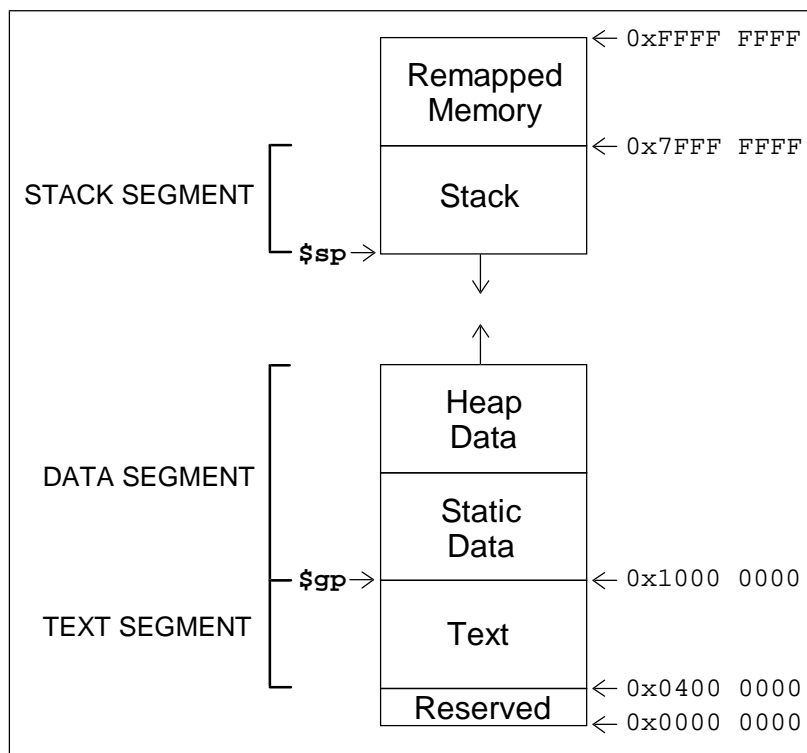By Joshua Cantrell
jjc@cory.berkeley.edu

MIPS Interrupt[1] Handling

Input/Output Devices

This MIPS microprocessor must deal with many I/O devices to create a fully working and functioning computer. It must accept input from a user (the keyboard and/or mouse), and output information to the user (the monitor). Other additional features often have to do with writing to hard drives, which is an I/O device. Even the main memory is typically an external I/O component to the microprocessor, however it is so commonly used that it's I/O support is integrated almost seamlessly into the design of a microprocessor, whereas the other devices mentioned have require special support, such as memory mapping and interrupts.

Memory Revisited

Recall that we have 32-bit addressing in regard to memory. You'll be surprised to know that an actual program can only be within 31-bits of that address! Why is this? Let's take a look at what memory looks like:

```
                                    ← 0xFFFF FFFF
                   Remapped
                   Memory
                                    ← 0x7FFF FFFF
  STACK SEGMENT
                   Stack
        $sp →

                   Heap
                   Data
  DATA SEGMENT
                   Static
                   Data
        $gp →                       ← 0x1000 0000
  TEXT SEGMENT
                   Text
                                    ← 0x0400 0000
                   Reserved
                                    ← 0x0000 0000
```

The stack segment doesn't start at the top of the memory as one might think! Instead there is a special region of memory reserved for non-program related memory. This memory contains things that don't change position from program to program. Everything under 0x80000000 is considered to be different for each program running on the operating system[2].

---

[1] In Patterson and Hennessy's *CO&D* interrupts are considered to be asynchronous, so a more general term would be exception handling.

[2] This is because a program runs under a type of virtual memory. The logical memory is actually the physical memory the microprocessor and operating system deals with, while the virtual memory is what the program sees and deals with. Virtual memory will be covered with the topic

By Joshua Cantrell
jjc@cory.berkeley.edu

What resides in this memory above 0x7FFFFFFF are special operating system functions and procedures, I/O registers mapped to memory addresses, and other things that programs need to access.

SPIM Keyboard Input

The SPIM simulator for the MIPS processor has a device emulator that allows you to read characters from the keyboard. The keyboard I/O registers are mapped to the locations 0xFFFF0000 and 0xFFFF0004. The register at 0xFFFF0000 is called the *Receiver Control* register. It consists of an input *ready* bit[3] and an output *interrupt enable* bit[4]. If *ready* is 0, then there is no data waiting to be read from the keyboard, otherwise, if *ready* is 1, then there is data waiting to be read from the keyboard. If *interrupt enable* is set to 0, an interrupt[5] is not triggered by the device, otherwise if it is set to 1, an interrupt is triggered whenever new data is ready to be read. The register at 0xFFFF0004 is called the *Receiver Data* register. Its 8 LSBs is an input byte representing the key pressed.

Input
Output

Receiver Control

0xFFFF0000	Unused (00...00)	I.E.	Ready
BITS:	31-2	1	0

Ready [Input]: 0 - No new data to be read.
         1 - New data to be read.
I.E. [Output]: 0 - Do not trigger an interrupt when data is available.
         1 - Trigger an interrupt when data is available.

Input

Receiver Data

0xFFFF0004	Unused (00...00)	Received Byte
BITS:	31-8	7-0

Received Byte [Input]: The value of the key pressed. (Sets Ready to 0 on read.)

When the *Receive Data* register is read, it also sets the *ready* bit to 0 until a new byte is received.

---

of caches. Currently, assume that 0x80000000 and higher is non-user program memory.

[3] It's an input bit because we can only read information from it. It does no good trying to write to that bit.
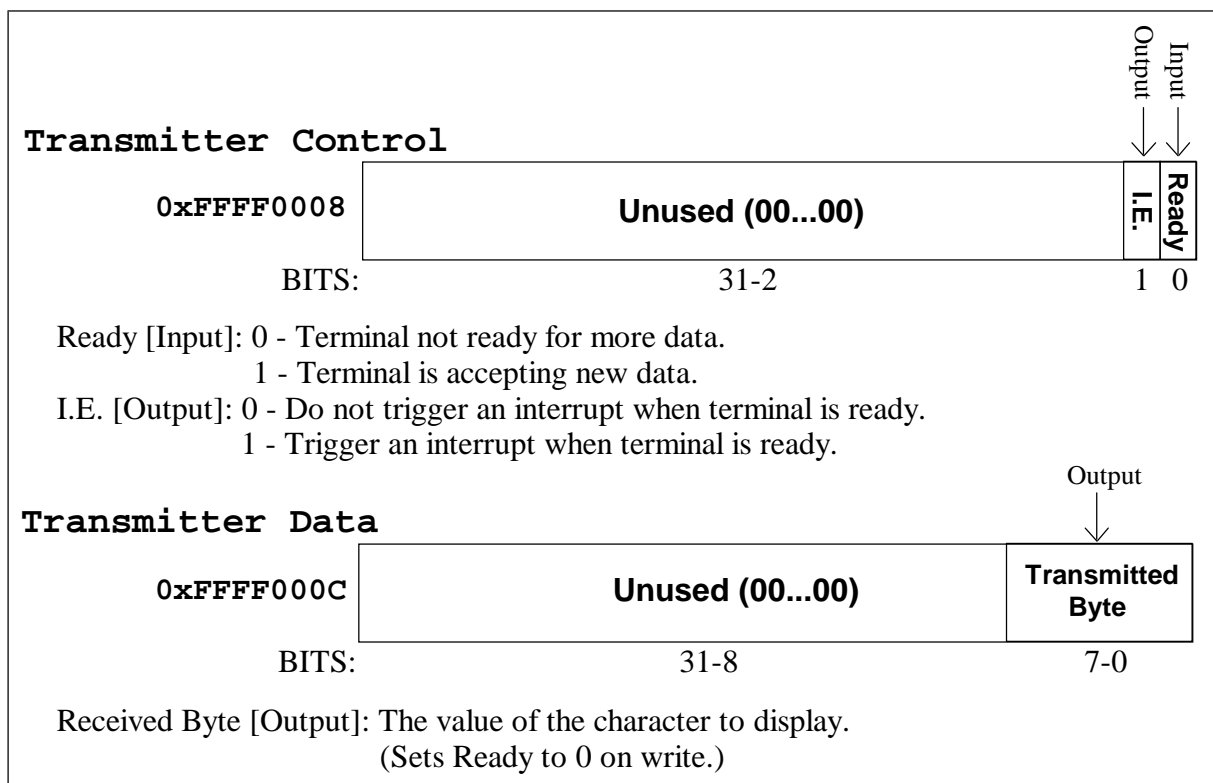
[4] It's an output bit because it's main purpose is for sending a value to the device's register. We can read the value we wrote on the register, but the device will never change this bit.

[5] I'll cover interrupts later in this document.

By Joshua Cantrell
jjc@cory.berkeley.edu

SPIM Terminal Output

The terminal output is similar to the keyboard except that you are sending data out to the user instead of receiving it. This means instead of waiting for data, you wait for the device to be ready to accept data.

The SPIM simulator for the MIPS processor has a device emulator that allows you to send characters to a terminal. The terminal I/O registers are mapped to the locations 0xFFFF0008 and 0xFFFF000C. The register at 0xFFFF0008 is called the *Transmitter Control* register. It consists of an input *ready* bit and an output *interrupt enable* bit. If *ready* is 0, then the terminal isn't ready to receive any data, otherwise, if *ready* is 1, then the terminal is accepting data. If *interrupt enable* is set to 0, an interrupt is not triggered by the device, otherwise if it is set to 1, an interrupt is triggered whenever the terminal is accepting new data. The register at 0xFFFF000C is called the *Transmitter Data* register. Its 8 LSBs is an output byte representing the data being sent to the terminal.

Transmitter Control

0xFFFF0008	Unused (00...00)	I.E.	Ready
BITS:	31-2	1	0

Input → Output (arrows pointing to Ready and I.E.)

Ready [Input]: 0 - Terminal not ready for more data.
           1 - Terminal is accepting new data.
I.E. [Output]: 0 - Do not trigger an interrupt when terminal is ready.
           1 - Trigger an interrupt when terminal is ready.

Transmitter Data

0xFFFF000C	Unused (00...00)	Transmitted Byte
BITS:	31-8	7-0

Output

Received Byte [Output]: The value of the character to display.
           (Sets Ready to 0 on write.)

When a byte is written to the *Transmitter Data* register, it sets the *ready* bit to 0 until a new byte is ready to be sent.

Polling Devices

As can be seen by examining the registers of the devices, there is a *ready* bit. We can't do anything with the data registers unless this ready bit is '1', so we must always wait until it is one before preceding. This can be done by simply making a loop that simply waits until the bit goes to '1' if it isn't already '1'. Because the registers have been mapped into memory, this is something we've known how to do for a long time, we only have to check the bit by looking at the correct address in memory.

By Joshua Cantrell
jjc@cory.berkeley.edu

Input Polling for the Keyboard

```
                                # do {
        lui $t0, 0xFFFF         #   $t0 = 0xFFFF0000;
WaitLoop:   lw $t1, 0($t0)      #   $t1 = *($t0);
        andi $t1, $t1, 0x0001   #   $t1 &= 0x00000001;
        beq $t1, $zero, WaitLoop #  } while ($t1 == 0);
        lw $v0, 4($t0)          # $v0 = *($t0 + 1);
```

Output Polling for the Terminal

```
                                # do {
        lui $t0, 0xFFFF         #   $t0 = 0xFFFF0000;
WaitLoop:   lw $t1, 8($t0)      #   $t1 = *($t0 + 2);
        andi $t1, $t1, 0x0001   #   $t1 &= 0x00000001;
        beq $t1, $zero, WaitLoop #  } while ($t1 == 0);
        sw $a0, 12($t0)         # *($t0 + 3) = $a0;
```

The problem with polling is that if we could be doing something else while we wait, then we'd be wasting a great deal of computing power doing nothing. I don't type fast, so I'm assuming that polling for input from the keyboard would go *really* slow. It certainly isn't a an efficient use of your microprocessor.

Interrupts and Program Flow

To more efficiently use your microprocessor, it makes sense that we will only get a character from the keyboard *if* there is one available. We shouldn't have to wait around or periodically check the keyboard to see if data is there. Why can't the keyboard just *tell* us when the data is ready, and run a special procedure to use the data? This is the purpose interrupts fulfill. When an interrupt is triggered, it causes the code being run to stop while a special procedure is run to handle the event that just occurred to cause the interruption.

Basic Triggered Interrupt Steps
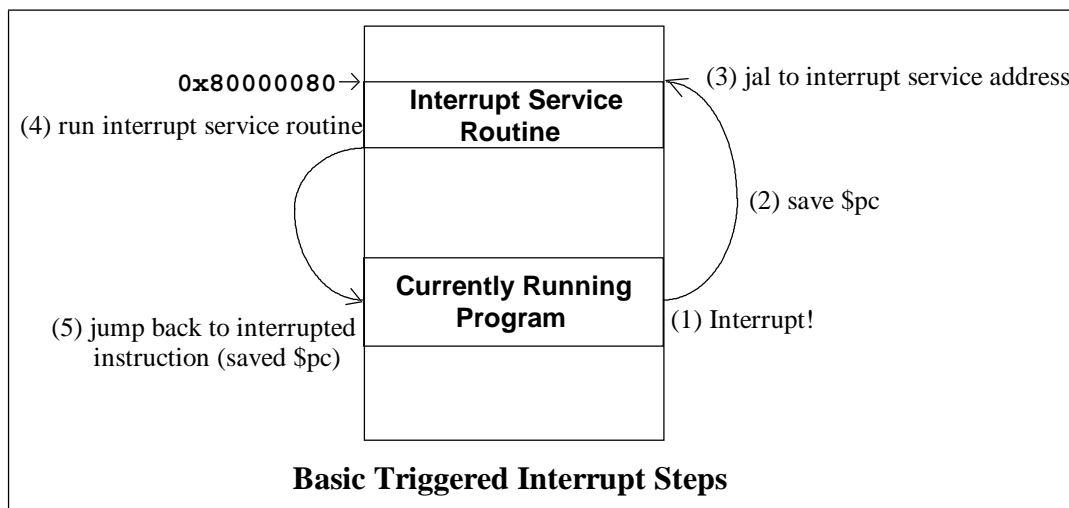
Notice how the interrupt occurs anywhere in the program and jumps straight to the *interrupt service routine*[6]. After the *interrupt service routine* is finished, it continues on running the program from where it was interrupted. This enables us to simply wait for the keyboard to tell us when it is ready rather than constantly polling it to see if it is ready.

---

[6] Often an *exception handler* is called which then calls the ISR. Some synchronous exceptions like those caused by the *syscall* instruction are not considered to be interrupts by *CO&D*.
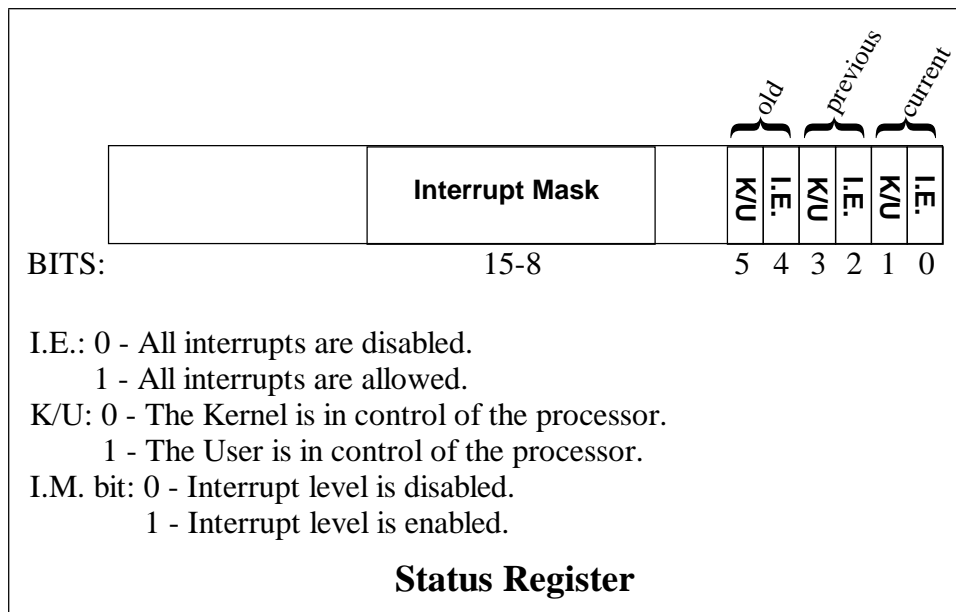
By Joshua Cantrell
jjc@cory.berkeley.edu

Registers Used in Interrupt Servicing

In the normal microprocessor, the registers $k0 and $k1 are both used as temporary variables in interrupt servicing routines. Coprocessor 0 is also used with interrupts. In Coprocessor 0, registers $8, $12, $13, and $14 are all important in servicing interrupts. These registers can be read and modified using the instructions *mfc0* (move from coprocessor 0) and *mtc0* (move to coprocessor 0).[7]

Register name	Register number	Usage
BadVAddr	8	register containing the memory address at which memory reference occurred
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	register containing address of instruction that caused exception

Status Register

The status register is composed of two major components, the *interrupt mask* and *kernel/user - interrupt enable* bits.

I.E.: 0 - All interrupts are disabled.
     1 - All interrupts are allowed.
K/U: 0 - The Kernel is in control of the processor.
     1 - The User is in control of the processor.
I.M. bit: 0 - Interrupt level is disabled.
     1 - Interrupt level is enabled.

Status Register

The *interrupt mask* allows there to be multiple levels of interrupts to be masked instead of requiring all interrupts to be masked at once. There are a totol of 8 bits in the *interrupt mask* allowing a total of 8 interrupt levels: 3 software levels and 5 hardware levels. If a bit is set to '0', that level of interrupts is disabled. If a bit is set to '1', that level of interrupts is enabled.

The *kernel/user - interrupt enable* bits allow multiple interrupts to occur within each other. This is to avoid having to block, or neglect other important interrupts from occurring, especially if a low priority interrupt is being handled and a higher priority interrupt occurs. A value of '1' in an *I.E.* bit means the interrupts are allowed, a '0' means they are all disabled. A value of '1' in a *K/U* bit means the interrupt handler is running at the kernel's level, a value of '0' means it is running at the user's level. The currently operating *K/U* and *I.E.* values in use by the
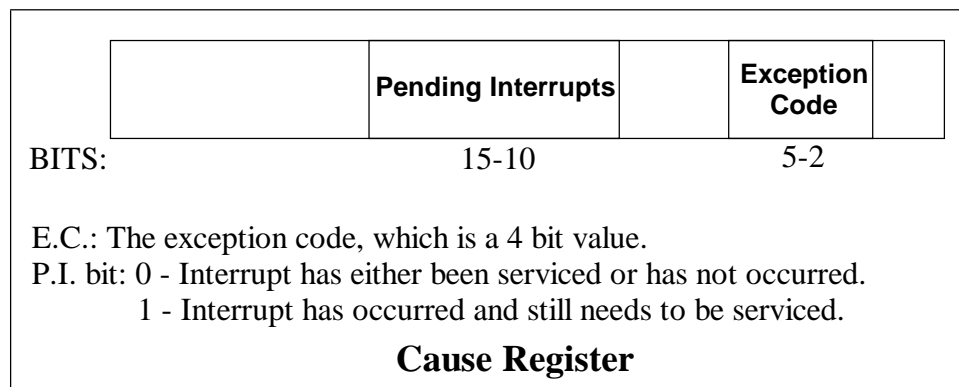
---

[7] The following is taken from A.7, page A-32, of *Computer Organization and Design: The Hardware/Software Interface, 2nd Edition*, by Patterson and Hennessy

By Joshua Cantrell
jjc@cory.berkeley.edu

coprocessor are in the first 2 LSBs. The other 4 LSBs are used as a stack for storing previous values. When an interrupt occurs, bits 3-0 are shift left by 2, and bits 5-4 are lost. Bits 1-0 are replaced with '0's, meaning the interrupts are turned off (to give us some time to save important values before being interrupted by any other interrupts which may overwrite them) and that we are in kernel mode.[8] When the *rfe* (return from exception) instruction is used, Bits 5-2 are shifted right by 2.

All of the bits in this register can be overwritten by a *mtc0* instruction, so when changing bits, you must be sure to preserve the unchanged bits. This can be done by first loading the register with *mfc0*, followed by using *and* and *or* operators to set/unset bits, and finally by storing the value with *mtc0*.

The Cause Register

The cause register is also composed of two major components. A *pending interrupts* field and an *exception code* field.

	Pending Interrupts		Exception Code	
BITS:	15-10		5-2	

E.C.: The exception code, which is a 4 bit value.
P.I. bit: 0 - Interrupt has either been serviced or has not occurred.
　　　　1 - Interrupt has occurred and still needs to be serviced.

Cause Register

Number	E.C. Name	Description
0	INT	external interrupt
4	ADDRL	address error exception (load or instruction fetch)
5	ADDRS	address error exception (store)
6	IBUS	bus error on instruction fetch
7	DBUS	bus error on data load on store
8	SYSCALL	syscall exception
9	BKPT	breakpoint exception
10	RI	reserved instruction exception
12	OVF	arithmetic overflow exception

The *pending interrupts* field consists of bits, each corresponding to a separate interrupt level. If a bit is '1', then an interrupt has occurred and still needs to be serviced. The *exception code* field consists of a 4 bit value that tells us what exception occurred to cause the interrupt.

The EPC Register

The EPC register contains the value of the program counter, $pc, at the time of the interrupt. This is where the program will return after handling the interrupt.

---

[8] Because bits 5-4 are lost, if more than 3 interrupts occur, the program will probably have to save the bits elsewhere so they wouldn't be lost if another interrupt occurs. Another possibility would be to only allow 3 interrupts to occur! I'll have to ask the professor about this.

By Joshua Cantrell
jjc@cory.berkeley.edu

A Simple Interrupt Service Routine

As an example, I'm modifying the code from Professor Patterson's lecture notes. This is an interrupt for reading a value from the keyboard. Assuming that the keyboard's *I.E.* bit is set to '1', this will run when someone hits a key on the keyboard. The first line, ".text 0x80000080" places the code explicitly at the memory location where the *interrupt service routine* is called.

```
.text 0x80000080
     mfc0 $k0, $13          # $k0 = $Cause;
     mfc0 $k1, $14          # $k1 = $EPC;
     andi $k0, $k0, 0x003c  # $k0 &= 0x003c;   // Only keep Exception Code.
     bne $k0, $zero, NotIO   # if ($k == 0) {   // Exception Code 0 is I/O.
     sw $ra, save0($0)      #    save0 = $ra;
     jal ReadByte           #    ReadByte();   // Get the byte.
     lw $ra, save0($0)      #    $ra = save0;
     jr $k1                 #    return;
                            #    }
```

Notice how we had to save any registers we used if it was something other than $k0 or $k1. Also take notice that the procedure ReadByte must save all registers it uses and cannot follow the same pattern as normal procedures because if it corrupts any registers, when the interrupt service routine finishes, the procedure that was interrupted will act erratically because the registers values have been modified!