Lab 11. Interrupts & IO programming

Goals

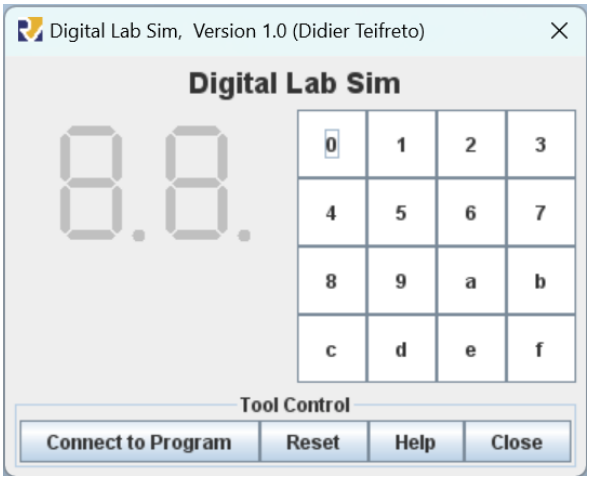
After this laboratory exercise, you should understand the basic principles of interrupts and how interrupts can be used for programming. You should also know the difference between polling and using interrupts and the relative merits of these methods.

Preparation

Polling or Interrupts

A computer can react to external events either by polling or by using interrupts. One method is simpler, while the other one is more systematic and more efficient. You will study the similarities and differences of these methods using a simple "toy" example program.
Each peripheral device connects to the CPU via a few ports. CPU uses address to find out the respective port, and after that, CPU could read/write the new value to these ports to get/control the device.

Home Assignment 1 – POLLING



Write a program using assembly language to detect key pressed in Digi Lab Sim and print the key number to console.
The program has an unlimited loop, to read the scan code of key button. This is POLLING.

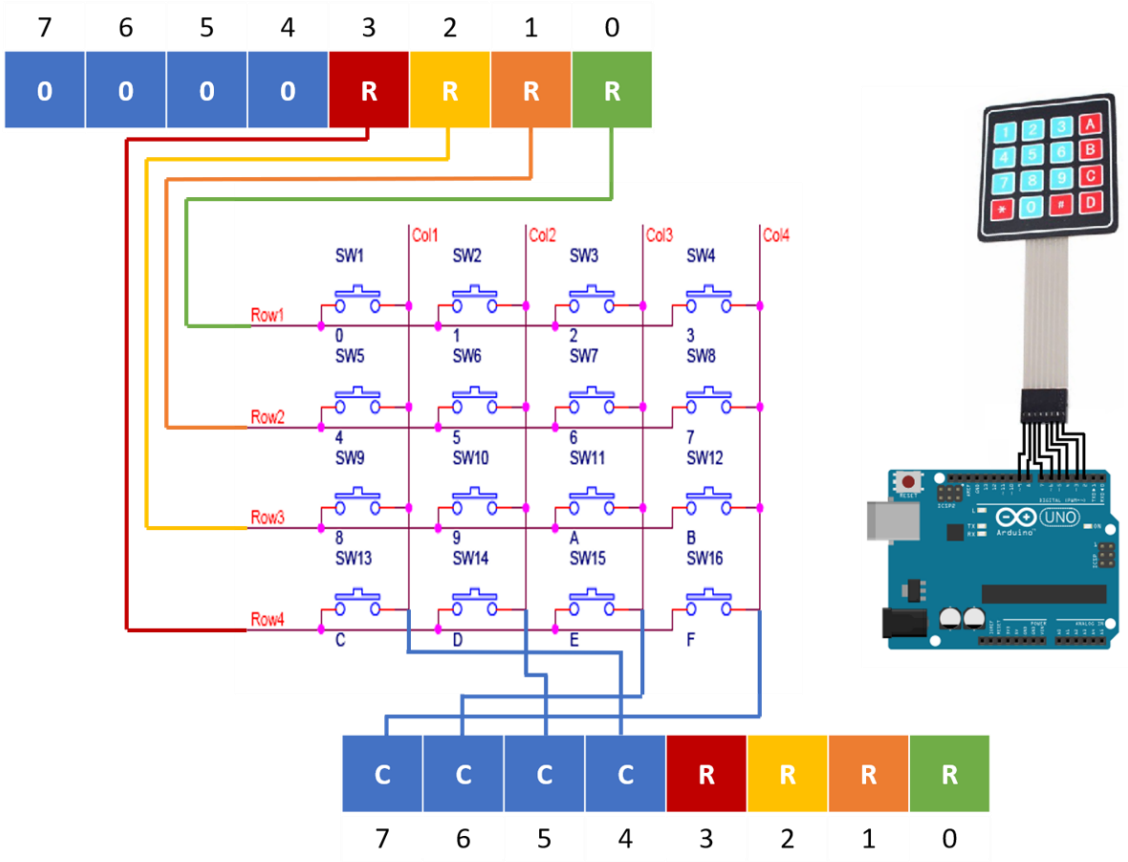
To use the key matrix[1], you should:

1. Assign the expected row index into the byte at the address 0xFFFF0012
2. Read byte at the address 0xFFFF0014, to detect which key button was pressed.

Note: Run the program at the speed of 30 ins/s to avoid RARS stop working.

[1] Key matrix animation: http://hackyourmind.org/public/images/keypad12keys_anim.gif

IN_ADDRESS_HEXA_KEYBOARD
Address **0xFFFF0012**



OUT_ADDRESS_HEXA_KEYBOARD
Address **0xFFFF0014**

```
# -------------------------------------------------------
#           col 0x1    col 0x2    col 0x4     col 0x8
#  row 0x1     0          1          2           3
#             0x11       0x21       0x41        0x81
#  row 0x2     4          5          6           7
#             0x12       0x22       0x42        0x82
#  row 0x4     8          9          a           b
#             0x14       0x24       0x44        0x84
#  row 0x8     c          d          e           f
#             0x18       0x28       0x48        0x88
# -------------------------------------------------------
# Command row number of hexadecimal keyboard (bit 0 to 3)
# Eg. assign 0x1, to get key button 0,1,2,3
#     assign 0x2, to get key button 4,5,6,7
# NOTE must reassign value for this address before reading,
# eventhough you only want to scan 1 row
.eqv IN_ADDRESS_HEXA_KEYBOARD       0xFFFF0012

# Receive row and column of the key pressed, 0 if not key pressed
```

```
# Eg. equal 0x11, means that key button 0 pressed.
# Eg. equal 0x28, means that key button D pressed.
.eqv OUT_ADDRESS_HEXA_KEYBOARD      0xFFFF0014

.text
main:
    li  t1, IN_ADDRESS_HEXA_KEYBOARD
    li  t2, OUT_ADDRESS_HEXA_KEYBOARD
    li  t3, 0x08        # check row 4 with key C, D, E, F

polling:
    sb  t3, 0(t1 )      # must reassign expected row
    lb  a0, 0(t2)       # read scan code of key button
print:
    li  a7, 34          # print integer (hexa)
    ecall
sleep:
    li  a0, 100         # sleep 100ms
    li  a7, 32
    ecall
back_to_polling:
    j     polling       # continue polling
```

Home Assignment 2 – INTERRUPT

Introduction to interrupt and interrupt routine

Interrupts are mechanisms that allow peripheral devices to send notifications to the CPU about events that need attention. When an interrupt occurs, the peripheral device sends a signal to the CPU, when the CPU receives this signal, it will perform the following tasks in the following order:

1. Back up the context of the current program.
2. Execute the interrupt service subroutine.
3. Restore the context and continue executing the main program.

Interrupts can occur from many sources: external interrupts from peripheral devices, timer interrupts, software interrupts or exceptions.

The RISC-V ISA defines three levels of access privileges including **User/Application**, **Supervisor**, and **Machine**. Access privileges define which resources (**registers**, **instructions**, ...) can be accessed by software. This mechanism will limit the execution of software and protect the system from software that intentionally performs unauthorized operations. Machine privileges have the highest level of access, User/Application privileges have the lowest level of access. **RARS simulates the User/Application level.**

Registers used for interrupt handling

The RISC-V architecture defines Control and Status Registers (CSRs) that indicate the state of the CPU and allow software to control the behavior of the CPU. The RISC-V ISA also includes a set of instructions that allow software to read and write the contents of the CSRs. The CSRs related to interrupt handling are:

• **mstatus**: Status register contains fields providing information or control over the interrupt handling mechanism. The **UIE** field (bit 0) enables or disables interrupts.
• **mcause** (Machine Interrupt Cause): Consisting of 2 fields: **INTERRUPT** (bit 31) indicates whether the cause is an interrupt or an exception and the **EXCCODE** field (bits 0 to 30) indicates the cause of the interrupt (or exception).

• **mtvec** (Machine Trap Vector): Register containing information about the subroutine that the CPU will execute when an interrupt occurs.

• **mie** (Machine Interrupt Enable): Register that sets whether to enable or disable specific interrupt sources.

• **mip** (Machine Interrupt Pending): Register containing information about interrupts that are not yet processed by the CPU.

• **mepc** (Machine Exception Program Counter): Register containing the value of the PC register when an interrupt occurs.

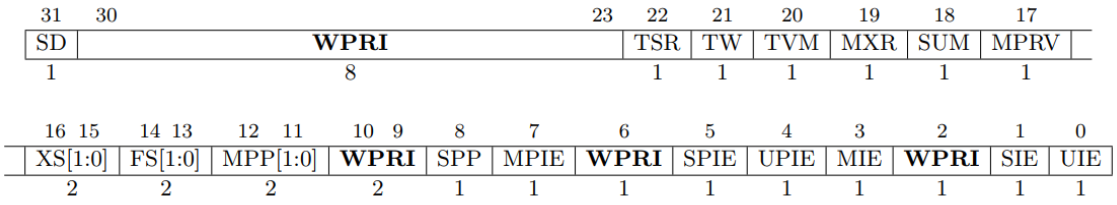


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

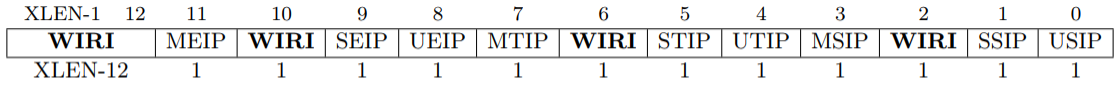


Figure 3.11: Machine interrupt-pending register (`mip`).

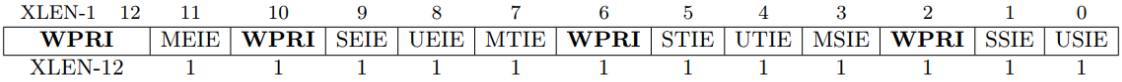


Figure 3.12: Machine interrupt-enable register (`mie`).

Interrupt handling flow (For RARS emulator, User access privilege)

1. Declare the interrupt handling routine (Interrupt Service Routine - ISR), the content of the ISR usually includes:
    a. Save the registers used in the subroutine.
    b. Classify the interrupt, depending on the interrupt type, perform the corresponding processing.
    c. Restore the saved registers.
    d. Return to the main routine
2. Load the interrupt handling routine address into the **mtvec** register.
3. Depending on the program, set the interrupt source in the **mie** register.

4. Enable global interrupts, set the **uie** bit of the **mstatus** register.
5. Set up the simulation tool to enable interrupts (Keypad, Timer Tool, ...)

Note:

- *RARS has renamed the CSRs registers to **ustatus**, **ucause**, **utvec**, **uie**, **uip**, **uepc** to emphasize the simulation in User access mode.*
- *With emulators, it is recommended to press the "Connect to Program" button before running the emulator. Otherwise, the interrupt event will not occur.*
- *Using breakpoints to stop the program when an interrupt occurs will not be effective, you can use the **ebreak** instruction to pause the program.*

RISC-V provides special instructions to manipulate CSRs:

- **csrrc t0, fcsr, t1** Atomic Read / Clear CSR, read from the CSR into t0 and clear bits of the CSR according to t1.
- **csrrci t0, fcsr, 10** Atomic Read / Clear CSR Immediate, read from the CSR into t0 and clear bits of the CSR according to a constant.
- **csrrs t0, fcsr, t1** Atomic Read / Set CSR, read from the CSR into t0 and logical or t1 into the CSR.
- **csrrsi t0, fcsr, 10** Atomic Read / Set CSR Immediate, read from the CSR into t0 and logical or a constant into the CSR.
- **csrrw t0, fcsr, t1** Atomic Read / Write CSR, read from the CSR into t0 and write t1 into the CSR.
- **csrrwi t0, fcsr, 10** Atomic Read / Write CSR Immediate, read from the CSR into t0 and write a constant to the CSR.

Note:

- *The immediate values in the above instructions are limited to the range of 5 bits (from -16 to 15).*
- *CSRs are used as name or number in an instruction (for example ustatus(0), uie(4), uepc(65), etc.)*
- *If the old value of CSR is not important, **zero** register can be used as the destination register.*

For example: **csrrsi zero, uie, 1** => set the first bit and do not store old value of **uie**.

The example below illustrates the setup and handling of interrupts generated by the Keypad tool. Read carefully and understand how the program works.

```
.eqv IN_ADDRESS_HEXA_KEYBOARD       0xFFFF0012
.data
    message: .asciz "Someone's presed a button.\n"
# ---------------------------------------------------------------
# MAIN Procedure
# ---------------------------------------------------------------
.text
```

```
main:
    # Load the interrupt service routine address to the UTVEC register
    la      t0, handler
    csrrs   zero, utvec, t0

    # Set the UEIE (User External Interrupt Enable) bit in UIE register
    li      t1, 0x100
    csrrs   zero, uie, t1       # uie - ueie bit (bit 8)
    # Set the UIE (User Interrupt Enable) bit in USTATUS register
    csrrsi  zero, ustatus, 1    # ustatus - enable uie (bit 0)

    # Enable the interrupt of keypad of Digital Lab Sim
    li      t1, IN_ADDRESS_HEXA_KEYBOARD
    li      t3, 0x80  # bit 7 = 1 to enable interrupt
    sb      t3, 0(t1)

    # ---------------------------------------------------------
    # No-end loop, main program, to demo the effective of interrupt
    # ---------------------------------------------------------
loop:
    nop
    # Delay 10ms
    li      a7, 32
    li      a0, 10
    ecall
    nop
    j       loop
end_main:

# ---------------------------------------------------------------
# Interrupt service routine
# ---------------------------------------------------------------
handler:
    # ebreak # Can pause the execution to observe registers
    # Saves the context
    addi    sp, sp, -8
    sw      a0, 0(sp)
    sw      a7, 4(sp)

    # Handles the interrupt
    # Shows message in Run I/O
    li      a7, 4
    la      a0, message
    ecall

    # Restores the context
    lw      a7, 4(sp)
```

```
    lw      a0, 0(sp)
    addi    sp, sp, 8

    # Back to the main procedure
    uret
```

Home Assignment 3 – INTERRUPT & STACK

The stack is used to save and restore the registers used in the interrupt service routine, avoiding affecting the operation of the main procedure.
The program below performs the following functions:
1. The main procedure sets up an interrupt from the keypad device of the Digital Lab Sim tool.
2. The main procedure prints a series of consecutive integers on the Run I/O screen.
3. Whenever the user presses one of the keys C, D, E or F, an interrupt is triggered, the interrupt service routine prints the key code on the Run I/O screen.
Read carefully and understand how the program works.

```
.eqv IN_ADDRESS_HEXA_KEYBOARD       0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD      0xFFFF0014
.data
    message: .asciz "Key scan code: "
# ---------------------------------------------------------------
# MAIN Procedure
# ---------------------------------------------------------------
.text
main:
    # Load the interrupt service routine address to the UTVEC register
    la      t0, handler
    csrrs   zero, utvec, t0

    # Set the UEIE (User External Interrupt Enable) bit in UIE register
    li      t1, 0x100
    csrrs   zero, uie, t1       # uie - ueie bit (bit 8)
    # Set the UIE (User Interrupt Enable) bit in USTATUS register
    csrrsi  zero, ustatus, 1    # ustatus - enable uie (bit 0)

    # Enable the interrupt of keypad of Digital Lab Sim
    li      t1, IN_ADDRESS_HEXA_KEYBOARD
    li      t3, 0x80  # bit 7 = 1 to enable interrupt
    sb      t3, 0(t1)

    # -----------------------------------------------------
    # Loop to print a sequence numbers
    # -----------------------------------------------------
    xor     s0, s0, s0      # count = s0 = 0
loop:
```

```
    addi    s0, s0, 1       # count = count + 1
prn_seq:
    addi    a7, zero, 1
    add     a0, s0, zero    # Print auto sequence number
    ecall
    addi    a7, zero, 11
    li      a0, '\n'        # Print EOL
    ecall
sleep:
    addi    a7, zero, 32
    li      a0, 300         # Sleep 300 ms
    ecall
    j       loop
end_main:

# ------------------------------------------------------------------
# Interrupt service routine
# ------------------------------------------------------------------
handler:
    # Saves the context
    addi    sp, sp, -16
    sw      a0, 0(sp)
    sw      a7, 4(sp)
    sw      t1, 8(sp)
    sw      t2, 12(sp)

    # Handles the interrupt
prn_msg:
    addi    a7, zero,  4
    la      a0, message
    ecall
get_key_code:
    li      t1,   IN_ADDRESS_HEXA_KEYBOARD
    li      t2,   0x88      # Check row 4 and re-enable bit 7
    sb      t2,   0(t1)     # Must reassign expected row
    li      t1,   OUT_ADDRESS_HEXA_KEYBOARD
    lb      a0,   0(t1)
prn_key_code:
    li      a7, 34
    ecall
    li      a7, 11
    li      a0, '\n'        # Print EOL
    ecall

    # Restores the context
    lw      t2, 12(sp)
    lw      t1, 8(sp)
```

```
    lw      a7, 4(sp)
    lw      a0, 0(sp)
    addi    sp, sp, 16

    # Back to the main procedure
    uret
```

Home Assignment 4 – Multiple Interrupts

In case multiple interrupts are enabled, when an interrupt occurs the CPU executes a common interrupt service routine. Therefore, within the routine, it is necessary to distinguish the interrupt source to perform the corresponding handling.
The **ucause** register provides information about the interrupt source. This register consists of two fields:

- **INTERRUPT** (31st bit): Takes the value 1 if the cause is an interrupt, the value 0 if the cause is an exception.
- **EXCCODE** (bits from 0 to 30): Indicates the cause of the interruption (interrupt source), described in the following table.

mcause fields		Cause
INTERRUPT	EXCCODE	
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	11	Machine external interrupt
1	12-15	Reserved for future standard use
1	≥ 16	Reserved for platform use

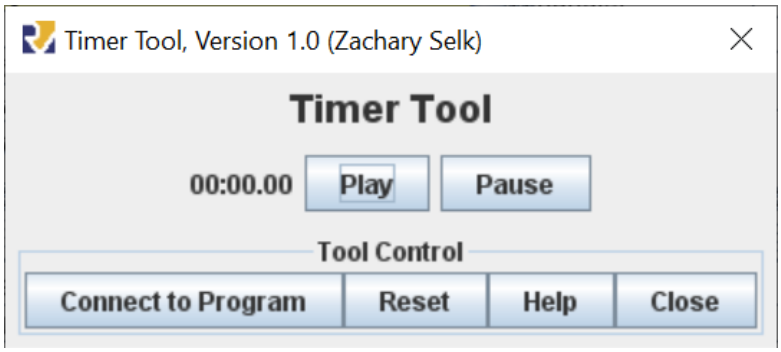
This program performs the following functions:

1. The main procedure simultaneously triggers 2 interrupts from the keypad (Digital Lab Sim) and from the timer (Timer Tool).
2. The main procedure runs an infinite loop.
3. When running the simulation, after each time interval or the user presses a button on the keypad, the program prints out the corresponding message on the Run I/O screen.

Note: Timer Tool user guide

- The word at address 0xFFFF0018 returns the current value of the timer (in ms).

- The word at address 0xFFFF0020 contains the comparison value (in ms). An interrupt occurs when the current value of the timer exceeds the comparison value.
- In the timer interrupt subroutine, the comparison value must be updated if the next interrupt is to occur.
- The Timer Tool (illustrated below) is used to control the timer.



Read this source code carefully and understand how the program works.

```
.eqv IN_ADDRESS_HEXA_KEYBOARD   0xFFFF0012
.eqv TIMER_NOW                  0xFFFF0018
.eqv TIMER_CMP                  0xFFFF0020
.eqv MASK_CAUSE_TIMER           4
.eqv MASK_CAUSE_KEYPAD          8

.data
    msg_keypad: .asciz "Someone has pressed a key!\n"
    msg_timer: .asciz "Time inteval!\n"

# ----------------------------------------------------------------
# MAIN Procedure
# ----------------------------------------------------------------
.text
main:
    la      t0, handler
    csrrs   zero, utvec, t0

    li      t1, 0x100
    csrrs   zero, uie, t1     # uie - ueie bit (bit 8) - external interrupt
    csrrsi  zero, uie, 0x10   # uie - utie bit (bit 4) - timer interrupt

    csrrsi  zero, ustatus, 1    # ustatus - enable uie - global interrupt

    # ---------------------------------------------------
    # Enable interrupts you expect
    # ---------------------------------------------------
```

```
    # Enable the interrupt of keypad of Digital Lab Sim
    li      t1, IN_ADDRESS_HEXA_KEYBOARD
    li      t2, 0x80  # bit 7 of = 1 to enable interrupt
    sb      t2, 0(t1)

    # Enable the timer interrupt
    li      t1, TIMER_CMP
    li      t2, 1000
    sw      t2, 0(t1)

    # ---------------------------------------------------------
    # No-end loop, main program, to demo the effective of interrupt
    # ---------------------------------------------------------
loop:
    nop
    li  a7, 32
    li  a0, 10
    ecall
    nop
    j   loop
end_main:

# ---------------------------------------------------------------
# Interrupt service routine
# ---------------------------------------------------------------
handler:
    # Saves the context
    addi    sp, sp, -16
    sw      a0, 0(sp)
    sw      a1, 4(sp)
    sw      a2, 8(sp)
    sw      a7, 12(sp)

    # Handles the interrupt
    csrr    a1, ucause
    li      a2, 0x7FFFFFFF
    and     a1, a1, a2      # Clear interrupt bit to get the value

    li      a2, MASK_CAUSE_TIMER
    beq     a1, a2, timer_isr
    li      a2, MASK_CAUSE_KEYPAD
    beq     a1, a2, keypad_isr
    j       end_process

timer_isr:
    li      a7, 4
    la      a0, msg_timer
```

```
    ecall

    # Set cmp to time + 1000
    li      a0, TIMER_NOW
    lw      a1, 0(a0)
    addi    a1, a1, 1000
    li      a0, TIMER_CMP
    sw      a1, 0(a0)

    j       end_process

keypad_isr:
    li      a7, 4
    la      a0, msg_keypad
    ecall
    j       end_process

end_process:

    # Restores the context
    lw      a7, 12(sp)
    lw      a2, 8(sp)
    lw      a1, 4(sp)
    lw      a0, 0(sp)
    addi    sp, sp, 16
    uret
```

Home Assignment 5 – Exception Handling

Exceptions are events generated by the CPU in response to exceptional conditions when executing instructions. Exceptions often trigger a handling mechanism to ensure that the exceptional conditions are handled before the CPU continues executing the program. Exception handling is used to protect the system from executing invalid instructions. Exception handling is similar with the interruption handling, including the following steps:
1. Save the program context.
2. Handle the exception condition.
3. Restore the context and continue program execution (Depending on the type of exception, the system will decide whether to continue executing the program or not).
The following example illustrates the implementation of the **try-catch-finally** structure in high-level programming languages to handle exceptions.

```
.data
    message: .asciz "Exception occurred.\n"
.text
main:
try:
    la      t0, catch
```

```
    csrrw   zero, utvec, t0 # Set utvec (5) to the handlers address
    csrrsi  zero, ustatus, 1  # Set interrupt enable bit in ustatus (0)

    lw zero, 0          # Trigger trap for Load access fault
finally:
    li a7, 10           # Exit the program
    ecall
catch:
    # Show message
    li  a7, 4
    la  a0, message
    ecall
    # Since uepc contains address of the error instruction
    # Need to load finally address to uepc
    la  t0, finally
    csrrw zero, uepc, t0
    uret
```

Assignment 1

Create a new project, type in, and build the program of Home Assignment 1. Run the program step by step to understand each line of the source code. Upgrade the source code so that it could detect all 16 key buttons, from 0 to F.

Assignment 2

Create a new project, type in, and build the program of Home Assignment 2. Run the program step by step to understand each line of the source code.

Assignment 3

Create a new project, type in, and build the program of Home Assignment 3. Run the program step by step to understand each line of the source code. Upgrade the source code so that it could detect all 16 key buttons, from 0 to F.

Assignment 4

Create a new project, type in, and build the program of Home Assignment 4. Run the program step by step to understand each line of the source code.

Assignment 5

Create a new project, type in, and build the program of Home Assignment 5. Run the program step by step to understand each line of the source code.

Assignment 6: Software interrupt

Software interrupts can be triggered by setting the **USIP** bit in the **uip** register (It needs to enable software interrupt by setting **USIE** bit in **uie** register in advance). Write a program

that raises software interrupt when an overflow occurs while adding two signed integers (Lab 4). ISR will display a message in the console and then terminate the program.

Conclusion

Answer the following questions before ending the lab:

- What is Polling?
- What is Interrupt?
- What is Interrupt Service Routine?
- What are the advantages of Polling?
- What are the advantages of Interrupt?
- Distinguish between Interrupt, Exception and Trap?