Exceptions

Exceptions form the basis for many advanced microprocessor features such as initialization after a reset, interfaces to operating systems, virtual memory, memory protection, i/o interrupts, hardware fault detection, and co-processor interfaces.
This lecture describes the types of exceptions available on the 68000 and how they affect the processor state.
You should acquire a detailed understanding of how the processor responds to different types of exceptions. This will allow you to make use of the above features and write exception handling routines.

Exceptions

An exception is an event that causes the CPU to stop it's normal flow of control, save it's current state ("context") and begin execution of a new sequence of instructions. Exceptions are used for several different purposes.

Causes of exceptions include interrupts (requests for attention from a peripheral device to the CPU) as well as requests by programs for the intervention of some sort of operating system routine ("traps").

User and Supervisor Modes

The 68000 CPU (and many other microprocessors) operates in one of two modes: user mode and supervisor mode.

The supervisor bit (S, bit 13) in the status register (SR) is 1 when the CPU is in supervisor mode. This bit is set automatically when an exception is serviced.

When the CPU is in user mode certain operations that might interfere with the supervisor routines or with other tasks (such as halting the CPU, or switching into supervisor mode) are not allowed. This provides some isolation between tasks and increases the overall reliability of the system.

Every exception (including the initial reset) causes the CPU to switch to supervisor mode. When the CPU is in supervisor mode it can switch back to user mode by changing a bit in the status register (SR). However, *the only way to return to supervisor mode is through an exception.*

There are four other bits in the upper 8 bits of the status register: three interrupt mask bits (IPL0-IPL2, bits 8 to 10) and the trace bit (T, bit 15) that affect exception behaviour. These bits cannot be modified by a user-mode program. The lower 8 SR bits (the condition code register) indicate the result of arithmetic/logic operations and *can* be modified when in user mode.

Exception Handling

The following steps happen when an exception occurs:

- the current instruction is completed before the exception is processed (if possible)
- the CPU makes a temporary copy of the status register (SR) and sets the S bit of the SR to put CPU into supervisor mode
- the CPU obtains the address of the routine to be executed to handle the exception (the *exception handler*) from a table at memory location 0 (the exception vector table)
- the CPU pushes the current PC (which points to the address of what would have been the next instruction) and the SR in that order onto the *supervisor* stack. The arrangement of variables on a stack is called a *frame*.
- the CPU loads the PC with the value obtained from the exception vector table. This causes the execution to continue at the exception handler

Exercise: Why is a copy of the SR made before the S bit is changed? There are 256 different exceptions. How large is the table of exception vectors? If the SSP was \$1000 where would the values of the current PC and SR be stored when an exception occurred?

The Appendix (p. 879) shows the layout of the exception vector table.

Exception Groups

Most exceptions allow the current instruction to complete execution and then save only the PC and SR on the supervisor stack. These are known as either Group 2 exceptions (caused by execution of an instruction) or Group 1 exceptions (most other exceptions).

However, three exceptions (reset, bus error, and address error) are so serious that the current instruction cannot be completed. These are called Group 0 exceptions. In addition to saving the PC and SR registers the CPU also saves 4 additional words of information about the cause of the exception. Although the 68000 can only use this information for diagnostic purposes, later versions of the 68000 family save enough information for the interrupted bus cycle to be repeated if the cause of the exception can be fixed.

The Appendix (p. 878) summarizes the exception groups and shows the organization of their stack frames.

Termination of Exception Handlers

An exception handler terminates using the RTE instruction that restores the PC and SR from the supervisor stack. This may cause a change from supervisor mode to user mode and may also change the interrupt mask.

Exercise: What is the difference between RTS and RTE? Why would RTE be privileged instruction?

We can't do a normal RTE from group 0 exceptions since stack frame is in the wrong format. Normally the routine causing the exception is terminated.

Types of Exceptions

This section describes the different types of exceptions in the order that their vectors appear in the exception vector table. Some of the less useful exceptions have been left out.

If any of these exceptions are not required their vectors should point to an RTE instruction.

Bus Error

This Group 0 exception is used by hardware to indicate to the CPU that something has gone wrong with a memory access. This allows the operating system to terminate the offending program. The hardware generates this exception by asserting the BERR* signal during a bus cycle. Different systems will generate BERR* for different reasons. Typical reasons include access to non-existent memory, write to read-only memory, write to protected memory areas when not in supervisor mode, parity error, etc. All of these are optional and require additional hardware not included in the 68000 CPU.

Address Error

This last Group 0 exception happens when an instruction attempts to read a word at an odd address. This is sometimes called an alignment error. Since compilers and assemblers don't generate code that does this, an access to a word on an odd address indicates a that there is a serious error with the code and that that task should be terminated.

Illegal instruction

This Group 1 exception indicates the CPU fetched and tried to execute an undefined instruction. As before, this usually indicates a serious problem, for example a program is trying to execute something that is not code.

Divide by Zero

This exception is caused by attempting to divide by zero.

Privilege violation

This exception happens when a program running in user mode tries to modify the SR or to do a STOP or RESET instruction.

1010 and 1111 Emulators

Instructions whose most-significant 4 bits are $A cause an emulator exception. This is similar to the undefined instruction exception, but a separate vector is provided so that users can emulate instructions that might be implemented in future processors.

The $F instruction is the similar but is intended for co-processor instructions. Some of these instructions are recognized by later processors in the 68000 series. This exception allows systems without, for example, floating point processors to transparently emulate these instructions in software when a floating point co-processor is not available.

Trap

This exception is provided for user level programs to request services from supervisor-level routines (e.g. the operating system) since the only way to get into supervisor mode is through an exception.

When a user program wishes to obtain O/S services (e.g. to read a file, start another program, get more memory, etc) it pushes the appropriate function arguments on the stack or loads them into registers and then executes a TRAP instruction.

The exception handler performs the required function and returns control back using an RTE instruction.

The 68000 has 16 trap instructions with separate exception vectors. These are in the exception vector table at slots 32 to 47.

Reset

The effect of asserting the RESET* pin (on power-up or when a user pushes the reset button when a program "hangs") is similar to an exception. A reset exception is different in that it does not cause the processor context to be saved and in addition to loading the PC from address 4 (vector 1) the reset operation also loads the SSP from location 0.

Interrupts

The last set of exceptions are interrupts. These exceptions are caused by devices that assert one or more of the $\overline{\text{IPL0}}$-$\overline{\text{IPL2}}$ (*interrupt request*) pins on the CPU.

The processing of interrupts differs from other exceptions in two significant ways: interrupts can be *masked* and the device that generated the interrupt can pass the exception number to the CPU (called a *vectored interrupt*).

After the initial exception processing the CPU does an interrupt acknowledge (IACK) bus cycle which is simply a read from memory location x where x is twice the interrupt level (i.e. the interrupt level appears on A1 to A3). The CPU also sets the function code lines (FC0 to FC2) to all '1' so that external hardware can differentiate between a normal read and an IACK cycle.

When the interrupting peripheral sees an acknowledgement cycle it can either place an 8-bit interrupt number on the LS 8 bits of the data bus and assert DTACK* (to generate a "vectored" interrupt) or it can assert VPA* (to generate an "autovectored" interrupt). The difference between these is explained below.

The CPU then fetches the address of an interrupt service (ISR) routine (an exception handler) from the exception vector table at the bottom of memory. By convention exception vectors 64 to 255 are used for interrupts.

Exercise: A peripheral asserts IPL1 and IPL2. During the IACK cycle it puts $50 on the data bus. What level interrupt is this? How would you find out where the interrupt handler is located?

Masking Interrupts

Since the interrupt mechanism is level-sensitive it is necessary to "mask" (prevent) interrupts of the same priority so that the device does not continue to generate interrupts after it has been recognized.

The 3 bits of the interrupt mask (I2–I0) indicate the level of the interrupt currently being serviced. Only interrupts of a higher level will be recognized.

Interrupt level 7 is a "non-maskable interrupt" (NMI) and can't be masked. Since interrupt level 7 cannot be masked, it is edge-sensitive. This means that the interrupt is only recognized on the falling edge of the signal.

Exercise: What interrupt levels would be acknowledged after the instruction `MOVE #$2700,SR` was executed? Can you think of other reasons why interrupts might need to be masked?

Vectored and Autovectored Interrupts

A disadvantage of vectored interrupts is that they require more complex hardware. Each interrupting device must contain an additional register that will respond during an IACK cycle. In addition, some means must be provided so that only one device at a given interrupt level can return a vector. The advantage of vectored interrupts is that the interrupt can be serviced more quickly since the CPU does not have to check all the possible interrupting devices to see which one caused the interrupt.

During the interrupt acknowledge cycle the interrupting device can assert VPA* instead of DTACK*. This tells the CPU that the device will not put an interrupt number on the data bus. Instead the CPU should use interrupt number 24 plus the interrupt *level*.

This method allows the CPU to directly identify up to 7 different interrupt sources, one per possible interrupt level. This is often sufficient for simple systems. It is also possible to combine multiple devices on the same interrupt and let the ISR poll the possible interrupt sources to determine which device caused the interrupt.

Function Code Outputs

The CPU uses 3 bits to indicate the type of bus cycle it is executing. The encoding is given in the appendix (p. 878) and table 4.3. The bits indicate whether the CPU is in supervisor or use mode, whether the bus cycle is accessing data or instructions or whether the CPU is in an interrupt acknowledge cycle.

These outputs can be used to provide simple memory protection. For example, hardware can be used to prevent access by user mode programs to the lower 1kB of memory where the exception vectors are stored, to addresses where i/o devices are located or to other operating system data areas.

Interrupt-Driven vs Polled I/O

When a CPU must deal with input and output (i/o) from various devices it can *poll* them. Polling requires checking the device (usually by reading a bit in a status register) to see if data can be read or written to the device. The polling must be done sufficiently often that data is not lost. For example, if characters can arrive on a serial port at up to 2000 characters per second, we must make sure we check the port at least 2000 times per second.

An alternative is to use hardware devices that can interrupt the CPU when they require attention. An interrupt is a signal that causes the CPU to interrupt the normal flow of control and to execute a different piece of code called an interrupt service routine (ISR). The ISR does the required i/o function and returns to the original flow of control.