MIPSfpga

by Imagination

Lab 10

Interrupts

Imagination Community

MIPSfpga Lab 10: Interrupts

1. Introduction

This lab shows the basic usage of interrupts in MIPS CPUs. The lab also demonstrates how the interrupts can enable the processor to avoid constantly polling I/O ports. Interrupts can thus increase the number of cycles available to the processor to spend on computation and other non-I/O tasks. The lab example can be used as a basis for a number of follow-up exercises that explore MIPS CPU features related to the interrupts. Another group of follow-up exercises can link the interrupts to software parallelism and context switching in operating systems. A number of recommended follow-up exercises are listed at the end of this lab.

2. Overview of Interrupts

Interrupts are a key concepts in computer programming and system design. Interrupts force the processor to suspend the regular flow of instruction execution and jump to a certain program address in response to some external event. The external event is usually a change in a hardware signal from outside of the CPU. The memory location (or, in some processors, an index in an array of memory locations) where the processor jumps to, is called the **interrupt vector**. The piece of code where the control goes after landing on the interrupt vector is called the **interrupt service routine**, or ISR. After the ISR is executed, the control goes back to the program location that was executed by the processor when it received the interrupt.

The external event that causes the interrupt can be a tick of a timer clock. Such timer interrupts are useful to organize task switching in a system where several software tasks, or programs, share the same processor, which switches between them. Another example of an external event that causes an interrupt is a signal about the completion of an input/output operation. Such I/O interrupts are useful to offload the regular program flow from constant polling of I/O registers, checking their status, as shown in Figure 1.

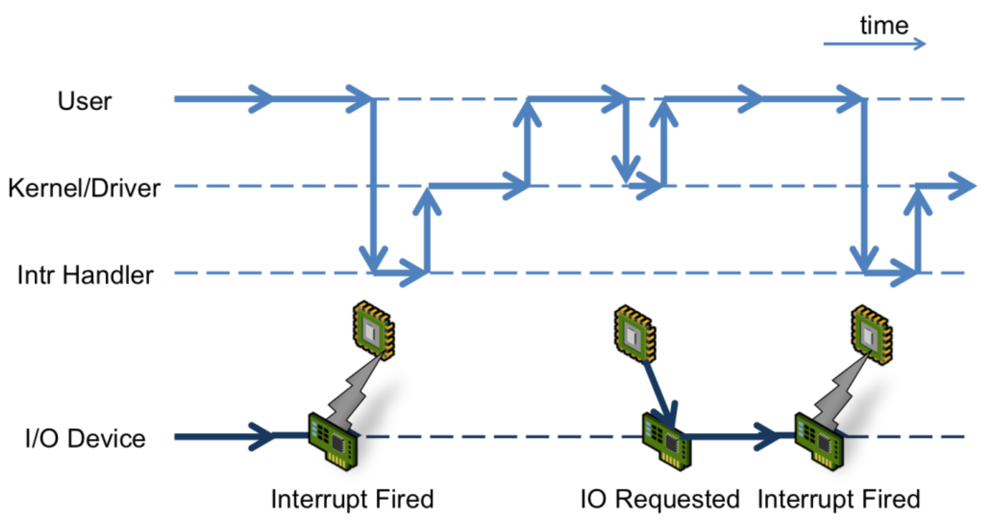

Figure 1. The action of an I/O interrupt. (Figure source: http://virtualirfan.com/history-of-interrupts)

Interrupts, sometimes called more specifically hardware interrupts, are a special case of a more general term: **exceptions**. An exception suspends the regular instruction flow and jumps to a vector in response to not only an external signal but also to some internal conditions of the CPU. Such conditions require immediate attention, for example: arithmetic overflows, accessing out of range address, running a privileged instruction in non-privileged (user) mode, bus errors, and other unusual conditions. Some of those errors should terminate the offending program, while others should cause the program to recover, through the action of the exception service routine. Some of those conditions, like memory address exceptions, may be not considered errors at all, but parts of the mechanics of virtual memory implementation. There are also so-called software interrupts, the exceptions that are intentionally caused by the program to request the services of the operating system. Exceptions are also used in the processor's debug interface. During this lab we will deal strictly with "true" hardware interrupts. To learn about the other aspects of exceptions, please consult the core and architecture documentation.

The need for exceptions and interrupts was so obvious that they appeared very early in the history of computers, as shown on Figure 2.

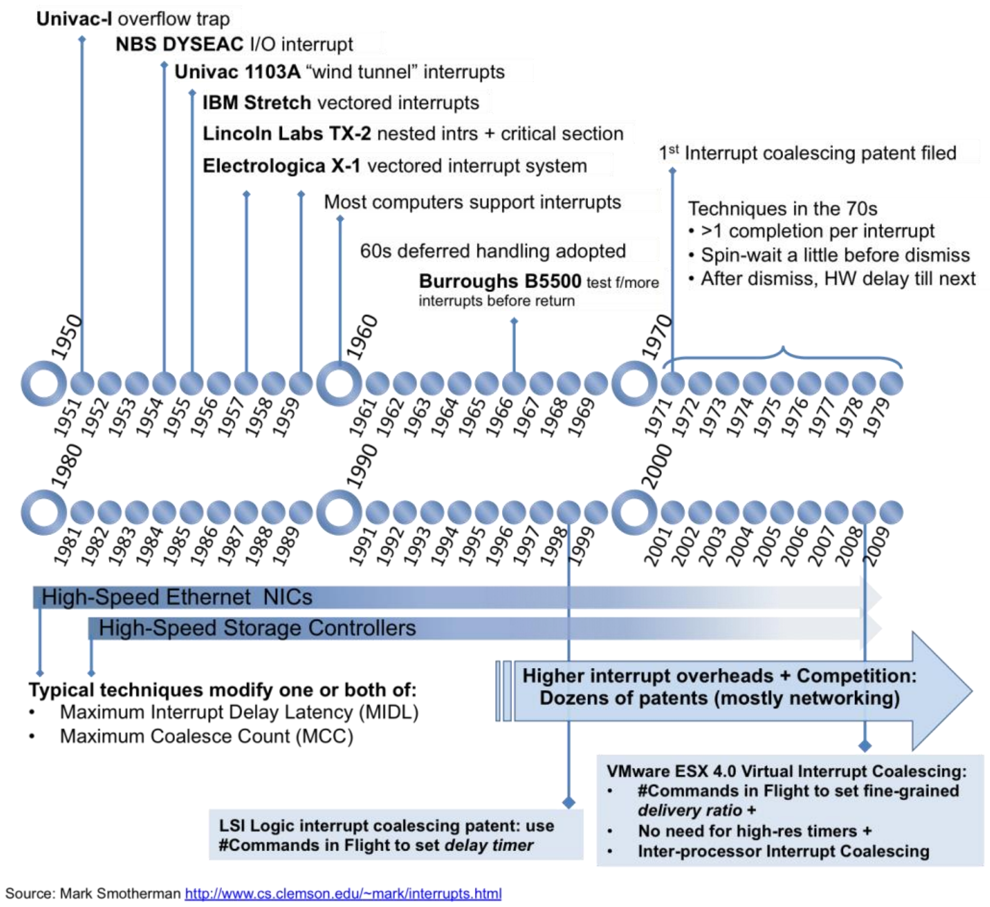
Univac-I overflow trap
NBS DYSEAC I/O interrupt
Univac 1103A "wind tunnel" interrupts
IBM Stretch vectored interrupts
Lincoln Labs TX-2 nested intrs + critical section
Electrologica X-1 vectored interrupt system
Most computers support interrupts
60s deferred handling adopted
Burroughs B5500 test f/more interrupts before return
1st Interrupt coalescing patent filed
Techniques in the 70s
• >1 completion per interrupt
• Spin-wait a little before dismiss
• After dismiss, HW delay till next
1950
1960
1970
1980
1990
2000
High-Speed Ethernet NICs
High-Speed Storage Controllers
Typical techniques modify one or both of:
• Maximum Interrupt Delay Latency (MIDL)
• Maximum Coalesce Count (MCC)
Higher interrupt overheads + Competition: Dozens of patents (mostly networking)
VMware ESX 4.0 Virtual Interrupt Coalescing:
• #Commands in Flight to set fine-grained delivery ratio +
• No need for high-res timers +
• Inter-processor Interrupt Coalescing
LSI Logic interrupt coalescing patent: use #Commands in Flight to set delay timer
Source: Mark Smotherman http://www.cs.clemson.edu/~mark/interrupts.html

Figure 2. The history of exceptions and interrupts. (Figure source: http://virtualirfan.com/history-of-interrupts)

According to the article *Interrupts* by Mark Smotherman (https://people.cs.clemson.edu/~mark/interrupts.html#dyseac), the first computer that employed I/O interrupts was DYSEAC, the second version of SEAC, the Standards Electronic Automatic Computer. According to Wikipedia (https://en.wikipedia.org/wiki/DYSEAC), "DYSEAC was a first-generation computer built by the National Bureau of Standards for the US Army Signal Corps. It went into operation in April 1954". According to Smotherman, DYSEAC was perhaps also the first mobile computer, carried in two tractor trailers at 12 and 8 tons, as shown on Figure 3.

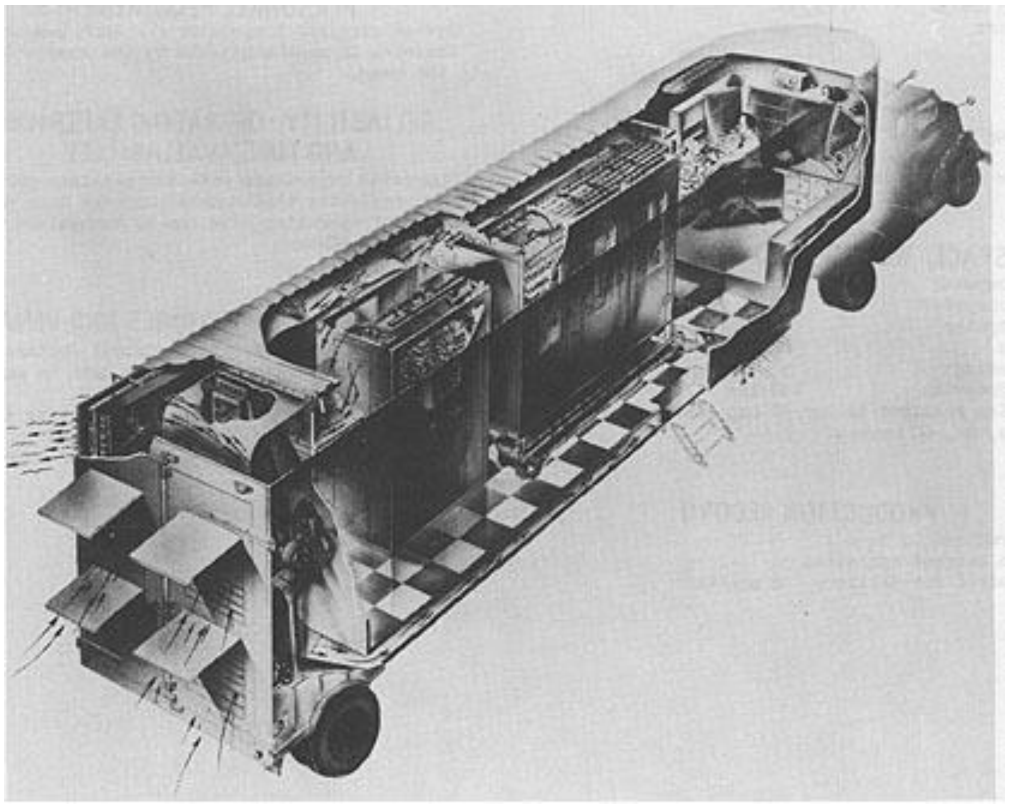

Figure 3. DYSEAC, the first computer with I/O interrupts. (Figure source: http://ed-thelen.org/comp-hist)

3. Lab Steps

This section outlines the sequence of steps, necessary to complete the lab.

3.1. Briefly review the reference materials

Look into *Appendix A. A list of recommended materials to review before and during the interrupt lab*. Briefly review the listed materials to gain understanding as to where to look for the reference information during the lab.

3.2. Review the information about interrupt-related hardware signals

Review *MIPS32® microAptiv™ UP Integrator's Guide*, *Chapter 4: Interrupt Interface*. This manual is included in *MIPSfpga* package. The most important information is the description of the interrupt pin signal *SI_Int* used in the lab. In the course of the lab this multi-bit signal is connected to the pushbuttons on FPGA board. This allows triggering interrupts by pressing the corresponding buttons.

3.3. Add provided Hardware Support for Interrupts

The modified hardware modules to support interrupts are available in the Lab10_Interrupts\Verilog directory. These Verilog files support:

1. A slow clock option so that interrupts can be viewed in real-time, and
2. The hardware changes required to support interrupts.

1. Slow Clock: The slow clock option (see module nexys4_ddr.v and mfp_clock_dividers.v) introduces a clock divider module (mfp_clock_divider_100_MHz_to_25_MHz_12_Hz_0_75_Hz, see mfp_clock_dividers.v) that allows the MIPSfpga to run at a slower rate. The right-most switches (SW[1:0]) control the speed of the clock, as shown in Table 1. When SW[0] is 1, the system runs at 0.75 Hz. Otherwise (when SW[0] is 0), the clock runs at 12 Hz when SW[1] is 1 and 25 MHz when SW[1] is 0.

Table 1. Slow clock frequency options

SW[1:0]	System clock frequency
?1	0.75 Hz
10	12 Hz
00	25 MHz

Note that when using the Bus Blaster probe or UART to download a program, the highest speed (25 MHz) must be used – i.e. SW[1:0] must both be in the OFF (down) position. The clock divider is instantiated in the top-level module, mfp_nexys4_ddr (see mfp_nexys4_ddr.v) as follows:

```
`ifdef MFP_USE_SLOW_CLOCK

  wire        muxed_clk;
  wire [1:0]  sw_db;

  mfp_multi_switch_or_button_sync_and_debouncer
  # (.WIDTH (2))
  mfp_multi_switch_or_button_sync_and_debouncer
  (
        .clk    ( CLK100MHZ ),
        .sw_in  ( SW [1:0]  ),
        .sw_out ( sw_db     )
  );

  mfp_clock_divider_100_MHz_to_25_MHz_12_Hz_0_75_Hz
  mfp_clock_divider_100_MHz_to_25_MHz_12_Hz_0_75_Hz
  (
        .clki    ( CLK100MHZ ),
        .sel_lo  ( sw_db [0] ),
        .sel_mid ( sw_db [1] ),
        .clko    ( muxed_clk )
  );
```

The MFP_USE_SLOW_CLOCK (and MFP_INTERRUPTS) constants are defined in mfp_config.vh as follows:

```
`define MFP_USE_SLOW_CLOCK
`define MFP_INTERRUPTS
```

To use the usual 50 MHz clock (i.e., the PLL instantiated in Lab 1), simply comment out the definition in the mfp_config.vh file, i.e.:

```
// `define MFP_USE_SLOW_CLOCK
```

2. Interrupt Hardware Support: To support hardware interrupts, the 8-bit SI_Int signal must be tied to the interrupt pins. In this case, we tie four of the pushbuttons (IO_PB[3:0]) to the least significant interrupt pins.

Table 2. Interrupt pins

Pushbutton	SI_Int bit
BTND (down pushbutton)	[3]
BTNL (left pushbutton)	[2]
BTNC (center pushbutton)	[1]
BTNR (right pushbutton)	[0]

The mfp_sys module (see mfp_sys.v) makes this connection using the following lines of Verilog:

```
    `ifdef MFP_INTERRUPTS
      assign SI_Int = {4'b0, IO_PB[3:0]};
    `else
      assign SI_Int = 8'b0;
    `endif
```

Now download the MIPSfpga system with interrupt support onto the Nexys4 DDR board. You can either use the provided bitfile (located in Lab10_Interrupts\Verilog\mfp_nexys4_ddr.bit) or create a Vivado project and generate a bitfile yourself. To do the latter, use the files provided in the Lab10_Interrupts\Verilog folder along with the existing MIPSfpga rtl_up files to create a Vivado project and a bitfile (refer to Lab 1 for a refresher of how to do this). Notice that some of the files (mfp_nexys4_ddr.v, mfp_nexys4_ddr.xdc, mfp_sys.v, and mfp_uart_receiver.v and the program files: ram_b?.txt) replace the original files.

After downloading to the board, make sure the system works by testing the default program (the IncrementLEDs program, unless you've changed it) by resetting the system – i.e., pressing the CPU RESET button. Note that the LEDs will appear to all be on if you are running the

IncrementLEDs program (i.e., without added delay between incrementing the value displayed on the LEDs) and also running the system at the highest clock speed (see Section 3.5).

3.4. Download Example Interrupt Program

The program for testing the interrupts is provided in the **Lab10_Interrupts\InterruptsProgram** directory. It is repeated below for your convenience.

```
#include <mips/cpu.h>
#include "mfp_io.h"

volatile int n;

void __attribute__ ((interrupt, keep_interrupts_masked)) _mips_general_exception ()
{
    unsigned cause = mips32_getcr ();  // Coprocessor 0 Cause register

    // check cause of interrupt
    if ((cause & 0x7c) != 0) {
        MFP_LEDS = 0x8001;  // Display 0x8001 on LEDs to indicate exception state
        while (1);  // Loop forever non-interrupt exception detected
    }
    if (cause & CR_HINT0)  // Checking whether interrupt 0 is pending
        n = 0xffff;
    else if (cause & CR_HINT1)  // Checking whether interrupt 1 is pending
        n = 0x0;
    else if (cause & CR_HINT2)  // Checking whether interrupt 2 is pending
        n = 0xf0f;
    else if (cause & CR_HINT3)  // Checking whether interrupt 3 is pending
        n = 0x1111;
      MFP_LEDS = n;
}

int main () {
    MFP_LEDS = 0x5555;

    // set up interrupts
    // Clear boot interrupt vector bit in Coprocessor 0 Status register
    mips32_bicsr (SR_BEV);

    // Set master interrupt enable bit, as well as individual interrupt
    // enable bits in Coprocessor 0 Status register
    mips32_bissr (SR_IE | SR_HINT0 | SR_HINT1 | SR_HINT2 | SR_HINT3 | SR_HINT4);

    while (1) {  // loop forever
      asm ("di");  // Disable interrupts
      n++;
      asm ("ei");  // Enable interrupts
        MFP_LEDS = n;
    }


    return 0;
}
```

The program sets up interrupts by:

1. clearing the boot interrupt bit and
2. enabling interrupts in general as well as 4 of the individual hardware interrupts

The code also shows how to disable and enable interrupts using MIPS assembly. The code then loops forever and is affected only by interrupts. The interrupt service routine (ISR) is given by the following code, also in main.c:

```
volatile int n;

void __attribute__ ((interrupt, keep_interrupts_masked)) _mips_general_exception ()
{
    unsigned cause = mips32_getcr ();  // Coprocessor 0 Cause register

    // check cause of interrupt
    if ((cause & 0x7c) != 0) {
        MFP_LEDS = 0x8001;  // Display 0x8001 on LEDs to indicate exception state
        while (1);  // Loop forever non-interrupt exception detected
    }
    if (cause & CR_HINT0)  // Checking whether interrupt 0 is pending
        n = 0xffff;
    else if (cause & CR_HINT1)  // Checking whether interrupt 1 is pending
        n = 0x0;
    else if (cause & CR_HINT2)  // Checking whether interrupt 2 is pending
        n = 0xf0f;
    else if (cause & CR_HINT3)  // Checking whether interrupt 3 is pending
        n = 0x1111;
       MFP_LEDS = n;
}
```

The ISR first checks if a non-hardware interrupt is detected. If so, it displays 0x8001 on the LEDs as before and enters an infinite loop. If hardware interrupt 0 (CR_HINT0) is detected (i.e., BTNR was pressed), the LEDs will display 0xFFFF. If hardware interrupt 1 (CR_HINT1) is detected (i.e., BTNC was pressed), the LEDs will display 0x0. And so on.

3.5. Exercising the slow clock

A simple program to view the effects of the slow clock is provided in the **Lab10_Interrupts\SlowClockProgram** directory. The program increments a variable (cnt) and displays its value on the LEDs with no delay. With the 25 MHz clock (SW[1:0] = 2'b00), all of the LEDs seem lit as the clock is faster than the response time of the LEDs. Now assert switch 1 (SW[1]) to make the clock run at 12 Hz. Now you will observe the LEDs showing the incremented value. Next, assert SW[0] (note: it doesn't matter if SW[1] is on or off). Now the clock runs at 0.75 Hz, so approximately 1 MIPS instruction is executed every 1.3 seconds. With this slow clock, the LEDs will increment at a very slow pace.

4. Proposed Exercises

Exercise 1. Interrupts in assembly

Rewrite the whole lab in assembly. Can you hand-write an interrupts service routine in assembly that has fewer instructions and works faster than the routine generated by GNU C compiler? Below are some pointers.

1. Observe the disassembly file: program_dasm.txt

2. Review and explain the interrupt-related sections of the code

For example, the exception vector, which starts here:

```
Disassembly of section .exception_vector:
80000000 <__exception_entry>:
/scratch/mpf/jobs/54395/48094/240198/shared/gcc/libgloss/mips/hal/mips_exc
pt_entry.S:68
80000000:   3c1b8000    lui   k1,0x8000
80000004:   277b2428    addiu k1,k1,9256
```

The code for setting up the interrupts:

```
    // set up interrupts
    // Clear boot interrupt vector bit in Coprocessor 0 Status register

    mips32_bicsr (SR_BEV);
800002cc:   40026000    mfc0  v0,c0_status
800002d0:   7c02b584    ins   v0,zero,0x16,0x1
800002d4:   40826000    mtc0  v0,c0_status
800002d8:   000000c0    ehb

    // Set master interrupt enable bit, as well as individual interrupt
    // enable bits in Coprocessor 0 Status register
    mips32_bissr (SR_IE | SR_HINT0 | SR_HINT1 | SR_HINT2 | SR_HINT3 |
SR_HINT4);
800002dc:   40026000    mfc0  v0,c0_status
800002e0:   34427c01    ori   v0,v0,0x7c01
800002e4:   40826000    mtc0  v0,c0_status
800002e8:   000000c0    ehb
```

The general exception code:

```
void __attribute__ ((interrupt, keep_interrupts_masked))
_mips_general_exception ()
{
80000204:   401b7000    mfc0  k1,c0_epc
```

```
80000208:   27bdfff0    addiu sp,sp,-16
8000020c:   afbb000c    sw    k1,12(sp)
80000210:   401b6000    mfc0  k1,c0_status
80000214:   afbb0008    sw    k1,8(sp)
```

Exercise 2. Interrupts in simulation

Modify the system testbench to study interrupts using Verilog simulation, without synthesizing the design. Create all the necessary scripts to show the main interrupt-related signals on the waveform.

Exercise 3. Using the slow clock and observing the program counter.

Improve the interrupt lab by exposing program counter (PC) to the outside seven-segment display. Create a variant of the interrupt lab that connects the processor's program counter (PC) to the outside multiple-digit seven-segment display. This connection should be muxed with the regular seven-segment display output and should be dependent on whether some selected switch or button is pressed. With this setup, when the system clock is turned into slow mode, it will be possible to observe how the processor enters interrupt service routune.

Which Coprocessor 0 register bits should be exposed to outside LEDs to observe interrupts in action in slow-clock mode?

Find on the Imagination Technologies website http://imgtec.com a manual called *MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture*. Investigate which Coprocessor 0 register fields are interesting to observe when the processor core takes an interrupt and enters the interrupt service routine in slow-clock mode. An example of such field is the *EXL* (*Exception Level*) bit of the Coprocessor 0 *Status* register.

Modify the core and system RTL to connect these Coprocessor 0 registers to external LEDs on FPGA board. Some fields, like *Status.EXL* do not require changes in core RTL because the original core RTL already outputs its value to the external pin, *SI_EXL*. Exposing such fields requires only changes in the system RTL (*system/mfp_*.v* files) . Other fields may require adding additional ports and connections to the core's RTL (*core/m14k_*.v* files) as well.

Document your changes and create a post on MIPSfpga forum of the Imagination Technologies website.

Exercise 4. Synchronising memory accesses in interrupts

Observe the effect of not disabling the interrupts around critical section that updates the same variable as the interrupt handler.

Write code that modifies the same variable as the interrupt handlers, variable n. For example, the main code could do the following:

```
asm ("di");
n ++;
asm ("ei");
```

Observe the operation of the program. Then comment out the assembly commands that enable and disable interrupts (*asm ("di");* and *asm ("ei")*) around incrementing the counter variable *n* in *main* function inside *main.c* file:

Build and re-run the program, using both fast and slow clocks. Can you see the difference in system responsiveness to the interrupts? You may observe cases when the system takes the interrupt and executes the interrupt service routine that resets the counter. However after returning from the interrupt, the counter appears not to be reset. Instead it continues to increment its old value, set before entering the interrupt service routine. Can you explain what is going on? We recommend to analyze the assembly output of *gcc* compiler by running *01_compile_c_to_assembly.sh* under Linux or *01_compile_c_to_assembly.bat* under Windows.

Exercise 5. Synchronizing the updates of a counter variable using the *LL/SC* (Load-Linked / Store-Conditional) pair of instructions

MIPS architecture provides a way of synchronizing variable updates without disabling interrupts, using a special pair of instructions *LL/SC* (Load-Linked / Store-Conditional). Create a version of *main.c* that uses this feature of the processor instead of *DI/EI* pair of instructions used in this lab and *Exercise 5.2.1*. If you are not familiar with *LL/SC*, you can review the following materials:

- *MIPS32® microAptiv™ UP Processor Core Family Software User's Manual*, section 12.3, the descriptions of *LL* and *SC* instructions. This manual is included in *MIPSfpga* package
- Book *See MIPS Run, Second Edition, by Dominic Sweetman*, sections *5.8.4 Critical Regions with Interrupts Enabled: Semaphores the MIPS Way* and *8.5.2 Load-Linked / Store-Conditional*.

You can implement this exercise using either a function written in assembly and called from *main*, or, alternatively, using *asm* construct in C with parameters, as described in online GCC documentation (https://gcc.gnu.org/onlinedocs), see 6.45.2 Extended Asm - Assembler Instructions with C Expression Operands

Exercise 6. Why do we need the *volatile* qualifier for the counter variable in the main() code?

Review the code of *main.c* in the **InterruptsProgram** directory. Why do we need counter variable *n* to be declared with *volatile* qualifier? How does the absence of this qualifier change the result of program execution? Try to run the program without *volatile* with different levels of compiler optimizations. Is there any change in behavior? Review the code produced by *gcc* compiler with different *-O* settings.

Exercise 7. Timer interrupts

Exercise 7.1: Timer interrupt lab using *Count/Compare* pair of Coprocessor 0 registers present inside *MIPS microAptiv UP* core

The timer interrupt is a special kind of hardware interrupt that occurs regularly, with a set frequency, usually in kilohertz range. Such interrupts are used to measure time intervals and to implement software multitasking, including parallel task execution in operating systems.

Read both hardware and software documentation about an embedded timer interrupt used in MIPS cores in conjunction with the *Count/Compare* pair of Coprocessor 0 registers. Is this feature present in MIPSfpga? If yes, implement a lab that uses this feature. This lab may for example measure time between pressing some button, or do some computation while the interrupt service routine polls some input.

Exercise 7.2. Timer interrupt lab using a custom timer module implemented outside *MIPS microAptiv UP* core

Create an alternative implementation of the timer interrupt lab (Exercise 5.3.1), without using the timer interrupt already implemented in *MIPS microAptiv UP* core. In order to create such implementation, write a custom timer interrupt-generating module in Verilog. Then connect it to one of hardware interrupt pins, bits of *SI_Int* external signal.

Exercise 7.3. Programmable timer interrupt lab using a custom timer module implemented outside *MIPS microAptiv UP* core

Extend the student project 5.3.2 by interfacing the timer interrupt-generating module not only to system clock, reset and *SI_Int* signal, but also to the system's *AHB-Lite* bus. The goal is that the frequency of interrupts can be controlled by the software running on MIPSfpga CPU core. For the details of *AHB-Lite* interfacing, see *the MIPSfpga Getting Started Guide* as well as the other labs.

Exercise 7.4. A variant of programmable timer interrupt lab that uses multiple counters and interrupt pins

Create a version of the student project 5.3.3 that uses multiple counters to generate hardware interrupts with different frequences on different hardware pins (bits of the signal *SI_Int*).

Exercise 7.5. Medium-to-advanced student project: Create a lab demonstrating multitasking / context switching

An important application of timer interrupts is to facilitate a variant of parallel programming called task switching, or context switching. Context switching is widely used in operating systems, from relatively simple, like FreeRTOS, to complicated, like Linux.

The idea of context switching is to periodically switch between different threads of execution, that are also sometimes called processes or tasks. The "context" is a reference to a set of information associated with the task, including program counter (PC) and general-purpose registers (GPR). The switch happens inside the timer interrupt service routine that saves the current context in some memory structure (called in some systems Process Control Block - PCB), then restores the context of a different process from another PCB and exits the timer interrupt into the new thread of execution.

Create a lab that switches between two different C functions, running in parallel. You don't need to use any operating system to do this. Just save and restore all the necessary registers inside the timer interrupt service routine, and maintain an illusion of parallel execution of two C programs for the end-user. The end-user would observe the outputs from two programs on LEDs or other output devices.

Exercise 7.6. Advanced student project: Port some open-source RTOS, like FreeRTOS, to MIPSfpga

FreeRTOS is a popular real-time operating system, used as an example RTOS in the course *Connected MCU* created by Dr.Alexander Dean of North Carolina State University. The *Connected MCU* course demonstrates FreeRTOS on Microchip PIC32MZ microcontroller that uses a processor core common with MIPSfpga. However PIC32MZ uses External Interrupt Controller which is absent in the default MIPSfpga system which uses Interrupt Compatibility Mode. Porting FreeRTOS on MIPSfpga should be an appropriate project for a graduate student of embedded programming or computer architecture.

Exercise 7.7. Medium-to-advanced student project: Create a lab demonstrating switching from the user mode to the kernel mode when entering the interrupt service routine

Processor mode is one of the most important concepts in computer science, particularly in the operating system design. Different CPUs have somewhat different modes with different names: user, kernel, priviledged, superuser and others. MIPS32 architecture supports user mode, kernel mode and supervisor mode. According to *See MIPS Run, Second Edition, by Dominic Sweetman* the supervisor mode in MIPS is a historical accident, a feature that was added by a customer request, but was practially never used in a real system. So only the user and kernel modes are really used.

In the latest releases of MIPS32 architecture there are some additional modes: root mode and guest mode. These modes are related to hardware-assisted CPU virtualization, which should not to be confused with the virtual memory feature. The virtual memory is present even in CPU cores without hardware-assisted virtualization, including MIPS microAptiv UP core used in

MIPSfpga. However the hardware-assisted virtualization is not present in MIPSfpga, so we are not concern about root and guest modes here.

The processor in kernel mode has complete access to all its architecture resources, it can execute all instructions, access all memory locations and all system Coprocessor 0 registers. This mode is used to run operating system and device drivers.

The processor in user mode can execute only a subset of instructions, has no access to system coprocessor registers, and controlled access to memory locations. This mode is used to run application programs that may be less reliable than the operating system.

When an interrupt or, in general, an exception occures, a processor switches its mode from user to kernel and jumps to an exception handler, a piece of code that handles the exception. When process the exception, the operating system performs the necessary actions. Examples of such actions:

- If the user code attempted to access a location in memory it should not acess, the operating system can terminate the user code, keeping the memory of the rest of the system (both OS and other application's memory) in order.
- If the exception is caused by the system timer interrupt, the interrupt service routine can switch the task/process context and return from the exception into another user task/process.
- If the interrupt came from an input-output device, the interrupt service routine can perform some action related to access to memory-mapped input-output registers. An example of such action is getting a byte transmitted by SPI protocol and putting it into a buffer that can be later accessed by the application code.
- There are also special user-mode instructions that cause the intentional exeptions, requesting the system to perform some action in kernel mode. In MIPS such instruction is called *syscall.*
- Finally, an exception can be caused by an illegal instruction or an instruction that is not permitted in user mode. The exception handler can try to emulate the instruction instead of terminating the user code.

Create a lab using MIPSfpga that demonstrates how the processor entering the user mode. This lab can expose *KSU* ("kernel/supervisor/user") field of Coprocessor 0 *Status* register by connecting it to the LEDs on FPGA board and running the system using slow 0.75 Hz or 12.5 Hz clock.

The main challenge in creating this lab is the need to modify the boot code of the software example. The current boot code exists the reset into kernel mode, this should be changed into user mode. The current boot code also does not initialize virtual address mapping for the user mode. See the documentation on translation lookaside buffer (TLB) and memory-management unit and modify the boot code to support the new lab.

Exercise 8. Special kinds of interrupts, interrupt features and interrupt-like exceptions

Exercise 8.1. Student project and investigation: Non-maskable-interrupt (NMI) lab

Study the information about non-maskable-interrupts (NMI) in *MIPS32® microAptiv™ UP Processor Core Family Software User's Manual* and *MIPS32® microAptiv™ UP Integrator's Guide*, included in *MIPSfpga* package.

Create a lab that demonstrates NMI interrupts. Investigate using the internet the history of NMI in different computers. Why was this feature necessary in the past? How useful is this feature for the modern applications?

Exercise 8.2. Evaluate the usefulness and applications of Vectored Interrupt (VI) mode in MIPS architecture

MIPS microAptiv UP core, used in MIPSfpga, supports three interrupt modes: Interrupt Compatibility mode, Vectored Interrupt (VI) mode, and External Interrupt Controller (EIC) mode. So far, the lab and exercises were using the Interrupt Compatibility mode. An alternative Vectored Interrupt (VI) mode adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt. Vectored Interrupt mode also allows assigning a GPR (General Purpose Register) shadow set for use during interrupt processing.

Study the documentation and create a lab demonstrating interrupts in Vectored Interrupt mode. How many clock cycles does this mode save when an interrupt is taken? When does it make sense to use this mode? Consider very low-power / low-frequency applications, interrupt response time, usage of shadow GPR registers (not present in MIPSfpga).

Which interrupt mode is used in Linux? Can you offer an explanation why?

Exercise 8.3. The purpose of *IV* bit of *Cause* Coprocessor 0 register

Read about *IV* bit of *Cause* Coprocessor 0 register. Why would anybody need such interrupt option? What problem is solved by this option? Propose possible answers.

Exercise 8.4.The purpose of *EBase* Coprocessor 0 register

Read about *EBase* Coprocessor 0 register and try to explain its usefulness. It is only for multiprocessor systems? How would you use it in a single-processor system?

Exercise 8.5. Advanced student project: Building External Interrupt Controller (EIC)

The External Interrupt Controller (EIC) mode is another interrupt mode, supported by MIPS microAptiv UP core used in MIPSfpga. EIC mode redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. EIC mode is used in MIPS-based microcontrollers from Microchip Technology, including some microcontrollers from the Microchip PIC32MZ family that are built around MIPS microAptiv UP core, the same core used in MIPSfpga.

The functionality of External Interrupt Controller created by Microchip is described in Microchip software documentation, the courses taught by Microchip during Microchip Master Conference, as well as in Imagination, Microchip and Digilent-sponsored course *Connected MCU* created by Dr. Alexander Dean of North Carolina State University.

MIPSfpga allows you to create an alternative External Interrupt Controller, with its own interrupt scheduling and prioritization, and compare your solution to the solution from Microchip Technology. This is a significant research project that can be combined with studies on Real-Time Operating Systems (RTOS).

Exercise 8.6. Student project: Create a lab demonstrating imprecise Bus Error exception

All interrupts are exceptions triggered by changes in external signals, but not all exceptions triggered by changes in external signals are in fact interrupts. There is an exception called Bus Error, which is triggered by a specific AHB-Lite bus response. This exception is also remarkable because it is not always precise, i.e. it may occur with *EPC* pointing to a different instruction.

Read about imprecise exceptions and bus errors in documentation and create a lab that demonstrates this feature. The useful chapters to read are:

- *MIPS32® microAptiv™ UP Integrator's Guide*, *Table 2.3 Signal Descriptions for m14k_cpu Level*, the description of *HRESP* signal: *"The transfer response. When LOW, the HRESP signal indicates that the transfer status is OKAY. When HIGH, the HRESP signal indicates that the transfer status is ERROR."*. This manual is included in *MIPSfpga* package.
- *MIPS32® microAptiv™ UP Processor Core Family Software User's Manual*, section *5.8.15 Bus Error Exception - Instruction Fetch or Data Access*. Notice the statement *"Bus errors taken on the requested (critical) word of an instruction fetch or data load are precise. Other bus errors, such as stores or non-critical words of a burst read, can be imprecise."* This manual is included in *MIPSfpga* package.
- Book *See MIPS Run, Second Edition, by Dominic Sweetman*, section *5.1 Precise Exceptions* and *Table 3.2 ExcCode Values: Different Kinds of Exceptions*.

Exercise 9. Effect of interrupts on the processor microarchitecture

Advanced student project: Create a lab demonstrating the effect of interrupts on processor pipelining

Create all the necessary scripts to show the main interrupt-related signals on the waveform during the simulation in Verilog simulator. Show on the waveform how an interrupt flushes the processor pipeline. Which stage of the pipeline is used to take a pending interrupt?

Connect the pipeline control signals to external LEDs on FPGA board and demonstrate how an interrupt flushes the processor pipeline from the instructions which are not going to graduate because of the interrupt.

If you need more information about CPU pipelining in MIPSfpga, also see Labs 14-19.

Appendix A. A list of recommended materials to review before and during the interrupt lab

- Book *See MIPS Run, Second Edition, by Dominic Sweetman*. *Chapter 5. Exceptions, Interrupts, and Initialization*, sections 5.1-5.8. If you don't have this book, see *Appendix B* in this lab for critical excerpts from Sweetman's book. The specific book sections to study:
    - *Chapter 5. Exceptions, Interrupts, and Initialization*, sections 5.1-5.8
    - *Chapter 3. Coprocessor 0: MIPS Process Control*, sections:
        - *3.3.1 Status Register (SR)*
        - *3.3.2 Cause Register*
        - *3.3.3 Exception Restart Address (EPC) Register*
        - *3.3.5 Count/Compare Registers: The On-CPU Timer*
        - *3.3.8 EBase and IntCtl: Interrupt and Exception Setup*
        - *3.3.9 SRSCtl and SRSMap: Shadow Register Setup*
        - *3.3.10 Load-Linked Address (LLAddr) Register*
    - *Chapter 8. Complete Guide to the MIPS Instruction Set*, section *8.5.2 Load-Linked / Store-Conditional*
- *MIPS32® microAptiv™ UP Integrator's Guide, Chapter 4: Interrupt Interface*. This manual is included in *MIPSfpga* package.
- *MIPS32® microAptiv™ UP Processor Core Family Software User's Manual*, relevant sections from:
    - *Chapter 5: Exceptions and Interrupts in the microAptiv™ UP Core*
    - *Chapter 6: CP0 Registers of the microAptiv™ UP Core*
    - section 12.3, the descriptions of *LL* and *SC* instructions

  This manual is included in *MIPSfpga* package.

- Relevant sections from the manual *MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture*. This manual can be downloaded from Imagination Technologies website http://imgtec.com

Appendix B. Suggested further reading and excerpts from the book *See MIPS Run, Second Edition, by Dominic Sweetman* about the interrupt processing

B.1. Review the information about interrupt-related hardware signals

Review *MIPS32® microAptiv™ UP Integrator's Guide*, *Chapter 4: Interrupt Interface*. This manual

B.2. Review the material that explains the *interrupt* function attribute

Use the following internet material to make sense of interrupt-specific function attributes in the code: *Using the GNU Compiler Collection (GCC). 6.31.18 MIPS Function Attributes* (http://gcc.gnu.org/onlinedocs/gcc/MIPS-Function-Attributes.html).

B.3. Review C macros that are used to access Coprocessor 0 registers

Search for *mips/cpu.h* header file in Codescape GCC compiler package. This file contains a set of macro definitions like *mips32_getcr* and *mips32_bicsr* that aid in accessing Coprocessor 0 registers. These registers are needed to setup the interrupts.

B.4. Review *Status* and *Cause* Coprocessor 0 register descriptions

The following materials are useful for the review:

- Book *See MIPS Run, Second Edition, by Dominic Sweetman*, *Chapter 3. Coprocessor 0: MIPS Process Control*
- *MIPS32® microAptiv™ UP Processor Core Family Software User's Manual*, *Chapter 6: CP0 Registers of the microAptiv™ UP Core*
- *MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture*

B.5. Excerpt from *See MIPS Run, Second Edition, by Dominic Sweetman*: Chapter 5. Exceptions, Interrupts, and Initialization

In the MIPS architecture interrupts, traps, system calls, and everything else that can disrupt the normal flow of execution are called exceptions and are handled by a single mechanism. What sort of events are they?

- *External events*: Some event outside the CPU core—that is, from some real "wire" input signal. These are interrupts. (Note: There are some more obscure noninterrupt external events like bus errors reported on a read—for now, just assume that they are a special sort of interrupt). Interrupts are used to direct the attention of the CPU to some external event: an essential feature of an OS that attends to more than one different event at a time. Interrupts are the only exception conditions that arise from something independent of the CPU's normal instruction stream. Since you can't avoid interrupts just by being careful, there have to be software mechanisms to inhibit the effect of interrupts when necessary.

- *Memory translation exceptions*: These happen when an address needs to be translated, but no valid translation is available to the hardware or perhaps on a write to a write-protected page.

  The OS must decide whether such an exception is an error or not. If the exception is a symptom of an application program stepping outside its permitted address space, it will be fixed by terminating the application to protect the rest of the system. The more common benign memory translation exceptions can be used to initiate operating system functions as complex as a complete demand-paged virtual memory system or as simple as extending the space available for a stack.

- *Other unusual program conditions for the kernel to fix*: Notable among these are conditions resulting from floating-point instructions, where the hardware is unable to cope with some difficult and rare combination of operation and operands and is seeking the services of a software emulator. This category is fuzzy, since different kernels have different ideas about what they're willing to fix. An unaligned load may be an error on one system and something to be handled in software on another.
- *Program or hardware-detected errors*: This includes nonexistent instructions, instructions that are illegal at user-privilege level, coprocessor instructions executed with the appropriate SR flag disabled, integer overflow, address alignment errors, and accesses outside *kuseg* in user mode.
- *Data integrity problems*: Many MIPS CPUs continually check data on the bus or data coming from the cache for a per-byte parity or for word- wide error-correcting code. Cache or parity errors generate an exception in CPUs that support data checking.
- *System calls and traps*: These are instructions whose whole purpose is to generate recognizable exceptions; they are used to build software facilities in a secure way (system calls, conditional traps planted by careful code, and breakpoints).

5.3 Exception Vectors: Where Exception Handling Starts

. . . .

Here's what a MIPS CPU does when it decides to take an exception:

1. It sets up **EPC** to point to the restart location.
2. It sets **SR(EXL)**, which forces the CPU into kernel (high-privilege) mode and disables interrupts.
3. **Cause** is set up so that software can see the reason for the exception. On address exceptions, **BadVAddr** is also set. Memory management system exceptions set up some of the MMU registers too; more details are given in Chapter 6.
4. The CPU then starts fetching instructions from the exception entry point, and everything else is up to software.

. . . .

5.5 Returning from an Exception

The return of control to the exception victim and the change (if required) back from kernel to a lower-privilege level must be done at the same time (“atomically,” in the jargon of computer science). It would be a security hole if you ran even one instruction of application code at kernel-privilege level; on the other hand, the attempt to run a kernel instruction with user privileges would lead to a fatal exception.

MIPS CPUs have an instruction, **eret**, that does the whole job; it both clears the **SR(EXL)** bit and returns control to the address stored in **EPC**.

5.8.1 Interrupt Resources in MIPS CPUs

MIPS CPUs have a set of eight independent (Note: Not so independent if you're using EIC mode; see section 5.8.5.) interrupt bits in their **Cause** register. On most CPUs you'll find five or six of these are signals from external logic into the CPU, while two of them are purely software accessible. The on-chip counter/timer (made of the **Count** and **Compare** registers, described in section 3.3.5) will be wired to one of them; it's sometimes possible to share the counter/timer interrupt with an external device, but rarely a good idea to do so.

An active level on any input signal is sensed in each cycle and will cause an exception if enabled.

The CPU's willingness to respond to an interrupt is affected by bits in **SR**.

There are three relevant fields:

- The global interrupt enable bit **SR(IE)** must be set to 1, or no interrupt will be serviced.
- The **SR(EXL)** (exception level) and **SR(ERL)** (error level) bits will inhibit interrupts if set (as one of them will be immediately after any exception).
- The status register also has eight individual interrupt mask bits **SR(IM)**, one for each interrupt bit in **Cause**. Each **SR(IM)** bit should be set to 1 to enable the corresponding interrupt so that programs can determine exactly which interrupts can happen and which cannot.

To discover which interrupt inputs are currently active, you look inside the **Cause** register. Note that these are exactly that—current levels—and do not necessarily correspond to the signal pattern that caused the interrupt exception in the first place. The active input levels in **Cause(IP)** and the masks in **SR(IM)** are helpfully aligned to the same bit positions, in case you want to “and” them together. The software interrupts are at the lowest positions, and the hardware interrupts are arranged in increasing order.

In architectural terms, all interrupts are equal. (Note: That's not quite true in vectored interrupt and “EIC mode,” described in section 5.8.5, but they're not widely used). When an interrupt exception is taken, an older CPU uses the “general” exception entry point—though MIPS 32/64 CPUs and some other modern CPUs offer an optional distinct exception entry point reserved for interrupts, which can save a few cycles. You can select this with the **Cause(IV)** register bit.

Interrupt processing proper begins after you have received an exception and discovered from **Cause(ExcCode)** that it was a hardware interrupt. Consulting **Cause(IP)**, we can find which interrupt is active and thus which device is signaling us. Here is the usual sequence:

- Consult the **Cause** register IP field and logically “and” it with the current interrupt masks in **SR(IM)** to obtain a bit map of active, enabled interrupt requests. There may be more than one, any of which would have caused the interrupt.
- Select one active, enabled interrupt for attention. Most OSs assign the different inputs to fixed priorities and deal with the highest priority first, but it is all decided by the software.
- You need to save the old interrupt mask bits in **SR(IM)**, but you probably already saved the whole **SR** register in the main exception routine.
- Change **SR(IM)** to ensure that the current interrupt and all interrupts your software regards as being of equal or lesser priority are inhibited.
- If you haven't already done it in the main exception routine, save the state (user registers, etc.) required for nested exception processing.
- Now change your CPU state to that appropriate to the higher-level part of the interrupt handler, where typically some nested interrupts and exceptions are permitted.

  In all cases, set the global interrupt enable bit **SR(IE)** to allow higher priority interrupts to be processed. You'll also need to change the CPU privilege-level field **SR(KSU)** to keep the CPU in kernel mode as you clear exception level and, of course, clear **SR(EXL)** itself to leave exception mode and expose the changes made in the status register.

- Call your interrupt routine.
- On return you'll need to disable interrupts again so you can restore the preinterrupt values of registers and resume execution of the interrupted task. To do that you'll set **SR(EXL)**. But in practice you're likely to do this implicitly when you restore the just-after-exception value of the whole **SR** register, before getting into your end-of-exception sequence.

When making changes to **SR**, you need to be careful about changes whose effect is delayed due to the operation of the pipeline — “CP0 hazards.” See section 3.4 for more details and how to program around the hazards.

Appendix C. Excerpts from the article *Using the GNU Compiler Collection (GCC). 6.31.18 MIPS Function Attributes*

http://gcc.gnu.org/onlinedocs/gcc/MIPS-Function-Attributes.html

These function attributes are supported by the MIPS back end:

interrupt

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present. An optional argument is supported for the interrupt attribute which allows the interrupt mode to be described. By default GCC assumes the external interrupt controller (EIC) mode is in use, this can be explicitly set using eic. When interrupts are non-masked then the requested Interrupt Priority Level (IPL) is copied to the current IPL which has the effect of only enabling higher priority interrupts. To use vectored interrupt mode use the argument vector=[sw0|sw1|hw0|hw1|hw2|hw3|hw4|hw5], this will change the behavior of the non-masked interrupt support and GCC will arrange to mask all interrupts from sw0 up to and including the specified interrupt vector.

You can use the following attributes to modify the behavior of an interrupt handler:

use_shadow_register_set

Assume that the handler uses a shadow register set, instead of the main general-purpose registers. An optional argument intstack is supported to indicate that the shadow register set contains a valid stack pointer.

keep_interrupts_masked

Keep interrupts masked for the whole function. Without this attribute, GCC tries to reenable interrupts for as much of the function as it can.

use_debug_exception_return

Return using the deret instruction. Interrupt handlers that don't have this attribute return using eret instead.

You can use any combination of these attributes, as shown below:

```
void __attribute__ ((interrupt)) v0 ();

void __attribute__ ((interrupt, use_shadow_register_set)) v1 ();
void __attribute__ ((interrupt, keep_interrupts_masked)) v2 ();

void __attribute__ ((interrupt, use_debug_exception_return)) v3 ();

void __attribute__ ((interrupt, use_shadow_register_set,
            keep_interrupts_masked)) v4 ();

void __attribute__ ((interrupt, use_shadow_register_set,
            use_debug_exception_return)) v5 ();

void __attribute__ ((interrupt, keep_interrupts_masked,
            use_debug_exception_return)) v6 ();

void __attribute__ ((interrupt, use_shadow_register_set,
            keep_interrupts_masked,
            use_debug_exception_return)) v7 ();

void __attribute__ ((interrupt("eic"))) v8 ();

void __attribute__ ((interrupt("vector=hw3"))) v9 ();
```