Project: Interrupts

Instructions: Follow lab instructions below and complete the following:

1. **Create a lab report and submit/upload via canvas as a .pdf . See Lab Report Specs document for formatting details.**
2. **Commit code to your personal branch as instructed**

Key

- Methods / Procedures: are enumerated.
- Questions: are italicized and generally ask you to share your observations and conclusions.
- Commit instructions: Code to commit to your branch are underlined.

Objectives: Simulate LC3 Interrupts for Keyboard Input.

Before we begin, it is important to note that the simulator will be using Verilog.

1. To install Verilog, see links to instructions at

directory, projects/tools/ Verilog/

Note this may take some time – allocate a few hours. Once Verilog is installed we can continue. This Project has you tracing execution of some code that handles interrupts on the LC3. There is a directory,

projects/LC3-OS/ keyboard_interrupts/

with the source material.

2. Read the Makefile in this directory. In the comments are instructions on usage and how to successfully complete the simulation. There are multiple steps. Please read each thoroughly before beginning.

3. The test program, testINT.asm, has a keyboard interrupt handler – it is very simple and doesn't even save and restore registers. That's ok for this simple simulation: This code counts on the registers not being saved in order to communicate between the code that was interrupted and the interrupt handler. *You are welcome to write your own code, instead of using mine.*

4. Follow the instructions in the Makefile. Simulate the execution of testINT.asm, and answer the questions below.

```
.ORIG x0200

   ;------- Initialize keyboard interrupt handling.

                     ;---Set up super's stack:
   ld   r6, stackPtr ;-- 0200:  r6 <== pointer to bottom of stack.

                     ;---Set up IVT:
   ld   r0, kbIVTloc ;-- 0201:  r0 <== addr of kb's vector.
   lea  r1, kbCode   ;-- 0202:  r1 <== kbCode's address.
   str  r1, r0, 0    ;-- 0203:  MEM[vec addr] <== kbCode's address.
   jsr EnableINTS    ;-- 0204:  enable interrupts, Priority <== 0.

                     ;---Enable KB INTs:
   ldi  r3, KBSR     ;-- 0205:  r3 <== current content of KBSR.
   ld   r4, ENmask   ;-- 0206:  r4 <== interrupt enable mask.
   not  r3, r3       ;-- 0207:  r3 <== r3 OR r4
   not  r4, r4       ;-- 0208:  ...
   and  r3, r3, r4   ;-- 0209:  ...
   not  r3, r3       ;-- 020a:  ...
   sti  r3, KBSR     ;-- 020b:  KBSR <== r3

                     ;---Loop waiting for INTs in kernel mode
   ld r3, Limit      ;-- 020c:  cnt <== Limit
 LOOP:               ;--        Loop
   add r3, r3, 0     ;-- 020d:     test(cnt)
   BRp LOOP          ;-- 020e:  Until(cnt == 0)
   BRz NEXT          ;-- 020f:  go to NEXT, try user mode
   BRnzp DONE        ;-- 0210:  quit

 KBcode:             ;---KB INT handler. Has side effects!
   add  r3, r3, -1   ;-- 0211:  Do something on INT: cnt--
   ldi  r0, KBDR     ;-- 0212:  read KBDR (resets kb_ctl, KBSR[15])
   rti               ;-- 0213:  return from INT
```

The first portion of the source code is shown at right. It should be obvious what it is up to. You should trace through its execution to watch that it works correctly. The interrupt handler is called "KBcode". The only sort of tricky part is the subroutine, EnableINTS, which globally enables interrupts (see, below).

After setting up the IVT and globally enabling interrupts and enabling keyboard interrupts, it loops,

waiting for an interrupt to occur. As you trace its execution, try to see exactly when an interrupt occurs, and check that what happens next is what you expected. (Note this vacuous loop is part of the simulation. In reality, the computer would be doing something of relevance rather that repeatedly doing nothing.) Note that KBcode decrements the counter the loop is testing.

Q.1 What is the IVT?

Q.2 What address is the address of the keyboard interrupt entry in the IVT? What is stored there (in general and in this specific example what exact value is stored there?)

Q.3 Identify what occurs when an interrupt is raised? How do you know an interrupt is raised? What state changes are observable in the simulator?

```
NEXT:           ;---------- go user mode
  lea  r0, B        ;-- 021c: get tmp user PC value
  str  r0, r6, -2   ;-- 021d: put PC value on stack
  ld   r0, uPSR     ;-- 021e: get user PSR value
  str  r0, r6, -1   ;-- 021f: put PSR value on stack
  add  r6, r6, -2   ;-- 0220: set SP to PC value
  rti               ;-- 0221: change to user mode (Saved_SSP <== R6)
B: ld  r6, uStack   ;-- 0222: set clobbered SP to user stack
                ;---------- in user mode, wait for INT
  ld r3, Limit      ;-- 0223:   cnt <== Limit
HERE:               ;--         Loop
  add r3, r3, 0     ;-- 0224:      test(cnt)
  BRp HERE          ;-- 0225:   Until(cnt==0)
DONE:
  ld   r4, STOP     ;-- 0226:   r4      <== Disable sys_clk signal.
  sti  r4, MCR      ;-- 0227:   MCR     <== r4: shuts down hardware.
```

Q.4 *What are the KSR and KDR?*

Q.5 *Speculate. In this simulation is the KSR set in hardware or software? Why?*

Q.6 When the keyboard interrupt handler has completed, where is the asci character stored?

Q.7 Does this vacuous execute in user mode or kernel mode? How do you know this?

After the vacuous loop is exited, it is time to try having an interrupt occur when in user mode, see NEXT.

Q.8 When goUSER routine is called, does the simulation successfully transition to User mode? If so, explain how this is accomplished. If not, explain why?

```
EnableINTS:
  SetPSR:  ld r0, PSRval    ;--022b:  get new PSR
           add r6,r6, -1    ;--022c:  push new PSR
           str r0, r6, 0    ;--022d:  push new PSR
           lea r0, donePSR  ;--022e:  get dummy PC
           add r6,r6, -1    ;--022f:  push new PC
           str r0, r6, 0    ;--0230:  push new PC
           rti              ;--0231:  pop dummy PC, pop new PSR
  donePSR: ret              ;--0232:  return
```

Note: I believe there is a hardware bug in kb_ctl. Did you notice? Something funny happens during simulation. BONUS: What is it?

By the way, this code eventually shuts down the LC3. If you do not want to step through the simulation, you can run the entire simulation in one shot,

vvp a.out < commands.txt > out.txt

Just put some lines with "cont" in them in commands.txt. You can read out.txt to see what happened.

The program's data areas are a little scattered. I've collected them here (at right). Note that every program word, instruction or data, is labeled with its runtime memory location.

```
;------- Data area.
kbIVTloc: .FILL x0180  ;-- 0214: keyboard's vector slot in IVT.
stackPtr: .FILL x3000  ;-- 0215: address of bottom of stack.
ENmask:   .FILL x4000  ;-- 0216: ENmask[14]=1 enable INTs
Limit:    .FILL x0001  ;-- 0217: Allow for 1 interrupts
KBSR:     .FILL xFE00  ;-- 0218: address of KBSR.
KBDR:     .FILL xFE02  ;-- 0219: address of KBDR.
MCR:      .FILL xFFFE  ;-- 021a: address of mach. ctl reg
STOP:     .FILL x8000  ;-- 021b: MCR[15]=1 disables sys_clk
uPSR:     .FILL x0000  ;-- 0228: user's PSR value
uStack:   .FILL xF000  ;-- 0229: user's SP value

PSRval: .FILL x0000    ;-- 022a: PSR.Priv=0, PSR.Priority=0, PSR.CC=0
```