Lab 6: We Got This Together

Brief

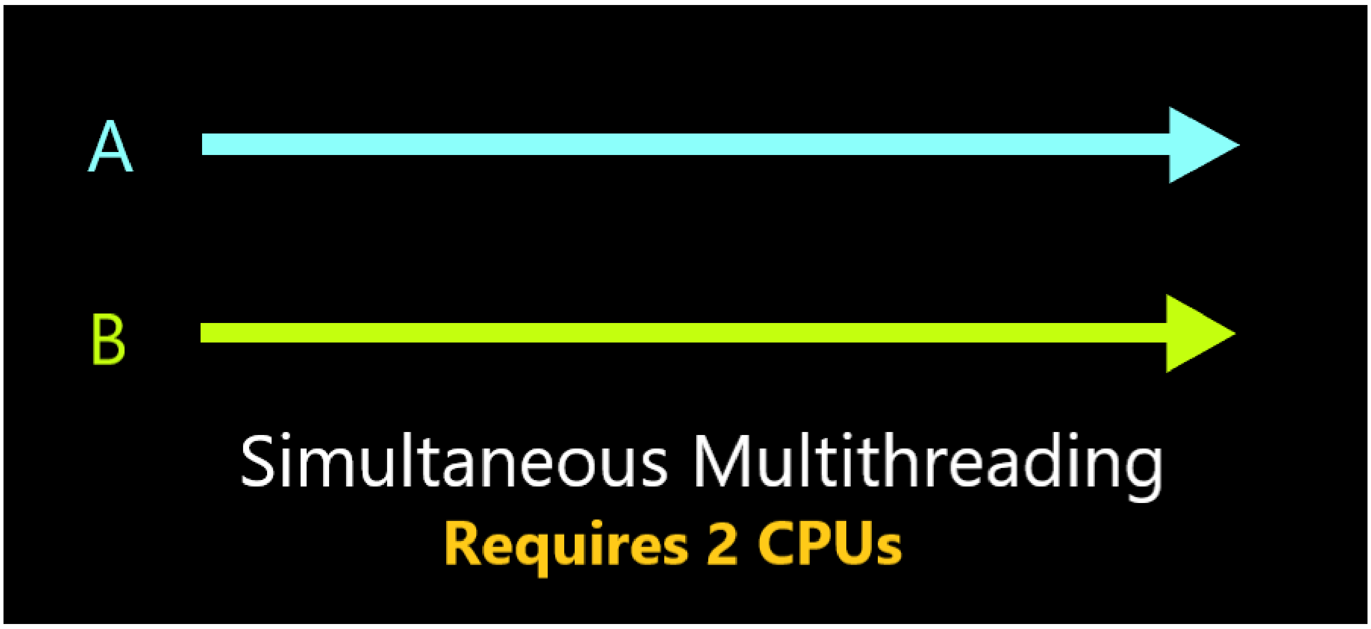
> A rising tide lifts all boats.

Welcome to the last assembly-based lab of the series! Easy or not, buggy or not, informative or not, you've went through the previous five labs. Congratulations I shall say, but it's still not time for relaxing. If your passion towards LC-3 is still there, or at least your LC3Tools still works, then it's time. The final lab is **by no means easy**, but is indeed interesting. You'll learn **how to run two programs in parallel** using LC-3. Let's cover the tasks, so you can have the most fun (or the least pain) finishing this lab.

Intro

LC-3 runs one program at a time. That is to say, if we have two programs to run, we must run program A first, wait for it to exit, then run program B. This is not a problem for singleton programs as we have interrupts to perform urgent operations in a higher priority. However, if we want to run two programs at the same time, it's not trivial to implement.

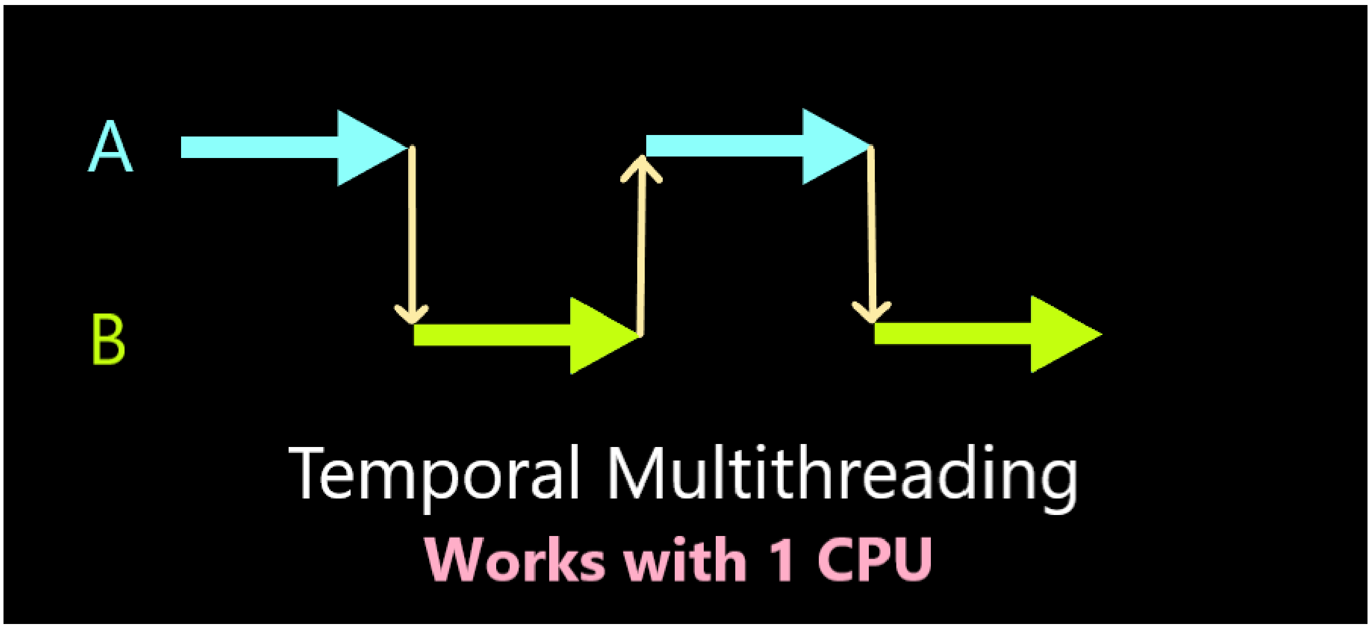
You might already heard of **threads**. Technically, one thread runs on one (logical) CPU core. If our LC-3 machine had more than one CPU, then it would be fairly fine to simply run two programs on separated threads. But it doesn't. Therefore, we cannot play with **simultaneous multithreading (SMT)** —— It's just impossible to perform two instructions at exactly the same time.



Is there a way we can still run two programs, using only one decoder, one controller, and one set of registers?

Maybe.

Although we cannot execute the instructions **simultaneously**, there is another technique called **temporal multithreading (TMT)**. In TMT, instructions **share** (rather than seize) the time of CPU. This is suitable for LC-3, as even though we have only one CPU, we can first execute some instructions from program A, then switch to program B for a while, then switch back to A... At any given time there is only **one** instruction running, but in general it **looks like** that the two programs are running alongside each other.



There is still another problem though. TMT is usually implemented based on clock signal. By creating an interrupt handler for clock interrupts, we can manage to change the states of two programs somehow. A program itself never knows it has been suspended.

But LC-3 doesn't have a clock, so instead of getting scheduled passively, the programs must work **cooperatively**. Program A should suggest "Hey, I've already done enough work and it's time to switch to program B!", and vice versa. Programs can utilize `TRAP` instruction to create an interrupt manually. Two programs work together to help each other —— much like our school, right? However, if one of the program stucks and falls into an endless loop, it cannot switch back to another program. Therefore, we also want a way of terminating them.

Tasks

Implement **temporal multithreading** for two LC-3 programs A and B.

- The user program A and B will be created and loaded for you at test time.

- When key `Q` is pressed, halt the machine (thus stopping both programs).
- Whenever a program requests task switching via `TRAP`, serve it. The trap vector for switching is defined as `x77`.

As context switching is quite cumbersome, we've implemented it for you. See section "Boilerplate" for details.

Examples

Suppose our user programs look like:

Program A:

```
LOOP ADD R0, R0, x1
ST R0, Somewhere
TRAP x77                ; Yield the time for B
BR LOOP
```

Program B:

```
LOOP LD R1, Somewhere
TRAP x77                ; Yield the time for A
BR LOOP
```

The system should run like:

```
    A        B
    |
R0->x0001
<TRAPPED>
    |------->|
         R1->x0001
         <TRAPPED>
    |<-------|
R0->x0002
<TRAPPED>
    |------->|
         R1->x0002
         <TRAPPED>
    |<-------|
    |
   ...
```

Requirements

- The two user programs locate at:
  - A: `x4090`
  - B: `x8090`

  (Yes, I made them deliberately)

  It's guaranteed that the code sections of A and B never overlaps.
- The user programs are arbitrary and is loaded at test time.
- For your information about the user program:
  - They never require any input.
  - They won't call `HALT`.
  -
  - They won't use `R6` or `R7`.
- Your code starts at `x800` and runs **in supervisor mode**.
  - Your code will be called by the bootloader automatically.
- You need to setup the machine to receive interrupts from the keyboard.
  - This will be discussed in section "How To".
- We'll provide code snippets and **you can copy them directly**. Apart from this, do not use any code that does not belong to you.

How To

Program Structure

Comparing to previous labs, Lab 6 is quite complex. Here is a descriptive overview of the program structure:

- **Entry**, starting at `x200`.
  - LC-3 sets PC to here at the beginning.

    - **We've implemented this for you.**
- **Setup program**, starting at `x800`.
    - Enables keyboard interrupt.
    - Sets handler for keyboard interrupt.
    - Sets handler for `TRAP x77`.
    - Call user program A at `x4090`.
- **Keyboard interrupt handler**, starting at `x1000`.
    - Check the pressed key.
    - If the key is `Q`, halt the machine.
- **Trap handler**, starting at `x1800`.
    - Switch to another program.
- **Program switch**, starting at `x2000`.
    - **We've implemented this for you.**
- **Program A, B**, starting at `x4090` and `x8090`.
    - **Will be provided when testing.**

Setup Keyboard Interrupt

Bit 14 of `KBSR` controls the enable of interrupts. By setting it to 1, the system will receive interrupts. You can use `OR` to achieve this (yes, you need to implement it using `AND` and `NOT`).

You might ask "Where is `KBSR`?". Well, since register `KBSR` is part of memory-mapped I/O, it's part of the memory, locates at `xFE00`. You can operate it just like regular memory units. e.g. use `STI` to assign a value to it directly.

After we've **enabled** the interrupt, we should also **register** the corresponding **interrupt service routine (ISR)**. Keyboard interrupt uses vector `x80`, which means whenever a key is pressed, the system queries interrupt table, trying to find a handler whose address is located at `x180` (`x100 + x80` as the interrupt vector table starts at `x100`). Therefore, we only need to **put the starting address of the ISR** at `x180`, the system will then call it automatically when a key is pressed.

Get the Keycode

If you look detailed into the book, you'll notice that `KBDR` stores the keycode striked. If you look even more detailed, you'll find that `KBSR` is automatically cleared when you read `KBDR`. As `KBDR` is also memory-mapped register, you can read its value using `LDI`. By the way, `KBDR` locates at `xFE02`.

Call Program Switch

We've already implemented program switch for you, but how to call it? Well, the switch locates at `x2000` and you can access it using code like this:

```
LD R7, SwitchProgramAddr
JMP R7
SwitchProgramAddr .FILL x2000
```

Is that all? Maybe, but there is still one thing we should be aware of. **The program switch saves the current context and loads the other context**. What does this mean? Well, the program switch will expect that **the current values of the registers are the same as they were in the user program** (except `R7` which has claimed that it won't be used). In other word:

Your keyboard ISR and trap handler should not change the value of any register besides `R7`. If you happened to have changed any, make sure to restore it before you call `JMP`.

This is trivial once you realize it, but can be tricky and hard to reveal if not.

Handle Traps

Registering a trap handler is more than similar comparing to setting up the keyboard ISR, except that the handler address of `TRAP x77` should be placed at `x77` (`x0` + `x77` as the trap vector table starts at `x0`).

Call User Program

As the last part of the setup program, you need to transfer the control to user program. You know that program A starts at `x4090`, so it's tempting to write:

```
LD R0, AddrA
JMP R0
AddrA .FILL x4090
```

But this will not work, at least not safe. Keep in mind that we're now in **supervisor mode**. If you jump directly to the user program, and it happens that program A contains something like `ST`

`R1, x180`, then the whole system will break.

The only instruction that leaves the supervisor mode is `RTI`. What it does is simple:

- Pops a value from the system stack, assign it to PC.
- Pops a value from the system stack, assign it to PSR.
- As PC has changed, the next instruction will be fetched from the new address in PC, leading to an implicit jump.
- As PSR has changed, the next instruction will be interpreted in the new mode in PSR, leading to an implicit mode change.

Knowing this, we only need to **prepare two values for PC and PSR, push them into the stack**, and call `RTI`. Nice!

Oh, in case you've missed anything, here are two snippets for pushing and popping values from the stack:

```
; Push R0 into the stack
ADD R6, R6, x-1
STR R0, R6, x0

; Pop a value from the stack and assign it to R0
LDR R0, R6, x0
ADD R6, R6, x1
```

Assemble Multiple Code Segments

LC-3 assembler is capable for processing multiple code segments at once, e.g.:

```
.ORIG x200
; ...
.END

.ORIG x1000
; ...
.END

.ORIG x3000
; ...
.END
```

When you assemble it using LC3Tools, it will place these segments at their correct location.

Also, to avoid a potential pitfall, please keep in mind that you cannot place anything after `.END` and before `.ORIG`:

```
.ORIG x200
; ...
.END

SOME_VALUE .FILL x1234 ; No!

.ORIG x3000
LD, R0, SOME_VALUE ; Won't work!
.END
```

Test and Run

Like other labs you've played with, you can assemble this lab just like you've done before.

If you're very interested in the result of TMT, you can try editing user program A and B, and see if they run concurrently.

Tips and Tricks

- To make the program switch work correctly, you **should not change any register** (except `R7`) in both keyboard ISR and trap handler. If you need to, make sure to use `ST` and `LD` to save and restore any registers used.
- LC3Tools may got stuck when running your program. If nothing is getting printed and you're quite sure that your code is correct, try restarting the software.

Boilerplate

It's highly recommended to use this boilerplate to complete this lab. Copy and paste it, then fill in the blanks marked with `; TODO`.

The user programs, A and B, have also been created for you. You may change it if you want, but if you use the provided examples, you shall see that messages from A and B both gets printed, and B prints roughly twice as fast as A.

Note that the speed that LC3Tools runs depends on your PC. If you find that the output is being laggy, try to reduce the values of `TIMER_LIMIT_A` and `TIMER_LIMIT_B`.

```
; Entry point
.ORIG x200

; Loads system stack
LD R6, SystemStack

; Calls your code
LD R0, SetupAddr
JMP R0

SystemStack .FILL x2FFF
SetupAddr .FILL x0800

.END

; Setup program
.ORIG x0800

; ==============================
; TODO
; - Enable keyboard interrupt
; - Setup ISR for keyboard
; - Setup ISR for trap x77
; - Call user program A
; ==============================

; Some values defined for your convenience
KBSR .FILL xFE00
KBSRMask .FILL x4000
KBVecAddr .FILL x0180
KBISRAddr .FILL x1000
TrapVecAddr .FILL x0077
TrapISRAddr .FILL x1800

.END

; Keyboard ISR
.ORIG x1000

; ==============================
; TODO
; - Get the keycode
; - If Q is pressed, halt the machine
; ==============================

KBDR .FILL xFE02
.END

; Trap x77 ISR
.ORIG x1800
```

```
; ==============================
; TODO
; - Call switch program
; ==============================

SwitchProgramAddr .FILL x2000
.END

; Switch Program
.ORIG x2000
LD R7, CurrentProgram
BRp SwitchB2A
; A to B
LEA R7, PC_PSR_A
ST R7, PC_PSR_Save
LEA R7, PC_PSR_B
ST R7, PC_PSR_Load

AND R7, R7, x0
ADD R7, R7, x1
ST R7, CurrentProgram

LEA R7, RegA

BR SwitchEnd
SwitchB2A
; B to A
LEA R7, PC_PSR_B
ST R7, PC_PSR_Save
LEA R7, PC_PSR_A
ST R7, PC_PSR_Load

AND R7, R7, x0
ST R7, CurrentProgram

LEA R7, RegB

SwitchEnd

STR R0, R7, x0
STR R1, R7, x1
STR R2, R7, x2
STR R3, R7, x3
STR R4, R7, x4
STR R5, R7, x5

LD R2, PC_PSR_Save

; Pop PC
LDR R7, R6, x0
```

```
ADD R6, R6, x1
STR R7, R2, x0

; Pop PSR
LDR R7, R6, x0
ADD R6, R6, x1
STR R7, R2, x1

LD R2, PC_PSR_Load

; Push PSR
LDR R7, R2, x1
ADD R6, R6, x-1
STR R7, R6, x0

; Push PC
LDR R7, R2, x0
ADD R6, R6, x-1
STR R7, R6, x0

LD R7, CurrentProgram
BRz LoadB2A
LEA R7, RegB
BR LoadEnd
LoadB2A
LEA R7, RegA
LoadEnd

LDR R0, R7, x0
LDR R1, R7, x1
LDR R2, R7, x2
LDR R3, R7, x3
LDR R4, R7, x4
LDR R5, R7, x5

RTI

PC_PSR_Save .BLKW 1
PC_PSR_Load .BLKW 1

SaveR0 .BLKW 1
CurrentProgram .FILL x0

RegA .BLKW 8
RegB .BLKW 8

PC_PSR_A
.FILL x4090
.FILL x8002

PC_PSR_B
```

```
.FILL x8090
.FILL x8002
.END

; Program A
.ORIG x4090
LOOP_A
LEA R0, MSG_A
PUTS

LD R1, TIMER_LIMIT_A
LOOP_IA
ADD R1, R1, x-1
TRAP x77
BRp LOOP_IA

BR LOOP_A

TIMER_LIMIT_A .FILL x200
MSG_A .STRINGZ "Program A reporting.\n"
.END

; Program B
.ORIG x8090
LOOP_B
LEA R0, MSG_B
PUTS

LD R1, TIMER_LIMIT_B
LOOP_IB
ADD R1, R1, x-1
TRAP x77
BRp LOOP_IB

BR LOOP_B

TIMER_LIMIT_B .FILL x100
MSG_B .STRINGZ "Program B reporting.\n"
.END
```

Questions

Please answer these questions in your lab report:

1. Talk about how you understand that we must not change any register in the ISR.
    - No need to talk about the special case `R7`, it's excluded for your convenience.

2. Using the program above, you may notice that if you hit some key (other than `Q`) quickly, LC-3 will crash. Suggest some possible cause of this.

3. Implementing temporal multithreading is cumbersome in terms of scheduling, while for simultaneous multithreading things will be tricky when managing data synchronization. You may have also noticed that the program switch is very long (~100 lines). Paying such a high price, why computers of nowadays still implement multithreading?

   - It's not about multiple cores —— you may have thousands of threads running in your system which is more than the number of cores of any commercial CPU product. The system switches between these threads just like we did in this lab. Ask yourself, why is it needed?

Report

Create your report as you've done before, but make sure to include answers to the questions this time.