Chapter 4 – Input/Output Organization

4.1. After reading the input data, it is necessary to clear the input status flag before the program begins a new read operation. Otherwise, the same input data would be read a second time.

4.2. The ASCII code for the numbers 0 to 9 can be obtained by adding $30 to the number. The values 10 to 15 are represented by the letters A to F, whose ASCII codes can be obtained by adding $37 to the corresponding binary number.

Assume the output status bit is b_4 in register Status, and the output data register is Output.

```
            Move        #10,R0      Use R0 as counter
            Move        #LOC,R1     Use R1 as pointer
Next        Move        (R1),R2     Get next byte
            Move        R2,R3
            Shift-right #4,R3       Prepare bits b7-b4
            Call        Convert
            Move        R2,R3       Prepare bits b3-b0
            Call        Convert
            Move        $20,R3      Print space
            Call        Print
            Increment   R1
            Decrement   R0
            Branch>0    Next        Repeat if more bytes left
            End

Convert     And         #0F,R3      Keep only low-order 4 bits
            Compare     #9,R3
            Branch≥0    Letters     Branch if [R3] ≥ 9
            Or          #$30,R3     Convert to ASCII, for values 0 to 9
            Branch      Print
Letters     Add         #$37,R3     Convert to ASCII, for values 10 to 15
Print       BitTest     #4,Status   Test output status bit
            Branch=0    Print       Loop back if equal to 0
            Move        R3,Output   Send character to output register
            Return
```

4.3. 7CA4, 7DA4, 7EA4, 7FA4.

4.4. A subroutine is called by a program instruction to perform a function needed by the calling program. An interrupt-service routine is initiated by an event such as an input operation or a hardware error. The function it performs may not be at

all related to the program being executed at the time of interruption. Hence, it must not affect any of the data or status information relating to that program.

4.5. If execution of the interrupted instruction is to be completed after return from interrupt, a large amount of information needs to be saved. This includes the contents of any temporary registers, intermediate results, etc. An alternative is to abort the interrupted instruction and start its execution from the beginning after return from interrupt. In this case, the results of an instruction must not be stored in registers or memory locations until it is guaranteed that execution of the instruction will be completed without interruption.

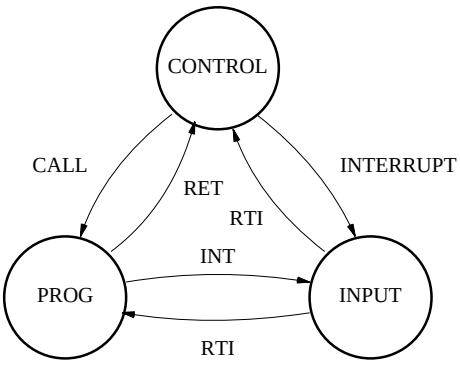
4.6. (*a*) Interrupts should be enabled, except when C is being serviced. The nesting rules can be enforced by manipulating the interrupt-enable flags in the interfaces of A and B.

(*b*) A and B should be connected to INTR_2, and C to INTR_1. When an interrupt request is received from either A or B, interrupts from the other device will be automatically disabled until the request has been serviced. However, interrupt requests from C will always be accepted.

4.7. Interrupts are disabled before the interrupt-service routine is entered. Once device i turns off its interrupt request, interrupts may be safely enabled in the processor. If the interface circuit of device i turns off its interrupt request when it receives the interrupt acknowledge signal, interrupts may be enabled at the beginning of the interrupt-service routine of device i. Otherwise, interrupts may be enabled only after the instruction that causes device i to turn off its interrupt request has been executed.

4.8. Yes, because other devices may keep the interrupt request line asserted.

4.9. The control program includes an interrupt-service routine, INPUT, which reads the input characters. Transfer of control among various programs takes place as shown in the diagram below.



A number of status variables are required to coordinate the functions of PROG and INPUT, as follows.

BLK-FULL: A binary variable, indicating whether a block is full and ready for processing.

IN-COUNT: Number of characters read.

IN-POINTER: Points at the location where the next input character is to be stored.

PROG-BLK: Points at the location of the block to be processed by PROG.

Two memory buffers are needed, each capable of storing a block of data. Let BLK(0) and BLK(1) be the addresses of the two memory buffers. The structure of CONTROL and INPUT can be described as follows.

```
CONTROL   BLK-FULL := false
          IN-POINTER := BLK(0)
          IN-COUNT := 0
          Enable interrupts
          i := 0
          Loop
                Wait for BLK-FULL
                If not last block then {
                     BLK-FULL := false          Prepare to read the next block
                     IN-POINTER := BLK(1 − i)
                     IN-COUNT := 0
                     Enable interrupts }
                PROG-BLK := BLK(i)             Process the block just read
                Call PROG
                If last block then exit
                i := 1 − i
          End Loop

Interrupt-service routine
INPUT:    Store input character and increment IN-COUNT and IN-POINTER
          If IN-COUNT = N Then {
                disable interrupts from device
                BLK-FULL := true }
          Return from interrupt
```

4.10. *Correction: In the last paragraph, change "equivalent value" to "equivalent condition".*

Assume that the interface registers for each video terminal are the same as in Figure 4.3. A list of device addresses is stored in the memory, starting at DEVICES, where the address given in the list, DEVADRS, is that of DATAIN. The pointers to data areas, PNTRn, are also stored in a list, starting at PNTRS.

Note that depending on the processor, several instructions may be needed to perform the function of one of the instructions used below.

POLL	Move	#20,R1	Use R1 as device counter, i
LOOP	Move	DEVICES(R1),R2	Get address of device i
	BitTest	#0,2(R2)	Test input status of a device
	Branch$\neq$0	NXTDV	Skip read operation if not ready
	Move	PNTRS(R1),R3	Get pointer to data for device i
	MoveByte	(R2),(R3)+	Get and store input character
	Move	R3,PNTRS(R1)	Update pointer in memory
NXTDV	Decrement	R1	
	Branch$>$0	LOOP	
	Return		

INTERRUPT	Same as POLL, except that it returns once a character is read. If several devices are ready at the same time, the routine will be entered several times in succession.

In case *a*, POLL must be executed at least 100 times per second. Thus $T_{max} = 10$ ms.

The equivalent condition for case *b* can be obtained by considering the case when all 20 terminals become ready at the same time. The time required for interrupt servicing must be less than the inter-character delay. That is, $20 \times 200 \times 10^{-9} < 1/c$, or $c < 250,000$ char/s.

The time spent servicing the terminals in each second is given by:

Case *a*: Time $= 100 \times 800 \times 10^{-9}$ ns $= 80$ μs
Case *b*: Time $= 20 \times r \times 200 \times 10^{-9} \times 100 = 400r$ ns

Case *b* is a better strategy for $r < 0.2$.

The reader may repeat this problem using a slightly more complete model in which the polling time, P, for case a is a function of the number of terminals. For example, assume that P increases by 0.5 μs for each terminal that is ready, that is, $P = 20 + 20r \times 0.5$.

4.11. (*a*) Read the interrupt vector number from the device (1 transfer).
Save PC and SR (3 transfers on a 16-bit bus).
Read the interrupt vector (2 transfers) and load it in the PC.

(*b*) The 68000 instruction requiring the maximum number of memory transfers is:

MOVEM.L D0-D7/A0-A7,LOC.L

where LOC.L is a 32-bit absolute address. Four memory transfers are needed to read the instruction, followed by 2 transfers for each register, for a total of 36.

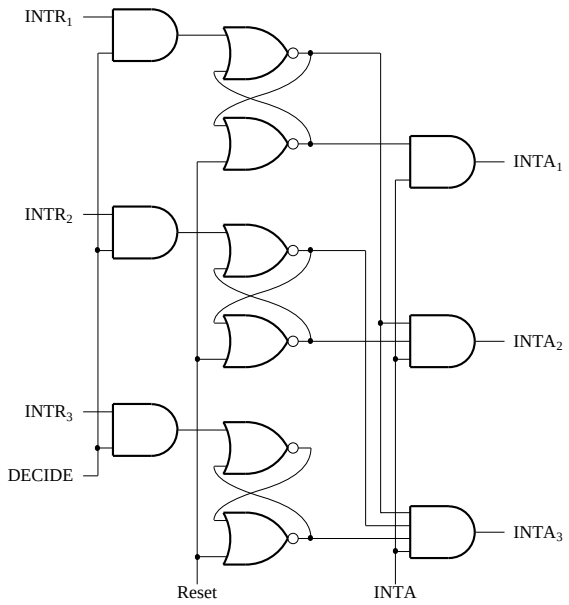
(*c*) 36 for completion of current instruction plus 6 for interrupt handling, for a total of 42.

4.12. (*a*) $\text{INTA1} = \text{INTR1}$
$\text{INTA2} = \text{INTR2} \cdot \overline{\text{INTR1}}$
$\text{INTA3} = \text{INTR3} \cdot \overline{\text{INTR1}} \cdot \overline{\text{INTR2}}$

(*b*) See logic equations in part *a*.

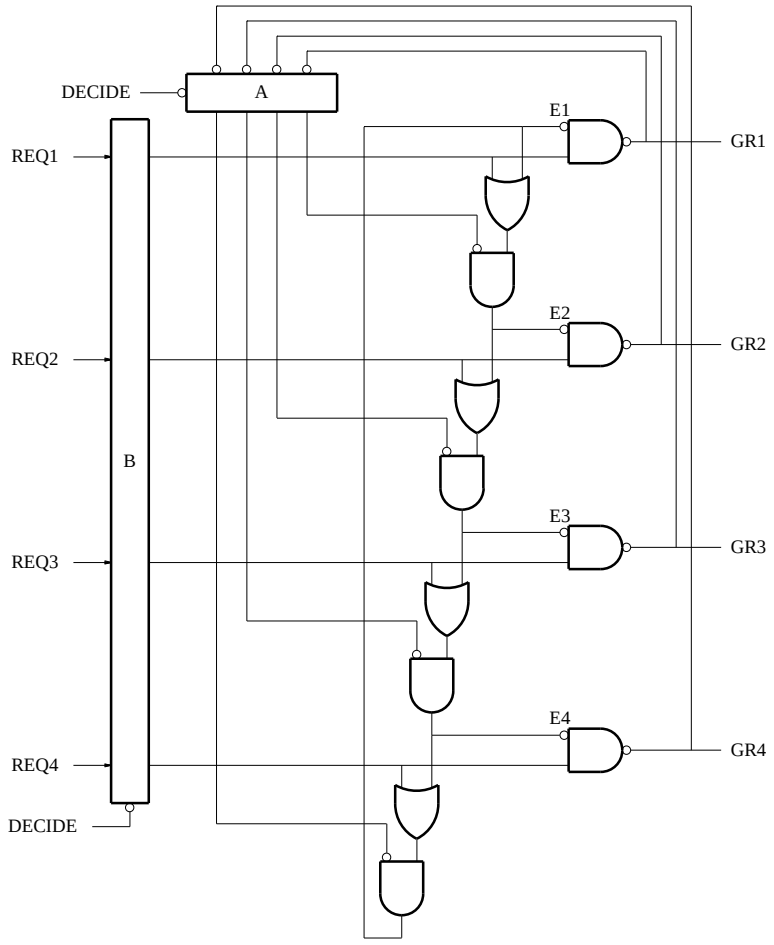
(*c*) Yes.

(*d*) In the circuit below, DECIDE is used to lock interrupt requests. The processor should set the interrupt acknowledge signal, INTA, after DECIDE returns to zero. This will cause the highest priority request to be acknowledged. Note that latches are placed at the inputs of the priority circuit. They could be placed at the outputs, but the circuit would be less reliable when interrupts change at about the same time as arbitration is taking place (races may occur).



4.13. In the circuit given below, register A records which device was given a grant most recently. Only one of its outputs is equal to 1 at any given time, identifying the highest-priority line. The falling edge of DECIDE records the results of the current arbitration cycle in A and at the same time records new requests in register B. This prevents requests that arrive later from changing the grant.

The circuit requires careful initialization, because one and only one output of register A must be equal to 1. This output determines the highest-priority line during a given arbitration cycle. For example, if the LSB of A is equal to 1, point E2 will be equal to 0, giving REQ2 the highest priority.



4.14. The truth table for a priority encoder is given below.

1	2	3	4	5	6	7	IPL_2	IPL_1	IPL_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
x	1	0	0	0	0	0	0	1	0
x	x	1	0	0	0	0	0	1	1
x	x	x	1	0	0	0	1	0	0
x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	1	1	1	1

A possible implementation for this priority circuit is as follows:

$$\mathrm{IPL}_2 = q_4 + q_5 + q_6 + q_7$$

$$\mathrm{IPL}_1 = q_6 + q_7 + \overline{\mathrm{IPL}_2}(q_2 + q_3)$$

$$\mathrm{IPL}_0 = q_7 + q_5 \cdot q_6 + \overline{\mathrm{IPL}_2}(q_3 + q_1 \cdot q_2)$$

4.15. Assume that the interface registers are the same as in Figure 4.3 and that the characters to be printed are stored in the memory.

```
* Program A (MAIN) points to the character string and calls DSPLY twice
MAIN    MOVE.L    #ISR,VECTOR     Initialize interrupt vector
        ORI.B     #$80,STATUS     Enable interrupts from device
        MOVE      #$2300,SR       Set interrupt mask to 3
        MOVEA.L   #CHARS,A0       Set pointer to character list
        BSR       DSPLY
        MOVEA.L   #CHARS,A0
        BSR       DSPLY
        END       MAIN
* Subroutine DSPLY prints the character string pointed to by A0
* The last character in the string must be the NULL character
DSPLY   ...
        RTS
* Program B, the interrupt-service routine, points at the number string and calls DSPLY
ISR     MOVEM.L   A0,-(A7)        Save registers used
        MOVE.L    NEWLINE,A0      Start a new line
        BSR       DSPLY
        MOVEA.L   #NMBRS,A0       Point to the number string
        BSR       DSPLY
        MOVEM.L   (A7)+,A0        Restore registers
        RTE
* Characters and numbers to be displayed
CHARS   CC        /AB ... Z/
NEWLINE CB        $0D, $0A, 0     Codes for CR, LF and Null
NMBRS   CB        $0D, $0A
        CC        /01 ... 901 ... 901 ... 9/
        CB        $0D, $0A, 0
```

When ISR is entered, the interrupt mask in SR is automatically set to 4 by the hardware. To allow interrupt nesting, the mask must be set to 3 at the beginning of ISR.

4.16. Modify subroutine DSPLY in Problem 4.15 to keep count of the number of characters printed in register D1. Before ISR returns, it should call RESTORE, which sends a number of space characters (ASCII code 20_{16}) equal to the count in D1.

```
DSPLY     ...
          MOVE    #$2400,SR        Disable keyboard interrupts
          MOVEB   D0,DATAOUT       Print character
          ADDQ    #1,D1
          MOVE    #$2300,SR        Enable keyboard interrupts
          ...
RESTORE   MOVE.L  D1,D2
          BR      TEST
LOOP      BTST    #1,STATUS
          BEQ     LOOP
          MOVEB   #$20,DATAOUT
TEST      DBRA    D2,LOOP
          RTS
```

Note that interrupts are disabled in DSPLY before printing a character to ensure that no further interrupts are accepted until the count is updated.

4.17. The debugger can use the trace interrupt to execute the saved instruction then regain control. The debugger puts the saved instruction at the correct address, enables trace interrupts and returns. The instruction will be executed. Then, a second interruption will occur, and the debugger begins execution again. The debugger can now remove the program instruction, reinstall the breakpoint, disable trace interrupts, then return to resume program execution.

4.18. (*a*) The return address, which is in register R14_svc, is PC+4, where PC is the address of the SWI instruction.

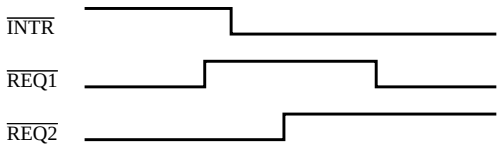
```
LDR   R2,[R14,#-4]          Get SWI instruction
BIC   R2,R2,#&FFFFFF00      Clear high-order bits
```

(*b*) Assume that the low-order 8 bits in SWI have the values 1, 2, 3, ... to request services number 1, 2, 3, etc. Use register R3 to point to a table of addresses of the corresponding routines, at addresses [R3]+4, [R3]+8, respectively.

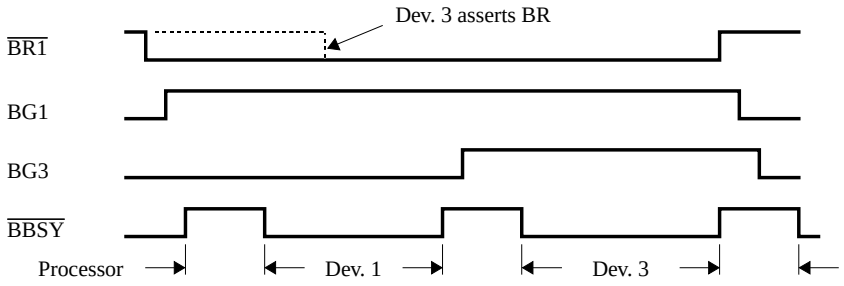
```
ADR   R3,EntryTable          Get the table's address
LDR   R15,[R3,R2,LSL #2]     Load starting address of routine
```

4.19. Each device pulls the line down (closes a switch to ground) when it is not ready. It opens the switch when it is ready. Thus, the line will be high when all devices are ready.

4.20. The request from one device may be masked by the other, because the processor may see only one edge.



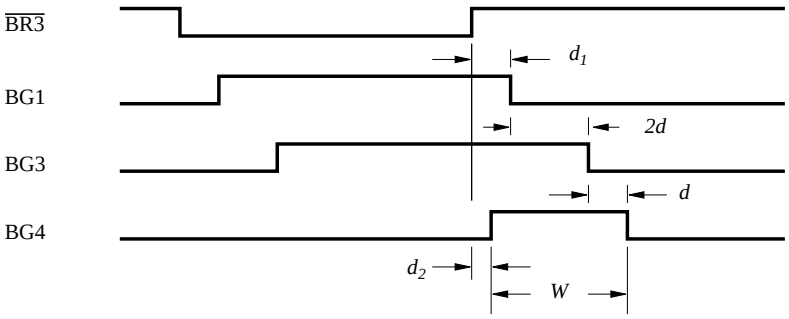
4.21. Assume that when BR becomes active, the processor asserts BG1 and keeps it asserted until BR is negated.



4.22. (*a*) Device 2 requests the bus and receives a grant. Before it releases the bus, device 1 also asserts BR. When device 2 is finished nothing will happen. BR and BG1 remain active, but since device 1 does not see a transition on BG1 it cannot become the bus master.

(*b*) No device may assert BR if its BG input is active.

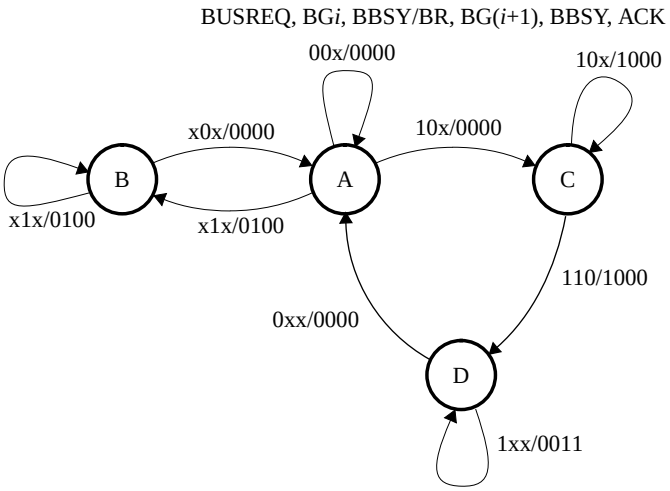
4.23. *For better clarity, change* BR *to* $\overline{\mathrm{BR}}$ *and use an inverter with delay* d_1 *to generate* BG1.



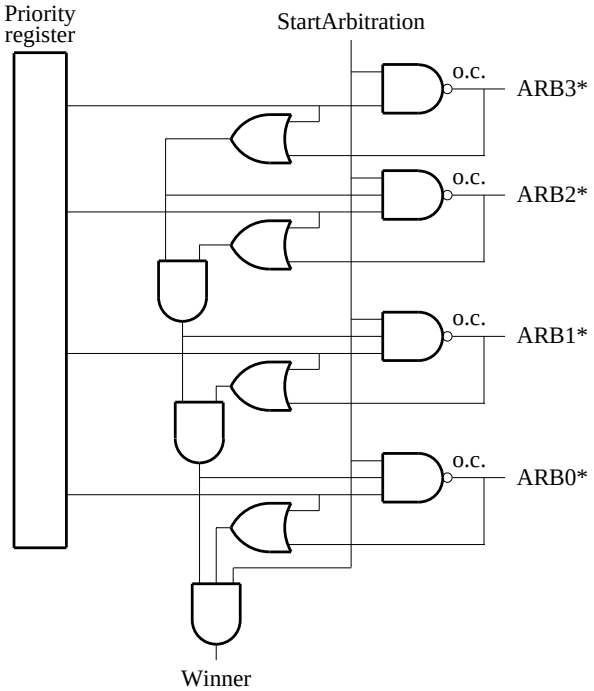
Assuming device 3 asserts BG4 shortly after it drops the bus request (delay d_2), a spurious pulse of width $W = d_1 + 3d - d_2$ will appear on BG4.

4.24. Refer to the timing diagram in Problem 4.23. Assume that both BR1 and BR5 are activated during the delay period d_2. Input BG1 will become active and at the same time the pulse on BG4 will travel to BG5. Thus, both devices will receive a bus grant at the same time.

4.25. A state machine for the required circuit is given in the figure below. An output called ACK has been added, indicating when the device may use the bus. Note that the restriction in Solution 4.22*b* above is observed (state B).



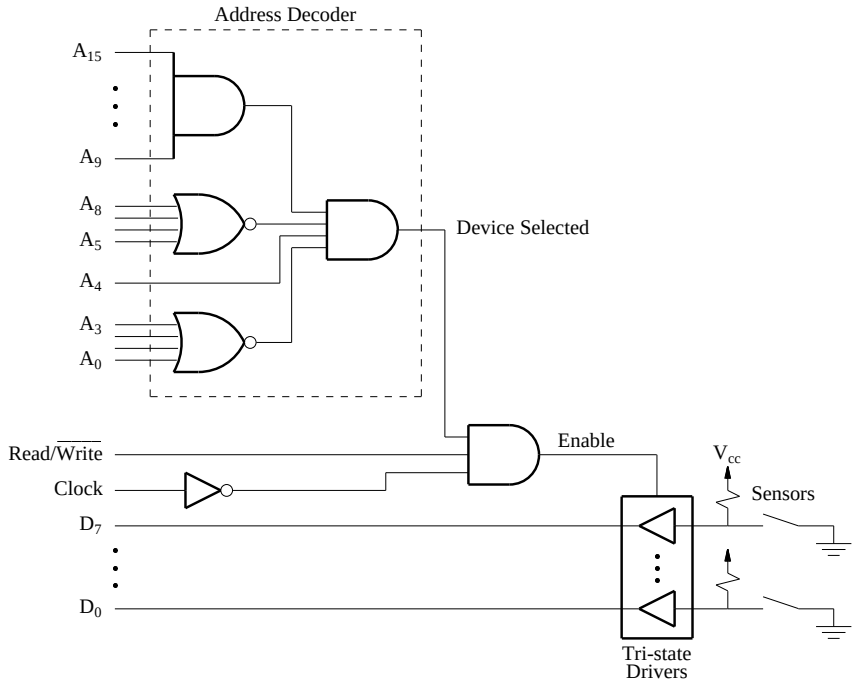
4.26. The priority register in the circuit below contains 1111 for the highest priority device and 0000 for the lowest.



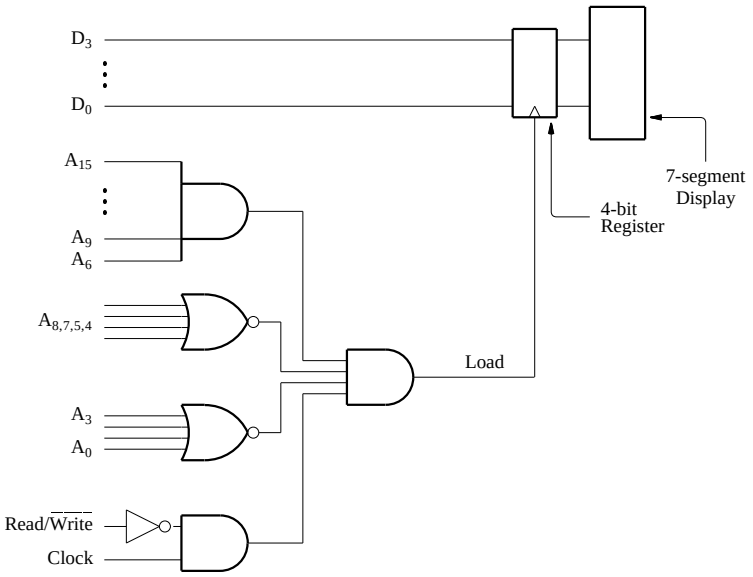
4.27. A larger distance means longer delay for the signals traveling between the processor and the input device. Primarily, this means that $t_2 - t_1$, $t_3 - t_2$ and $t_5 - t_3$ will increase. Since longer distances may also mean larger skew, the intervals $t_1 - t_0$ and $t_4 - t_3$ may have to be increased to cover worst-case differences in propagation delay.

In the case of Figure 4.24, the clock period must be increased to accommodate the maximum propagation delay.

4.28. A possible circuit is given below.

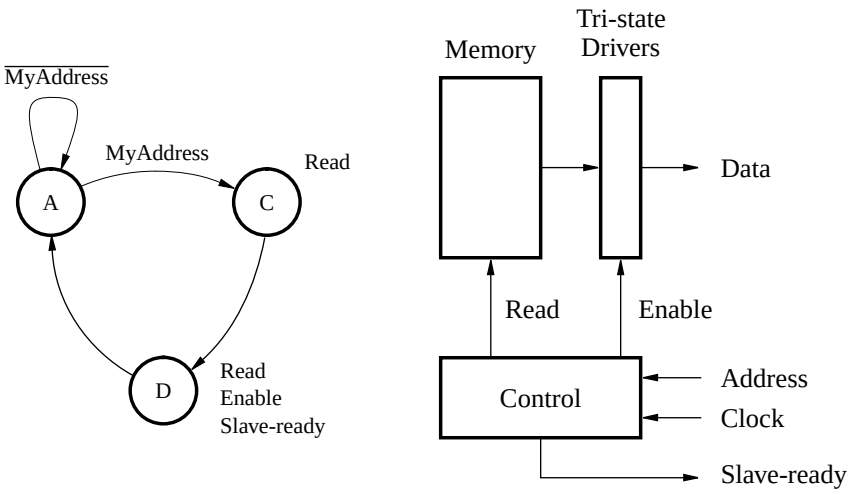


4.29. Assume that the display has the bus address FE40. The circuit below sets the Load signal to 0 during the second half of the write cycle. The rising edge at the end of the clock period will load the data into the display register.



4.30. Generate SIN_{write} in the same way as Load in Problem P4.29. This signal should load the data on D6 into an Interrupt-Enable flip-flop, IntEn. The interrupt request can now be generated as $\overline{\text{INTR}} = \overline{\text{SIN} \cdot \text{IntEn}}$.

4.31. Hardware organization and a state diagram for the memory interface circuit are given below.



4.32. (*a*) Once the memory receives the address and data, the bus is no longer needed. Operations involving other devices can proceed.

(*b*) The bus protocol may be designed such that no response is needed for write operations, provided that arrival of the address and data in the first clock cycle is guaranteed. The main precaution that must be taken is that the memory interface cannot respond to other requests until it has completed the write operation. Thus, a subsequent read or write operation may encounter additional delay.

Note that without a response signal the processor is not informed if the memory does not receive the data for any reason. Also, we have assumed a simple uniprocessor environment. For a discussion of the constraints in parallel-processing systems, see Chapter 12.

4.33. In the case of Figure 4.24, the lack of response will not be detected and processing will continue, leading to erroneous results. For this reason, a response signal from the device should be provided, even though it is not essential for bus operation. The schemes of both Figures 4.25 and 4.26 provide a response signal, Slave-ready. No response would cause the bus to hang up. Thus, after some time-out period the processor should abort the transaction and begin executing an appropriate bus error exception routine.

4.34. The device may contain a buffer to hold the address value if it requires additional time to decode it or to access the requested data. In this case, the address may be removed from the bus after the first cycle.

4.35. Minimum clock period = 4+5+6+10+3 = 28 ns
Maximum clock speed = 35.7 MHz

These calculations assume no clock skew between the sender and the receiver.

4.36. $t_1 - t_0 \geq$ bus skew = 4 ns
$t_2 - t_1$ = propagation delay + address decoding + access time
= 1 to 5 + 6 + 5 to 10 = 12 to 21 ns
$t_3 - t_2$ = propagation delay + skew + setup time
= 1 to 5 + 4 + 3 = 8 to 12 ns
$t_5 - t_3$ = propagation delay = 1 to 5 ns
Minimum cycle = 4 + 12 + 8 + 1 = 25 ns
Maximum cycle = 4 + 21 + 12 + 5 = 42 ns