Input/Output & Subroutines

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 familiar are you with the idea of pixels, RGB, and how those relate to video displays?

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Logistics

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

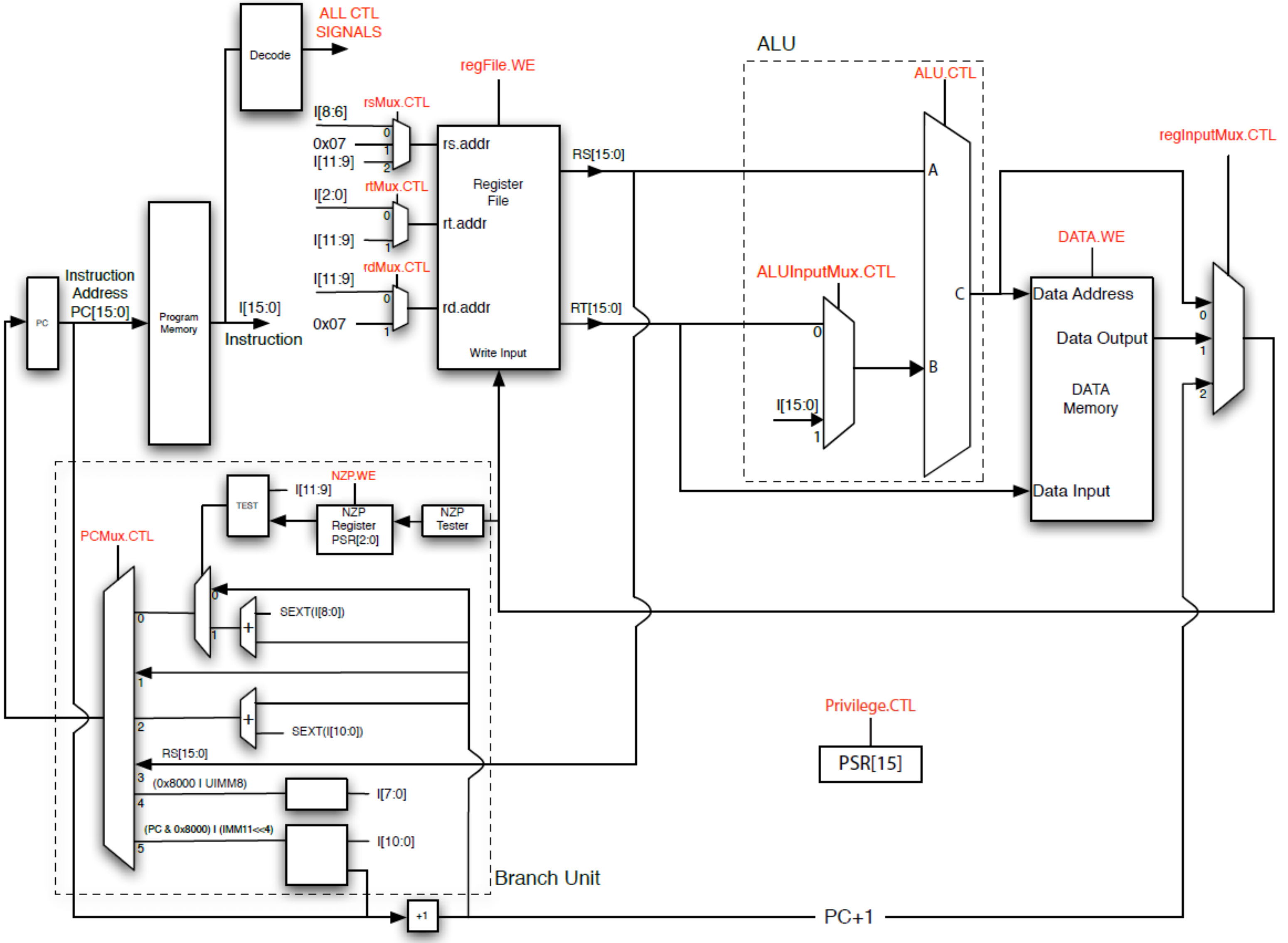
- Midterm Exam: Wednesday Next Week “in lecture”
  - More details to be released soon

Lecture Outline

- **I/O Devices in LC4 Overview**
- Interacting with I/O in LC4 Assembly
  - Memory Mapped I/O
  - Keyboard & ASCII Display
  - Timer
  - Video Display
- Subroutines in LC4

Last Couple Lectures:

Single Cycle Implementation of the LC4 ISA



LC4 is Little

- "LC4" -> Little Computer 4

- What is LC4 missing when you think of a "typical" modern computer?
  - Graphics
  - Keyboard & Mouse input
  - Files
  - Printing
  - Multiple Programs running at once
  - …

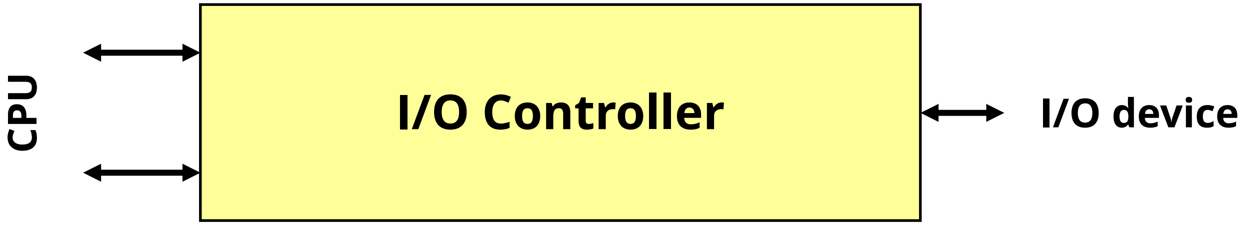
I/O

- Reading/writing anything “beyond” memory is called I/O
  - We call the locations we read/write to I/O devices

- I/O devices include:
  - Keyboard
  - Mouse
  - Files
  - Graphics Displays
  - Networks
  - Etc.

I/O Devices & Controllers

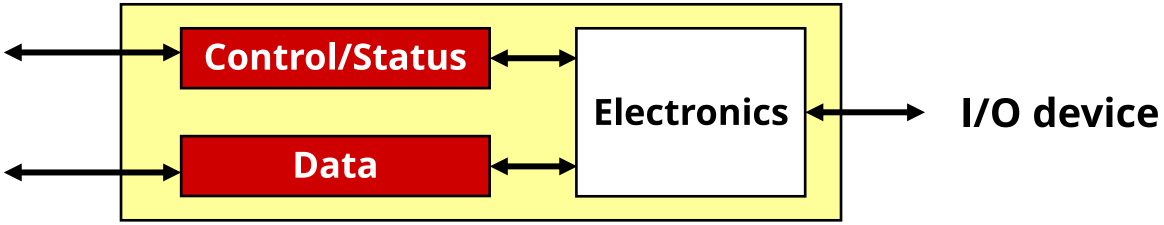
- Most I/O devices are not purely digital, they have their own hardware
  - Electro-mechanical: e.g. keyboard, mouse, disk, motor
  - Analog/digital: e.g. touchscreen, network interface, monitor, speaker, mic
- … all have digital interfaces presented by an I/O Controller
  - I/O Device (analog/digital mix) talks to controller
  - CPU (digital) talks to controller
  - Controller acts as a translator: digital (CPU) <-> analog (device)



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

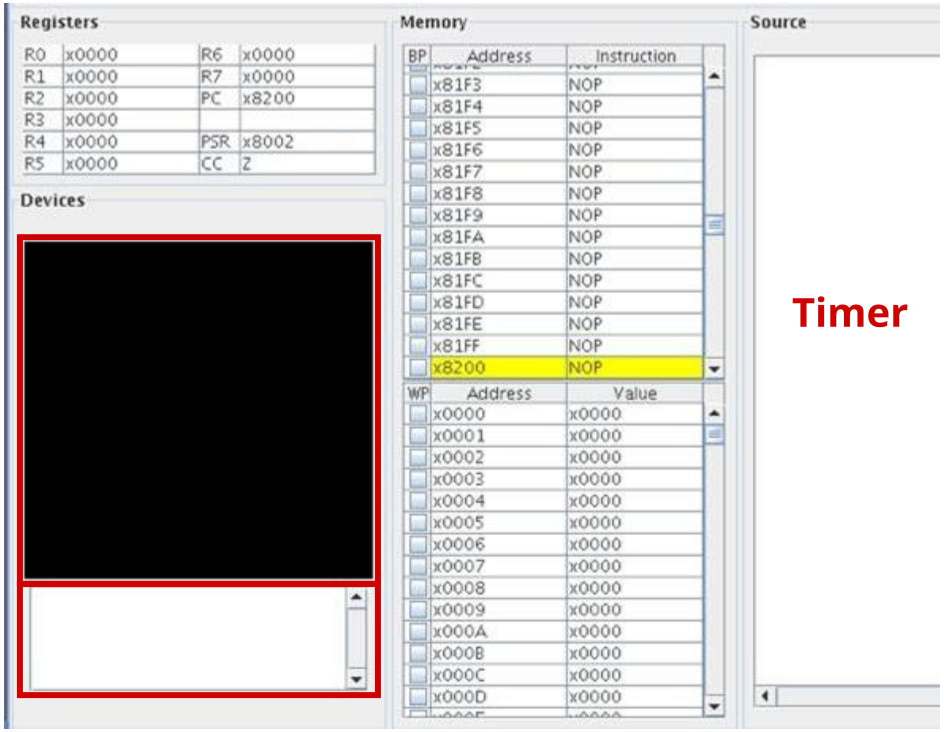
Similar steps for writing.
More details later!



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)

Video display



Registers

R0	x0000	R6	x0000
R1	x0000	R7	x0000
R2	x0000	PC	x8200
R3	x0000		
R4	x0000	PSR	x8002
R5	x0000	CC	Z

Devices

Memory

BP	Address	Instruction
□	x81F3	NOP
□	x81F4	NOP
□	x81F5	NOP
□	x81F6	NOP
□	x81F7	NOP
□	x81F8	NOP
□	x81F9	NOP
□	x81FA	NOP
□	x81FB	NOP
□	x81FC	NOP
□	x81FD	NOP
□	x81FE	NOP
□	x81FF	NOP
□	x8200	NOP

WP	Address	Value
□	x0000	x0000
□	x0001	x0000
□	x0002	x0000
□	x0003	x0000
□	x0004	x0000
□	x0005	x0000
□	x0006	x0000
□	x0007	x0000
□	x0008	x0000
□	x0009	x0000
□	x000A	x0000
□	x000B	x0000
□	x000C	x0000
□	x000D	x0000

Source

Timer

Keyboard/console

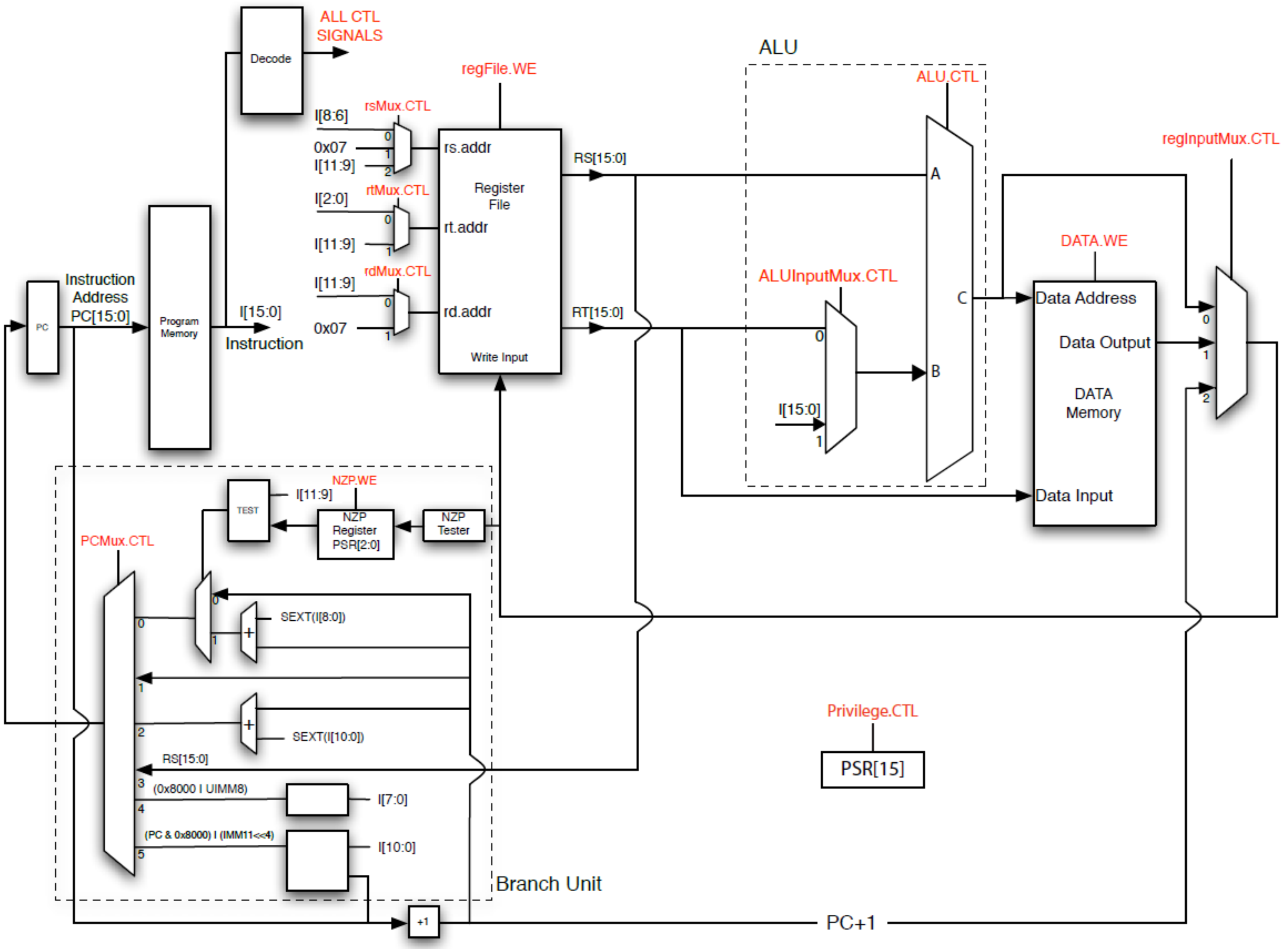
Demo: Breakout/Brick-breaker

Lecture Outline

- I/O Devices in LC4 Overview
- **Interacting with I/O in LC4 Assembly**
  - **Memory Mapped I/O**
  - **Keyboard & ASCII Display**
  - **Timer**
  - **Video Display**
- Subroutines in LC4

Where is I/O accessed in this computer?

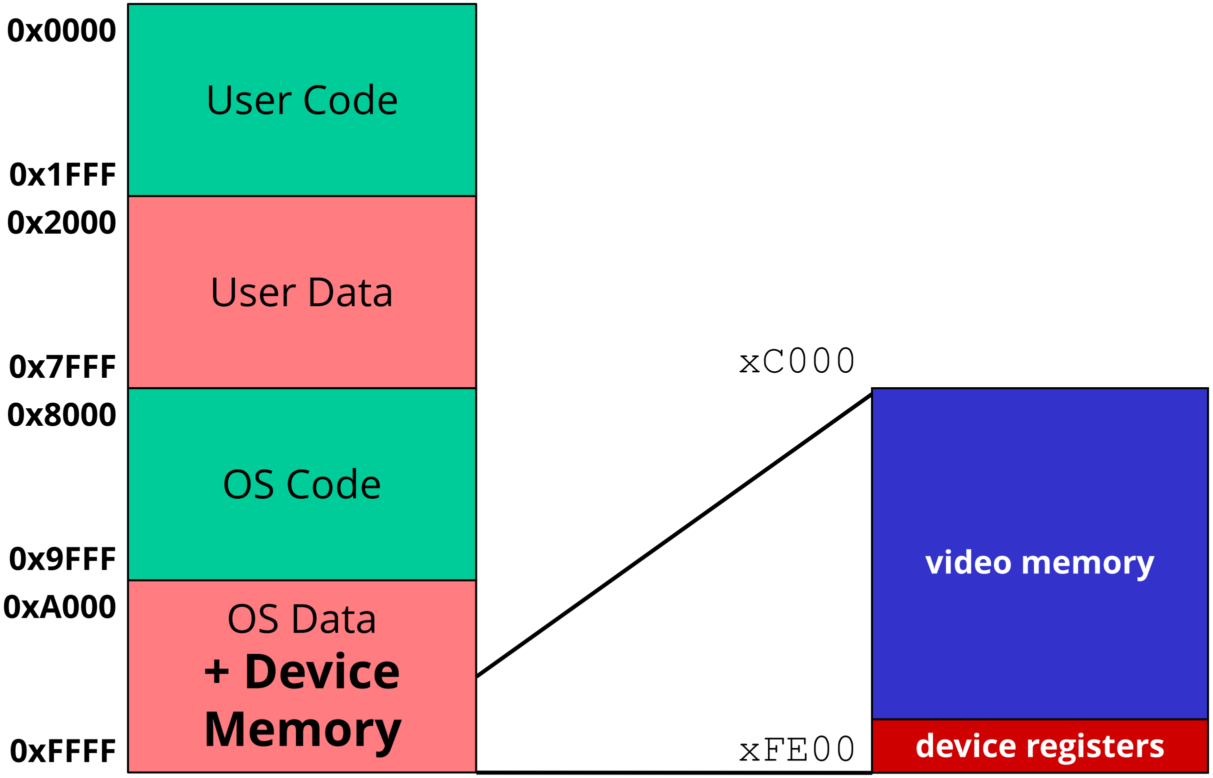
Single Cycle Implementation of the LC4 ISA



How can we handle I/O in LC4?

- **Two common options**
- We could create new “I/O instructions” for the ISA
  - Designate opcode(s) for I/O
  - Register and operation encoded in instruction
- Memory-mapped I/O (Using LDR/STR for LC4)
  - Assign a memory address to each device register
  - Use conventional loads and stores
  - Hardware intercepts loads/stores to these address
  - No actual memory access performed
  - LC4 (and most other platforms) do this

LC4 Device Memory



LC4 ASCII I/O Device Registers

These are NOT registers like R0-R7, PC, and PSR

These are memory locations from the ASM perspective

- Keyboard status register (KBSR): `xFE00`
  - KBSR[15] is 1 if keyboard has new character
- Keyboard data register (KBDR): `xFE02`
  - KBDR[7:0] is last character input on keyboard
- ASCII display status register (ADSR): `xFE04`
  - ADSR[15] is 1 if console ready to display next character
- ASCII display data register (ADDR): `xFE06`
  - ADDR[7:0] is written to console

`xFE00` **device registers**

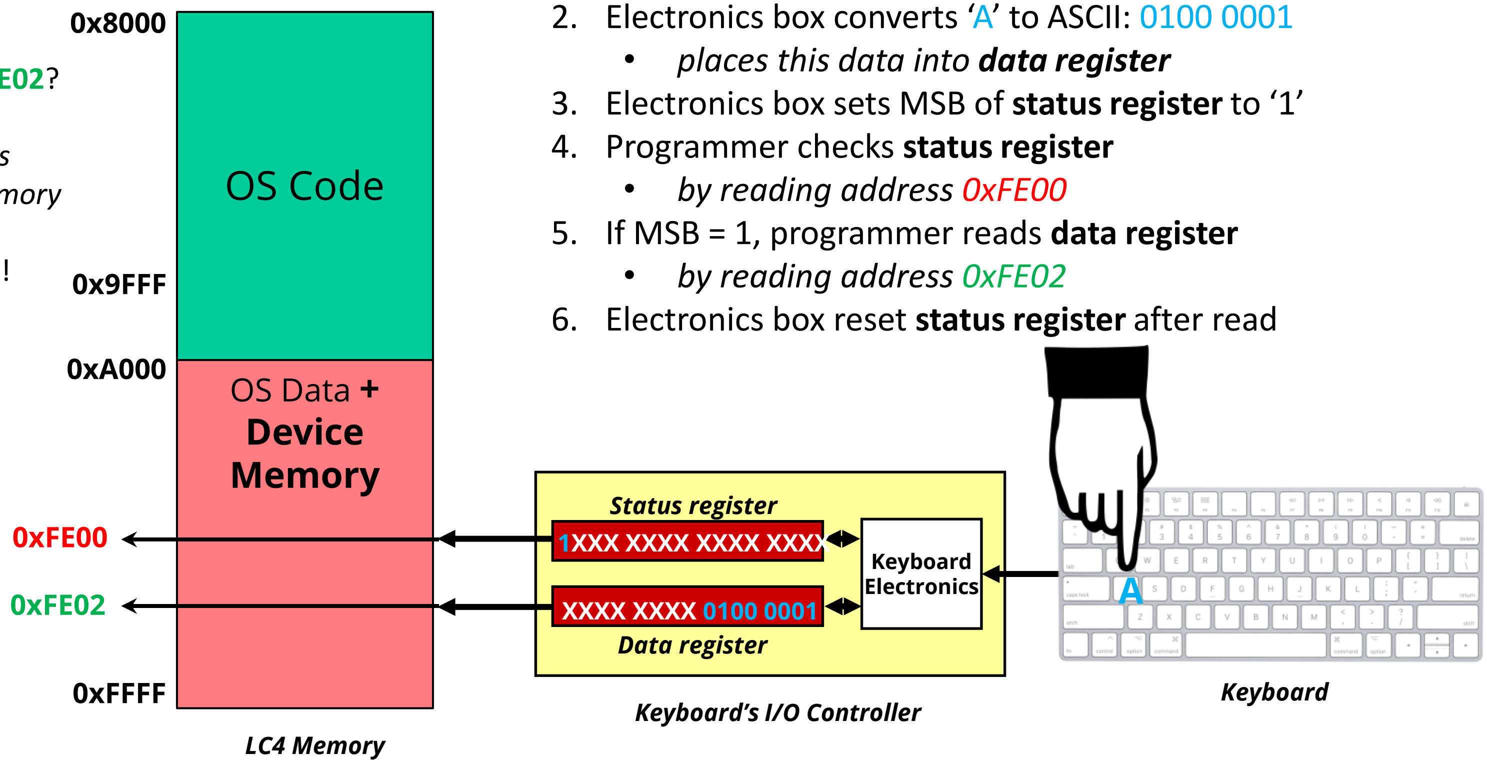
Memory Mapped I/O Demo

1. User presses ‘A’ key on keyboard
2. Electronics box converts ‘A’ to ASCII: 0100 0001
   - *places this data into **data register***
3. Electronics box sets MSB of **status register** to ‘1’
4. Programmer checks **status register**
   - *by reading address 0xFE00*
5. If MSB = 1, programmer reads **data register**
   - *by reading address 0xFE02*
6. Electronics box reset **status register** after read

How to read **xFE00** & **xFE02**?

***status** & **device** Regs*
are mapped to data memory

LDR will do the trick!



LC4 Memory

Keyboard’s I/O Controller

Keyboard

Aside: Constants in LC4

- Can declare signed/unsigned constants using **.CONST**/**.UCONST**

```
OS_KBSR_ADDR .UCONST xFE00  ; 'alias' for keyboard status reg
```

  - Recall, this is an assembly “directive”
  - Mnemonic: **.UCONST UIMM16**
  - Function: associate **UIMM16** with preceding label
  - Defines an *unsigned* 16-bit constant (**.CONST** is for signed) that doesn’t show up in memory.
  - Handy tool for us to declare an “alias” for a integer value to use for the **LC** pseudo-instruction (**LC** details on next slide)
  - Why not just use **.FILL**?
    - **.FILL** directives show up in data memory
    - **.UCONST** directives don’t

Aside: Using Constants in LC4

- Set registers to a constant with the **LC** pseudo-instruction

```
OS_KBSR_ADDR .UCONST xFE00  ; 'alias' for keyboard status reg
LC R0, OS_KBSR_ADDR         ; R0 = address of keyb status reg
```

- **LC** (Load Constant)
  - Assembler pseudo-instruction similar to **LEA**
  - Expands into **CONST**, **HICONST** pair
  - Loads value at label rather than address of label
    - **LEA** reads address of the label

Example: Reading from Keyboard

```
; code will read 1 character from the keyboard, store it in R0

OS_KBSR_ADDR .UCONST xFE00  ; ‘alias’ for keyboard status reg
OS_KBDR_ADDR .UCONST xFE02  ; ‘alias’ for keyboard data reg


.CODE
GETC                  ; a LABEL for now (perhaps subroutine someday)
   LC R0, OS_KBSR_ADDR; R0 = address of keyboard status reg
   LDR R0, R0, #0     ; R0 = value of keyboard status reg & updates NZP
   BRzp GETC          ; if R0[15]=1, data is waiting!
                      ;   else, loop and check again...
                      MSB = 1, means value is negative
   ;; reaching here, means data is waiting in keyboard data reg

   LC R0, OS_KBDR_ADDR ; R0 = address of keyboard data reg
   LDR R0, R0, #0      ; R0 = value of keyboard data reg
```

When complete, R0 contains ASCII character from keyboard

Poll Everywhere

pollev.com/tqm

- What instructions do we need to change to PUTC (print a character) from GETC (read a character)? (Ignore changes to inputs, e.g. registers/labels/constants used)

A. **Line 4 (last LDR)**

B. **Line 2 (BRzp)**

C. **Both**

D. **Neither**

E. **I'm not sure**

```
; code will read 1 character from the
; keyboard, store it in R0. What if
; we wanted to change it to write the
; character in R0 to ASCII display

OS_KBSR_ADDR .UCONST xFE00
OS_KBDR_ADDR .UCONST xFE02

.CODE
GETC
0    LC R0, OS_KBSR_ADDR
1    LDR R0, R0, #0
2    BRzp GETC
3    LC R0, OS_KBDR_ADDR
4    LDR R0, R0, #0
```

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

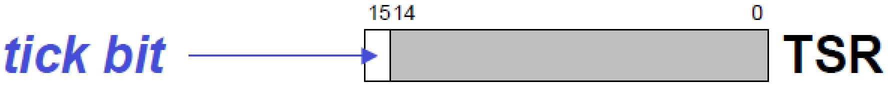
Aliases for ASCII display status & data regs

Get character from keyboard

Print character to ASCII display

LC4 Device Register: Timer

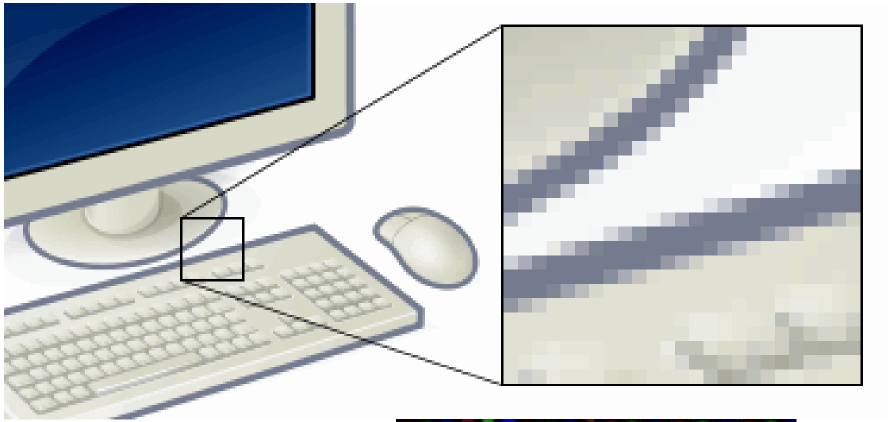
- TIMER:
  - Timer interval register (**`TIR`**): **`xFE0A`**
    - Set desired time in **`TIR`** (in msec)
  - Timer status register (**`TSR`**): **`xFE08`**
    - **`TSR[15]`** is 1 if timer has "gone off", sets itself to 0 after read

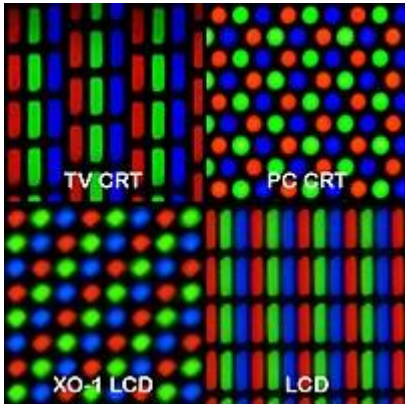


- Works like an egg timer, set desired time in **`TIR`**, Then poll/check **`TSR`** to see if time has expired

Aside: Displays & Pixels

- Pixel: Smallest addressable element of most images and video display devices
  - Usually a pixel will represent a single square on a display.
  - The whole display is made of these small pixels





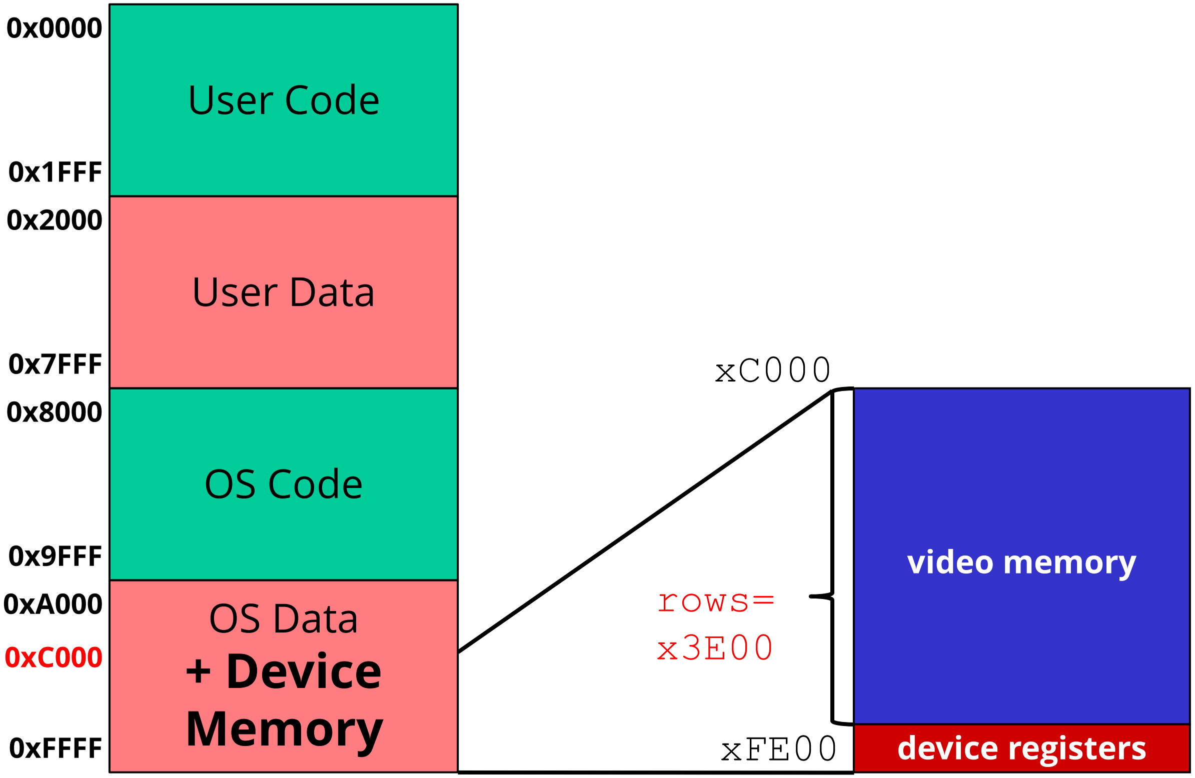
- Each Pixel's color is created by some amount of **R**ed, **G**reen and **B**lue (RGB)

- Short video on how RGB works for those interested: First 2.5 min of "This Is Not Yellow" By Vsauce on YouTube

LC4 Device Register: Video

- VIDEO:
  - Video display **control** register (**VDCR**): **xFE0C**
    - Can be used to clear screen or swap video buffers
  - Video display's many data registers: xC000-xFDFF
    - There are 15,872 pixels, each pixel needs its own register containing the color for that pixel

Video Memory



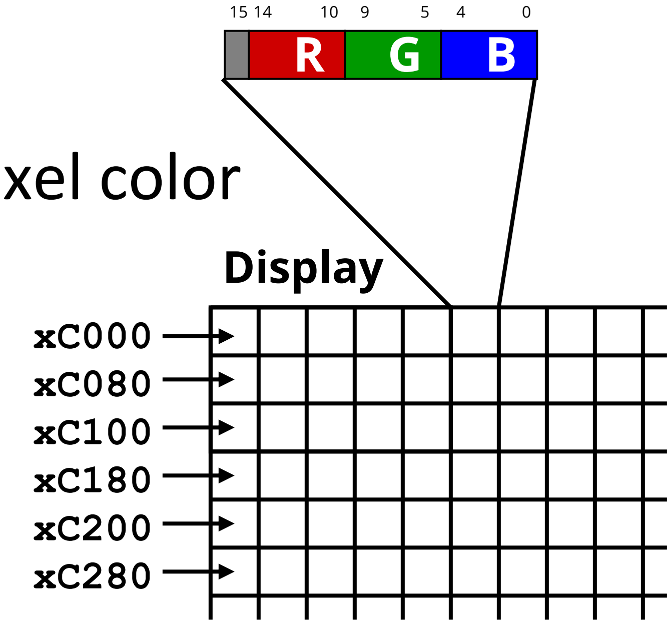
0x3E00 converts to
=15,872

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

One memory location
per pixel on video display

LC4 Pixel-Based Video Display

- LC4 has a 128x124 16b RGB (32K color) pixel display
  - **128** columns (0-127) and **124** rows (0-123)
  - Entire display is memory-mapped
    - One memory location per pixel
    - Memory region xC000-xFDFF
    - xC000-xC07F is first row,
      xC080-xC0FF is second row, etc.
- Write to memory location to set pixel color
  - Your job: compute location of pixel
  - Then STR color to that address

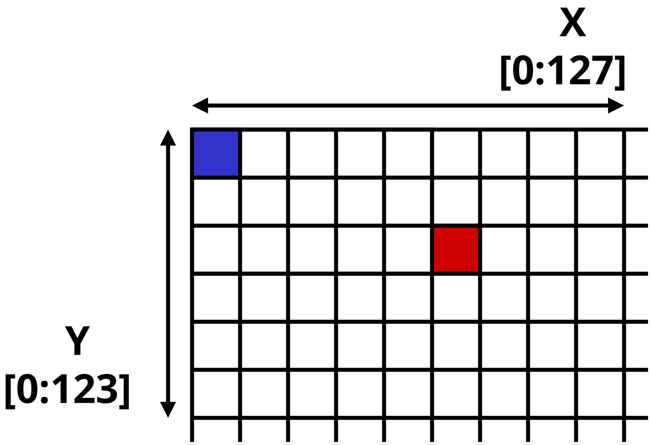


Addressing a Pixel

- 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)



Poll Everywhere

pollev.com/tqm

- If I drew a pixel at offset 261 (vmem[2][5]) into OS_VIDEO_MEM and wanted to draw the pixel above it on the display, which offset should I write to?

```
.ADDR xC000
OS_VIDEO_MEM        .BLKW x3E00
OS_VIDEO_NUM_COLS   .UCONST #128
OS_VIDEO_NUM_ROWS   .UCONST #124
```

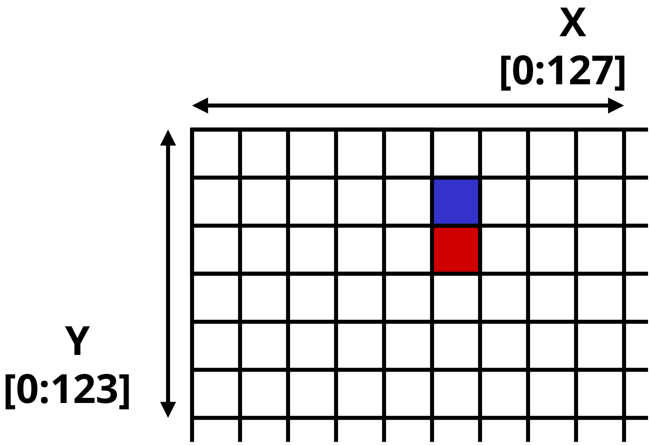
A. 259

B. 261

C. 133

D. 389

E. I'm not sure



Poll Everywhere

pollev.com/tqm

- If I drew a pixel at offset 261 (vmem[2][5]) into OS_VIDEO_MEM and wanted to draw the pixel above it on the display, which offset should I write to?

```
.ADDR xC000
OS_VIDEO_MEM         .BLKW x3E00
OS_VIDEO_NUM_COLS .UCONST #128
OS_VIDEO_NUM_ROWS .UCONST #124
```

A. 259

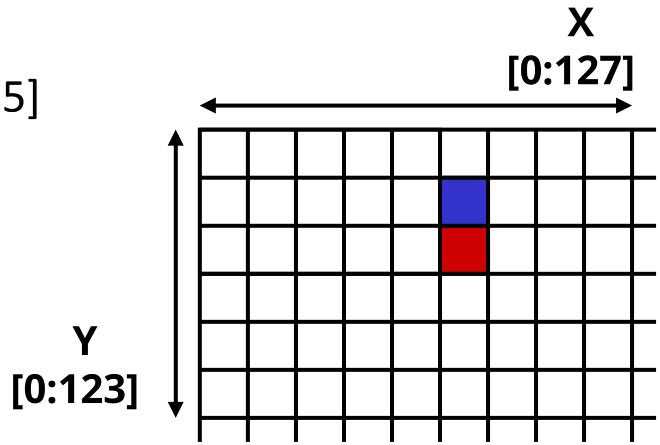
B. 261

[✓ C. 133]

D. 389

E. I’m not sure

Pixel above: vmem[1][5]
offset = (1 * 128) + 5



Demo: Drawing a Horizontal Line

- `draw_horizontal_line.asm` on course website

Lecture Outline

- I/O Devices in LC4 Overview
- Interacting with I/O in LC4 Assembly
  - Memory Mapped I/O
  - Keyboard & ASCII Display
  - Timer
  - Video Display
- **Subroutines in LC4**

"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
  - Actual implementation: `JMPR R7`

Creating a Subroutine:

- Consider the multiply program from 2 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
```

- NOTE: the same registers R0-R7 are used inside and outside a subroutine. (These are NOT parameters)
  - We can’t always be sure that a certain register will not be changed
  - If we wanted to keep any values in registers the same after the subroutine, we must store them in memory
    (we’ll return to this much later in the semester)

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? (More next lecture)