Subroutines & OS TRAPs

Introduction to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

How are you feeling about LC4 I/O?

Logistics

- HW05 Control Signals: **This Friday** 10/21 @ 11:59 pm
  - Should have everything you need
  - Practice in Recitations this week
  - Normal programming assignment ☺

- Midterm Exam: Wednesday Next Week "in lecture"
  - Details released on the course website

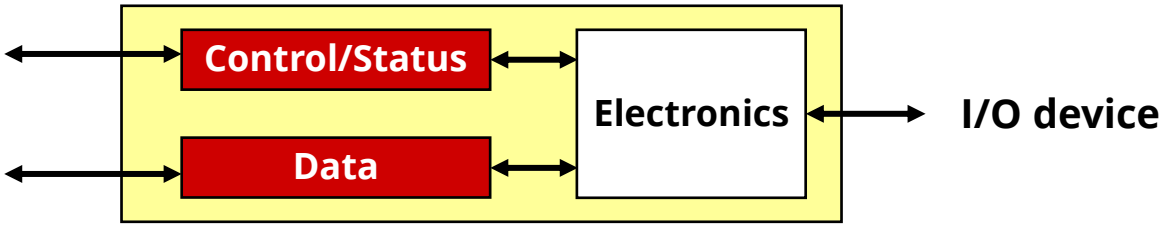
Lecture Outline

- **I/O Devices in LC4 Wrap-up**
- Calling “functions” in LC4
- Traps & The OS

I/O Controller to CPU Interface

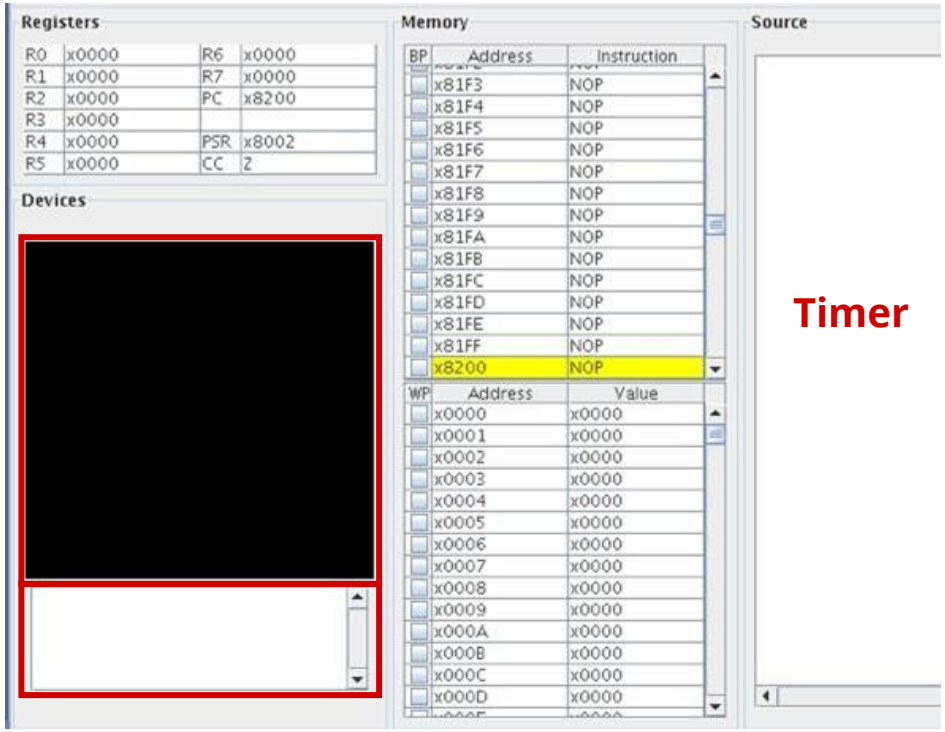
- I/O controller interface abstracts I/O device as "device registers"
  - Control/Status: may be one register or two
    - Control: lets us toggle options on the device (we won't focus on this)
    - Status: lets us know if we are data is ready to be read/written
  - Data: may be more than one register
    - The data we are reading/writing
- Example: CPU reading data from input device
  - CPU checks status register if input is available
  - Reads input the data register

Similar steps for writing.
More details later!



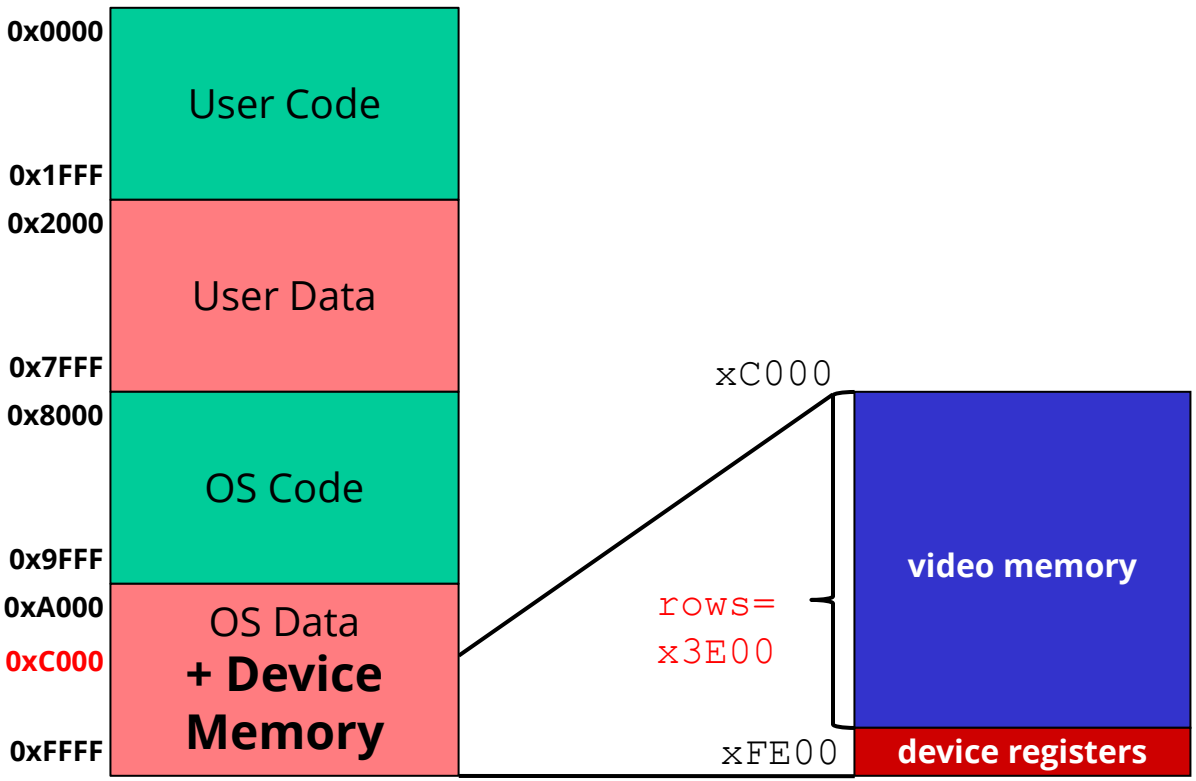
LC4 I/O Devices

- LC4 has 4 I/O devices
  - Keyboard (input)
  - ASCII console (output)
  - 128x124 16-bit RGB pixel display (output)
  - Timer (not really an I/O device but looks like one to software)



Demo: Breakout/Brick-breaker

I/O "Memory"



`0x3E00` converts to =15,872

LC4's video display has:
128 cols x124 rows
=15,872 pixels

One memory location per pixel on video display

Example: Print character to Screen

```
; reads 1 character from the keyboard, prints it to ASCII display

OS_KBSR_ADDR .UCONST xFE00
OS_KBDR_ADDR .UCONST xFE02
OS_ADSR_ADDR .UCONST xFE04
OS_ADDR_ADDR .UCONST xFE06

.CODE
GETC
   LC R0, OS_KBSR_ADDR ;; loop while KBSR[15]==0
   LDR R0, R0, #0
   BRzp GETC
   LC R0, OS_KBDR_ADDR
   LDR R0, R0, #0      ;; read data from keyboard

PUTC
   LC R1, OS_ADSR_ADDR ;; loop while ADSR[15]==0
   LDR R1, R1, #0
   BRzp PUTC
   LC R1, OS_ADDR_ADDR
   STR R0, R1, #0      ;; write R0 to ASCII display
```

Aliases for keyboard status & data regs

Aliases for ASCII display status & data regs

Get character from keyboard

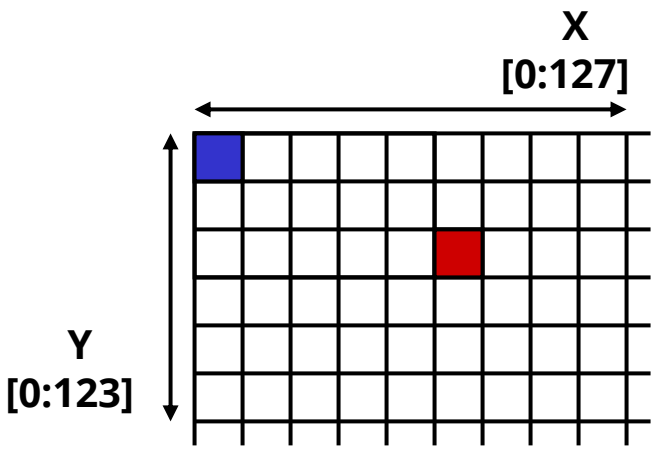
Print character to ASCII display

Addressing a Pixel

- Need to calculate the address that corresponds to a pixel

```
.ADDR xC000
OS_VIDEO_MEM       .BLKW x3E00   ; why 3E00?
OS_VIDEO_NUM_COLS  .UCONST #128
OS_VIDEO_NUM_ROWS  .UCONST #124
```

- Logically display is 2D, but 1D in memory
  - Row-major order (vmem[y][x])
    vmem[y][x] – pixel on row y, col x
  - Pixel at vmem[2][5] stored at
    xC000 + (2 * 128) + 5
  - In general vmem[y][x] stored at
    xC000 + (y * 128) + x
  - Note indexing from upper
    left corner of the display (0, 0)



Demo: Drawing a Horizontal Line

- `draw_horizontal_line.asm` on course website

Lecture Outline

- I/O Devices in LC4 Wrap-up
- **Calling "functions" in LC4**
- Traps & The OS

"Functions" in LC4

- To avoid repeating code, we group code together in one cohesive and invocable (e.g. callable) unit.
  - Typically this is in the form of a function.

- In LC4, we do this with **subroutines**
  - Subroutines don't necessarily follow the same ideas of variable scope, parameters, return values, etc.
  - In LC4, a subroutine is just a callable sequence of instructions.
  - We use JSR, JSRR and RET instructions for handling subroutines

JSR, JSRR and RET

- **`JSR IMM11`**
  - Action: `R7 = PC + 1`,
    `PC = (PC & 0x8000) | (IMM11 << 4)`
  - "Jump Subroutine"
  - Stores `PC + 1` in `R7` before jumping so that after the subroutine, we can return to right after `JSR`
- **`JSRR Rs`**
  - Action: `R7 = PC + 1, PC = Rs`
  - "Jump Subroutine Register"
- **`RET`**
  - **Ret**urn from a subroutine
  - Is a Pseudo Instruction
  - Actual implementation: `JMPR R7`

Creating a Subroutine:

- Consider the multiply program from 3 lectures ago:
- How do we make this a subroutine?
  - Add a RET pseudo-instruction wherever we are "done" with the subroutine
  - Add the .FALIGN directive before the first label/instruction
    - .FALIGN makes sure the code starts at an address that is a multiple of 16.
    - This is needed since JSR stores a IMM11 that is then shifted to the left by 4
    - (x << 4) == x * 16

```
;; Multiplication program
;; C = A*B
;; R0 = A, R1 = B, R2 = C
        .CODE
        .FALIGN
MULT
        CONST R2, #0
LOOP
        CMPI R1, #0
        BRnz END
        ADD R2, R2, R0

        ADD R1, R1, #-1
        BRnzp LOOP
END
        RET
```

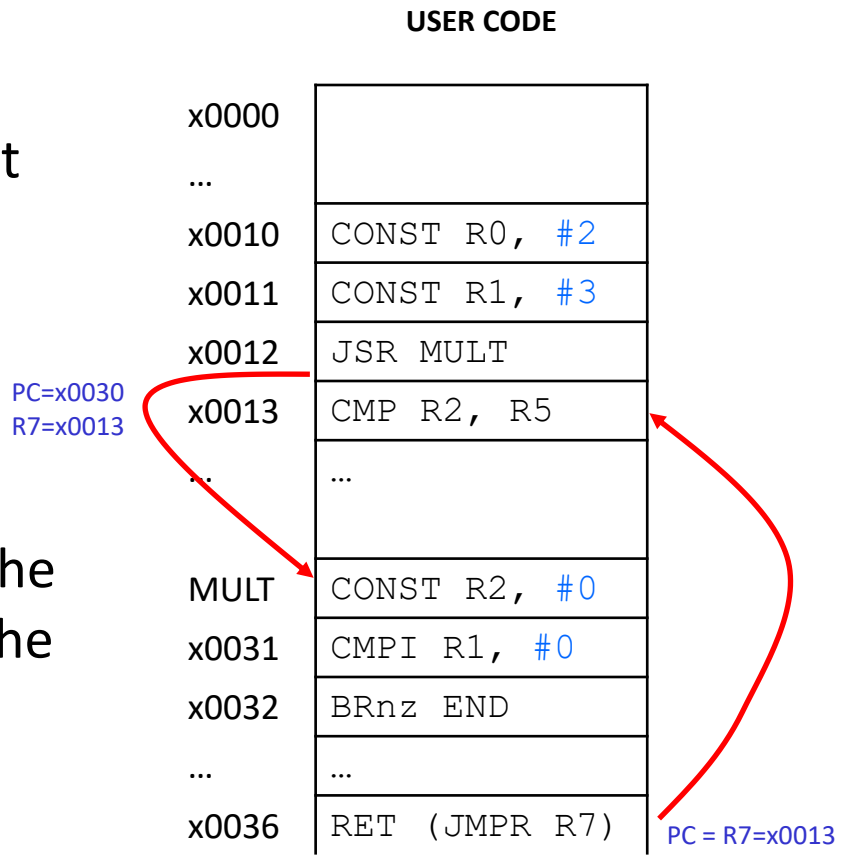
Calling a Subroutine:

- If we wanted to call a subroutine from other LC4 Code

```
.CODE
.ADDR  0x0000
CONST R0, #5  ; Initialize input "parameters"
CONST R1, #6

JSR MULT      ; call the subroutine

; resume execution here after MULT returns
```

Subroutine Walkthrough

- When a JSR is executed:
  - Stores PC + 1 in R7
  - PC jumps to the address of the start of the subroutine (which must be a multiple of 16).
- During Subroutine:
  - R0-R7 are possibly modified
  - R7 should have the same value at the end of the subroutine. It contains the address needed to return to Caller
- After Subroutine is complete:
  - Returns using RET (which is JMPR R7)
  - R7 should contain the return address



Subroutine Data Passing

- "Parameters"
  - Similar to HW04, we can designate some registers to contain "inputs" that are set by the caller. These values can be:
    - Some 16-bit value
    - Address to a memory location containing values (e.g. strings or arrays)
- "Return Values"
  - Subroutines will also designate a register to store their "result" in, assuming that there is a result to return

- NOTE: the same registers R0-R7 are used inside and outside a subroutine.
  - We can't always be sure that a certain register will not be changed

Backing Up the Register File

- The register file will be used inside a subroutine
  - It will likely overwrite everything in the REGFILE
  - BEFORE you call a subroutine, save relevant content of REGFILE
  - LDR and STR's "OFFSET" comes in handy here:

```
TEMPS .UCONST x4200     ; address of temporary storage
LC R7, TEMPS            ; load address into R7
STR R0, R7, #0          ; store R0 in TEMPS[0]
STR R1, R7, #1          ; store R1 in TEMPS[1]
STR R2, R7, #2          ; store R2 in TEMPS[2]
…
STR R6, R7, #6          ; store R6 in TEMPS[6]
JSR MULT                ; call the subroutine
LC R7, TEMPS            ; load address into R7
LDR R0, R7, #0          ; restore R0 from TEMPS[0]
LDR R1, R7, #1          ; restore R1 from TEMPS[1]
LDR R2, R7, #2          ; restore R2 from TEMPS[2]
```

Save content of REGFILE before you call subroutine

Restore content of REGFILE AFTER you return

I/O Subroutines?

```
; subroutine to read 1 character
; from the keyboard, return it in R0

OS_KBSR_ADDR .UCONST xFE00
OS_KBDR_ADDR .UCONST xFE02



.CODE
.FALIGN
SUB_GETC
    LC R0, OS_KBSR_ADDR ; load status register addr
    LDR R0, R0, #0
    BRzp GETC

    LC R0, OS_KBDR_ADDR ; load Data register addr
    LDR R0, R0, #0

    RET
```

- How can we make I/O easier?
  - Can we make subroutines to handle I/O?

Lecture Outline

- I/O Devices in LC4 Wrap-up
- Calling “functions” in LC4
- **Traps & The OS**

Operating Systems

- An operating system is software the directly interacts with hardware. The OS is trusted to do this for a few reasons:
  - To prevent users from breaking things
  - To abstract away messy details about hardware devices into a standardized and more portable/convenient interface
    - Think of how there are many types of keyboards, computer mice, network cards, hard drive types etc. OS abstracts away these details
    - Users typically don't want to handle the status and data registers directly. Users can call an "OS function" to do things for them.
  - Manages (allocates, schedules, protects) hardware resources
    - Modern computers will have more than one program running, how are resources (files, screen display, etc) shared across these programs?

LC4 Memory Map

Recall the full LC-4 Memory Map

- We have 2 Program Memories
- And 2 Data Memories

User Region

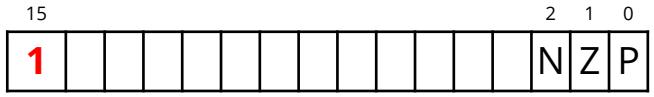
- Programs run by users (e.g.: factorial program)
- Processes run in user mode have **PSR[15]=0**
- ***NOT allowed*** to access OS locations in memory

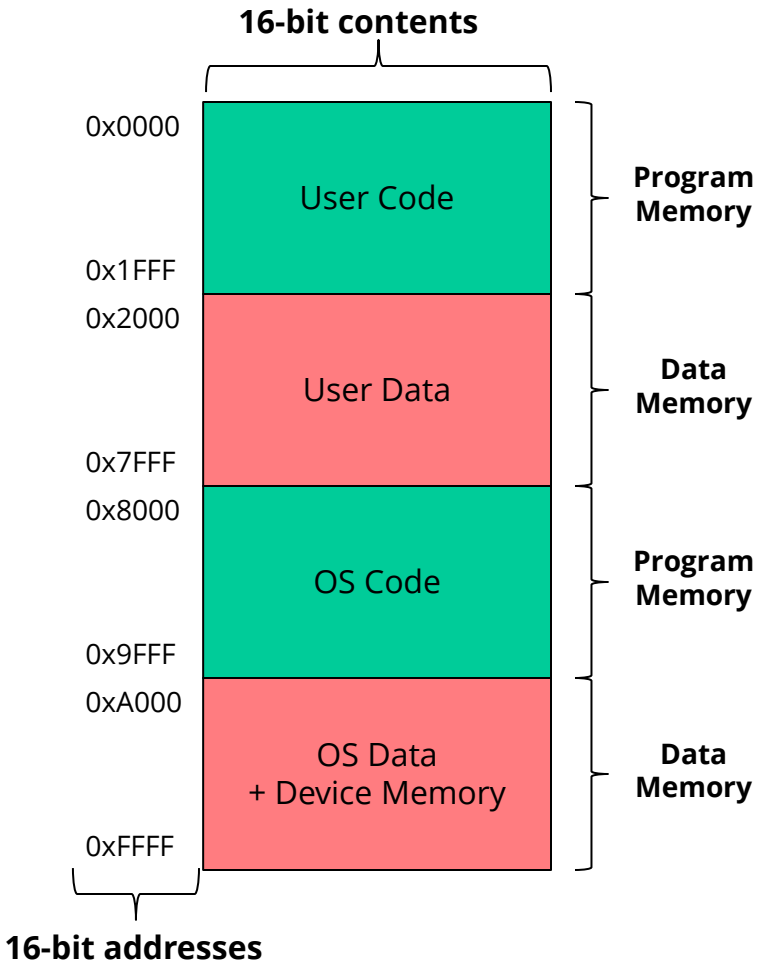
Operating System Region

- Programs run by OS (e.g.: I/O device programs)
- Processes run in OS mode have **PSR[15]=1**
- ***Allowed*** to access OS & User locations in memory

Processor Status Register

- Contains the privilege bit
- Contains the three NZP bits





GETC in User Issues

```
; subroutine to read 1 character
; from the keyboard, return it in R0

OS_KBSR_ADDR .UCONST xFE00
OS_KBDR_ADDR .UCONST xFE02



.CODE
.ADDR x0000
GETC
   LC R0, OS_KBSR_ADDR ; load status register addr
   LDR R0, R0, #0
   BRzp GETC

   LC R0, OS_KBDR_ADDR ; load Data register addr
   LDR R0, R0, #0
```

- There is a slight problem with this code
  - Since it will be loaded into program memory: **x0000**
    - *the LDR statements will fail!*
  - Programs running in USER program memory:
    - *have PSR[15]=0*
    - *they cannot access OS data memory (where Device registers are)*

GETC in OS

```
; subroutine to read
; 1 character from the
; keyboard, return it in R0

OS_KBSR_ADDR .UCONST xFE00
OS_KBDR_ADDR .UCONST xFE02


.OS
.CODE
.ADDR x8000
SUB_GETC
    LC R0, OS_KBSR_ADDR
    LDR R0, R0, #0

    BRzp GETC


    LC R0, OS_KBDR_ADDR
    LDR R0, R0, #0

    RET
```

- These 3 red bolded directives:
  - .OS .CODE .ADDR x8000
  - instruct the assembler, to tell the loader, to load this program into OS program memory
- When the LC4 executes this code, PSR[15] must be 1
  - *since the PSR[15]=1,*
  - *this program will be allowed to LDR from OS data memory*
- We have one small problem…
  - *what if we turn this into a subroutine*
  - *how can we call this subroutine from user space?*

Calling GETC in User Memory

- Currently, we can't easily run GETC
  - When a program is running in User Program Memory, PSR[15] = 0 We can't LDR/STR to device memory
  - If we put GETC subroutine in OS program memory, then PSR[15] must already be 1 to execute it
- How do we call the OS code from a USER program? (PSR[15]=0)…
  - JSR and JMP won't allow it!
  - Neither change the privilege of the program
  - LC4 will kill any program with PSR[15]=0 that attempts to jump into OS memory.
- Answer: TRAP instruction

TRAP vs JSR

Mnemonic	Semantics	Encoding
TRAP UIMM8	R7 = PC+1, PC = (x8000 | UIMM8), PSR[15] = 1	1111----UUUUUUUU
JSR IMM11	R7 = PC+1, PC = (PC&x8000) | (IMM11<<4)	01101IIIIIIIIIII

- The TRAP instruction is very similar to a JSR:
  - It saves PC+1 into R7
  - It updates the PC to an offset you specify
  - But it also elevates the privilege level of the CPU from 0 to 1
- The purpose of the TRAP instruction:
  - Allow a program running in USER Program Memory, to call a subroutine installed in OS Program Memory
- Subroutines in OS code are called TRAPS

RTI vs RET

Mnemonic	Semantics	Encoding
RTI	PC = R7, PSR[15] = 0	1000----------
RET	JMPR R7, *which simply sets: PC = R7*	11000--111------

- The RTI instruction is very similar to a RET:
  - It restores the PC back to the value saved in R7 (just like RET)
  - BUT, it also lowers the privilege level of the CPU from 1 to 0
- The purpose of the RTI instruction:
  - Allow a subroutine running in the OS program memory to return to a caller in the USER program memory

Installing GETC into the OS

```
; User Program Memory
.CODE
.ADDR x0000

; doing some fun stuff, like computing factorials!

; now, let's get a character from the keyboard!

TRAP x00        ; saves R7=PC+1, sets PC = x8000 | x00,
                ; and PSR[15]=1

; upon return, do something with R0

; OS Program Memory
.OS
.CODE
.ADDR x8000
SUB_GETC
   LC R0, OS_KBSR_ADDR
   LDR R0, R0, #0
   BRzp GETC

   LC R0, OS_KBDR_ADDR
   LDR R0, R0, #0   ; loads char from keyboard into R0
   RTI              ; sets PC = R7 and restores PSR[15]=0
```

The Limits of the TRAP Instruction

Mnemonic	Semantics	Encoding
TRAP UIMM8	R7 = PC+1, PC = (x8000 | UIMM8), PSR[15] = 1	1111----UUUUUUUU
JSR IMM11	R7 = PC+1, PC = (PC&x8000) | (IMM11<<4)	01101IIIIIIIIIII

- The TRAP instruction is limited.
  - Can't jump to anywhere in OS program memory, only the first 256 memory locations
  - We could expand the immediate to be more than 8 bits, why this limitation?
    - To control what portion of OS memory the USER can jump to
  - How it limits the user:
    - In the semantics: PC = (x8000 | UIMM8)
    - What is the largest 8-bit unsigned number you can make? xFF = 255
    - e.g.: PC = x8000 | xFF = x80FF

Installing GETC into the OS Properly

```
; User Program Memory
.CODE
.ADDR x0000

; doing some fun stuff, like computing factorials!

; now, let's get a character from the keyboard!

TRAP x00


; OS Program Memory
.OS
.CODE
.ADDR x8000
SUB_GETC
    LC_R0, OS_KBSR_ADDR
    LDR R0, R0, #0
    BRzp GETC

    LC_R0, OS_KBDR_ADDR
    LDR R0, R0, #0
    RTI
```

We shouldn't install our "TRAPS" starting at x8000

Why not?
- For one, user's might jump into the middle of our trap! Imagine: TRAP x01? We'd jump right into LDR R0,…

Another reason?
- Since traps take up multiple locations that can be jumped to, longer traps restrict how many traps we can have in the OS

Controlling User Access to the OS

- Since TRAP can only jump to the first 256 locations in OS program memory…
  - Make those locations all JMP to the beginning of a TRAP routine
  - Allows us to have complete control over how users enter the OS. Users can't JUMP into the middle of an OS TRAP routine
  - Allows us to put OS TRAPs deeper into OS Memory

```
.OS
.CODE
.ADDR x8000
    JMP TRAP_GETC              ; x00
    JMP TRAP_PUTC              ; x01
    JMP TRAP_DRAW_H_LINE       ; x02
    …
    JMP BAD_TRAP               ; xFF
```

The first 256 lines of OS Program Memory called the: **TRAP VECTOR TABLE**

We publish this list to the user
user can call the TRAPS by number:
e.g.: TRAP x01, will call TRAP: PUTC

the table listing helps them map # to TRAP

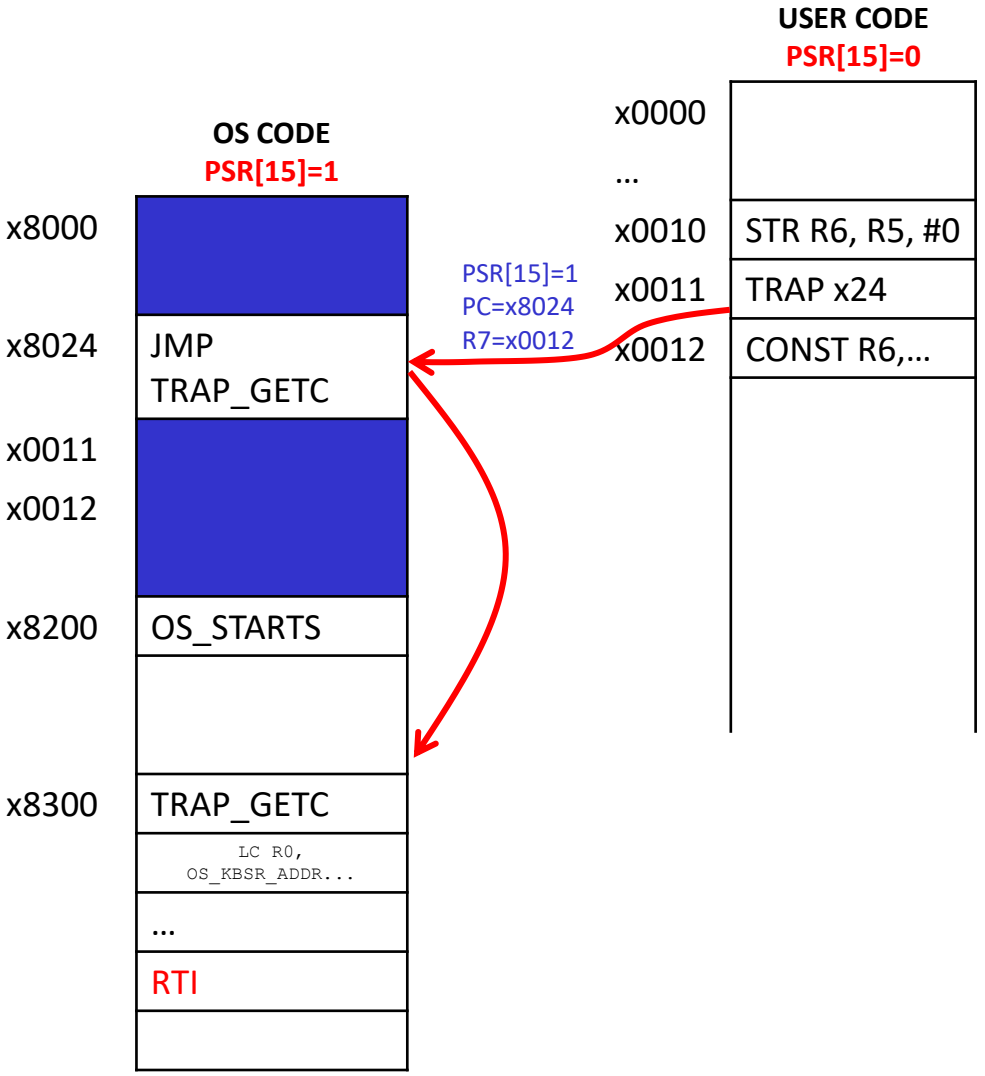
Installing TRAPs into the OS Properly

```
; OS Program Memory
.OS
.CODE
.ADDR x8300
TRAP_GETC               ;; this is TRAP x00
   LC R0, OS_KBSR_ADDR
   LDR R0, R0, #0
   BRzp GETC

   LC R0, OS_KBDR_ADDR
   LDR R0, R0, #0
   RTI

TRAP_PUTC               ;; this is TRAP x01
   LC R1, OS_ADSR_ADDR
   LDR R1, R1, #0
   BRzp TRAP_PUTC
   LC R1, OS_ADDR_ADDR
   STR R0, R1, #0
   RTI
```

Start at a memory location in OS program memory, but AFTER the TRAP Vector Table

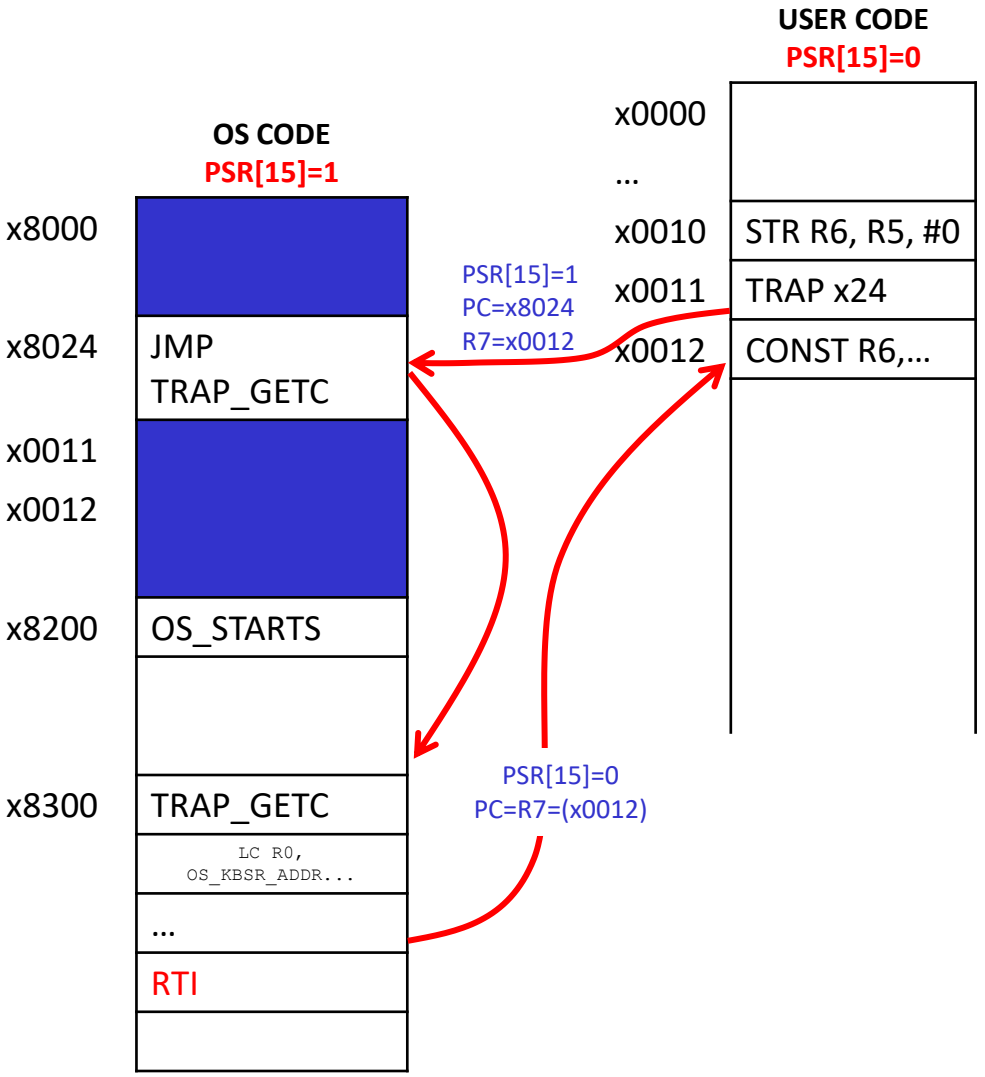
Trap Execution Walkthrough

- When a TRAP is called:
  - CPU sets PSR[15]=1,
  - stores PC+1 in R7
  - and Jumps to entry in the TRAP Table
  - This address is a JMP instruction which redirects to the TRAP routine



Trap Execution Walkthrough

- After the TRAP routine is complete:
  - it returns by using RTI,
  - which sets the PC to R7
  - which should contain the return address
  - and sets PSR[15] = 0



TRAP vs SUBROUTINE

- TRAPs behave very similar to subroutines
  - Data is passed in the same way
  - Registers may still be overwritten by a TRAP or Subroutine
  - TRAPs can access user data to read string/array inputs
  - R7 contains the return address to go back to the caller

- Key Differences:
  - Different instructions to enter/leave TRAPs and Subroutines
  - TRAPs exist in the OS and require OS privilege
  - Can’t call a TRAP from within another TRAP

OS in the Real World:

- What we just created highlights the role of the OS
  - Protecting & Abstracting away details of hardware
  - Creating a system of handling I/O calls
- Real OSs handle a lot more than I/O & System Calls
  - Sharing resources (CPU, memory, files) across multiple programs
  - Interrupts for handing I/O instead of "polling" (manually checking if I/O devices are ready)
  - Still follow similar practices with TRAP Vector Table
  - (Take 3800 or 5480 for more!)
- The OS in LC4 pretty much only handles I/O
  - There is only one program running in LC4 at a time, so these other features don't make sense to implement.