[20 pts]

1- Consider a 1-address CPU that has a memory unit with 128K words of 32 bits each. An instruction is stored in one word of memory. The instruction format is divided into four fields: opcode field, 2-bit addressing mode field that specifies direct, indirect, indirect with pre-decrement, or indirect with post-increment addressing mode, a register field that specifies one of 32 registers, and an address field. For your information, given an address

- Direct addressing is where the operand is located in M[address].
- Indirect addressing is where the operand is located in M[M[address]].
- Indirect addressing with pre-decrement is where the operand is located in M[-M[address]].
- Indirect addressing with post-increment is where the operand is located in M[M[address]+].

(a) What is the maximum number of opcodes that can be incorporated into the CPU? How many bits are in the opcode field, the register field, and the address field? Draw the instruction format and indicate the number of bits in each field.

(b) How many bits are in the registers PC, MAR, MDR, IR, and AC?

Solution:

(a) Since the memory contains 128K ($2^{17}$) words, 17 bits are required for the address field. There are 32 registers, thus 5 bits are needed for the register field. This leaves 32 minus 17+5+2 (addressing mode) = 24, or 8 bits for the opcode field. Thus, this allows $2^8 = 256$ different opcodes (or instructions). The instruction format is shown below.

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>AM</th>
<th>Reg</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>5</td>
<td>17</td>
</tr>
</tbody>
</table>
```

(b) From (a) we see that 17 bits are required to address the entire memory, thus MAR and PC each have 17 bits. Since a memory word consists of 32 bits MDR and AC each have 32 bits. The number of bits required for IR depends on whether you assume the IR can hold the entire instruction or just the opcode. If IR can hold the entire instruction then it is 32 bits. If it just holds the opcode, then it is 8 bits. Depending on the assumption, the other possible answers are 10 bits and 15 bits.

[20 pts]

2- Consider the internal structure of a pseudo-CPU discussed in class. Suppose the pseudo-CPU can be used to implement the hypothetical instructions given below (Actually, these are instructions from PDP-8, which is the first commercially successful minicomputer from the 60’s). Give the sequence of microoperations required to implement the Fetch and Execute cycle for each of the instructions shown below. Your solution should result in minimum number of microoperations. Assume PC is currently pointing to the instruction and only PC and AC have the capability to increment itself.

(a) Logical AND:
   \[ \text{AND} \ Y \ ; \ AC \leftarrow AC \land M[Y] \]

(b) Increment and skip if zero:
   \[ \text{ISZ} \ Y \ ; \ M[Y] \leftarrow M[Y] + 1, \text{If}(M[Y]+1=0) \text{ Then } PC \leftarrow PC + 1 \]

(c) Deposit and clear the accumulator:
   \[ \text{DCA} \ Y \ ; \ M[Y] \leftarrow AC, AC \leftarrow 0 \]

(d) Jump to subroutine:
   \[ \text{JMS} \ Y \ ; \ M[Y] \leftarrow PC, PC \leftarrow Y + 1 \]

Solution:

To implement the instruction set, the pseudo-CPU continuously performs Fetch and Execute cycles. Since the Fetch cycle involves fetching and decoding instructions, the sequence of microoperations performed will always be the same. The Fetch cycle is given by

Fetch Cycle
Cycle 1: MAR \leftarrow PC;
Once the Fetch cycle completes, the CPU performs the Execute cycle. The sequence of microoperations performed in the Execute cycle depends on the assembly instruction decoded in the Fetch cycle. However, it is important to remember that since only a skeleton description of the pseudo-CPU has been given, some reasonable assumptions must be made about the architecture in order to implement the instruction set. The following are the sequence of microoperations performed for each assembly instruction:

(a) \textbf{AND} \ Y \quad ; \quad \text{AC} \leftarrow \text{AC} \land \text{M}[Y]

\textbf{Execute Cycle}

\begin{align*}
\text{Cycle 1: } & \text{MDR} \leftarrow \text{M}[\text{MAR}] \quad ; \quad \text{Read the operand from memory} \\
\text{Cycle 2: } & \text{AC} \leftarrow \text{AC} \land \text{MDR} \quad ; \quad \text{AND the contents of AC and MDR (i.e., M}[Y])}
\end{align*}

(b) \textbf{ISZ} \ Y \quad ; \quad \text{M}[Y] \leftarrow \text{M}[Y] + 1, \text{If(M}[Y]+1=0) \text{ Then } \text{PC} \leftarrow \text{PC} + 1

One straightforward way of doing this is given below:

\textbf{Execute Cycle}

\begin{align*}
\text{Cycle 1: } & \text{MDR} \leftarrow \text{M}[\text{MAR}] \quad ; \quad \text{Read M}[Y] \text{ from memory} \\
\text{Cycle 2: } & \text{AC} \leftarrow \text{MDR} \quad ; \quad \text{Transfer M}[Y] \text{ into AC} \\
\text{Cycle 3: } & \text{AC} \leftarrow \text{AC} + 1 \quad ; \quad \text{Increment M}[Y] \\
\text{Cycle 4: } & \text{MDR} \leftarrow \text{AC} \quad ; \quad \text{Transfer M}[Y]+1 \text{ into MDR} \\
\text{Cycle 5: } & \text{M}[\text{MAR}] \leftarrow \text{MDR}, \quad \text{If (AC=0) Then } \text{PC} \leftarrow \text{PC} + 1 \quad ; \quad \text{Store the M}[Y]+1 \text{ back into memory, and at the same time increment PC if AC =0}
\end{align*}

The problem with this method is that the original content of the AC is destroyed. So the proper way to do this is to save AC into a temporary register (called TEMP) added to the pseudo-CPU datapath. Then the sequence of microoperations is given by:

\textbf{Execute Cycle}

\begin{align*}
\text{Cycle 1: } & \text{MDR} \leftarrow \text{M}[\text{MAR}], \text{TEMP} \leftarrow \text{AC} \quad ; \quad \text{Read M}[Y] \text{ from memory} \\
\text{Cycle 2: } & \text{AC} \leftarrow \text{MDR} \quad ; \quad \text{Transfer M}[Y] \text{ into AC and save the contents of AC} \\
\text{Cycle 3: } & \text{AC} \leftarrow \text{AC} + 1 \quad ; \quad \text{Increment M}[Y] \\
\text{Cycle 4: } & \text{MDR} \leftarrow \text{AC} \quad ; \quad \text{Transfer M}[Y]+1 \text{ into MDR and restore AC} \\
\text{Cycle 5: } & \text{M}[\text{MAR}] \leftarrow \text{MDR}, \text{AC} \leftarrow \text{TEMP}, \quad \text{If (AC=0) Then } \text{PC} \leftarrow \text{PC} + 1 \quad ; \quad \text{Store the M}[Y]+1 \text{ back into memory, and at the same time increment PC if AC =0}
\end{align*}

(c) \textbf{DCA} \ Y \quad ; \quad \text{M}[Y] \leftarrow \text{AC}, \text{AC} \leftarrow 0

\textbf{Execute Cycle}

\begin{align*}
\text{Cycle 1: } & \text{MDR} \leftarrow \text{AC} \quad ; \quad \text{Transfer contents of AC to MDR} \\
\text{Cycle 2: } & \text{M}[\text{MAR}] \leftarrow \text{MDR}, \text{AC} \leftarrow 0 \quad ; \quad \text{Store AC into memory location M}[Y], \text{and at the same time clear AC}
\end{align*}

(d) \textbf{JMS} \ Y \quad ; \quad \text{M}[Y] \leftarrow \text{PC}, \text{PC} \leftarrow Y + 1

\textbf{Execute Cycle}

\begin{align*}
\text{Cycle 1: } & \text{MDR} \leftarrow \text{PC} \quad ; \quad \text{Transfer contents of PC to MDR} \\
\text{Cycle 2: } & \text{M}[\text{MAR}] \leftarrow \text{MDR}, \text{PC} \leftarrow \text{MAR} \quad ; \quad \text{Store PC into memory and at the same time transfer contents of MAR to PC} \\
\text{Cycle 3: } & \text{PC} \leftarrow \text{PC} + 1 \quad ; \quad \text{Increment PC to point to Y+1}
\end{align*}
Consider the following hypothetical 1-address assembly instruction called "Store Accumulator Indirect with Pre-Decrement" of the form

\[
\text{STA } -(x) \quad ; M[x] \leftarrow M[x]-1, M[M[x]] \leftarrow AC,
\]

Suppose we want to implement this instruction on the pseudo-CPU discussed in class augmented with a temporary register TEMP. An instruction consists of 16 bits: A 4-bit operation code and a 12-bit address. All operands are 16 bits. PC and MAR each contain 12 bits. AC, MDR, and TEMP each contain 16 bits, and IR is 4 bits. Give the sequence of microoperations required to implement the Execute cycles for the above STA -(x) instruction. Your solution should result in minimum number of microoperations. Assume PC is currently pointing to the STA instruction and only PC and AC have the capability to increment/decrement itself. Fetch cycle is given below.

### Fetch Cycle

Cycle 1: \( \text{MAR} \leftarrow \text{PC}; \)
Cycle 2: \( \text{MDR} \leftarrow \text{MMAR}, \text{PC} \leftarrow \text{PC+1} \) ; Read inst. & increment PC
Cycle 3: \( \text{IR} \leftarrow \text{MDR}_{\text{opcode}}, \text{MAR} \leftarrow \text{MDR}_{\text{address}} \) ;

**Solution:**

Here is one possible solution.

### Execute Cycle

Cycle 1: \( \text{MDR} \leftarrow M[\text{MAR}] \) ; Get EA+1 from memory (i.e., \( M[x] \))
Cycle 2: \( \text{TEMP} \leftarrow \text{AC} \) ; Save AC to TEMP
Cycle 3: \( \text{AC} \leftarrow \text{MDR} \) ; Decrement EA+1
Cycle 4: \( \text{AC} \leftarrow \text{AC}-1 \) ;
Cycle 5: \( \text{MDR} \leftarrow \text{AC} \) ; Store EA into \( M[x] \)
Cycle 6: \( M[\text{MAR}] \leftarrow \text{MDR} \) ;
Cycle 7: \( \text{AC} \leftarrow \text{TEMP} \) ; Restore AC
Cycle 8: \( \text{MAR} \leftarrow \text{MDR} \) ; Have MAR point to EA
Cycle 9: \( \text{MDR} \leftarrow \text{AC} \) ; Store content of AC into \( M[M[x]]=M[\text{EA}] \)
Cycle 10: \( M[\text{MAR}] \leftarrow \text{MDR} \) ;

Note that it is also possible to perform steps 1-2 and 6-7 at the same time.
4- Based on the initial register and data memory contents shown below (represented in hexadecimal), show how these contents are modified (in hexadecimal), including N, Z, and C bits of the Status register (SREG), after executing each of the following AVR assembly instructions. Instructions are unrelated.

(i) MOV R1, R28
(ii) LD R4, Y+
(iii) LDI R4, 33
(iv) MUL R2, R3
(v) ROL R3

Solution:

(i) R28 (which is also the lower 8 bits of the Y register) contains $02$. Thus, R1 changes to $02$. No flags are modified.

(ii) Effective address is $0102$, thus M[Y]=$35$. N, Z, and C flags are not modified. R4 changes to $35$ and Y changes to $0103$.

(iii) LDI instruction only allows R31-R16 as the destination. Thus, this instruction is not allowed.

(iv) Since $1B \times 07 = 27 \times 07 = 189 = $BD
Since High byte of the result is stored in R1 and low byte of the result is stored in R0,
R1 changes to 00 and R0 changes to $BD_{16}$. C-bit is cleared since the 15th bit is zero, and Z-bit is cleared since the result is non-zero, i.e., SREG = 0b11111100 = $FC$. N-bit is not modified.

(v) Since C-bit = 1 (0th bit), so R3 = 0b00001111 = $0F$. N-bit is cleared since the result is positive (i.e., MSB of the result is zero), Z-bit is cleared since the result is non-zero, and C-bit is cleared since MSB of the R3 before the shift was zero. Thus, SREG = 0b11111000 = $F8$.

5- Consider the following code written in AVR assembly. Explain in words what the program accomplishes when it is executed. That is, explain what it does, how it does it, and how many times it does it. What is the value of location CTR when the execution completes?

```
.ORG 0x000F
LDI XH, high(CTR)
LDI XL, low(CTR)
LDI R31, 0xf0
CLR R5
LOOP:
  CLC
  ROL R31
  BRCC SKIP
  INC R5
SKIP:
  CPI R31, 0x00
  BRNE LOOP
  ST X, R5
DONE:
JMP DONE
.DSEG
CTR: .BYTE 1
```
**Solution:**

This program loads the value \( f_0_{16} = 11110000_2 \) and rotates a bit at a time through the carry to determine the number of 1’s. The number of 1’s is stored at location CTR. Since \( 11110000_2 \) has 4 1’s, the value at location CTR will be 4. This is done by

- Loading immediate value \( f_0 \) into R31
- Rotating R31 left through the carry to see if the shifted bit is set, and feed in 0’s.
- If set, increment counter (R5).
- BRNE checks if the loop should terminate. And the program terminates when R31 becomes all 0’s. The number times the loop executes depends on the number and position of 1’s. For the value \( f_0 \), the loop executes 4 times.
- BYTE assembler directive allocates 1 byte for storing the count.