
   }
   return 0;
}
```

Sample Machine Code

You may want to test your code too. (It'd be nice to see if it worked, right?) You should be able to quickly create assembly language routines that you can compile into machine language to run on your simulator. Okay, so the instruction set is not that robust, but there are some things you can do. For example, the code below adds the values stored in memory locations 0x1000, 0x1001, and 0x1002, takes the two's complement, and stores the result in 0x1003.

```
LOAD ACC, [0x1000]
ADD  ACC, [0x1001]
ADD  ACC, [0x1002]
XOR  ACC, 0xff
INC  ACC
STOR ACC, [0x1003]
HALT
```

LOAD ACC, [0x1000] and therefore, the most significant four bits are 0000_2. The function is a load, so the next bit is 1. ACC is what is being loaded, so the sixth bit is a 0. The operand is a 16-

bit address, so the method is 00. This gives us the opcode and operand for the first instruction: 0x08, 0x10, 0x00.

The next instruction, ADD ACC, [0x1001], is a mathematical operation so the most significant bit is a 1. It is an ADD function, so the next three bits are 011. The destination is ACC, so 01 follows. The source is a memory address which is identified by 11. This makes the opcode and operand for the next instruction 0xb7 0x10 0x01. The following instruction, ADD ACC, [0x1002], is the same except for the memory address. This gives us the opcode and operand for the third instruction: 0xb7 0x10 0x02.

There is no instruction to do a 2's complement, so we need to flip the bits and add one. We flip the bits doing a bitwise XOR with all 1's. This is a logic function, so the instruction opcode begins with 1. XOR is 010. The destination is ACC which is 01. The source is a constant, 10. This gives us the following opcode and operand: 0xa6 0xff. The increment doesn't require an operand, so the source is ignored (I'd just set it to 00). This gives us an opcode 0xd4.

Finally, use the data transfer command STOR to store the result at 0x1003. The operand and opcode 0x00 0x10 0x03. This gives us the initialization values for memory starting at address 0.

```
memory[0]  = 0x08;
memory[1]  = 0x10;
memory[2]  = 0x00;
memory[3]  = 0xb7;
memory[4]  = 0x10;
memory[5]  = 0x01;
memory[6]  = 0xb7;
memory[7]  = 0x10;
memory[8]  = 0x02;
memory[9]  = 0xa6;
memory[10]  = 0xff;
memory[11]  = 0xd4;
memory[12]  = 0x00;
memory[13]  = 0x10;
memory[14]  = 0x03;
memory[15]  = 0x19;
```

Questions

As part of your project, answer the following questions. You may discuss the concepts with others in the class, but each of you must submit your answers in your own words.

1. Of the 256 possible opcodes we can get from and 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?
2. 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?
3. If executeInstruction() were divided into two parts, decode and execute, what additional global resources would be needed for your simulator?

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