



Lab YP4

Introducing Interrupts



These materials produced in association with Imagination.
Join our University community for more resources.

community.imgtec.com/university

MIPSfpga 2.0. Lab YP4 – Introducing interrupts

1. Introduction

This lab shows the basic usage of interrupts in MIPS CPUs. The lab also demonstrates how the interrupts can offload the processor from constantly polling I/O ports. Such offloading increases 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.

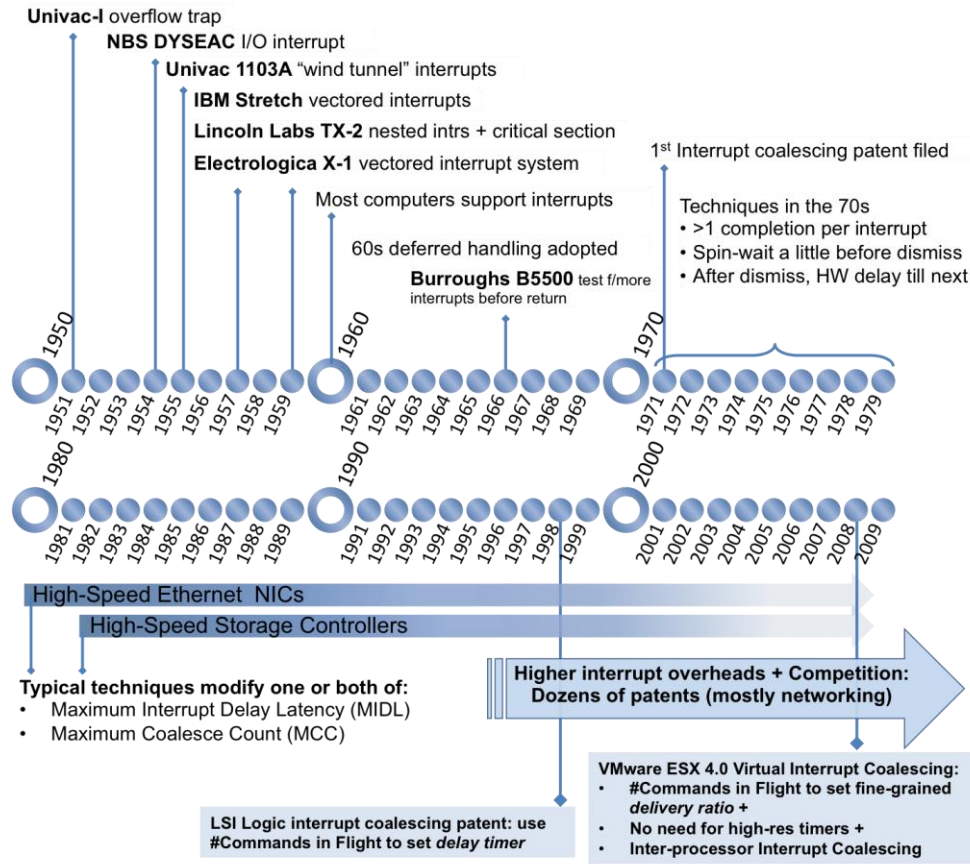
This lab can be also combined with the previous lab, *MIPSfpga 2.0. Lab YP3 – Integrating a peripheral: the light sensor example*, to create a student project that demonstrates the interrupt-driven approach to input/output, and compares this approach to polling I/O ports without using interrupts.

2. The theory of operation

Interrupt is one of the key concepts in computer programming and system design. Interrupt is basically a feature of a CPU that forces the processor to suspend the regular flow of instruction execution and to 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 on [Figure 1](#).

Figure 1. The action of an I/O interrupt. The source of the figure: <http://virtualirfan.com/history-of-interrupts>.

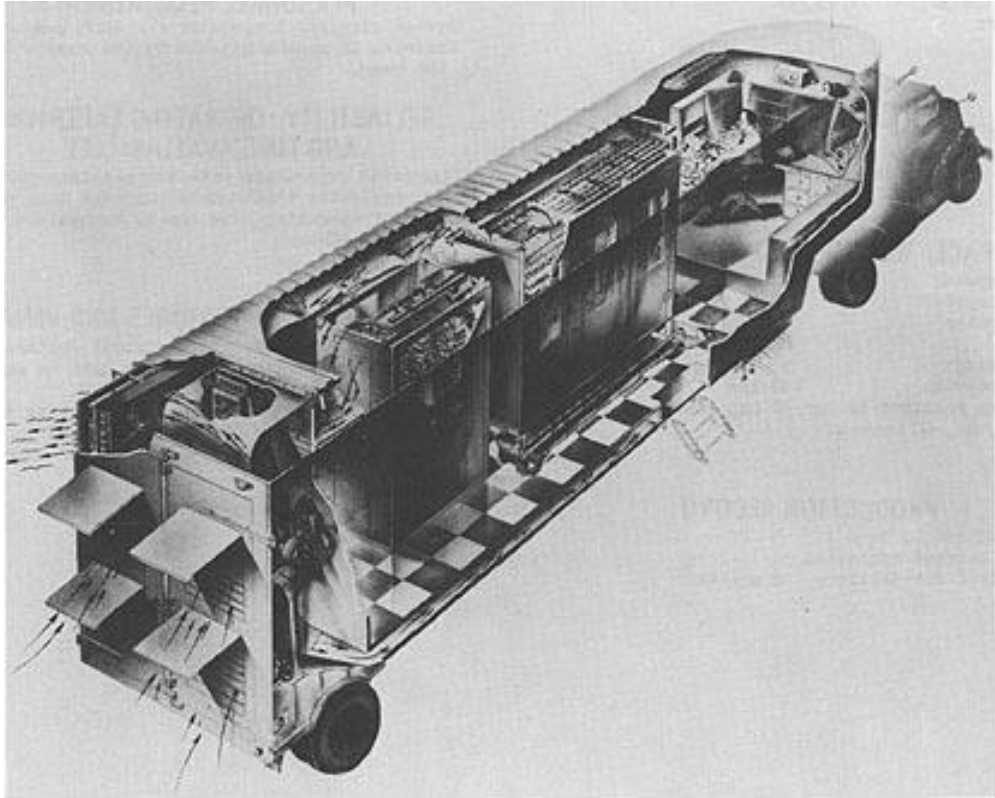


Source: Mark Smotherman <http://www.cs.clemson.edu/~mark/interrupts.html>

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. The source of the figure: <http://ed-thelen.org/comp-hist>.



3. Lab steps

This section outlines the sequence of steps, necessary to complete the lab. Almost all generic steps in this lab are the same as in *MIPSfpga 2.0 Lab YP1. Using MIPSfpga with Serial Loader Flow that does not require BusBlaster board and OpenOCD software*. Such generic steps are not described in this section. Only the steps different from *Lab YP1* are explained in details.

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 get understanding 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 interrupt pin signal *SI_Int* used in the lab. In the course of the lab this multi-bit signal is connected to the buttons on FPGA board. This allows triggering different interrupt by pressing the corresponding buttons.

3.3. Setup the hardware configuration parameters

Modify the configuration parameters in the file *system_rtl/mfp_ahb_lite_matrix_config.vh* as follows:

```

17 //
18 // Configuration parameters
19 //
20
21 // `define MFP_USE_WORD_MEMORY
22 // `define MFP_INITIALIZE_MEMORY_FROM_TXT_FILE
23 `define MFP_USE_SLOW_CLOCK_AND_CLOCK_MUX
24 `define MFP_USE_UART_PROGRAM_LOADER
25 // `define MFP_DEMO_LIGHT_SENSOR
26 `define MFP_DEMO_INTERRUPTS
27 // `define MFP_DEMO_CACHE_MISSES
28 // `define MFP_DEMO_PIPE_BYPASS

```

3.4. Review the lab-specific hardware modifications

Review the following fragment of *system_rtl/mfp_system.v*.

```

311 `ifdef MFP_DEMO_INTERRUPTS
312     assign SI_Int          = { 4'b0, IO_Buttons [3:0] };
313 `else
314     assign SI_Int          = 8'b0;
315 `endif

```

3.5. Connect the board to the computer

For *Digilent* boards, such as *Nexys4*, *Nexys4 DDR* or *Basys3*, this step is obvious.

For *Altera*/*Terasic* boards some additional steps required:

1. Connect USB-to-UART connector to FPGA board. Either *FT232RL* or *PL2303TA* that you can buy from AliExpress or Radio Shack will do the job. *TX* output from the connector (green wire on *PL2303TA*) should go to pin 3 from right bottom on Terasic DE0, DE0-CV, DE1, DE2-115 (right top on DE0-Nano) and *GND* output (black wire on *PL2303TA*) should be connected to pin 6 from right bottom on Terasic DE0, DE0-CV, DE1, DE2-115 (right top on DE0-Nano). Please consult photo picture in *Lab YP1* to avoid short-circuit or other connection problems.
2. For *FT232RL* connector: make sure to set 3.3V/5V jumper on *FT232RL* part to 3.3V.
3. For the boards that require external power in addition to the power that comes from USB, connect the power supply. The boards that require the extra power supply include *Terasic DE2-115*.
4. Connect FPGA board to the computer using main connection cable provided by the board manufacturers. Make sure to put USB cable to the right jack when ambiguity exists (such as in *Terasic DE2-115* board).
5. Make sure to power the FPGA board (turn on the power switch) before connecting the UART cable from USB-to-UART connector to the computer. Failing to do so may result in electric damage to the board.
6. Connect USB-to-UART connector to FPGA board.

3.6. Run the synthesis and configure the FPGA with the synthesized MIPSfpga system

This step is identical to the synthesis step in *Lab YP1*

3.7. Go to the lab directory and clean it up

Under Windows:

```
cd programs\04_interrupts
00_clean_all.bat
```

Under Linux:

```
cd programs/04_interrupts
./00_clean_all.sh
```

3.8. Review the portion of the lab program code that does not use interrupts

The *main()* function is located in file *programs/04_interrupts/main.c*. This function is executed after the reset and running the boot sequence. The main function in this lab simply runs a counter and outputs its value on a multi-digit seven-segment display on FPGA board.

The output value is constructed from both high and low bits of the counter. This allows the student to observe the digits on the display changing when running the synthesized system with either fast clock (50 MHz) and ultra-slow clock (0.75 Hz or 12.5 Hz). For more details about the switchable clock see *MIPSfpga 2.0 Lab YP2. Using switchable clock to observe the CPU cycle-by-cycle*.

Depending on the position of a switch 2, the program can either run with or without interrupts. In both cases it resets the counter to predefined values when either button 0 or button 1 is pressed.

Note that on some FPGA boards button 0 is used as system reset. For such boards you can either do the whole lab only using button 1 or resynthesize the system, connecting reset to a button or a switch other than button 0.

```

17 int main ()
18 {
19     if (MFP_SWITCHES & 4) // Switch 2 is on
20     {
21         // Count without interrupts, polling buttons in the loop
22
23         for (n = 0;; n++)
24         {
25             unsigned buttons = MFP_BUTTONS;
26
27             if (buttons & 1)
28                 n = 0;
29             else if (buttons & 2)
30                 n = 0x100000;
31
32             // Assigning to 7-segment indicator this value
33             // makes count visible with both slow and fast clock
34
35             MFP_7_SEGMENT_HEX = ((n >> 8) & 0xfffff00) | (n & 0xff);
36         }
37     }

```

3.9. Prepare the first software run

Following the procedure described in *Lab YPI*, compile and link the program, generate Motorola S-Record file and upload this file into the memory of the synthesized MIPSfpga-based system on the board.

Under Windows:

1. cd programs\04_interrupts
2. run 02_compile_and_link.bat
3. run 08_generate_motorola_s_record_file.bat
4. run 11_check_which_com_port_is_used.bat
5. edit 12_upload_to_the_board_using_uart.bat based on the result from the previous step – set the working port in “set a=” assignment.
6. Make sure the switches 0 and 1 on FPGA board are turned off and switch 2 is turned on. Switches 0 and 1 control the speed of the clock, while switch 2 determines whether the program uses interrupts (switch 2 is off) or does not use interrupts (switch 2 is on). See *3.8. Review the lab program code*. If the switches 0 and 1 are not off, the loading through UART is not going to work.
7. run 12_upload_to_the_board_using_uart.bat

Under Linux:

If uploading program to the board first time during the current Linux session, add the current user to *dialout* Linux group. Enter the *root* password when prompted:

```

sudo adduser $USER dialout
su - $USER

```

After that:

1. cd programs/04_interrupts
2. run ./02_compile_and_link.sh

3. run `./08_generate_motorola_s_record_file.sh`
4. run `./11_check_which_com_port_is_used.sh`
5. edit `./12_upload_to_the_board_using_uart.sh` based on the result from the previous step – set the working port in “set a=” assignment
6. Make sure the switches 0 and 1 on FPGA board are turned off and switch 2 is turned on. Switches 0 and 1 control the speed of the clock, while switch 2 determines whether the program uses interrupts (switch 2 is off) or does not use interrupts (switch 2 is on). See *3.8. Review the lab program code*. If the switches 0 and 1 are not off, the loading through UART is not going to work.
7. `./run 12_upload_to_the_board_using_uart.sh`

3.10. The first run

1. Set the switches 0 and 1 on FPGA board to off position and switch 2 to on position. Make sure the switches 0 and 1 are off, otherwise the boot sequence (a sequence of processor instructions before *main* function) will take too long, since these switches control the clock frequency.
2. Reset the processor. The reset buttons for each board are listed in the table below:

Board	Reset button
Digilent Basys3	Up
Digilent Nexys4	Dedicated CPU Reset
Digilent Nexys4 DDR	Dedicated CPU Reset
Terasic DE0	Button/Key 0
Terasic DE0–CV	Dedicated reset button
Terasic DE0–Nano	Button/Key 0
Terasic DE1	Button/Key 0
Terasic DE2–115	Button/Key 0

Terasic DE10-Lite	Button/Key 0
-------------------	--------------

3. Notice the output on seven-segment display. Press button 1, observe how it resets the counter value.
4. Turn the switch 1 on. This will switch the system clock from 25 MHz to 12.5 Hz / beats per second. You should see LED 7 start blinking, it is connected straight to the system clock. Notice the speed of counting.
5. Press button 1, observe how it resets the counter value.

3.12. 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>).

3.13. 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_bicr* that aid in accessing Coprocessor 0 registers. These registers are needed to setup the interrupts.

3.14. 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*

3.15. Review the portion of the lab program code that uses interrupts

The interrupt handler in *programs/04_interrupts/main.c*.

```

1  #include <mips/cpu.h>
2
3  #include "mfp_memory_mapped_registers.h"
4
5  volatile int n;
6
7  void __attribute__((interrupt, keep_interruptions_masked)) general_exception_handler ()
8  {
9      unsigned cause = mips32_getcr (); // Coprocessor 0 Cause register
10
11      if (cause & CR_HINT0) // Checking whether interrupt 0 is pending
12          n = 0;
13      else if (cause & CR_HINT1) // Checking whether interrupt 1 is pending
14          n = 0x100000;
15  }
16

```

Setting up the Coprocessor 0 registers for handling the interrupts in *programs/04_interrupts/main.c*.

```

38      else
39      {
40          // Count with interrupts, without polling buttons in the loop
41
42          // Clear boot interrupt vector bit in Coprocessor 0 Status register
43
44          mips32_bicr (SR_BEV);
45
46          // Set master interrupt enable bit, as well as individual interrupt enable bits
47          // in Coprocessor 0 Status register
48
49          mips32_bissr (SR_IE | SR_HINT0 | SR_HINT1 | SR_HINT2 | SR_HINT3 | SR_HINT4 | SR_HINT5);
50
51          for (n = 0;;)
52          {
53              MFP_7_SEGMENT_HEX = ((n >> 8) & 0xfffff00) | (n & 0xff);
54
55              asm ("di"); // Disable interrupts
56              n ++;
57              asm ("ei"); // Enable interrupts
58          }
59      }
60
61      return 0;
62  }

```

Code to setup the exception vector in *programs/04_interrupts/exceptions.S*:

```

1      .set    noreorder      # Don't allow the assembler to reorder instructions.
2      .set    noat
3
4      .section .exceptions
5
6      .org    0x180
7
8      general_exception_vector:
9
10     .type    general_exception_vector, @function
11
12     j        general_exception_handler
13     nop

```

The linker script which includes a new section for the exception vector in *programs/04_interrupts/program.ld*:

```
59  SECTIONS
60  {
61      /*** Exception vectors ***/
62
63      .exceptions 0x80000000 : /* Exception vectors. */
64      {
65          *(.exceptions)
66          /* For some reason the following is necessary for
67             this section to be kept when .rec is produced */
68          BYTE(0)
69      } = 0
```

3.16. Running the program that uses interrupts

Re-run the step 3.10. *The first run* with switch 2 in "off" position. Do you notice the difference in program performance? Can you explain the difference?

4. Homework: doing everything in assembly

This section guides the student through the activity of analyzing the interrupt processing on assembly level. This activity helps to have a clear picture on how everything works.

4.1. Disassemble the ELF file created during the lab

Under Windows:

```
cd programs\04_interrupts
04_disassemble.bat
```

Under Linux:

```
cd programs/04_interrupts
./04_disassemble.sh
```

The generated file is called *program.dis*. It is located in the current directory.

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

Analyze the following sections of *programs/04_interrupts/program.dis* file:

The code for the interrupt vector:

```

1
2 program.elf:      file format elf32-tradlittlemips
3
4
5 Disassembly of section .exceptions:
6
7 80000000 <general_exception_vector-0x180>:
8     ...
9
10 80000180 <general_exception_vector>:
11 80000180:      08000469      j      800011a4 <general_exception_handler>
12 80000184:      00000000      nop
13     ...

```

The code for the interrupt handler:

```

369 800011a4 <general_exception_handler>:
370 800011a4:      401b7000      mfc0    k1,c0_epc
371 800011a8:      27bdfff0      addiu   sp,sp,-16
372 800011ac:      afbb000c      sw      k1,12(sp)
373 800011b0:      401b6000      mfc0    k1,c0_status
374 800011b4:      afbb0008      sw      k1,8(sp)
375 800011b8:      7c1b2004      ins     k1,zero,0x0,0x5
376 800011bc:      409b6000      mtc0    k1,c0_status
377 800011c0:      afa30004      sw      v1,4(sp)
378 800011c4:      afa20000      sw      v0,0(sp)
379 800011c8:      40026800      mfc0    v0,c0_cause
380 800011cc:      30430400      andi    v1,v0,0x400
381 800011d0:      1460000c      bnez    v1,80001204 <general_exception_handler+0x60>
382 800011d4:      30420800      andi    v0,v0,0x800
383 800011d8:      10400002      beqz    v0,800011e4 <general_exception_handler+0x40>
384 800011dc:      3c020010      lui     v0,0x10
385 800011e0:      af82801c      sw      v0,-32740(gp)
386 800011e4:      8fa30004      lw      v1,4(sp)
387 800011e8:      8fa20000      lw      v0,0(sp)
388 800011ec:      8fbb000c      lw      k1,12(sp)
389 800011f0:      409b7000      mtc0    k1,c0_epc
390 800011f4:      8fbb0008      lw      k1,8(sp)
391 800011f8:      27bd0010      addiu   sp,sp,16
392 800011fc:      409b6000      mtc0    k1,c0_status
393 80001200:      42000018      eret
394 80001204:      8fa30004      lw      v1,4(sp)
395 80001208:      8fa20000      lw      v0,0(sp)
396 8000120c:      8fbb000c      lw      k1,12(sp)
397 80001210:      af80801c      sw      zero,-32740(gp)
398 80001214:      409b7000      mtc0    k1,c0_epc
399 80001218:      8fbb0008      lw      k1,8(sp)
400 8000121c:      27bd0010      addiu   sp,sp,16
401 80001220:      409b6000      mtc0    k1,c0_status
402 80001224:      42000018      eret

```

The code for setting up the interrupts:

```

453 800012e0: 40026000 mfc0 v0,c0_status
454 800012e4: 7c02b584 ins v0,zero,0x16,0x1
455 800012e8: 40826000 mtc0 v0,c0_status
456 800012ec: 000000c0 ehb
457 800012f0: 40026000 mfc0 v0,c0_status
458 800012f4: 3442fc01 ori v0,v0,0xfc01
459 800012f8: 40826000 mtc0 v0,c0_status
460 800012fc: 000000c0 ehb

```

The code that guards incrementing the counter in the main loop against interrupts. Such guarding is accomplished using *di/ei* instruction pair (disable interrupts / enable interrupts). Can you explain why is it necessary? What wrong could happen if these instructions are missing? Can you guess a scenario when such guarding is not going to work?

```

470 80001324: 41606000 di
471 80001328: 8f82801c lw v0,-32740(gp)
472 8000132c: 24420001 addiu v0,v0,1
473 80001330: af82801c sw v0,-32740(gp)
474 80001334: 1000fff4 b 80001308 <main+0x98>
475 80001338: 41606020 ei

```

4.3. 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?

5. Follow-up projects and exercises

The topic of interrupts is an important one, and we recommend additional exercises to understand it fully.

5.1 Lab enhancements

5.1.1 Lab enhancement: 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.

5.1.2 Student exercise: 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 routine. For more details about the switchable clock see *MIPSfpga 2.0 Lab YP2. Using switchable clock to observe the CPU cycle-by-cycle.*

5.1.3 Student investigation: 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, *SL_EXL*. Exposing such fields requires only changes in the system RTL (*system_rtl/mfp_*.v* files). Other fields may require adding additional ports and connections to the core's RTL (*core_rtl/m14k_*.v* files) as well.

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

5.2 Synchronising memory access in interrupts

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

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:

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

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.

5.2.2 Exercise: Synchronize the updates of a counter variable using *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](#)

5.2.3 Investigation: Why do we need *volatile* qualifier for the counter variable?

Review the code of *main.c*. 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.

5.3 Timer interrupts

5.3.1 Student project: 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.

5.3.2 Student project: 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 *SL_Int* external signal.

5.3.3 Student project: 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 *SL_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 *Lab YP3 – Integrating a peripheral: light sensor example*.

5.3.4 Student project: 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 frequencies on different hardware pins (bits of the signal *SL_Int*).

5.3.5 Medium-to-advanced student project: Create a lab demonstrating the 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.

5.3.6 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.

5.3.7 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, privileged, 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 practically 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 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 concerned 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 occurs, a processor switches its mode from user to kernel and jumps to an exception handler, a piece of code that handles the exception. When processing 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 access, 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 exceptions, 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 enters 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.

5.4 Special kinds of interrupts, interrupt features and interrupt-like exceptions

5.4.1 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?

5.4.2 Investigation: 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?

5.4.3 Investigation: 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.

5.4.4 Investigation: 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?

5.4.5 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 Diligent-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).

5.4.6 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*.

5.5 Effect of interrupt on processor microarchitecture

5.5.1 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, you can review *MIPSfpga 2.0. Lab YP6 – The first glance into pipelining*.

5.6 Reimplement the light sensor lab making it interrupt-driven

After going through this lab, go back to the light sensor lab *MIPSfpga 2.0. Lab YP3 – Integrating a peripheral: the light sensor example*) and re-implement it using interrupt-driven approach. Modify the light sensor interfacing module so it issues an interrupt when the measured value changes. Connect the interrupt pin to *SL_Int* signal of MIPS microAptiv UP core. Measure the system performance improvement that comes from offloading input-output to the interrupt service routine.

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 SRSSMap: 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. Excerpts from a book *See MIPS Run, Second Edition, by Dominic Sweetman* about the interrupt processing.

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 — "CPO 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 reenables 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 ();
```