Sample exam subject

G.1. Design an 8 bit parallel printer interface connected to an ATmega64 microcontroller (to its ports). The transfer protocol should use 2 handshaking signals: STB output (informing the printer that data is available on the data lines) and BUSY or ACK (input) informing the microcontroller that the printer is either busy or is ready to receive new data. Explain the design. Draw the schematic of the interface. Draw the flow-chart of the procedure that sends a string of characters to the printer. Also write the procedure in assembly language.
G.2. Explain the principles of the input-caption function available through the AVR timer and comparator.
G.3. Design an ***EPROM*** memory interface for a 8086µP (**having a 16 bit data bus**). Total size: **64 KB**, mapped in the **lower part** of the memory space. **Use EPROM chips: 4K x 4 bits**. The interface design should contain: the memory chips layout, the address decoder, the data, address and control lines interconnection. Explain the design
G.4. Explain how are generated the control signals of an 8086 microprocessor in maximum mode.
G.5. Explain the interrupt response sequence (what happens when an interrupt request occurs).
G.6. What is the “daisy chain technique” and give 2 examples in which can be used ?

The solution is presented in the next pages …..

[redacted]

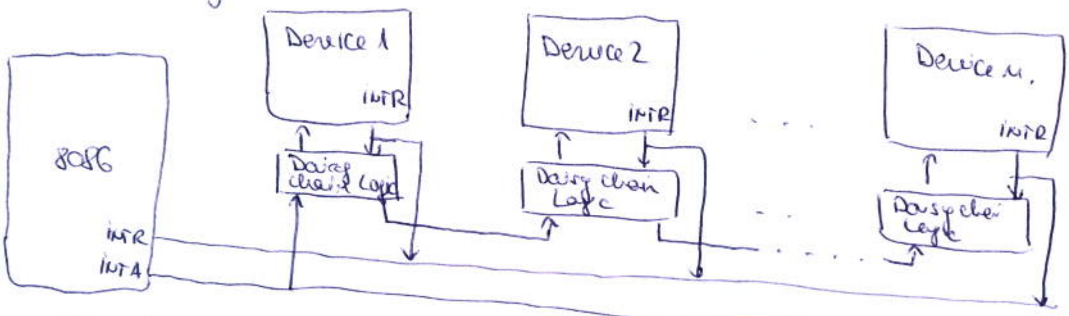
10

1p

G

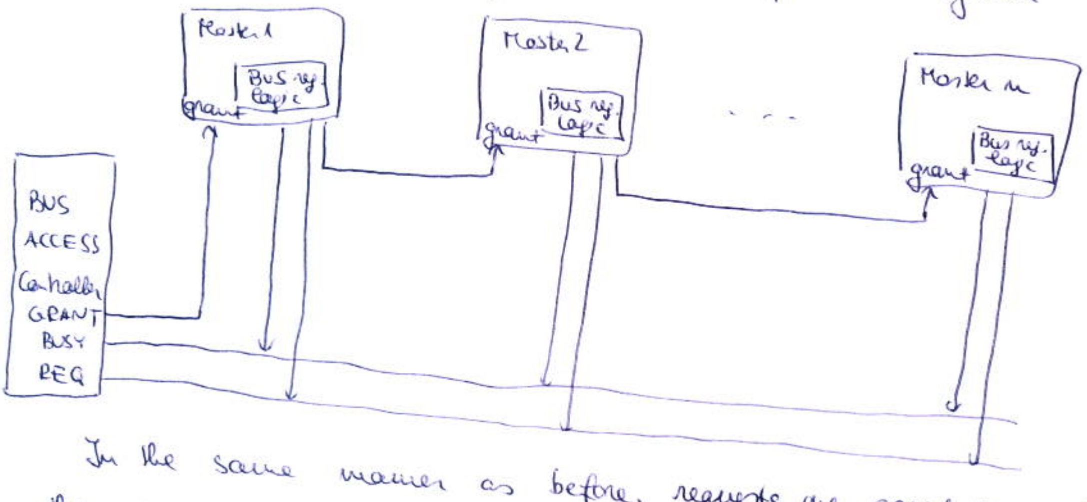
G6. Daisy chaining technique is a technique to resolve priority conflicts among simultaneous requests. It consists of the fact that the devices ~~requesting same feature~~ are put on a "chain" and the first one ~~to~~ on the chain that requests a feature will have it. It can be used in the following examples:

1. Interrupt requests.



Here the INTA signal will travel through the daisy chain logic. The first device requesting interrupt will have the acknowledge. (i.e. priority is from Device 1 → device m decreasing)

2. Bus request on a loosely coupled multiprocessor system



In the same manner as before, requests are asserted on the request bus, and the access grant will go through all masters until the first one that is requesting. Priority is decreasing from Master 1 → Master n.

1p

G5. The interrupt response sequence will include the following steps:

A. In the case of 8086

- flags are pushed on the stack
- interrupt and trap flags are cleared (IF and TF in ~~SREG~~ FLAG REGister)
- the code segment address (CS) and instruction pointer (IP) are pushed on stack
- the new CS and IP are set according to the contents of the interrupt vector table.

INT m has the entry at address 4*m. (4 bytes, 2 for CS and 2 for iP).

- the routine is executed
- on return the CS, iP and flags are popped from register. (as flags were pushed before clearing IF and TF, they do not need to be reset)

B. In the case of AVR (ATmega64)

- the response takes at least 4 cycles:
- PC is pushed on stack
- jump is made according to interrupt vector table
- interrupt flag is cleared from SREG (CLi)
- if the interrupt is a wakeup call, a 4 more cycle is introduced
- Upon return:
- pop PC from stack
- increment stack pointer by 2 (in fact, this is part of a pop)
- re-enable interrupts (SEi)

~~In addition~~

1p

G4. The control signals of an 8086 microprocessor in maximum mode are generated using a bus controller (8288 device). In maximum mode, 8086 has $S_2 - S_0$ status bits as outputs. The 8288 takes these outputs, and based on their combinations generates the needed control signals. These signals are: $\overline{IORC}$, $\overline{IOWC}$, $\overline{MEMRC}$, $\overline{MEMWC}$, $\overline{AIORC}$, $\overline{AMEMRC}$, $\overline{DEN}$, $DT/\overline{R}$, ALE

Note: the latter 3 signals are not obvious from their name, so I explain the usage: $\overline{DEN}$ ~~is the enable~~ is connected to the enable pin of the transceivers. This will enable data go through them.

$DT/\overline{R}$ (data transmit / $\overline{receive}$) is connected to the direction pin of the transcievers. it indicates if 8086 transmits or receives data.

ALE - (address latch enable) - enables addresses from latch to address bus.
↳ connected to the enable of the address latches.

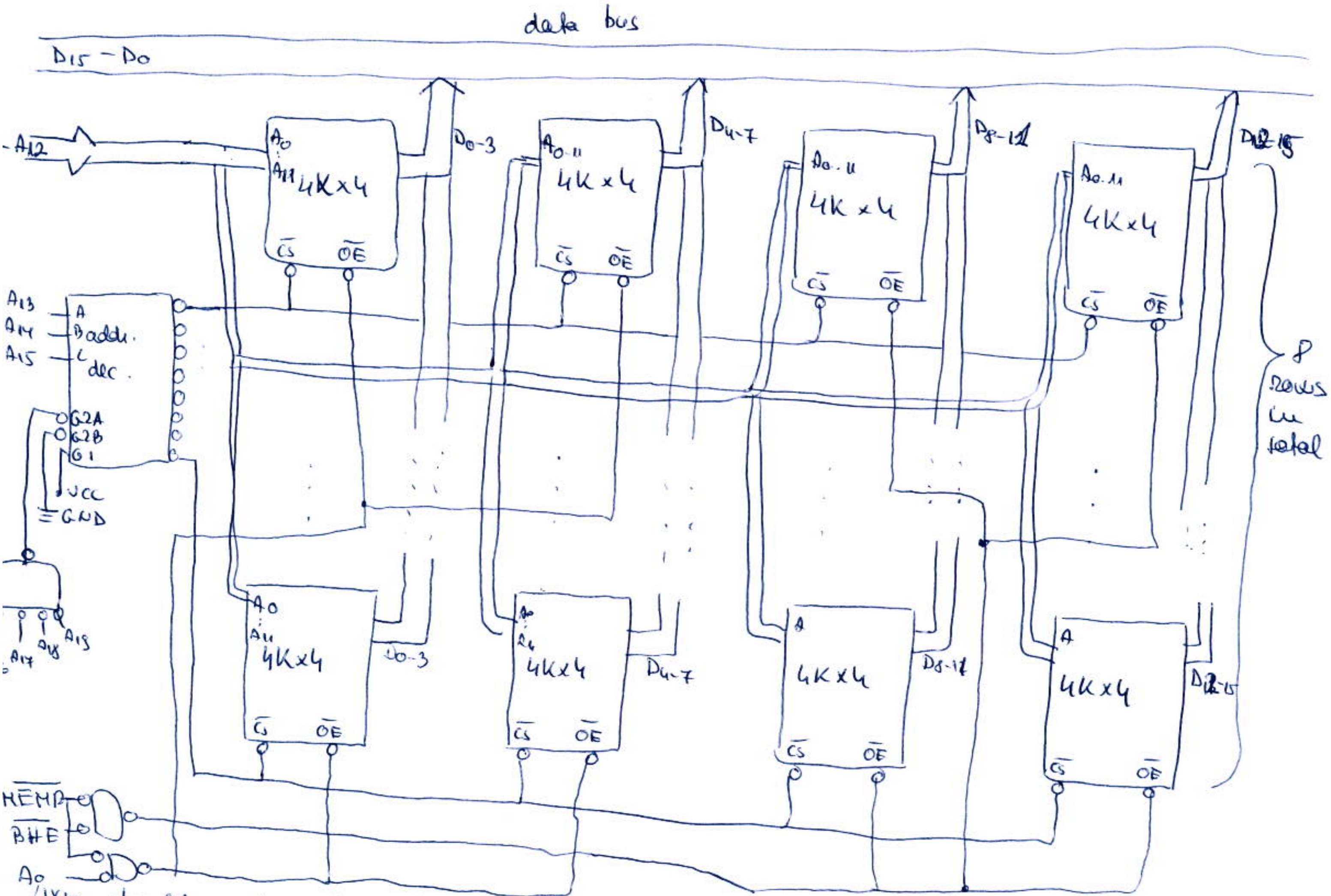
G3. EPROM memory interface design: 2p

we need a total size of 64KB = 64K × 8 bits, we have 4K × 4 bits chips => we need

$$\frac{64K \times 8}{4K \times 4} = 16 \times 2 = 32 \text{ chips}$$

We have EPROM, so we need to take care only of the read operations, as write operations does not occur.

For the 8086 we have a 16 bit data bus. For this reason, we have to connect 4 such chips in a row to ensure this width. (In fact it will be parallel connection, because, even though the width is enlarged, we should use separate control lines for accessing low and high bytes).



(We should also be able to address individual chips on a row. For this purpose I reserve not only $\overline{BHE}$ and A_0, but A_1 as well. (8086 will access only on bytes, but the correct design takes into account this feature as well.)

We have 4K blocks. Thus we need 12 address lines to address the contents of a block. We use $A_1 - A_{12}$ for this purpose. Other address lines (such as $A_{16} - A_{19}$) are designed, so that the memory is in the lower part of the address space. (Thus, these should be 0). The address-lines $A_{13} - A_{15}$ are used to select the rows.

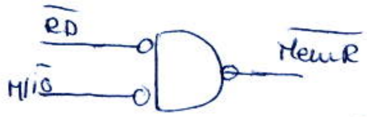
So: A_0 and $\overline{BHE}$ are used to select low/high byte or all word.

$A_1 - A_{12}$ are used for addressing inside one block of 4K.

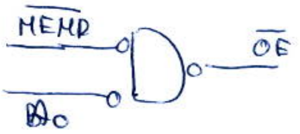
A_{13-15} - used for addressing the 8 rows.

A_{16-19} - set to 0. (to map into lower part of memory).

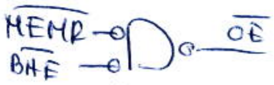
For selecting low bank we activate A_0 (active low) for high bank BHE, for both we activate both. In addition we need the $\overline{MEMR}$ control signal which comes from $M/\overline{IO}$ and $\overline{RD}$:



For Low bank the $\overline{OE}$ signal is connected to:



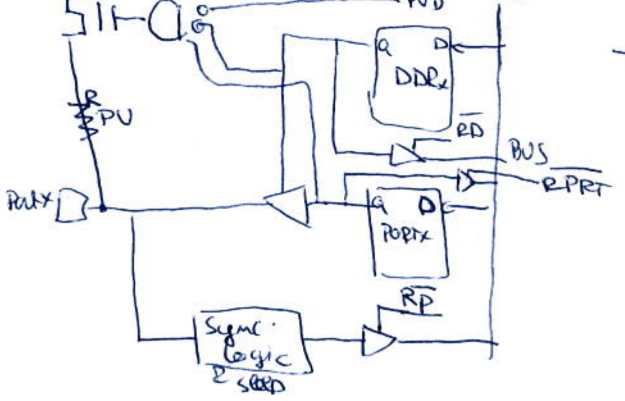
For High bank $\overline{OE}$ is connected to



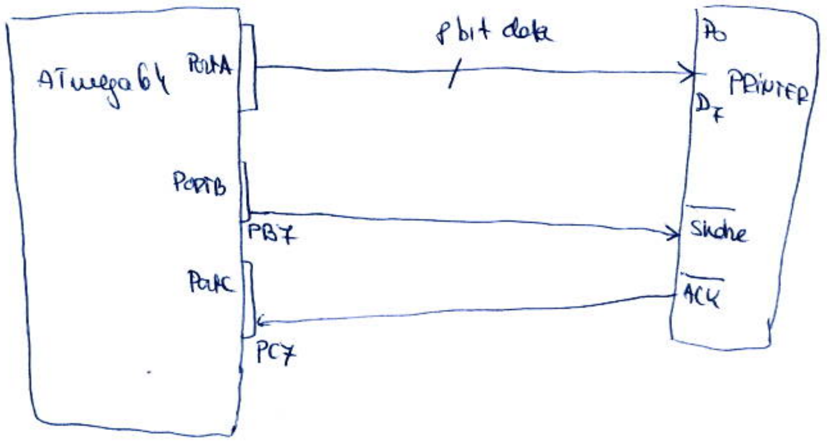
Note: 8086 will address only bytes. But in the hypetetical case that we want to access the banks (all 4 columns) separately we use $A_0 A_1$ and BHE for selecting the High/Low bank as before and A_1 for selecting the 4 bits in the byte, and the other addresses are shifted (i.e. we use $A_2 .. A_{13}$ and $A_{14} .. A_{16}$ for addressing blocks and for decoding rows.)

2,5P

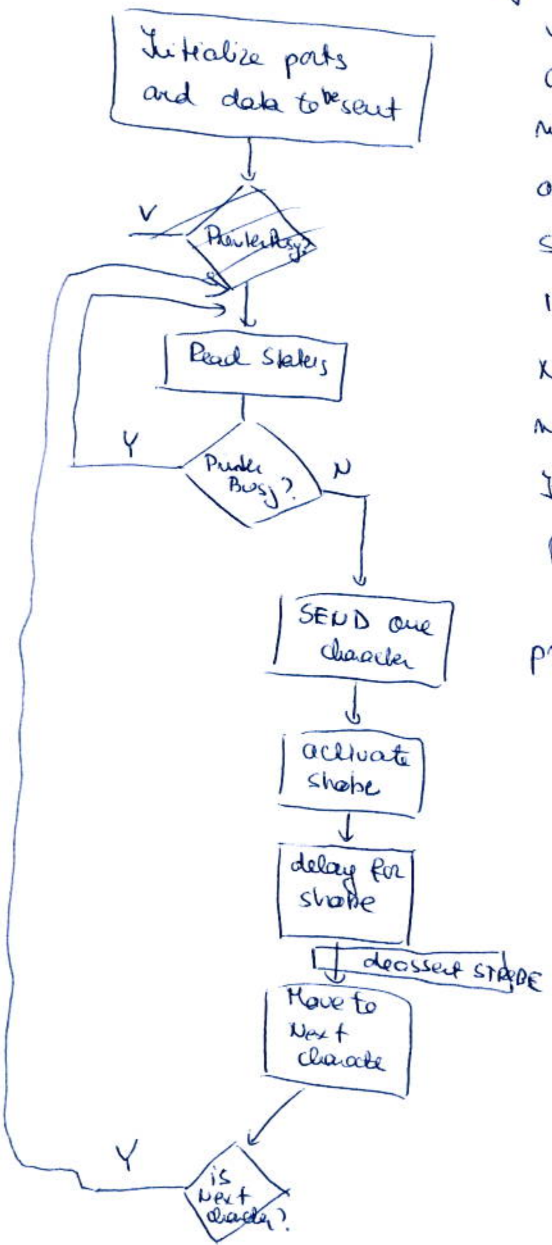
G1. We need an 8 bit port for sending data, and ~~a~~ two control lines: the STB and ACK. Because these are different directions (STB out, ack in) I will use separate ports for them, although we could use the same port, with different ~~ports~~ pins configured differently. ~~For the correct communication we notice that we should use latches for output ports and three-state buffers for input ports.~~ PORTA will be the data port, PORTB will be for strobe, and PORTC for acknowledge. We should note that in case of ATMega64 the Ports have internally their registers (in fact, every pin has its D flip-flop) So we do not need latches for output ports and three state buffers for input ports, as these effects are done automatically by the structure of the port of an ATmega64). The schematic will be then: (next page)



→ this is the schematic of a port, we can see that we have flipflops for Portx, DDRx and have a three state buffer for the PINx, so we do not need to explicitly introduce these into the design (as it would be the case when using 8086.)



The flow chart is the following:



when the printer is busy we poll (wait for the acknowledge to be low i.e. - data was read, and new data can come.) We send the data, and activate strobe signal. We wait for some time so that strobe is read by printer (we know that it is not busy, so it should be a little amount). Next we deassert strobe, and ~~wa~~ move to the next character. If no next character, we are done. If there is a next character, read again status of the printer and loop.

The assembly language program will be like:

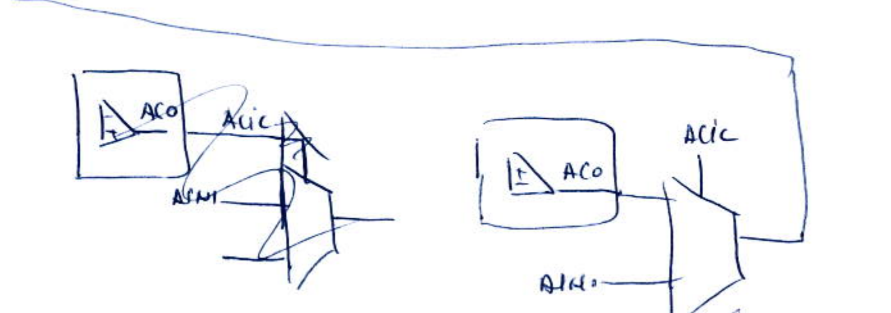
```
print:
        ldi r16, 0xFF
        out DDRA, r16       // A port as output
        out DDRB, r16       // B port as output
        ldi r16, 0x00
        out DDRC, r16       // port C as input.

        ldi Xh, high(2* stringaddress)
        ldi Xl, low(2* stringaddress)    // load into X
                                          the address of the
                                          string to be
                                          written

        ldi r17, charcount       // load to r17 the
                                   number of characters.

// now the procedure itself
poll:
        IN r18, PINC        // read ~~Port~~ pin C
        sbc r18, 7          // skip if bit 7 is cleared i.e.
        rjmp poll              ACK active.
```

```
    // now printer ready, send data
    ld r20, X        // load from memory the character.
    OUT PORTA, r20   // send character.
    ldi r18, 0x00
    OUT PORTB, r18   // out ~~cha~~ strobe active on low
    ldi r19, 0xFF
delay:
        dec r19
        ~~bre~~
        cpi r19,0        } delay loop
        brne delay
    ldi r18, 0x70
    OUT PORTB, r18   } deassert strobe.

    ADDi xl, 0x01
    ~~AD~~ ldi r18, 0    } Move to next address
    ADC xh, r18

    DEC r17          // decrement character count
    cpi r17,0
    brne poll        // if not zero, poll again
ret  - if it was zero, return.
```

1,5

(the 16bit timer!)

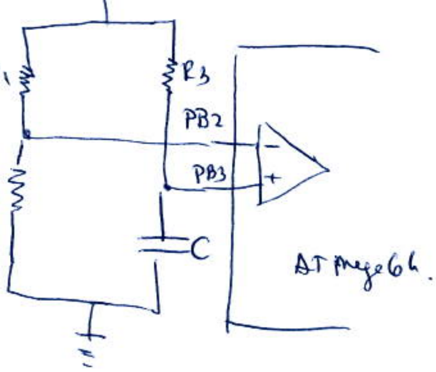
Q2 The AVR Timer has a function called input capture. In this case the contents of the register TCCNTn will be saved in IRC whenever an input capture interrupt is requested. This is used for measuring frequencies, periods or other analog characterstics of a signal. It is used in conjuction with the analog comparator. The following is the opration:



The analog comparator compares the signals that are set by the ACBG and ACME signals. (AIN0 can be external or Bandgap, AIN1 can be external or AD_{0-7}). If the ACIC is set in the ACSR register (Analg comparator Timer capture interupt enable.), on a match (ACO) a timer capture interrupt is requested. This is the interrupt of the Timer1 (16 bit timer) that can be validated in TIMSK register. When the interrupt is ~~requested~~, requested ~~saved~~, the value of TCCNT register is put into the ICR register and ISR is called. From the interrupt service routine ~~subroutine~~ we can read the value and use it for some purpose.

Ex: measuring frequency or even measuring capacity of a condense:

ex: measuring capacity of condenser:



1. configure timer and comparator.
2. let PB3 be output and write 0 to it to discharge the condenser
3. let PB3 be input (just like PB2)
4. start timer
5. On the ISR:
   - read ICR1
   - convert to seconds
   - calculate capacity using the characteristic equation of an RC low pass