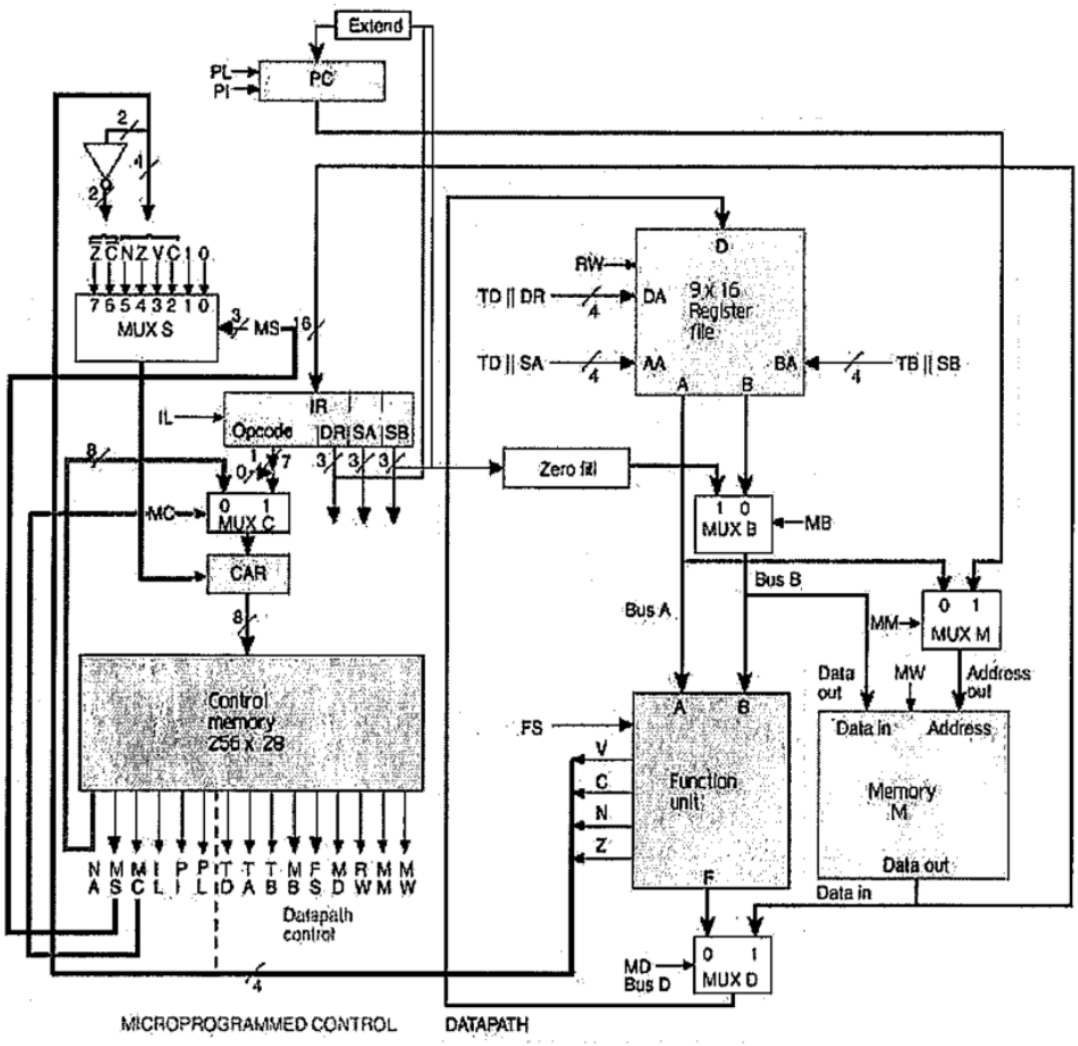
Q1 & Q2 have come up the last 4 years in a row. I don't think he changes the exam much. Please correct and critique these answers as much as possible xxx - C Nev

Q1)a) Explain how the following microprogrammed processor executes machine instructions.



Setup:

When turned on the PC (program counter) and CAR (control address register) are both reset.

The PC specifies the address of the next microprogram in Memory M, while the CAR specifies the address of the next set of machine code instructions in Control Memory. The PI signal increments the value of the PC, and the PL signal uses DR || SB (concatenated) as the next address.

When the PC is reset it points to the first microprogram in memory. Memory contains the addresses of the instructions that are stored in control memory. For example, instruction 0x002E may correspond to the memory address in Control Memory of the instruction ADI. The address accessed is determined by the PC.

Execution:

This microprogram (instruction) address is loaded into the IR (Instruction Register), where it is split into a 7-bit opcode (the first 7 bits) that corresponds to the instruction to be accessed in Control Memory, and three 3-bit values (the last 9 bits) that can either be used for register selection or as values (for immediate values or PC offset). They decide the Destination Register.

Opcode	DA	SA	SB
7 bit	3 bit	3 bit	3 bit

For example

```
ADD r0, r1, r2 =
```

0000011	000	001	010

```
= 0000011000001010    = 0x060A
```

The opcode is loaded into MUX C, and, as the control memory is in the Instruction Fetch State, it is loaded from MUX C into the CAR and then used to select the next set of machine code in the Control Memory. The CAR determines what the next address the Control Memory uses to access the next instruction.

The Control Memory accesses the instruction to be accessed (as dictated by CAR) based on the instruction address based in. An instruction could take more than one clock cycle (e.g Load).

The Control Memory controls the inputs to all of the multiplexers, Function Unit, Register File, PC, IR, CAR and Memory.

Values coming from either the output of the Function unit or Memory (as selected by MUX D) can be loaded into registers in the Register File if RW (Read=0/Write=1) is set to high.

The Register File contains an array of registers which holds data. A temporary register (R8) is also here for multi-cycle instructions so that the data the user may be storing is not disrupted.

The destination register is selected by DR coming from IR, or TD (for the temp register) coming from Control Memory.

The Function Unit takes in values from the Register File (selected by SA, SB, TA or TB) or immediate values and perform arithmetic logic or shifting operations on those values. It contains an ALU & a Shifter. The status bits from the Function Unit are used for conditional instructions in the microprogrammed control.

Values from the Register File can also be stored in Memory M if MW(Memory Write) is set.

The processor is pipelined so that a number of operations can be happening simultaneously, i.e. instruction fetching, execution of a previous instruction and writing the results to the Register File.

Q1)b) How would you microcode a conditional branch instruction?
If a branch instruction was run the PC would not be incremented. Instead the current PC would be offset by some value (the value from the 'Extend' unit) to jump somewhere else in main memory. In this case PL would be set in place of PI. Otherwise, an unconditional branch instruction and a fetch next instruction instruction would have similar Microprograms.

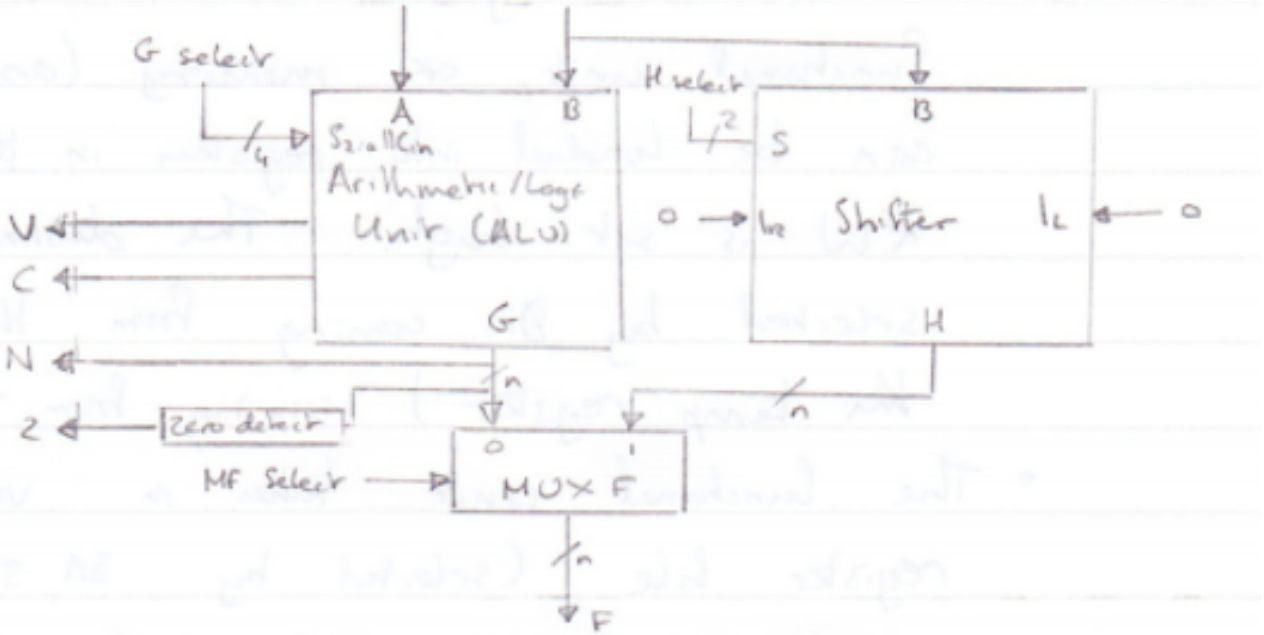
In the case of a conditional branch, the MS select value would depend on the status flag being selected. For example for a branch if equal (BEQ) instruction, MS would be 100 to select the Z flag. The value of the Z flag would then decide whether the CAR just increments (if Z = 0) or if it loads the unconditional branch microprogram (if Z = 1). If the CAR does just increment, the following control word would be the fetch next instruction microprogram.

i.e the two control word sequence for conditional branch would be:

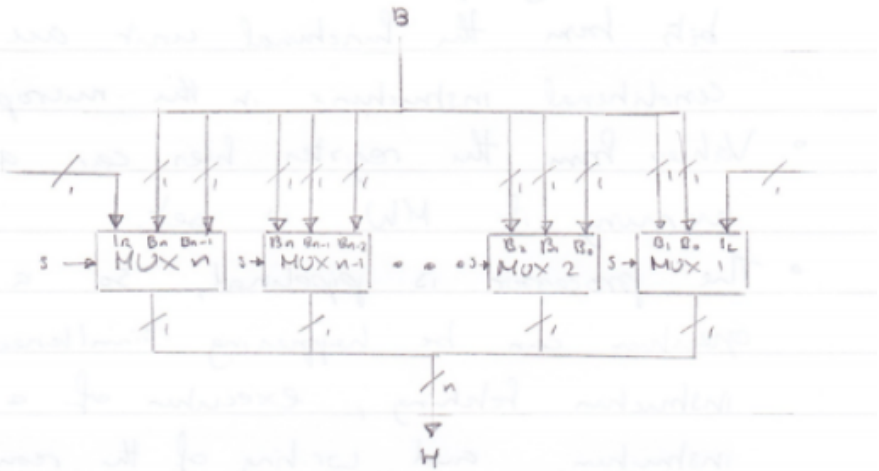
1. Go to unconditional branch microprogram (if Z = 1)
   else go to next control word (if Z = 0)
2. Go to 'fetch next instruction' microprogram

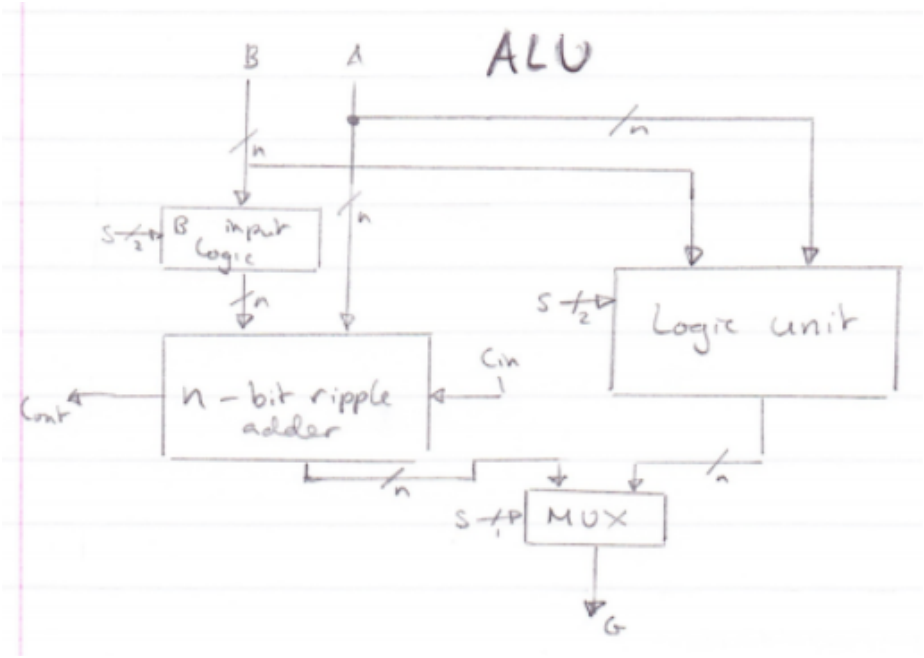
Q2) I'm not sure how low level Mankze wants us to go here but this answer goes all the way down to full adders.

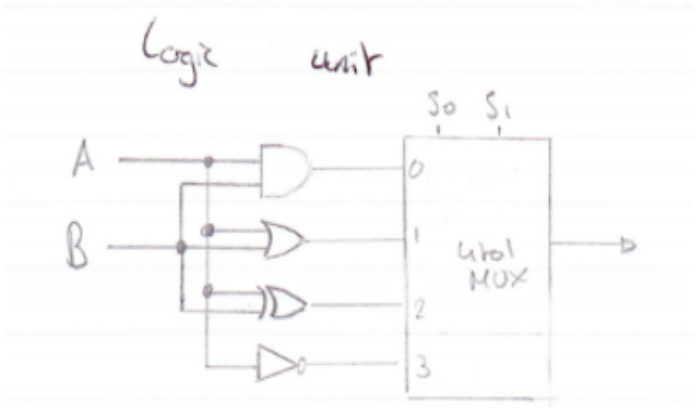
Function Unit

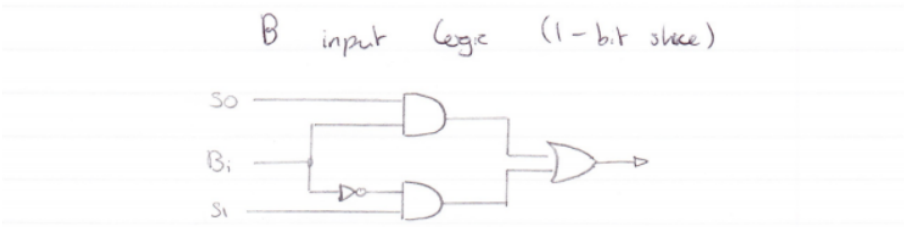


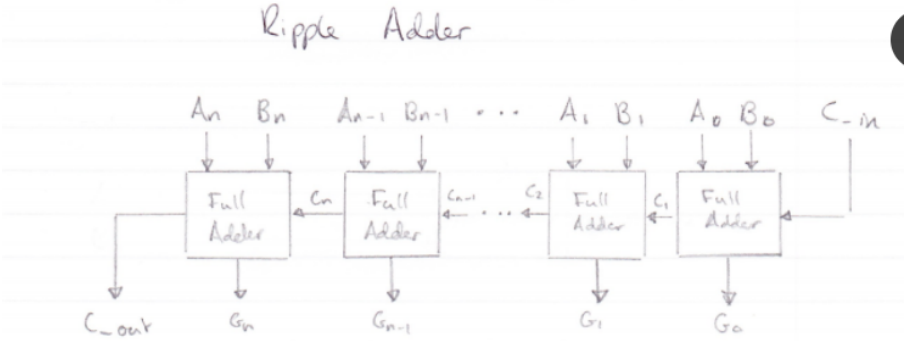
Shifter

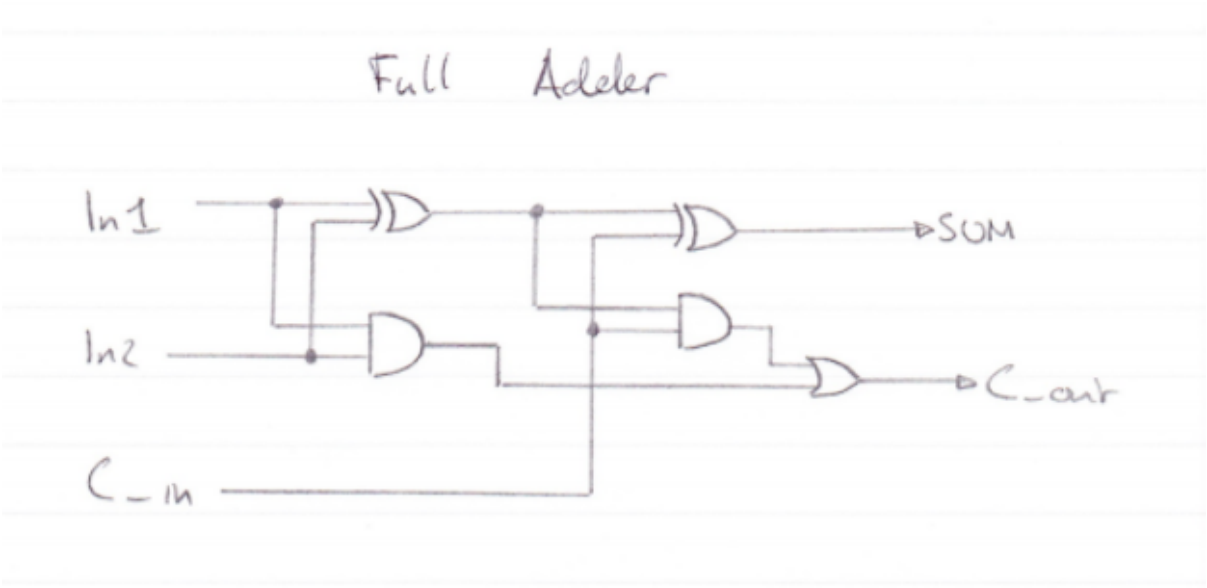



ALU
B
A
n
S
2
B input Logic
n
S
2
Logic unit
Cin
Cout
n - bit ripple adder
n
n
S
1
MUX
G

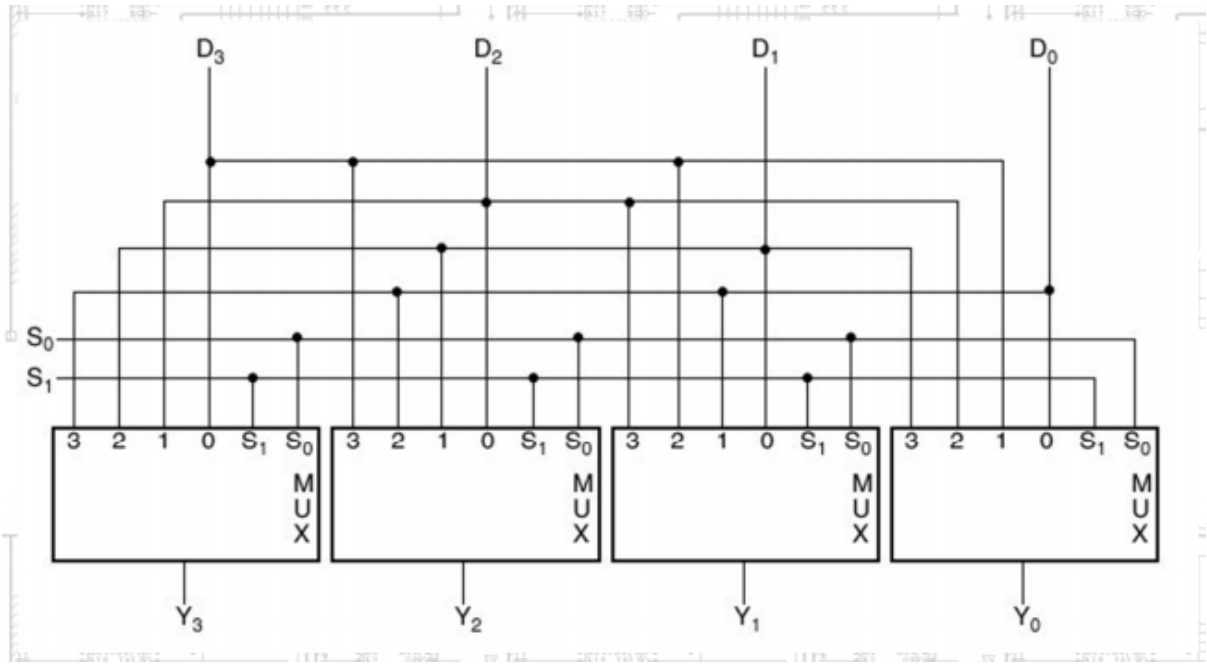

Logic unit
S0 S1
A
B
0
1
2
3
4to1 MUX


B input Logic (1-bit slice)
S0
Bi
Si


Ripple Adder
An Bn
An-1 Bn-1
A1 B1
A0 B0
C_in
Full Adder
Cn
Cn-1
C2
C1
C_out
Gn
Gn-1
G1
G0



b) Schematic:



Implementation:

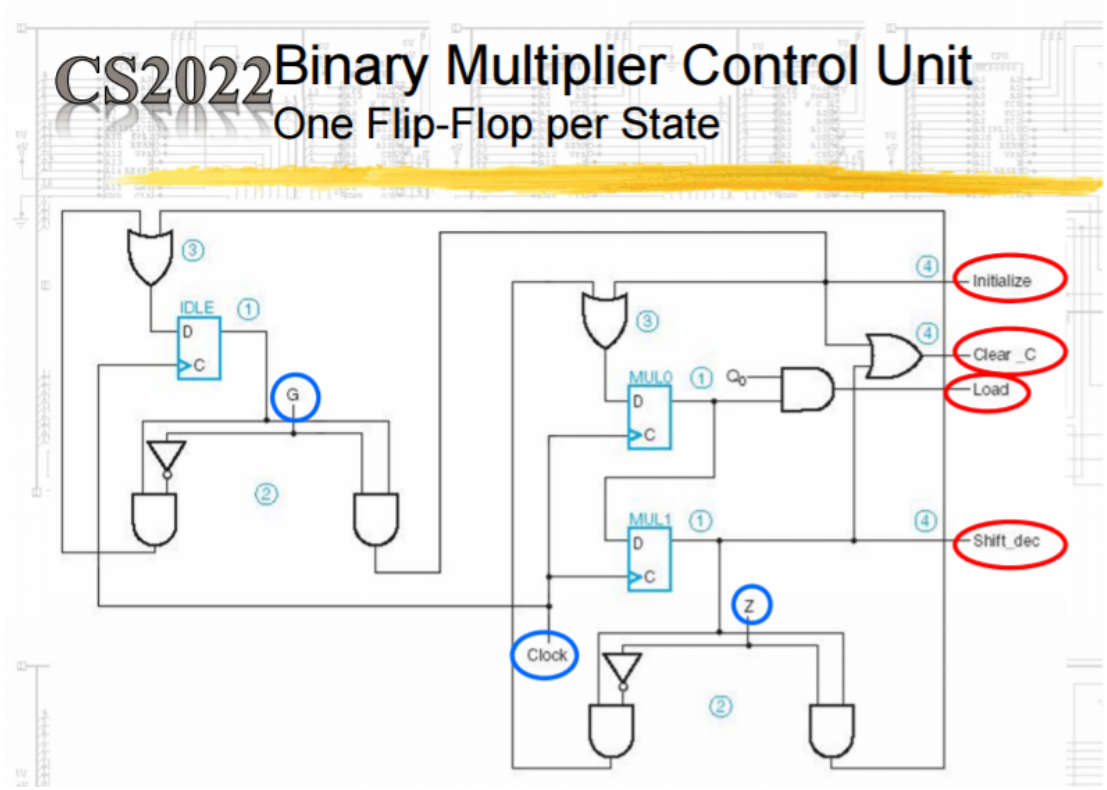
Not sure if this is asking for the code r wha.

Barrel Shifters are able to perform multiple shifts. It is implemented by supplying signal through the 4 line labelled D on the above diagram. The below table details the way in which multiple shift may be performed using this barrel shifter.

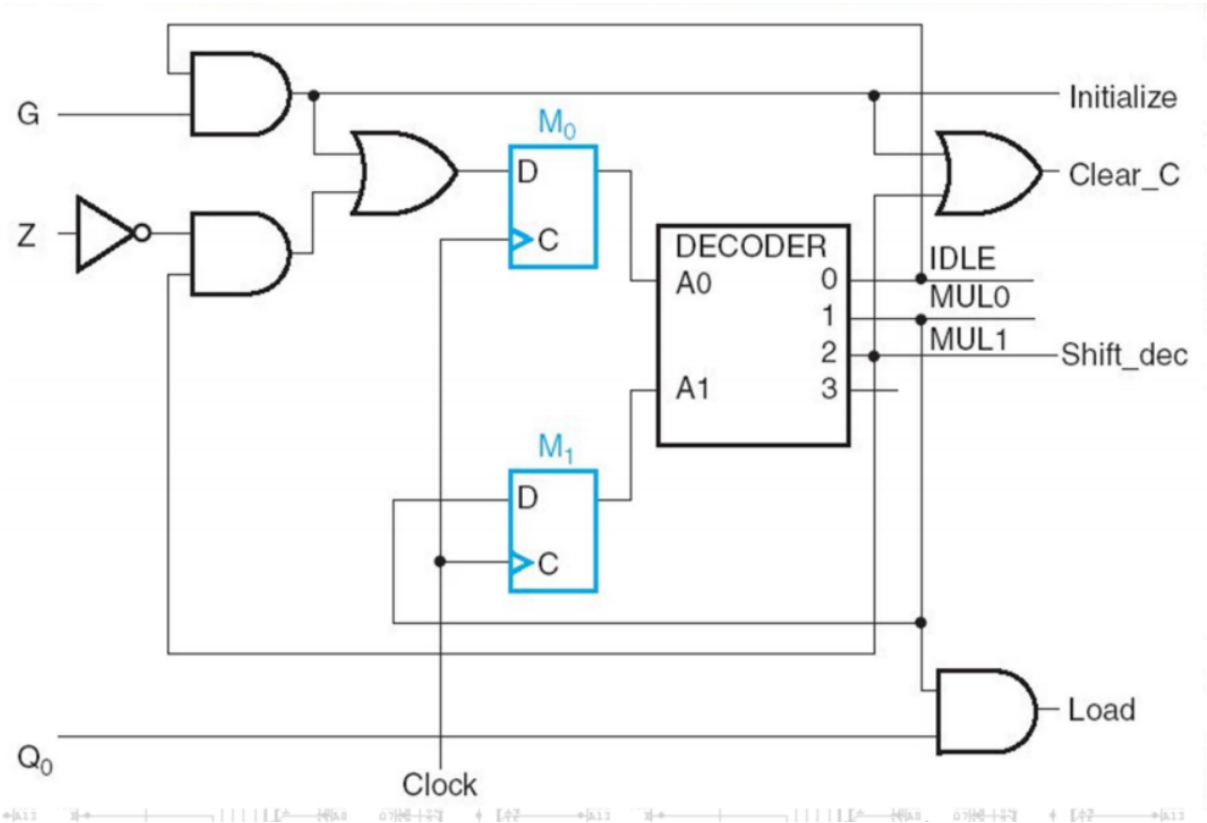
S_1 S_2	Y_3 Y_2 Y_1 Y_0	Micro-ops
0 0	D_3 D_2 D_1 D_0	No Rotate
0 1	D_2 D_1 D_0 D_3	Rotate One
1 0	D_1 D_0 D_3 D_2	Rotate Two
1 1	D_0 D_3 D_2 D_1	Rotate Three

Question 3

(a)



b)



(Additionally the VHDL code for a binary multiplier, i think it's more likely since he either asks code or schematic and asked schematic in 2016)

```
-- Binary Multiplier with n = 4: VHDL Description
-- See Figures 8-6 and 8-7 for block diagram and ASM Chart
-- in Mano and Kime
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity binary_multiplier is
      port(CLK, RESET, G, LOADB, LOADQ: in std_logic;
      MULT_IN: in std_logic_vector(3 downto 0);
      MULT_OUT: out std_logic_vector(7 downto 0));
end binary_multiplier;
architecture behavior_4 of binary_multiplier is
            type state_type is (IDLE, MUL0, MUL1);
            signal state, next_state : state_type;
            signal A, B, Q: std_logic_vector(3 downto 0);
            signal P: std_logic_vector(1 downto 0);
            signal C, Z: std_logic;
      begin
            Z <= P(1) NOR P(0);
            MULT_OUT <= A & Q;

      state_register: process (CLK, RESET)
      begin
            if (RESET = '1') then
                  state <= IDLE;
            elsif (CLK'event and CLK = '1') then
                  state <= next_state;
            end if;
      end process;

      next_state_func: process (G, Z, state)
      begin
            case  state is
                  when IDLE =>
                        if G = '1' then
                              next_state <= MUL0;
                        else next_state <= IDLE;
                        end if;
                  when MUL0 =>
                        next_state <= MUL1;
                  when MUL1 =>
                        if Z = '1' then
                              next_state <= IDLE;
                        else
                              next_state <= MUL0;
```

```
                    end if;
            end case;
      end process;

      datapath_func: process (CLK)
      variable CA: std_logic_vector(4 downto 0);
      begin
            if (CLK'event and CLK = '1') then
                  if LOADB = '1' then
                        B <= MULT_IN;
                  end if;
                  if LOADQ = '1' then
                        Q <= MULT_IN;
                  end if;
                  case state is ]
                  when IDLE =>
                        if G = '1' then
                              C <= '0';
                              A <= "0000";
                              P <= "11";
                        end if;
                  when MUL0 =>
                        if Q(0) = '1' then
                                    CA := ('0' & A) + ('0' & B);
                              else
                                    CA := C & A;
                              end if;
                              C <= CA(4);
                              A <= CA(3 downto 0);
                  when MUL1 => C <= '0';
                  A <= C & A(3 downto 1);
                  Q <= A(0) & Q(3 downto 1);
                  P <= P - "01";
            end case;
            end if;
      end process;
end behavior_4
```

(Memorizing that. Ruff.)