Exercise 11.1. Copy an array of $100$ bytes from a table beginning at location $\$A0000$ to a table beginning at location $\$A8000$. You might find the instruction BGT label (branch if greater than zero) useful.

Program 11.1: Similar (means "displaced or indexed $\times 100$")

```assembly
MOVE.L #$8000, A0
MOVE.W #$100, DO
LOOP: MOV.B (A0), ($100(A0))
ADDQ.L $1, A0
SUBQ.W DO
BGT LOOP
MOVE.L #$8000, A0
MOVE.L #$8100, A1
MOVE.W #$FF, DO
LOOP: MOV.B (A0)+, (A1)+
DBF DO, LOOP
```

Second method

- Source table
- Destination table
- Size: $100$
- Move byte
- Loop until done

(Solution)

- Two pointers
- Could use index instead

Move:

```assembly
movea. l #$A\phi\Phi\Phi\Phi, A\phi
movea. l #$A8\Phi\Phi, A1
movea. l #$A81\Phi\Phi, A2
```

Copy:

```assembly
movea. b (a\phi)+, (a1)+
cmpa. l A2, A1
b1.s. s 'copy'
```

(Point to source start)

(Point to copy start)

- Init end marker (address)
- Last good address plus $1$
- Copy, & advance pointer

```
ptr = end (ptr incremented)
Keep copying till pointer = end marker
```

(End (12 bytes)

Because no need to get constants from memory

End-marker in AA register: A2

But "A6F" - 16-byte storage

In faster still:

- 16-byte words
Exercise 11.2. Write a program to calculate the sum of 16-bit words in a table that begins at $10000$. Assume the length of the table, in words, is given as the first table entry (which should not be part of the sum); assume also that the sum will not overflow.

```
clr.l d2          ; init sum < 0
clr.l d3          ; init scratch
move.w $10000, d1 ; get table-length
movea.l #$10002, a1 ; point to table

pickup:
move.w (a1), a1   ; pick up 16 bits
add.l d3, d2      ; form sum (to 32 bits)

subq.w #1, d1     ; count down

6hi.s 'pickup'

stop
```

or use "dbf". replace $ with

dbf d1, 'pickup'

Here table-length entry should be length - minus one (see pgm 11.1).
Exercise 11.3. Write a program to reverse the order of bytes in a table of $100$ bytes beginning at $\$1000$. A straightforward (but slow) way is to reverse the order while copying to a temporary array, then copy the reversed version back. A faster method does the reversal "in place" (but be careful not to step on your own feet as you go). Program both methods.

A. Straightforward: copy to 2nd temporary array:

```
movea.w #$1000, a0 ; source
movea.w #$2100, a1 ; temp table top, plus 0
movea.w #$100, a2 ; end marker, (end + 1)
```

```
   tempcopy: move.b (a0)+, (a1)+ ; cross copy
   cmpa.w a2, a0 ; done?
   b1s.s 'tempcopy' ; not yet
   movea.w #$1000, a0 ; re-init 'source'
```

```
   tempcopy: move.b (a1)+, (a0)+ ; now copy table back from temp table;
                     (no cross copy this time)
```

```
   cmpa.w a2, a0
   b1s.s 'tempcopy' ; loop till hit end marker
```

Clever Method:

```
let
movea.w #$1000, a0
```

```
   move.b (a0), d0 ; hold "low", temporarily
   move.b (a1), (a0)+ ; put "high" side "low"
   move.b d0, -(a1) ; now temp upflow into high
   cmpa.w a1, a0 ; see if meeting (at middle) or
   b1s.s 'clever' ; loop till ear meets west (or at low)
```
Exercise 11.4. Assume we choose to live dangerously by omitting the '574 register, and that we have two asynchronous devices interrupting at the rate of 1000 interrupts/sec each. Assume that the vector fetch cycle has a critical time window of 1ns during which a change of asserted vector will lead to an incorrect fetch (i.e., the CPU will read a vector number different from both asserted vectors). Estimate how often the CPU will vector into the wild blue yonder (i.e., crash!).

Two asynchronous changes of '148 vectors/period:
2000/second.

This change can cause a crash if the change occurs during setup time 1ns wide.
In the 1ns period of the interrupting pair A, B,
there are two bad nanoseconds: \( \frac{2 \times 10^{-6}}{1 \text{ns}} = 2 \times 10^{-6} \): two chances in a million per period.
But there are 1k of these periods/s, so two changes:
\( \frac{2 \times 10^{-6}}{1 \text{ns}} \times 1000 \text{ seconds} = 2 \times 10^{-2} \text{ per second.} \)

We'll get a crash on average, after 500 seconds—less than 10 minutes!
Exercise 11.5. Use a 1-of-8 decoder ('138) to expand our original circuit to accommodate eight 8Kx8 RAMs.

Exercise 11.6. Modify the original circuit to accommodate four 32Kx8 RAMs.

The difference is that these bigger RAMs use 15 address lines, so will be selected by higher-order lines: \( A_{16} A_{15} \) (300 = T-wt example) RAM address space.

\[ \begin{array}{c}
\text{RAMEN} \\
\text{RAM} \\
\text{BASE} + 32K \end{array} \]
Exercise 11.7. Now change the circuit to accept a pair of 64Kx8 ROMs (27512).

Exercise 11.8. Redraw the memory map for each of the previous exercises.
Peripheral circuits

We have nine peripherals in this circuit, so we used a 1-of-8 decoder (‘138) as an address decoding switchyard (with the LED indicator and DIPswitch sharing one read/write port). The decoder is enabled on A19 asserted, which puts us in the I/O portion (the top half) of address space; it is also disabled during interrupt acknowledge, as we explained earlier. We used address lines A12-A14 into the decoder, which puts the successive peripherals at addresses $80000$, $81000$, $82000$, etc.; we ignored the remaining high-order address lines, as we did with memory, thus producing a "lazy" address decoding, with each peripheral making multiple appearances in address space. In fact, every address above $80000$, right up to the top of memory at $FFFFF$ (that's a half million addresses), has some peripheral living in it!

**Exercise 11.9.** Elaborate on this last statement by calculating exactly how many times one of these peripherals appears in memory. Then write down the general form for the LED indicator's address, using x's for "don't cares".

![Diagram showing address space and peripheral appearances.]

But then there are more reappearances, with 8 ports in $1\frac{1}{2}$ M locations, each port appears $0.5M/8$ times: that's $\frac{1M}{16} = 64K \times \text{time}$

The LED address is $A_{19} A_{18} \cdots A_9 A_{15} A_{14} 4_{13} 4_7 \quad x \quad x \quad x \quad x$

In HE: "$\geq 8 \quad 0 \text{ or } 9 \quad x \quad x \quad x$"
Exercise 11.10. The only real disadvantage of our incomplete address decoding is that it wastes a half megabyte of address space (most of which could hold memory) on a few paltry peripherals. Show how a designer who wants to use most of the 1Mbyte address space for memory might decode I/O addresses so the eight ports are memory-mapped as $FF000, $FF100, ..., $FF700, and do not respond to lower addresses. You could now install a megabyte of RAM, except for the fact that both memory and I/O would respond to those high I/O port addresses. Figure out a way to solve this problem.

Basic trick is to disable memory, while enabling I/O, at addresses $FF000 and above (assigning $\alpha = 4K$ locations to I/O).

![Diagram showing address decoding and memory mapping.]

Presumably pointless, but this would remap I/O to $FF000$ to $FF700$, as stated in the problem.

One note: RAM

Resume the RAMs are 1M chips:

18 chips, each capacity 1M x 1
Exercise 11.14. Add hardware (very little required) to make the LED port readable. Use cleverness to keep the additional address decoding simple.
In our example, Figure 11.20 and Program 11.2 show the operation of bad_int, whose job is to bring about an orderly shutdown of output signals, then make the LED display do something eye-catching. Its starting address, known to the linker after all the relocatable assembly code has been hooked together, is loaded (by our main program during the boot-up sequence) into all those reserved vector locations (in low RAM) shown in the table. An exception or spurious interrupt (i.e., any except level 5) causes the CPU to do the sequence described above, putting us into the code at bad_int. We first shut off the x-axis signal, because if we're unlucky enough the exception could have happened right in the middle of the x-axis software pulse, causing the XY display to stay at full intensity (and in one spot) after the crash. While we're at it, we might as well terminate the SWEEP output and assert the END output, since whatever we were doing is now surely a hopeless cause.

Now for the fun. We send 01H to the LED, then enter a loop that does a left circular shift, wastes a biologically-significant amount of time, then sends the shifted byte out, etc. The net effect is a dazzling "walking bit" display, guaranteed to arouse even the most jaded button pusher. We never do an RTE, so the thing just goes on forever. The operator is obliged to do a total RESET to get things going again.

Exercise 11.15. Think of something smarter, so the operator can find out which exception caused the trouble. Hint: There are slightly fewer than 256 possible exceptions; there are also 8 LED bits. Can you write the code for your brainchild?

Instead of providing the one response suggested above, let each of the response routines first output the vector number (formed by throwing out the two low-order address bits of the exception-response address: e.g.,

\[
\begin{align*}
\text{Address of vector} & \rightarrow \text{Vector Number} \\
868 & \rightarrow 5/2 \\
0110 & \rightarrow 0001 (0010)
\end{align*}
\]

Send it to displays, wait, darren displays, send 0 again, blank...forever.

### Common Responses

- Shift temp reg. right 2 bits
- Send 2 bytes
- Clear display
- Delay 1000

**Routines:**
- Move b d1, display
- Move b d1, display
- Move b d1, display
Even the features we included in our design could have additional modes. In particular, microprocessor systems with both A/D and D/A converters can benefit from built-in analog multiplexers that let you loop the DAC outputs back into the ADCs. That way you can test all the converters with software, as part of a power-on test sequence (that also includes memory and port tests, etc.). You can even check the power supply voltage(s), using the ADCs.

Exercise 11.1. Show how to do this, by connecting analog multiplexers at the input of the ADCs. You'll need to arrange a port address to which the CPU can ship its MUX-select commands.

---

New port defined
For purpose of MUX-select: MUX 586
(This can drive the similar analog mux that links the other A/D-D/A pair)

(Incidentally, this setup leaves open the question whether a failed self-test indicates A/D or D/A failure.)
But you can't safely make constant-duration software pulses if interrupts are going on underneath your code.

**Exercise 11.12. Why not?**

You can't because the interrupts by assumption are asynchronous with the processor's operations (otherwise they could be handled by program polling), and one cannot know, therefore, how many interrupts will occur during the "constant-duration... pulses." Thus pulse length can be stretched or shortened, depending on the number of interrupts (assuming that the interrupt routines themselves run at constant rate (no branches, equal delays)).
;SUBROUTINES
;"clear-arrays" -- clear DATA and NORM arrays

clear_arrays:
CLRL D0
MOVE L @data_array, A0 ;zero lives here
MOVE L @norm_array, A1 ;pointers
MOVE W @$FF, D1 ;counter
clr_loop:
MOVE L D0, (A0)+
MOVE W D0, (A1)+
DBF D1, clr_loop ;fastest looping primitive
RTS

Exercise 11.13. Rewrite clear_arrays using SUBQ and Bcc in place of DBF.
Rewrite it again using CLR in place of MOVE.

clear_arrays:
mov E @data_array, at
mov E @norm_array, a1 ;pointer, as before
mov W @$FF, dl
clr W D1
clr_loop:
{ clr L (at)+
  clr W (a1)+
  subz W #1, dl ;this misses pass zero, so
  bhi s, clr_loop
}
RTS
Exercise 11.14. Add hardware (very little required) to make the LED port readable. Use cleverness to keep the additional address decoding simple.
Bad_int

In our example, Figure 11.20 and Program 11.2 show the operation of bad_int, whose job is to bring about an orderly shutdown of output signals, then make the LED display do something eye-catching. Its starting address, known to the linker after all the relocatable assembly code has been hooked together, is loaded (by our main program during the boot-up sequence) into all those reserved vector locations (in low RAM) shown in the table. An exception or spurious interrupt (i.e., any except level 5) causes the CPU to do the sequence described above, putting us into the code at bad_int. We first shut off the z-axis signal, because if we’re unlucky enough the exception could have happened right in the middle of the z-axis software pulse, causing the XY display to stay at full intensity (and in one spot) after the crash. While we’re at it, we might as well terminate the SWEEP output and assert the END output, since whatever we were doing is now surely a hopeless cause.

Now for the fun. We send 01H to the LED, then enter a loop that does a left circular shift, wastes a biologically-significant amount of time, then sends the shifted byte out, etc. The net effect is a dazzling “walking bit” display, guaranteed to arouse even the most jaded button pusher. We never do an RTE, so the thing just goes on forever. The operator is obliged to do a total RESET to get things going again.

Exercise 11.15. Think of something smarter, so the operator can find out which exception caused the trouble. Hint: There are slightly fewer than 256 possible exceptions; there are also 8 LED bits. Can you write the code for your brainchild?

The LEDs could be used to display the exception number; each exception routine could do that task.

For example, an “address error” exception sends the CPU to address $c, which it picks up the start address of the response routine. This is called “vector number 3”.

The response routine could send 2 to LED’s and vector number; then blank LED’s to get attention, then send vector number again.

Perhaps you can invent something less tedious to code.