

# Operating Modes and Exceptions

---

CIS\*2030  
Lab Number 7

Name: \_\_\_\_\_

Mark: \_\_\_\_\_/83

## Overview

Like subroutine calls, *exceptions* can also change the control of flow in a running program. In general, exceptions are calls to the kernel for the purpose of having the kernel perform some form of processing on behalf of the caller. The caller can reside either in software or in hardware, and the calls can be either synchronous or asynchronous.

Software exceptions are the result of executing or attempting to execute a program instruction. For example, executing a trap instruction, or attempting to execute an illegal instruction, are examples of software exceptions. These types of exceptions are said to be synchronous because they occur at the same place in the program every time the program is executed. In contrast, asynchronous exceptions have no temporal connection to the running program, and can theoretically occur at any point in time. Examples of asynchronous exceptions include memory errors or notifications from input-output devices. Asynchronous exceptions that arise in hardware, versus software, are typically referred to as interrupts (and will be studied in detail in the next week's lab).

When an exception occurs, the execution of the current program is temporarily suspended, and control is passed to a specific program located in the kernel. This program is referred to as an *exception handler*. In practice, exception handlers are written by the system programmer, not the user. (Although, in this lab you will learn how to write exception-handling routines.) As the name suggests, the exception handler is written specifically for dealing with the exception. Since an exception handler can be called at any time, parameters cannot be passed to it, as this would require some prior preparation. After the exception handler finishes, control is (usually) returned to the program that was running when the exception occurred, and the program continues executing as though nothing had happened. In this sense, an exception is similar to a subroutine call. Moreover, just like a subroutine call, the exception handler must be transparent; that is, it must save any working registers upon entry, and then restore these registers prior exiting. The exception handler uses the system stack for the purpose of saving and restoring working registers.

Also, since exception handlers are part of the kernel, they must always be run when the CPU is in supervisor mode (versus user mode). This way, the handler has access to the entire instruction set that includes privileged instructions specifically for handling exceptions. Seamlessly moving between user mode and supervisor mode, however, requires help from both hardware and software.

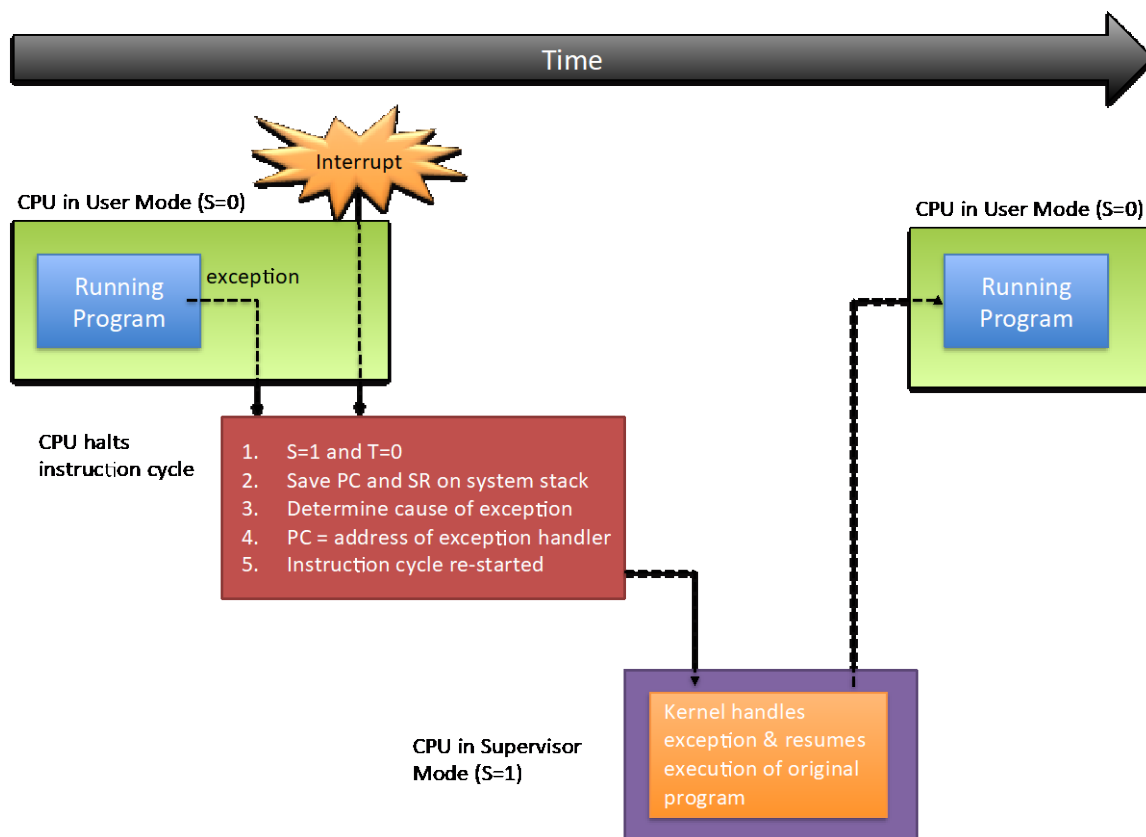
## Preparation

Prior to starting the lab, you should review your course notes and perform the following reading assignments from your textbook (if you have not already done so):

- Chapter 6

## Introduction

Figure 1 provides a high-level overview of the exception mechanism for the 68000. As explained in class, a significant number of steps are performed when handling an exception, some of which are performed automatically by the processor, and others that are performed manually in software.



**Figure 1:** Overview of exception and interrupt handling on 68000.

The scenario depicted in Fig. 1 is one where the processor is in user mode and executing a program. At some point in time, either the program generates an internal exception or a peripheral device generates an external interrupt. Either way, the processor responds by taking steps to handle the exception or interrupt with the help of the kernel. (The actual steps taken by the processor when processing an exception are described in detail both in your lecture notes, as well as in Chapter 6 of your textbook. You are encouraged to review these steps before proceeding.) In general, the following sequence of steps are typically performed when processing an exception:

- With the instruction cycle halted and, hence, the user program suspended, the contents of the status register are saved in an internal register.
- The S bit in the status register is set so that the exception handler, which is considered part of the kernel, can run in supervisor mode; also, the trace bit is cleared.
- The exception number used to identify the exception is obtained and then used to generate the vector address (location in the vector table that contains the address of the exception handler the kernel will run).
- The processor then saves the contents of the PC and internally saved status register on the system stack.
- The program counter is loaded with a new value read from the vector table using the vector address (computed earlier)
- The instruction cycle is turned back on, and execute resumes in the exception handler.

The exception handler typically performs the following steps:

- Saves any working registers upon entry on the system stack.
- Handles the exception
- Restores any working registers just before returning
- Restores the original (suspended) program's status register and program counter, thus passing control back to the (suspended) program; that program then continues as though nothing had happened. This last step assumes that the cause of the exception allows the program to resume execution. If this is not the case, for example the exception is the result of an address error, the kernel can terminate the offending program.

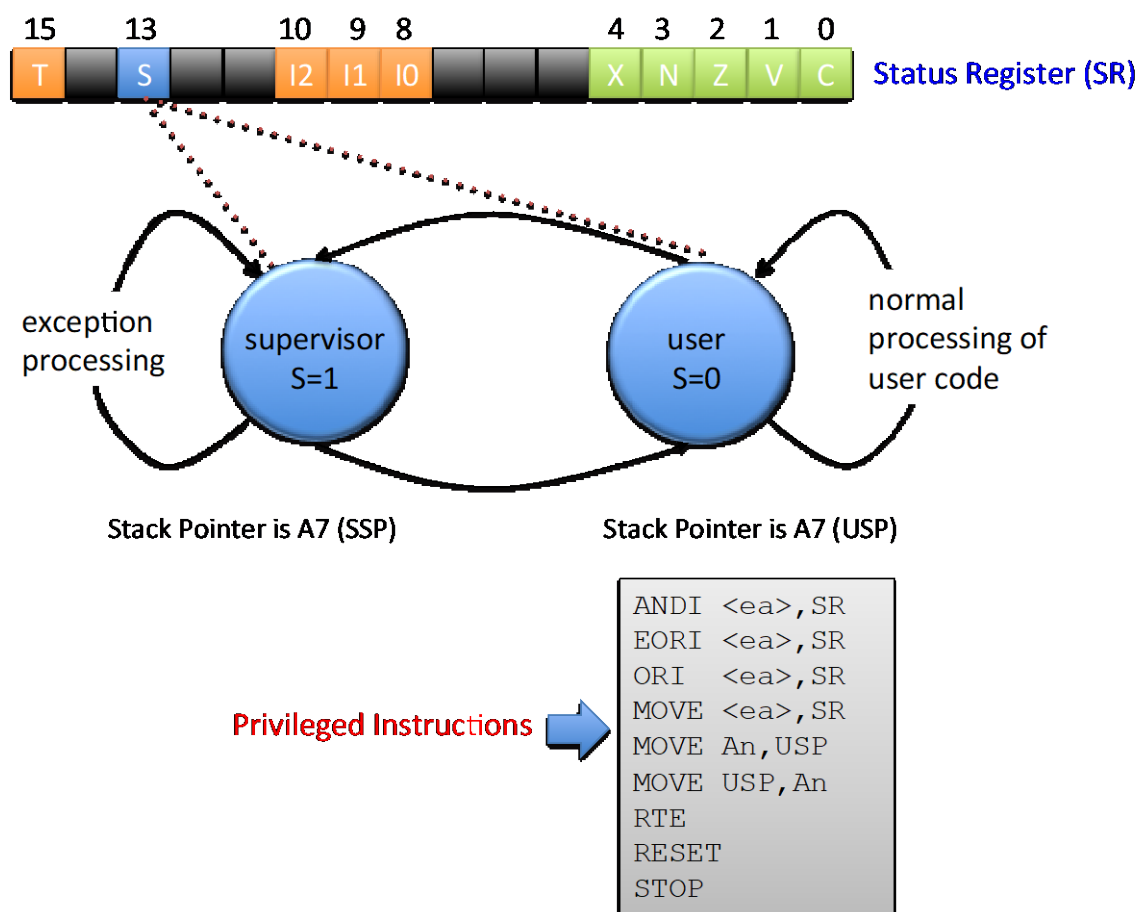
Next you will look at the different facets of the previous exception processing mechanism, before exploring the overall exception handling mechanism employed by the 68000.

### **Part 1: Operational Modes and Privileged Instructions**

As explained in class, computers typically execute two types of programs: user (or application) programs and system programs. Associated with these two types of programs are two CPU operational modes: *user* mode and *supervisor* mode. When the processor is supervisor mode the entire ISA is available to the program that is running. In contrast,

when the processor is in user mode, only a subset of the ISA is available to the running program. In practice, system programs, like the kernel, always run in supervisor mode, while user programs are forced, by the kernel, to run in user mode. The rationale behind this dichotomy is based on *protection*. It is assumed that system programs can be trusted. Therefore, they should have access to the full functionality of the computer defined by its ISA. This includes access to potentially dangerous instructions like STOP that can be used to halt the instruction cycle of the processor, or the RESET instruction, which can be used to reset all peripheral devices connected to the processor's bus. As a general rule, user programs are not trusted, and are required to run in the less privileged user mode. In this mode, any attempt to execute a privileged instruction or access a privileged resource results in a *privilege violation* exception being generated, which, in turn, leads to the termination of the offending user program.

Figure 2 shows how the status register (SR) is organized. Bit 13 is called the S-bit, and is used to determine if the processor is in supervisor mode (S=1) or user (S=0) mode. (The trace bit, bit 15, and the interrupt mask bits, bits 8-10, also play a role in how exceptions



**Figure 2:** User mode is for user code; supervisor mode is for exceptional or system code.

are processed, but for now we will focus only on the issue of privilege.) Figure 2 also shows the list of *privileged* instructions that *cannot* be executed in user mode. Many of these instructions can be used to directly or indirectly set bit 13 to 1, thus forcing the processor into supervisor mode. Clearly, a program running in user mode should not be allowed to execute these instructions in order to force its way into supervisor mode. Only the kernel, which runs in supervisor mode, should have permission to alter bit 13 of the SR. As stated above, any attempt to execute one of these privileged instruction while the processor is running in user mode will result in a privilege violation exception. Notice also that when S=1 the system stack pointer is active, but when S=0 the user stack pointer is active. Consequently, system programs and user programs have their own, separate, stacks. Once again, this is due to protect system programs from user programs that may corrupt their own stack.

### Step 1

Download the sample program called **Lab7a.X68** from the course website.

### Step 2

Start Easy68K. Once running, load the file Lab7a.X68 using the [File->Open File](#) menu choice. You should see something similar to the program below.

### Step 3

Assemble the program, then invoke the simulator. **Note that exceptions must be enabled in the simulator (ensure Options → Enable Exceptions is ticked).** The purpose of the program is to simulate a privilege violation exception that results in the termination of the offending user program. To do this, several segments of code must be created. However, for now, you should focus exclusively on the code segment labeled *main*, which consists of two instructions (ANDI and ORI) that appear on lines 12 and 13, respectively.

Set breakpoints at lines 12 and 13, then execute the program. Before the simulator executes the instruction on line 12, determine what 16-bit value is contained in the status register. **[1 point]**

SR = \_\_\_\_\_

What operational mode (user or supervisor) is the processor operating in? Explain.  
**[2 points]**


#### Step 4

Run the program until the next breakpoint is reached on line 13. Before the instruction on line 13 is executed, what 16-bit value is now contained in the status register? **[1 point]**

SR = \_\_\_\_\_

What operational mode (user or supervisor) is the processor operating in? Explain. **[2 points]**

#### Step 5

Now press the run  button to run the remaining program instruction. Notice that the message “`Privilege Violation has occurred. Main code segment terminated.`” is displayed in the Sim68K I/O window, indicating that the attempted execution of the instruction on line 11 resulted in an internal privilege violation exception being generated. Explain why this happened. **[3 points]**

#### Step 6

What 16-bit value is now contained in the status register? **[1 point]**

SR = \_\_\_\_\_

What operational mode (user or supervisor) is the processor operating in? Explain.  
[2 points]

This example illustrates a more general fact – the only way for the processor to enter supervisor mode from user mode is by means of an exception. Conversely, while in supervisor mode the processor can use a privileged instruction, like `ANDI`, to clear the S bit forcing the processor into user mode. This functionality provides the kernel with a way of running user programs in user mode. However, any attempt by a user program to modify bit S in the status register will result in a privilege violation, and control will be returned to the kernel, which runs in supervisor state. Later, we will see that user programs can safely access kernel resources through special instructions, like `TRAP`, which generate exceptions that act as system calls, and do not result in the termination of the user program.

## Part 2: Making Sense of an Exception

In the previous example, the main program segment caused a privilege-violation exception to be generated. But how did the processor manage to identify the exception as a privilege-violation?

As explained in class, the 68000 supports a maximum of 255 exceptions, each of which is assigned a unique 8-bit unsigned value known as a *vector number*. When an exception occurs, the vector number is either determined automatically by the processor in the case of an internal exception, or supplied by an external device in the case of an external exception (or interrupt). The processor uses the vector number to distinguish between each of the 255 possible exceptions.

Also in the previous example, when the main program segment caused a privilege-violation exception to be generated, the processor responded by running the exception handler in response to the exception. But how did the processor know where the exception handler was located in memory?

As explained in class, each of the 255 exceptions has its own exception-handling routine. The address of each routine is stored in the first 1K bytes of memory (i.e., addresses 0x000000 – 0x0003FF) in a *vector table*. When an exception occurs, the processor first obtains the vector number identifying the exception, and then the vector number is multiplied by 4 to form a *vector address* in the range of 0x000000 – 0x0003FF. The vector address points to the location in the vector table that contains the address of the routine used to handle the exception. In the C programming language, we would refer to this type of indirection as “a pointer to a pointer”.

Table 1 shows the first few vector number assignments for the 68000. (The complete set of assignments are available from your textbook.) Notice that the vector number for a privilege violation is  $8_{10}$ . Thus, when a privilege violation occurs, the processor automatically generates the vector number  $8_{10}$ , and multiplies it by  $4_{10}$ , to obtain the vector address  $32_{10}$  or  $20_{16}$ . The processor now knows that the address of the privilege violation exception handler can be found in memory location  $20_{16}$ . Of course, it is the responsibility of the system programmer to not only write the exception handler routines for each

Table 1: Vector Number Assignments.

Vector Number (Decimal)	Vector Address (Hexadecimal)	Assignment
0	000	Reset: initial SSP
	004	Reset: initial PC
2	008	Bus Error
3	00C	Address Error
4	010	Illegal instruction
5	014	Zero Divide
6	018	CHK instruction
7	01C	TRAPV instruction
8	020	Privilege Violation
.	.	.
.	.	.
.	.	.
64-255	100-3FF	User Interrupts

exception, but also to load the address of each handler into the exception table, so that the processor can respond to exceptions.

When implementing a computer system, it is common practice to implement the vector table using some form of non-volatile memory, like a Read-Only-Memory (ROM), so that the vector table is always present. If the decision is made to implement the vector table in RAM, which is volatile, then it is up to the kernel to store the address of each exception handler in the appropriate location in the vector table when the computer is powered on. The former approach has the benefit of providing additional protection, since the use of a read-only-memory prevents user programs from being able to modify the contents of the vector table to pass control to their program while the processor is seeking to process an exception in supervisor mode.

### Step 1

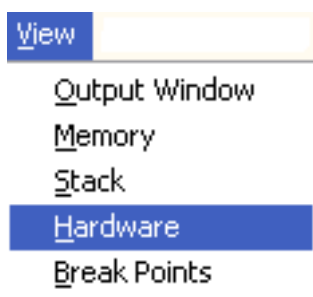
Re-load the sample program called **Lab7a.X68** from the course website, and read through the program carefully.



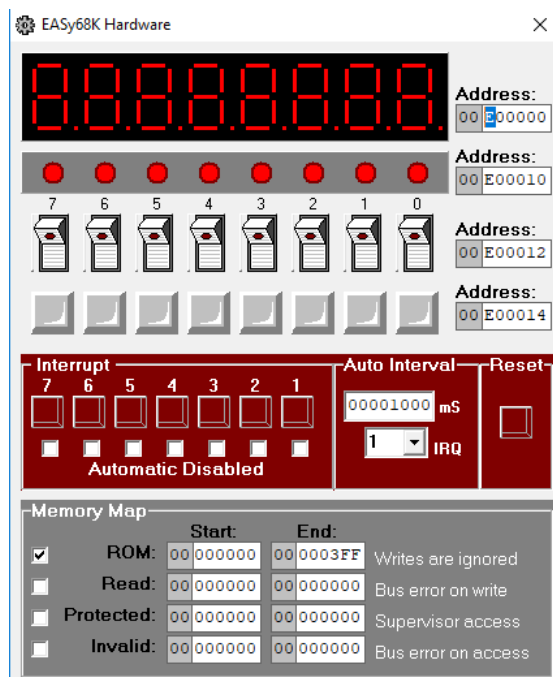
The first thing that you will notice is that on line 8 the program makes use of an assembler directive that you have not seen before:

```
MEMORY ROM 0,1023
```

This assembler directive informs the Easy68K simulator that memory addresses 0 through 1023 are to be implemented in a ROM. Recall that the vector table occupies the first 1024 bytes in memory, so the previous directive is effectively placing the contents of the vector table into a read-only memory. To see this, you can make use of the simulator's hardware simulation capabilities. On the simulator window, click **View** and then select the **Hardware** menu option, as shown below.



A hardware window, similar to the one below, will open.



← Vector Table Implemented in ROM

The hardware window displays various input-output devices, some of which you will work with in an upcoming lab. For now, notice that there is also a memory map at the bottom of the hardware window. This memory map shows that the range of memory addresses associated with the vector table (i.e., 0x000000 – 0x0003FF) are, in fact, located in a ROM rather than RAM.

## Step 2

As explained, the system must be programmed to respond to exceptions. This involves two separate, but related tasks. First, the vector table must be initialized with the address of the exception-handling routine. Second, the exception-handling routine must be written. Normally, both of these tasks fall to the system programmer.

Continuing with the original program, the segment of code shows how to achieve the previous two tasks for the privilege-violation exception. The first code segment places the address of the exception handler `PRVHANDLER` into location 0x000020 in the vector table. This, of course, is because the vector address for this particular exception is 20<sub>16</sub>. The second code segment is the exception-handling routine. In this case, a privilege exception is handled by displaying a message on the screen and terminating the program by not passing control back to the program.

```
* Update Vector Table
PRVADDR EQU    $20
          ORG    PRVADDR
          DC.L   PRVHANDLER

* Privilege violation exception handler
          ORG    $400
PRVHANDLER
          LEA     MSG,A1
          MOVE.L  #13,D0
          TRAP    #15

MSG       DC.B    'Privilege Violation has occurred. Main code
                  segment terminated.',0
```

To view the effect of the first code segment on the contents of the ROM, open the memory window. Recall that you can find the memory window by going to [View](#) on the menu bar and selecting [Memory](#) from the drop-down menu.) Now use the memory window to view the contents of the vector table starting at address 0x00000000.

According to the simulator, what 32-bit (hexadecimal) value is located at address 0x000020? What does this value represent? [4 points]

### Part 3: Saving Processor Information

As discussed in class, when an exception occurs the processor is responsible for saving part of the state of the suspended program. This includes the current program counter and status register. Two questions arise at this point. First, where do these values get saved? Second, why are the values saved in the first place?

With regards to first question, the processor saves the previous values by pushing them onto the *system* stack. The reason they are pushed onto the system stack (and not the user stack) is because the processor is already in supervisor state. It is the responsibility of the exception handler to save and then restore any working registers.

With regards to the second question, the processor saves the PC and SR on the system stack so that a privileged instruction, called RTE, can be used to return the processor from supervisor mode to user mode and control from the exception handler back to the suspended user program. The RTE instruction accomplishes this by first popping the word off of the top of the stack and putting it into the SR, and then popping the longword off of the top of the stack and putting it into the PC. Keep in mind that the PC contains the address of the next instruction to be executed in the suspended program, while the status register contains all of the state of the processor and the original condition-code flags at the time of the exception.

#### Step 1

Load the sample program called **Lab7b.X68** from the course website.

#### Step 2

Read through the program carefully. Notice that the program is very similar to the previous program, but some small changes have been made to the main code segment and the exception handler. With regards to the latter, the handler now saves and restores working register upon entry and prior to exit. The handler also uses a RTE instruction to return control to the main code segment upon completion.

```

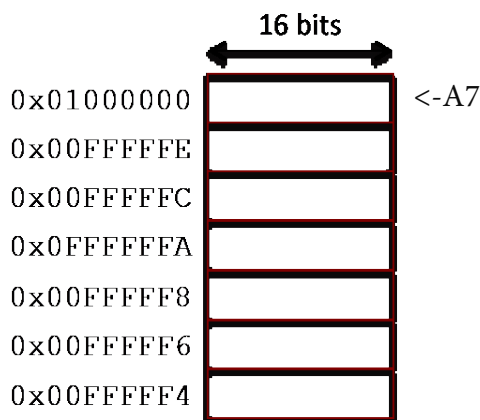
* Privilege violation exception handler
    ORG      $400
PRVHANDLER
    MOVEM.L  A1/D0,-(A7)      ;save working registers
    LEA      MSG,A1
    MOVE.L   #13,D0
    TRAP     #15
    MOVEM.L  (A7)+,A1/D0      ;restore working registers
    RTE                          ;return from exception

```

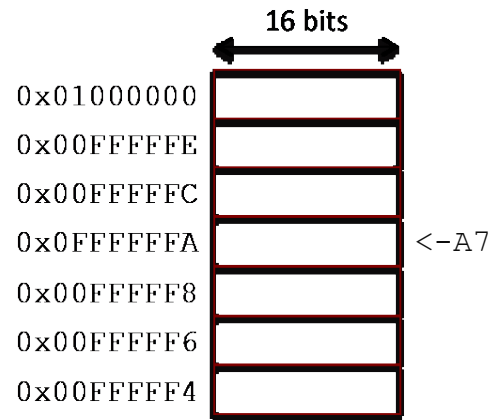
Upon return, the main code segment now displays an appropriate message on the screen indicating that a return from the privilege violation exception handler has, in fact, successfully occurred.

### Step 3

Now, assemble the program and execute the simulator. Set a breakpoint at the ORI instruction on line 14. Run the program to the breakpoint. When the program reaches the breakpoint, show the location of the system stack pointer (SSP) in the (left) memory map below:



Before ORI #\$2000, SR




After ORI #\$2000, SR

Now, generate the privilege violation exception by trying to execute the ORI #\$2000, SR using the simulator's trace facility.

Next, open the Stack window. (Recall that you can find the Stack window by going to [View](#) on the menu bar and selecting [Stack](#) from the drop-down menu.) In the (right) memory map above, show the saved values of the PC and SR on the stack. Also, show the new location of the system stack pointer (SSP). Don't forget to fill out the values in both memory maps. **[4 points]**

What are the addresses of the (saved) status register, lower word of the program counter, and higher word of the program counter on the system stack? **[3 points]**

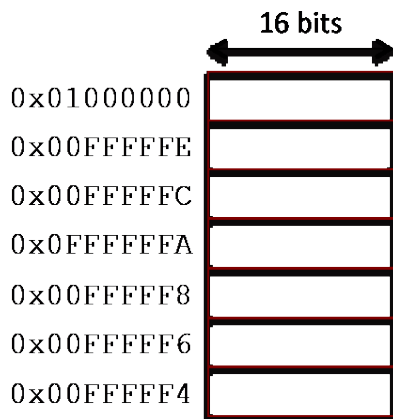
#### Step 4

Set a breakpoint at the RTE instruction on line 41. Run the program  until the breakpoint is reached. What values do the PC and SR contain, and what operational mode is the processor in? **[3 points]**

#### Step 5

Now use the trace facility to execute the RTE instruction. What values do the PC and SR now contain? What is the next instruction to be executed? What operational mode is the processor now in? **[3 points]**

In the memory map below, show the contents of the system stack, along with the new location of the system stack pointer (SSP). **[2 points]**



In general, saving a snapshot of the state of the processor on the system stack immediately when an exception occurs, and then restoring this information at the end of the exception handler, allows the suspended program to continue executing right where it left off.

Run the remainder of the program instructions in the main code segment. In the Sim58K I/O window you should see a message similar to the one below indicating that the main code is once again running.

```

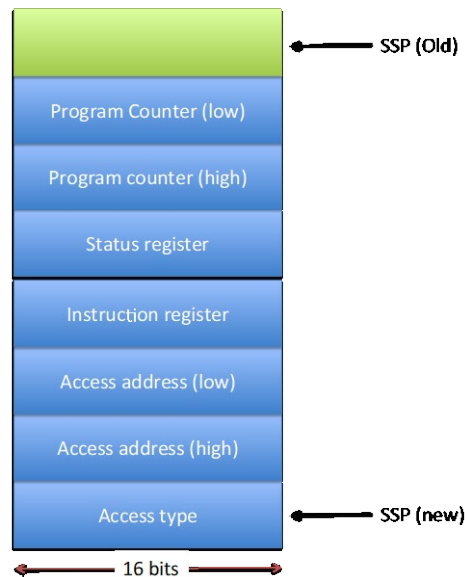
Sim68K I/O
Privilege Violation has occurred. Running exception handler.
Return from Exception (RTE) successful. Running main code.

```

Ultimately, it is up to the system programmer to determine how an exception is to be handled. In this example, the privilege violation handler allowed the suspended program to run. In the example before this one, the handler chose to (in effect) terminate the suspended program by not using a RTE instruction at the end. Thus, the same exception in different systems may be handled in different ways!

#### **Part 4: Saving Even More Processor Information**

As discussed in class, bus-error and address-error exceptions cause more information (than the PC and the SR) to be pushed onto the system stack. This is because the previous exceptions result from a hardware error in which the processor is unable to complete the current bus cycle. Failure to complete the current bus cycle can occur for a multitude of reasons. Therefore, additional information is pushed onto the system stack to aid in debugging the system. A total of seven words are pushed onto the stack, as illustrated below.



Before proceeding, make sure that you have read your textbook and understand the details associated with each of the seven items saved on the stack.

### Step 1

Download the sample program called **Lab7c.X68** from the course website.

### Step 2

Start Easy68K. Once running, load the file Lab7c.X68 using the **File->Open File** menu choice. You should see something similar to the figure below.

```

*-----*
* Title       : Lab8c
* Written by  : Your name goes here
* Date       : dd/mm/yyyy
* Description: Example of Address Error
*-----*

      ORG      $8000

      LEA      $9000,A0
      MOVE.B   (A0)+,D0
      MOVE.W   (A0),D1

      END      $8000
  
```

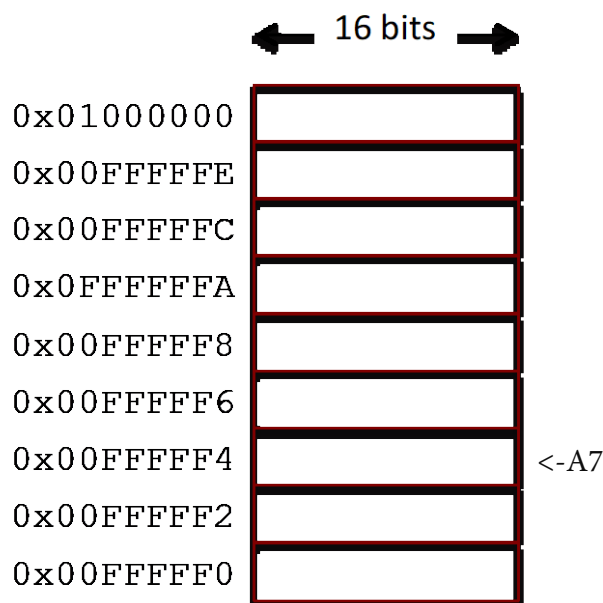
14: 9      Insert

The purpose of this program is to generate an *address-error* exception. An address error exception occurs whenever the processor attempts to access a word or longword at an odd memory address. In this case, the second instruction (`MOVE.B (A0)+, D0`) uses post-increment addressing, which causes A0 to contain the address 0x00009001 after the byte is copied from memory into D0. Since this is an odd address, the third instruction (`MOVE.W (A0), D1`) will cause an address error exception when it attempts to copy the word from memory into D1.

### Step 3

Assemble the program, and then start the simulator. Next, go to [View](#) and on the menu bar select [Stack](#) from the drop-down menu. Run the program, causing the address-error exception to be generated.

Notice that seven words have been pushed onto the system stack. Complete the memory map below, showing the contents of the system stack and the location of the system stack pointer. [8 points]



Use the information in the memory-map (above) to answer the following questions:



1. What is the first word of the instruction being executed at the time of the exception?  
**[1 point]**

2. What is the 32-bit address that was being accessed at the time of the exception?  
**[1 point]**

3. Based in the value in the memory access word, was the processor executing a read cycle or a write cycle at the time of exception? **[2 points]**

4. Based on the value in the memory access word, was the processor accessing data or an instruction at the time of the exception? **[2 points]**

### **Part 5: The Reset Exception**

As explained in class, the reset exception differs from all of the other exceptions defined in the ISA. Generated when the processor is first powered up, the reset exception is responsible for loading initial values from the vector table into the system stack pointer and the program counter, respectively. The SSP is loaded from memory locations 0-3 in memory, while the PC is loaded from memory locations 4-7. (In most systems, the initial value of the PC is the first address in a simple *monitor* program that does not end with a RTE instruction, but which passes control to another program, called a boot loader, once the boot loader is copied from disk to RAM by the monitor.) Also, memory addresses 0-7 are typically implemented in a non-volatile memory ROM, so that the initial values of the SSP and PC remain in memory even when the computer is powered off.

## Step 1

Download the sample program called **Lab7d.X68** from the course website.

## Step 2

Start Easy68K. Once running, load the file Lab7d.X68 using the [File->Open File](#) menu choice. Read through the program carefully!

The first thing that you will notice is that the program makes use of the following assembler directive:

```
MEMORY    ROM 0,7
```

The previous assembler directive informs the assembler that memory addresses 0 through 7 are implemented in a ROM. This information is subsequently used by the Easy68K simulator to setup the memory map used by the hardware window (described below in Step 5).

The remainder of the program contains three independent, but related, code segments. The first code segment, shown below, is a system-initialization code segment responsible for assigning the initial (32-bit) values for the system stack pointer and program counter to memory locations 0 and 4, respectively.

```
RESET     EQU      0           ;reset Vector Address
SSP        EQU     $FFFFFFFE    ;initial value of SSP
PC         EQU     $00000400    ;initial value of PC
```

```
* Segment 1: Update Vector Table
```

```
ORG       RESET
DC.L      SSP
DC.L      PC
```

The second code segment is a small piece of dummy code that represents the monitor program to be run on reset.

```
* Segment 2: Simple Monitor Program
```

```
ORG       $400
LEA       MSG1,A1
MOVE.L    #13,D0
TRAP      #15
```

```
MSG1      DC.B      'Reset Exception has occurred, terminating process, and
                    initiating monitor program',0
```

The final code segment is an infinite loop that starts running in user mode at the beginning of the simulation.

```
* Segment 3: Simple running program
    ORG      $8000
    ANDI     #$DFFF, SR
    LEA      MSG2, A1
    MOVE.L   #13, D0
LOOP   TRAP   #15
       BRA    LOOP
MSG2   DC.B   'Process running as normal', 0
```

The overall purpose of the program is to simulate the case where a user program is running normally, and then a *hardware reset* exception occurs causing the user program to be terminated and the monitor program to start running as part of the reset process.

### Step 3

Assemble the program, and then start the simulator. Notice that the first program instruction to be executed is the `ANDI #$DFFF, SR` instruction on line 34. Before running the program, answer the following questions:

1. What is the (16-bit) value in the status register? [1 point]

2. What is the operational mode (user or supervisor) of the processor? [1 point]


3. What are the values of the three interrupt mask bits in the status register? [1 point]

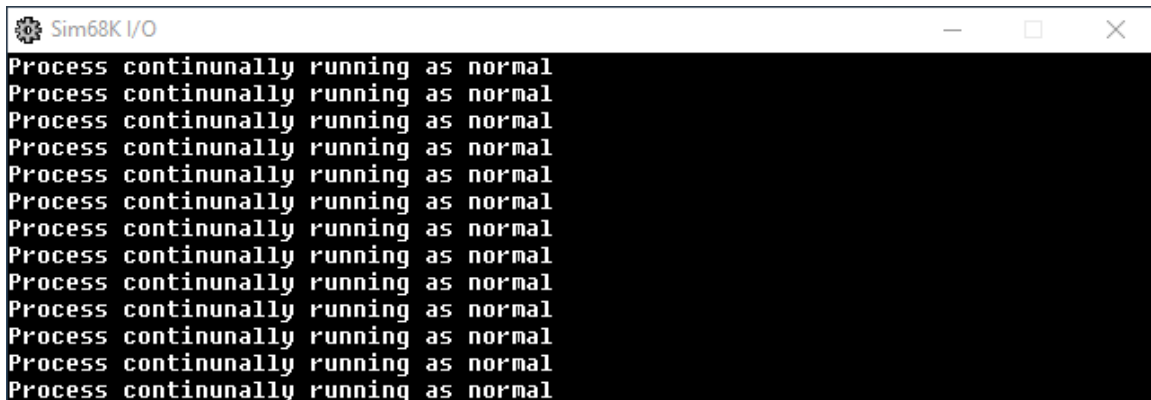
4. What is the value of the trace bit in the status register? [1 point]

Now, using the trace mode facility, execute the `ANDI $DFFF, SR` instruction on line 34.

5. What is the (16-bit) value in the status register? [1 point]

6. What change has been made to the operation of the processor as a result of executing this instruction? [1 point]

Now press the run button  to execute the remaining program instructions. You should see something similar to the following in the Sim68K I/O window:



```
Sim68K I/O
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
Process continually running as normal
```

Observe that the infinite loop in code segment 3 is executing as expected.

#### Step 4

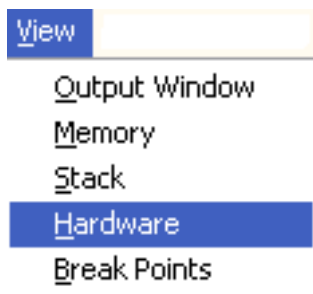
With the previous loop running, return to the simulator window and start the memory window. (Recall that you can find the memory window by going to [View](#) on the menu bar and selecting [Memory](#) from the drop-down menu.) Now use the memory window to view the contents of the vector table.

7. What 32-bit value is contained in memory at address 0? What does this value represent? Explain. [2 points]

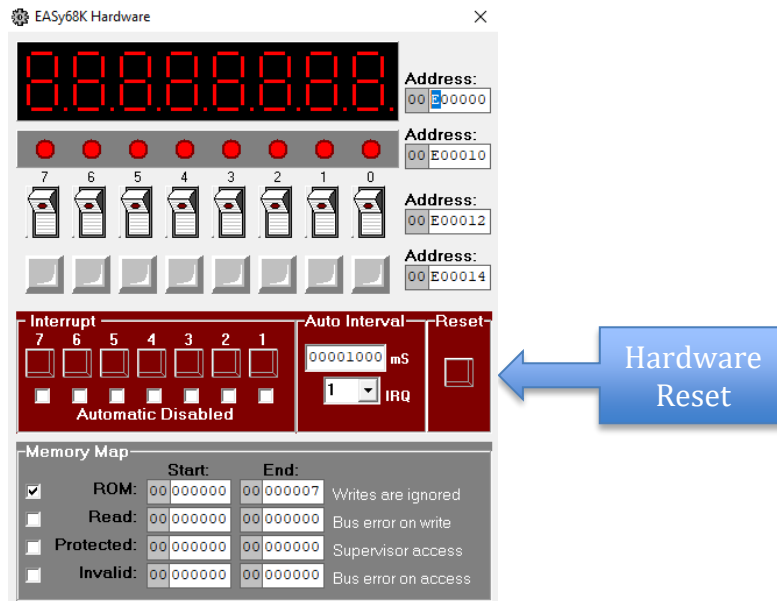
8. What 32-bit value is contained in memory at address 4? What does this value represent? Explain. [2 points]

### Step 5

You will now generate a hardware reset exception. To do this, you will make use of the simulator's hardware simulation capabilities. On the simulator window, click **View** and then select the **Hardware** menu option, as shown below.



A hardware window, similar to the one below, will open.



Observe that there is a hardware reset “button” on the right-hand side of the window. This button can be “pressed” by using the mouse to click on the button. Also, notice that the memory map at the bottom of the hardware window shows that memory addresses associated with the hardware reset exception (i.e., 0 – 7) are located in a ROM.

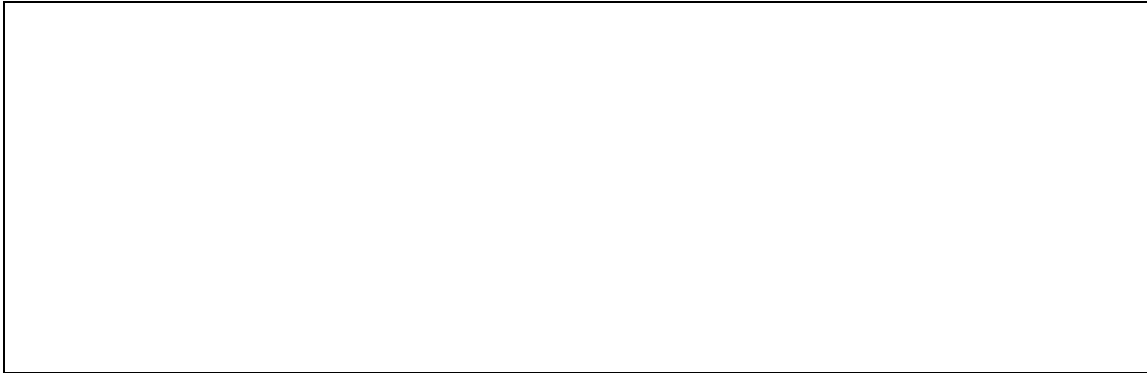
Before clicking the hardware reset button, return to the simulator window and set a breakpoint at the first instruction in the monitor program (i.e., `LEA MSG1, A1` on line 26). Now, click the hardware-reset button.

Notice that several things have occurred as a result of generating a hardware reset, including the termination of the user program, and control being passed to the monitor program.

Answer the following questions:

9. What 16-bit value does the status register contain? Does this value differ from the earlier value reported for question 5? If so, explain *how* it differs and *why* it differs. [4 points]

10. What values do the SSP and PC contain? How do these values compare with those earlier reported when answering questions 7 and 8? If they are different, explain why they are different. **[4 points]**



Run the program to completion. Notice that on a hardware reset, the original program is terminated and control is passed back to a simple monitor program that is responsible for “resetting” the system.

### Trap Instructions

In this portion of the lab, you will gain experience writing code to respond to exceptions. As illustrated above, responding to exceptions requires the system programmer to perform the following two tasks:

1. Writing code to implement the exception-handling routine.
2. Loading the address of the exception-handling routine into the appropriate location in the vector table.

When writing code for the exception handler, any input or output done inside the exception handler should be performed using a TRAP instruction, as described next.

As explained in class, *traps* are instructions that enable a programmer to generate exceptions from within a program. At the assembly-language level, traps provide the programmer with a way to enter supervisor mode to access system services and resources under control of the kernel. For example, one of the most common uses for trap instructions is to perform input-output operations.

Altogether, the 68000 supports a total of 16 trap instructions. These instructions have the following syntax:

TRAP #N, where N is an integer 0 through 15.

Each of the 16 TRAP instructions has a unique vector number (32-47) and unique vector address (0x080 – 0x0BC). When TRAP #1 is executed, the exception-handling routine associated with TRAP #1 is executed. Similarly, when TRAP #2 is executed, the

exception-handler associated with TRAP #2 is executed, and so. At first, it may seem that the number of unique calls that can be made to the kernel using a TRAP instruction is limited to 16. However, the actual number can greatly be expanded by using a data register to pass a parameter to an exception handler. In this way, different parameter values can be used with the same handler to request different services from the kernel. For example, in the case of Easy68K, TRAP #15 is used for I/O. However, different I/O tasks can be performed by putting different task numbers in D0 before executing the trap. The figure below shows just a few of the I/O tasks supported by the Easy68K. You are encouraged to read about these, and other tasks, using the Easy68K help facility.

**Sim68K Text I/O**

**TRAP #15** is used for I/O. Put the task number in D0.

Task	Description
0	Display n characters of string at (A1), n is D1.W (stops on NULL or max 255) with CR, LF. (see task 13)
1	Display n characters of string at (A1), n is D1.W (max 255) without CR, LF. (see task 14)
2	Read string from keyboard and store at (A1), NULL (0) terminated, length returned in D1.W (max 80)
3	Display signed number in D1.L in decimal in smallest field. (see task 15 & 20)
4	Read a number from the keyboard into D1.L.
5	Read single ASCII character from the keyboard into D1.B.
6	Display single ASCII character in D1.B.
7	Check for keyboard input. Set D1.B to 1 if keyboard input is pending, otherwise set to 0. Use task 2 or 5 to read pending key.
8	Return time in hundredths of a second since midnight in D1.L.
9	Terminate the program. (Halts the simulator)
10	Print the NULL terminated string at (A1) to the default printer. (Always send a Form Feed character to end printing. See below.)
11	Cursor Position, Set/Get or Clear Screen Set cursor position: The high byte of D1.W holds the COL number (0-255), The low byte holds the ROW number (0-128). 0,0 is top left. Out of range coordinates are ignored. Get cursor position: Set D1.W to \$00FF Returns COL number, ROW number in high and low byte of D1.W respectively. Clear Screen : Set D1.W to \$FFF0. (Task 95 supports exact pixel placement of text)
12	Keyboard Echo. D1.B = 0 to turn off keyboard echo. D1.B = non zero to enable. (default). Echo is restored on 'Reset' or when a new file is loaded.
13	Display the NULL terminated string at (A1) with CR, LF.
14	Display the NULL terminated string at (A1) without CR, LF.

## Part 6: Allowing a User to Change the Operational Mode of the Processor

Your task is to implement a TRAP #0 exception that causes the operational mode of the processor to flip. In other words, if TRAP #0 is executed while the processor is in user mode, upon return from the TRAP #0 exception handler, the processor should be supervisor mode. Similarly, if TRAP #0 is executed while the processor is in supervisor mode, upon return from exception handler, the processor should be in user mode.



Test your implementation with the following (simple) main code segment:

```
* The main program starts below
    ORG      $8000
main    TRAP   #0
        TRAP   #0
        TRAP   #0
        TRAP   #0
        SIMHALT
```

Your exception handler should save and restore any working registers. Also, save your program in a file called Lab7e.X68. **[20 points]**