COMP3221/COMP9221: MICROPROCESSORS AND EMBEDDED SYSTEMS

Worked Examples B

Example 8 (I/O Port Addressing)

An 8-bit microprocessor has a memory address space of 64 Kbytes and an I/O address space of 256 bytes. A designer intends to use only the I/O space for peripheral devices.

a. How many input ports and how many output ports can this microprocessor support?

b. Suppose each I/O port must serve both input and output. How many of these I/O ports can be implemented?

c. Assume that at most 8 I/O ports are needed. How can this fact be used to simplify I/O address-decoding?

d. Assume that 16 I/O ports are needed. How could port addresses be chosen so that address-decoding can be performed with two 74LS138 decoder ICs only?

Example 9 (I/O Port Addressing)

A particular system is based on a 16-bit microprocessor and has two I/O devices. Device #1 has 3 status lines and 4 control lines. Device #2 has 6 status lines and 5 control lines.

a. How many 8-bit ports are needed for status and control access to each device?

b. How many I/O addresses are needed for status and control access to each device?

c. Could a single 16-bit-wide I/O port be used for status and control access to both devices? (If so, how?)

d. Now assume Device #2 has 2 status lines and 4 control lines. How many I/O port addresses are now needed?

Example 10 (Memory-Mapped I/O)

A microprocessor has a 4 Mbyte address space for both memory and I/O. It is to be used in an application where memory capacity will never exceed 2 Mbyte.

a. How can the address space be divided between memory and I/O so that the hardware needed to select between memory and I/O is as simple as possible?

b. Assume that the memory is constructed from byte-wide 256 Kbyte devices only. Give a specification for the memory address decoder which would give the simplest hardware implementation.
Example 11 (Direct I/O)

A micro-based instrument has 8 modes of operation, selected by an 8-way rotary switch. The switch is to be interfaced using Direct I/O and ‘polling’. The bus lines include 16-bit Address, 8-bit Data, an IO/M address space select, RD and WR strobes. The switch is to be in the 256-byte I/O address space at address 255 only.

a. Design a suitable Direct I/O interface, using the simplest hardware possible.

b. Describe a programmer’s model for the switch interface.

Example 12 (Asynchronous Serial Communication)

An ACIA is programmed for full-duplex (i.e both receive and transmit), 1200-baud, 7-bit data, odd-parity, one-stop-bit RS232C communication.

a. Sketch a timing diagram for a frame containing the character ‘f’.

b. How many characters per second can be transmitted?

c. Suppose (by mistake) a frame is received which was an ‘f’ character transmitted at 2400 baud instead of 1200 baud. How would the ACIA respond (valid character, parity error, frame error?).

Example 13 (Interrupt-Driven System)

A 68000-based system is to implement a simple clock, receiving an interrupt every 50 milliseconds from an external 6840 Programmable Timer (PTM) device, and updating a time display on the VDU screen once every second.

The 6840 PTM is the only peripheral device, and can occupy half the address space (from 800000H up). It is to use Level-7 interrupts.

The 6840 PTM has these signals for interfacing to the microcomputer bus (Note that it has other signals for interfacing to external timing events, but which are not required in this application):

- \( RS_0 - RS_2 \) Register Select (input)
- \( CS_0, CS_1 \) Chip Select (input)
- \( \text{RESET} \) Reset (input)
- \( D_0 - D_7 \) Data (bidirectional)
- \( R/W \) Direction of data transfer (input)
- \( E \) ‘E’ clock (input)
- \( IRQ \) Interrupt Request (output)

A test program is to be written, which on receipt of each interrupt, updates the value of two 16-bit variables: ‘count’ representing a count of the (50-millisecond) interrupts, and ‘seconds’ representing a count of the number of elapsed seconds.
a. Briefly describe the interfacing hardware needed, aiming for hardware simplicity.

b. Describe (in English or pseudo-C) the steps needed to set up interrupt vectors and initialise the 6840 PTM for Timer #2 operation. Note that all 6840 Timers start operating when a ‘0’ is written to Bit 0 of Control Register #1. (See below for Control Register #2 usage).

c. Write a suitable interrupt handler for the test program.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>O/P Enable</td>
<td>0/1</td>
<td>If asserted, Timer Output is available on Output 2 (Pin 3).</td>
</tr>
<tr>
<td>6</td>
<td>Int. Enable</td>
<td>0/1</td>
<td>If asserted, Status Register Bit 7 and IRQ (Pin 9) are asserted when Timer Count reaches 0. (Interrupt cleared by CPU read of Status Register, followed by CPU read of Timer Count’s MS byte.)</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>0,0,0</td>
<td>Continuous Mode: Write to Timer Count re-initialises Timer.</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>0,0,1</td>
<td>Frequency Compare: Int. Req. if Gate Cycle &lt; Count Cycle.</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>0,1,0</td>
<td>Continuous Mode: Only RESET or Gate → 0 re-initialises.</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>0,1,1</td>
<td>Pulsewidth Compare: Int. Req. if Gate Pulse &lt; Count Cycle</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>1,0,0</td>
<td>Single Shot Mode: Write to Timer Count re-initialises Timer.</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>1,0,1</td>
<td>Frequency Compare: Int. Req. if Gate Cycle &gt; Count Cycle</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>0,1,0</td>
<td>Single Shot Mode: Only RESET or Gate → 0 re-initialises.</td>
</tr>
<tr>
<td>5,4,3</td>
<td>Mode Control</td>
<td>0,1,1</td>
<td>Pulsewidth Compare: Int. Req. if Gate Pulse &gt; Count Cycle</td>
</tr>
<tr>
<td>1</td>
<td>Clock Source</td>
<td>0</td>
<td>Timer uses Clock on Clock Input (pin 4).</td>
</tr>
<tr>
<td>1</td>
<td>Clock Source</td>
<td>1</td>
<td>Timer uses Clock on ‘E’ Input (pin 17).</td>
</tr>
<tr>
<td>0</td>
<td>Reg. Select</td>
<td>0</td>
<td>Control Register #3 accessible</td>
</tr>
<tr>
<td>0</td>
<td>Reg. Select</td>
<td>1</td>
<td>Control Register #1 accessible</td>
</tr>
</tbody>
</table>

*6840 Control Register #2: Bit Values and Functions*
Example 8 (I/O Port Addressing)
a. 256 of each type, because 256 read cycles and 256 write cycles can be distinguished.
b. 256 altogether, because now cannot use read/write line in addressing.
c. Use ‘linear addressing’ with each one (of eight) address lines enabling one I/O port.
d. Use A7 to enable one ’138, A6 to enable the other ’138, with A5, A4, A3 decoded by each ’138.

Example 9 (I/O Port Addressing)
a. 4 ports (input of up to 8 status bits per device, output of up to 8 control bits per device)
b. 1 address per device (reading gets status, writing sets control)
c. Yes, by using Upper/Lower addressing to transfer a byte between CPU and just one of the two devices.
d. 1 address (read 3 + 2 status bits, write 4 + 4 control bits)

Example 10 (Memory-Mapped I/O)
a. Choose 2 Mbyte for I/O as well as memory, with A21 selecting between them.
b. Can use ’138 decoder, enabled by AS and A21, decoding A20, A19, A18 to give 8 x 256 Kbyte blocks in
the 2 Mbyte memory address space.

Example 11 (Direct I/O)
(a) Simple interfacing hardware:
(b) If \([ XX \) means the contents of I/O address \( XX \) (hex), then a programmer’s model is simply:

\[
\text{switch\_value} = \log_2( [FF] )
\]

Or supply a look-up table:

<table>
<thead>
<tr>
<th>Code Read from FF(hex)</th>
<th>Switch Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111110</td>
<td>Position 0</td>
</tr>
<tr>
<td>11111101</td>
<td>Position 1</td>
</tr>
<tr>
<td>11111011</td>
<td>Position 2</td>
</tr>
<tr>
<td>11110111</td>
<td>Position 3</td>
</tr>
<tr>
<td>11101111</td>
<td>Position 4</td>
</tr>
<tr>
<td>11011111</td>
<td>Position 5</td>
</tr>
<tr>
<td>10111111</td>
<td>Position 6</td>
</tr>
<tr>
<td>01111111</td>
<td>Position 7</td>
</tr>
<tr>
<td>11111111</td>
<td>(Between Positions)</td>
</tr>
<tr>
<td>else</td>
<td>Invalid Data</td>
</tr>
</tbody>
</table>

**Example 12 (Asynchronous Serial Communication)**

(a) A timing diagram for a frame containing the character ‘f’.

```
Data
0
1
1
1
0
0
1
1
Parity Stop Idle
```

(b) A frame contains 10 bits (1 x start, 7 x data, 1 x parity, 1 x stop). So, transmitting frames continuously at 1200 baud (bits/sec), the character transmission rate is 120 chars/sec.

(c) Response to ‘f’ at 2400 baud instead of 1200 baud:

```
Data
0
1
1
1
0
0
1
1
Parity Stop Idle
```

‘f’ has the ASCII code 66 (hex). When transmitted at double the expected baud rate, there appears to be a valid frame (start bit OK, parity OK, stop bit OK), containing the ASCII code 7B (hex). This is the code for ‘{’, so the ACIA will report that a valid ‘{’ character was received.
**Example 13 (Interrupt-Driven System)**

*a. Briefly describe the interfacing hardware needed, aiming for hardware simplicity.*

![Diagram of interfacing hardware](image-url)
A test program updates the value of two 16-bit variables at each interrupt ('count' of interrupts and 'seconds')

b. Describe set up of interrupt vectors, initialising 6840 PTM, etc, for Timer #2 operation.

;program test

* ;define
* ;struct ptm{
  ctrl1 equ 1 ;offset 1
  ctrl13 equ 1 ;also offset 1
  ctrl12 equ 3 ;offset 3
  stat equ 3 ;also offset 3
  tim2u equ 9 ;offset 9
  tim2l equ 11 ;offset 11
* ;}
VEC_ADD equ $00007C ;interrupt vector
PTM_ADD equ $800000 ;ptm address
PERIOD equ 50000 ;period = 50000 (if 1MHZ E clock)
PU equ PERIOD/256 ;period (upper byte)
PL equ PERIOD-256*PU ;period (lower byte)

* use a0 ;struct ptm *ptr
* use d0 ;int temp
count: dc.w 0 ;int count = 0
second: dc.w 0 ;int second = 0

* ;void init() {
  * ;/*set interrupt vector*/
  init: lea VEC_ADD,a0 ;*VEC_ADD = &timer
  lea timer(pc),a1
  move.l a1,(a0)
  * ;/*initialise PTM*/
  lea PTM_ADD,a0 ;ptr = PTM_ADD
  move.b #PU,tim2u(a0) ;ptr->tim2u = PU
  move.b #PL,tim2l(a0) ;ptr->tim2l = PL
  move.b #$C3,ctrl2(a0) ;ptr->ctrl2 = $C3
  * ;/*start timer*/
  move.b #$00,ctrl11(a0) ;ptr->ctrl11 = $00
  rts
  .
  .
  .
c. Write an interrupt handler for the test program:

```
;program test

* ;define
* ;struct ptm{
ctrl1 equ 1 ;offset 1
ctrl3 equ 1 ;also offset 1
ctrl2 equ 3 ;offset 3
stat equ 3 ;also offset 3
tim2u equ 9 ;offset 9
tim2l equ 11 ;offset 11
*
VEC_ADD equ $00007C ;interrupt vector
PTM_ADD equ $800000 ;ptm address
PERIOD equ 50000 ;period = 50000 (if 1MHZ E clock)
PU equ PERIOD/256 ;period (upper byte)
PL equ PERIOD-256*PU ;period (lower byte)
*
use a0 ;struct ptm *ptr
use d0 ;int temp
count: dc.w 0 ;int count = 0
second: dc.w 0 ;int second = 0
.
.
.
;void timer() {
timer: movem.l d0/a0-a1,-(a7) ; *save_registers*
lea PTM_ADD,a0 ;ptr = PTM_ADD
move.b stat(a0),d0 ;temp = ptr->stat /*clear int.*/
move.b tim2u(a0),d0 ;temp = ptr->tim2u
lea count(pc),a1 ;count++ /*update time*/
addi.w #1,(a1)
timil: cmpl.w #20,(a1) ;if (count == 20) {
  bne timx1
  timdl: move.w #0,(a1) ;count = 0
    lea second(pc),a1 ;ntime++
    addi.w #1,(a1)
  *
  }
timx1: movem.l (a7)+,d0/a0-a1 ; /*restore_registers*/
rte ;}
.
.
.
```