I/O CONTROLLER INTERFACE AND I/O DEVICES ON THE LC4

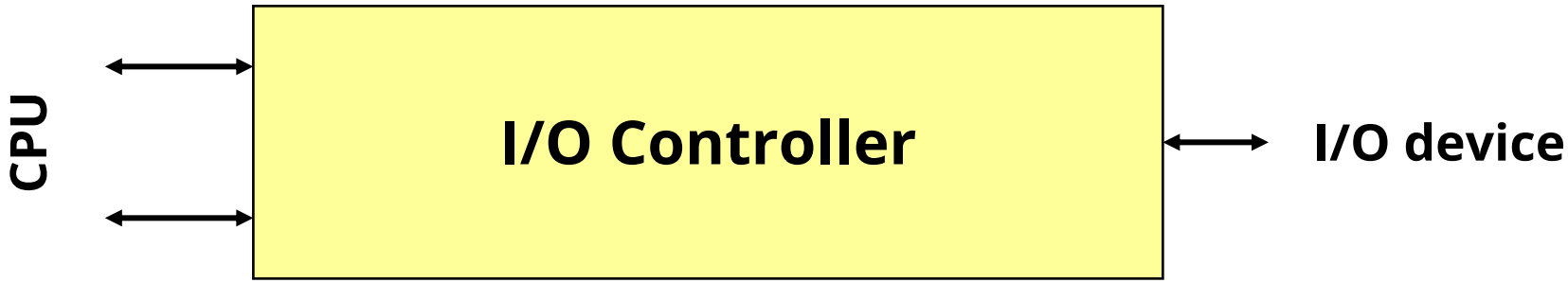
I/O Devices and Controllers

Most I/O devices are not purely digital themselves …

- Electro-mechanical: e.g. keyboard, mouse, disk, motor
- Analog/digital: e.g. touchscreen, network interface, monitor, speaker, mic

… all have digital interfaces presented by **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”

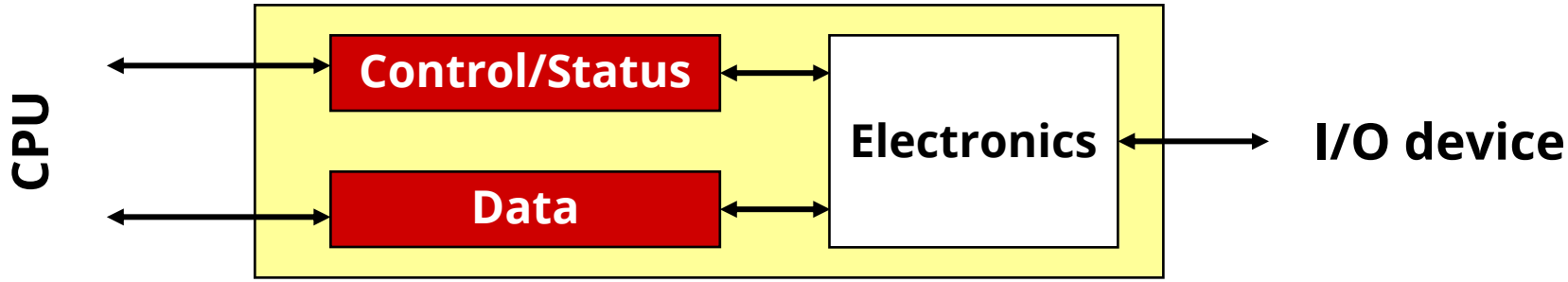
- **Control/Status**: may be one register or two
- **Data**: may be more than one of these

For input devices

- CPU checks **status register** if input is available
- Reads input from **data register** (or waits if no input available)

For output devices

- CPU checks **status register** to see if it can write
- Writes output to **data register**



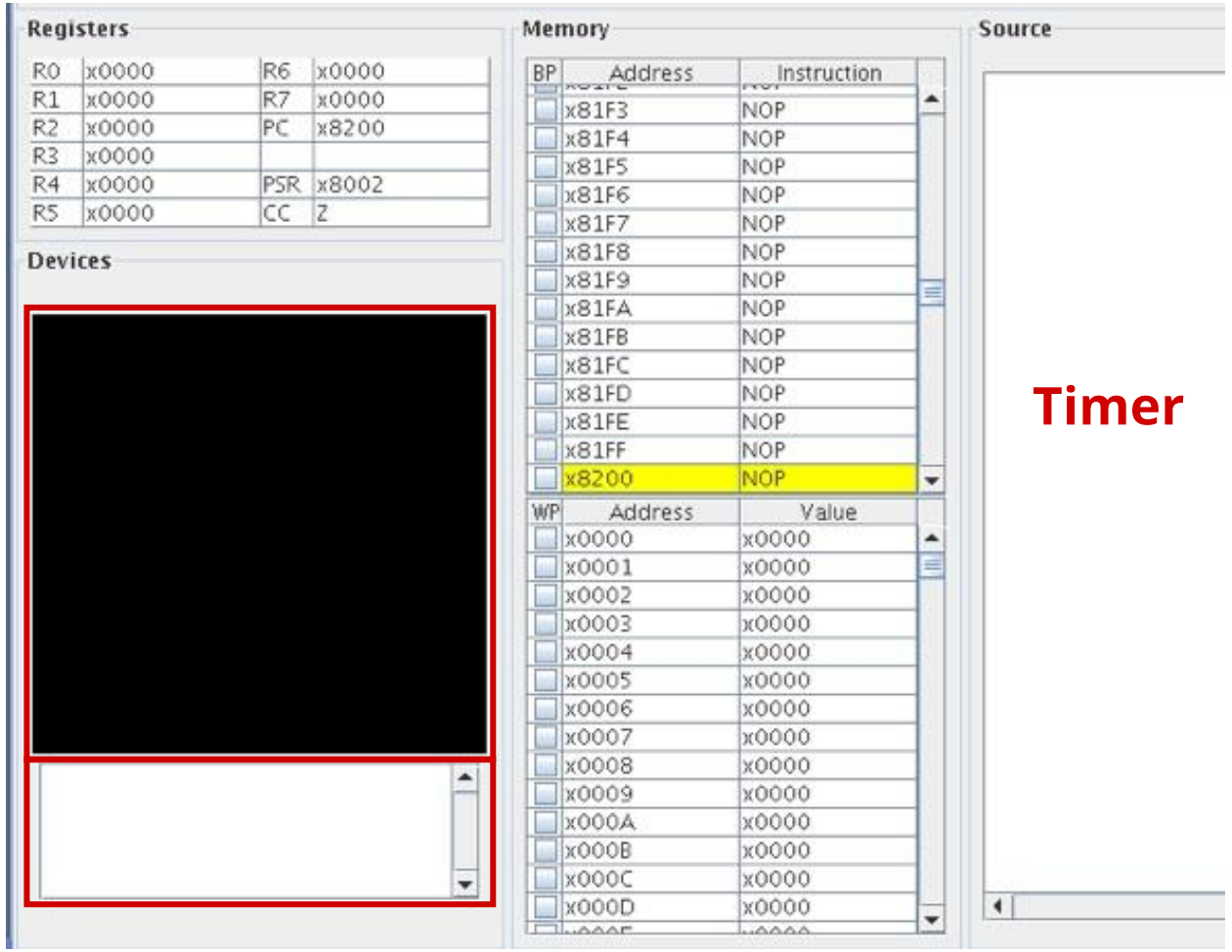
LC4 I/O Devices

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

Keyboard/console



HOW ARE DEVICE REGISTER READS/ WRITES PERFORMED?

How are Device Register Reads/Writes Performed?

Two options (aren't there always?)

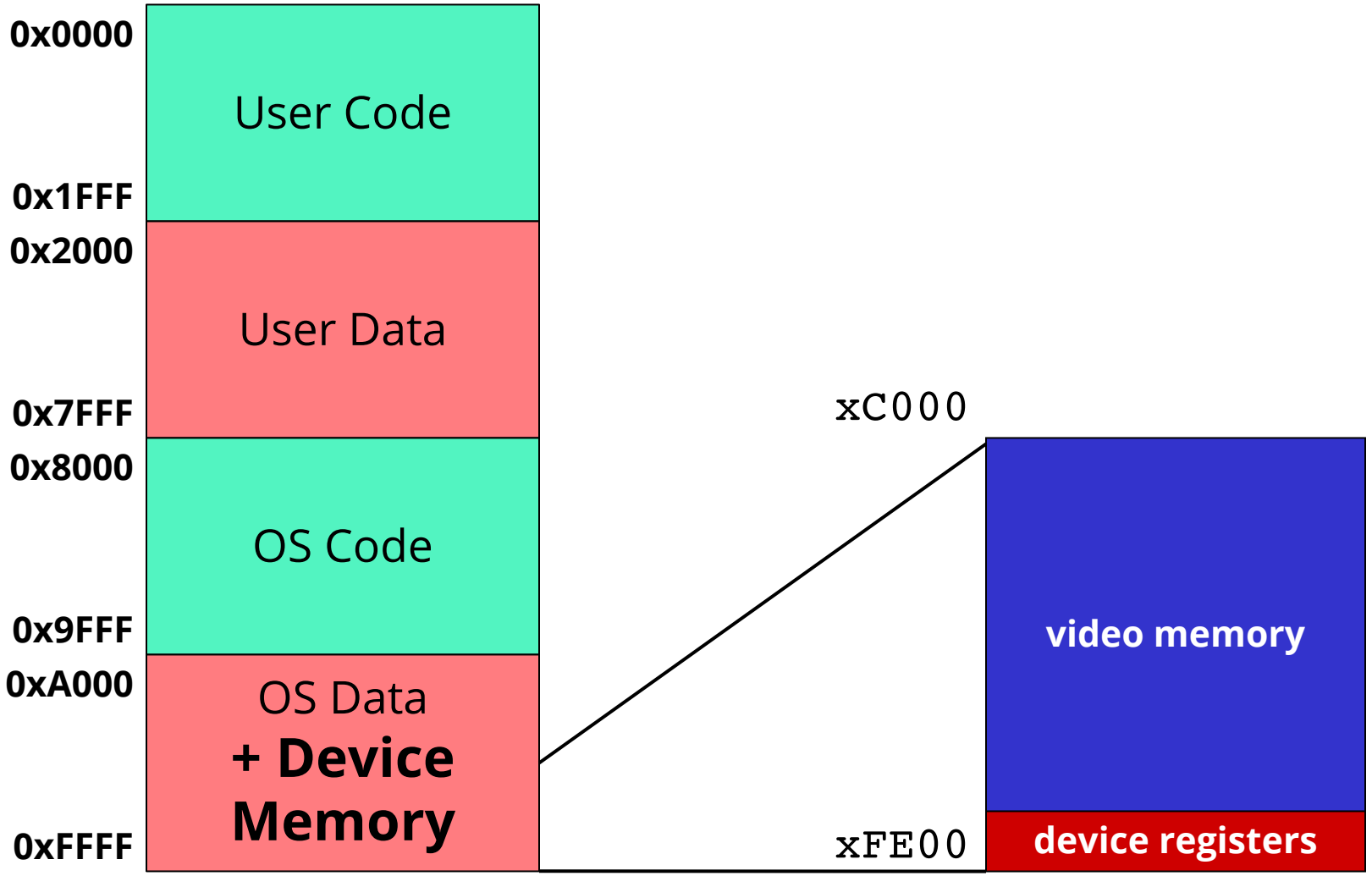
We could create new "I/O instructions" for the ISA

- Designate opcode(s) for I/O
- Register and operation encoded in instruction

Memory-mapped I/O *(we already have these in ISA: LDR/STR)*

- 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 Data Memory (The Other Half)



LC4 ASCII I/O Device Registers

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**

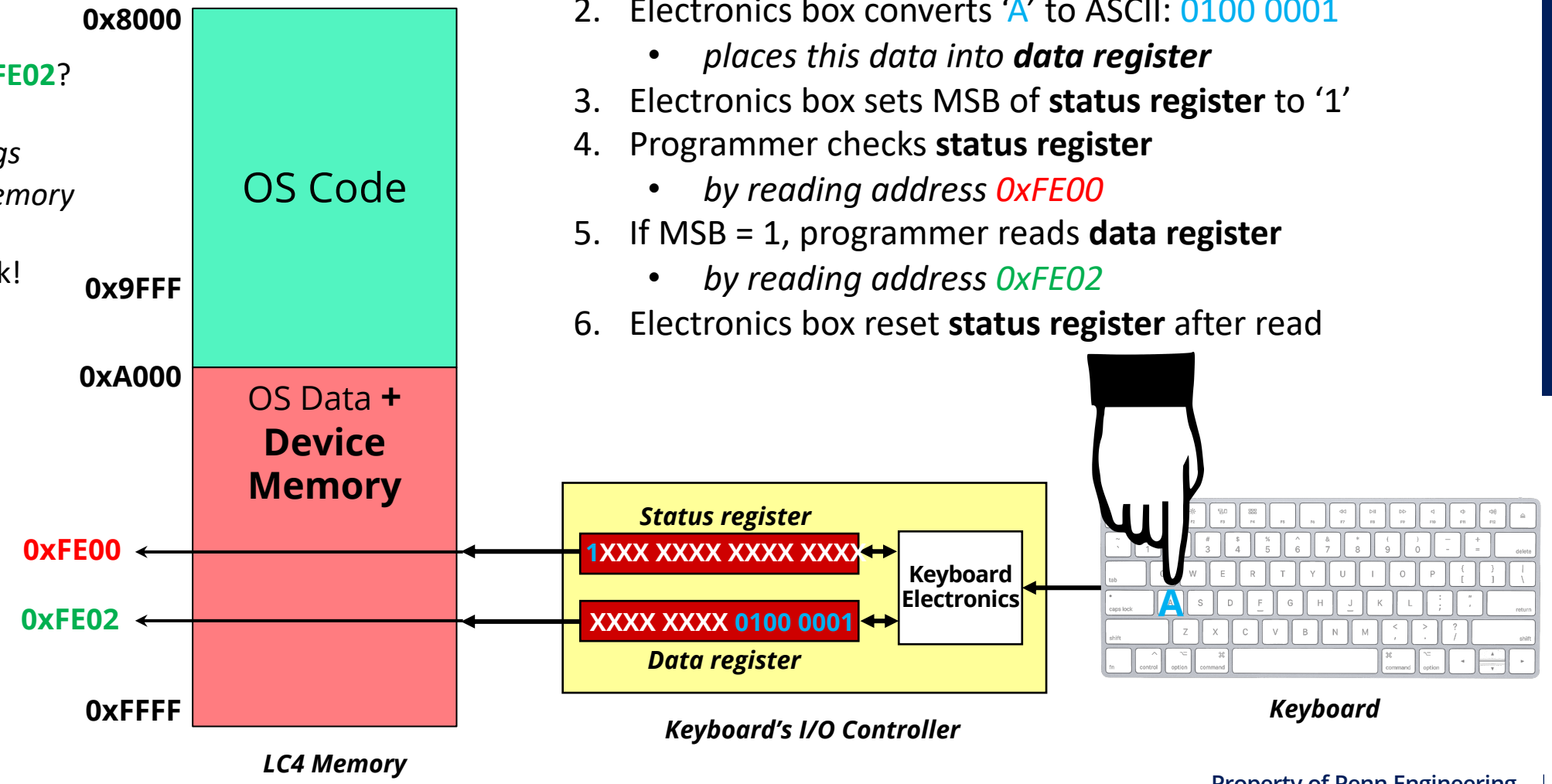
MEMORY “MAPPED” I/O – HOW DOES IT WORK?

Memory "Mapped" I/O – How Does It Work:

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

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

LDR will do the trick!



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

DEVICE 1: KEYBOARD – READING INPUT

Example 1: 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
   BRzp GETC                ; if R0[15]=1, data is waiting!
                            ; else, loop and check again...

;; 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

Assembler Constants Using Directives

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

`.UCONST`

- Recall, this is an assembly "directive"
- Mnemonic: `.UCONST UIMM16`
- Function: `associate UIMM16 with preceding label`

- Defines an *unsigned* 16-bit constant (unlike .CONST)
- Why not just use `.FILL`?
  - `.FILL` directives show up in data memory
  - `.UCONST` directives don't

Assembler Constants Using Directives

```
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***

DEVICE 2: ASCII DISPLAY – WRITING OUTPUT

Example 2: Read from Keyboard, Print 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

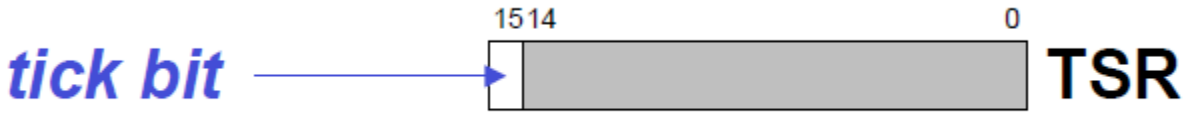
Print character to ASCII display

DEVICE 3 & 4: TIMER & VIDEO

LC4 I/O Device Registers: Timer / Video

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



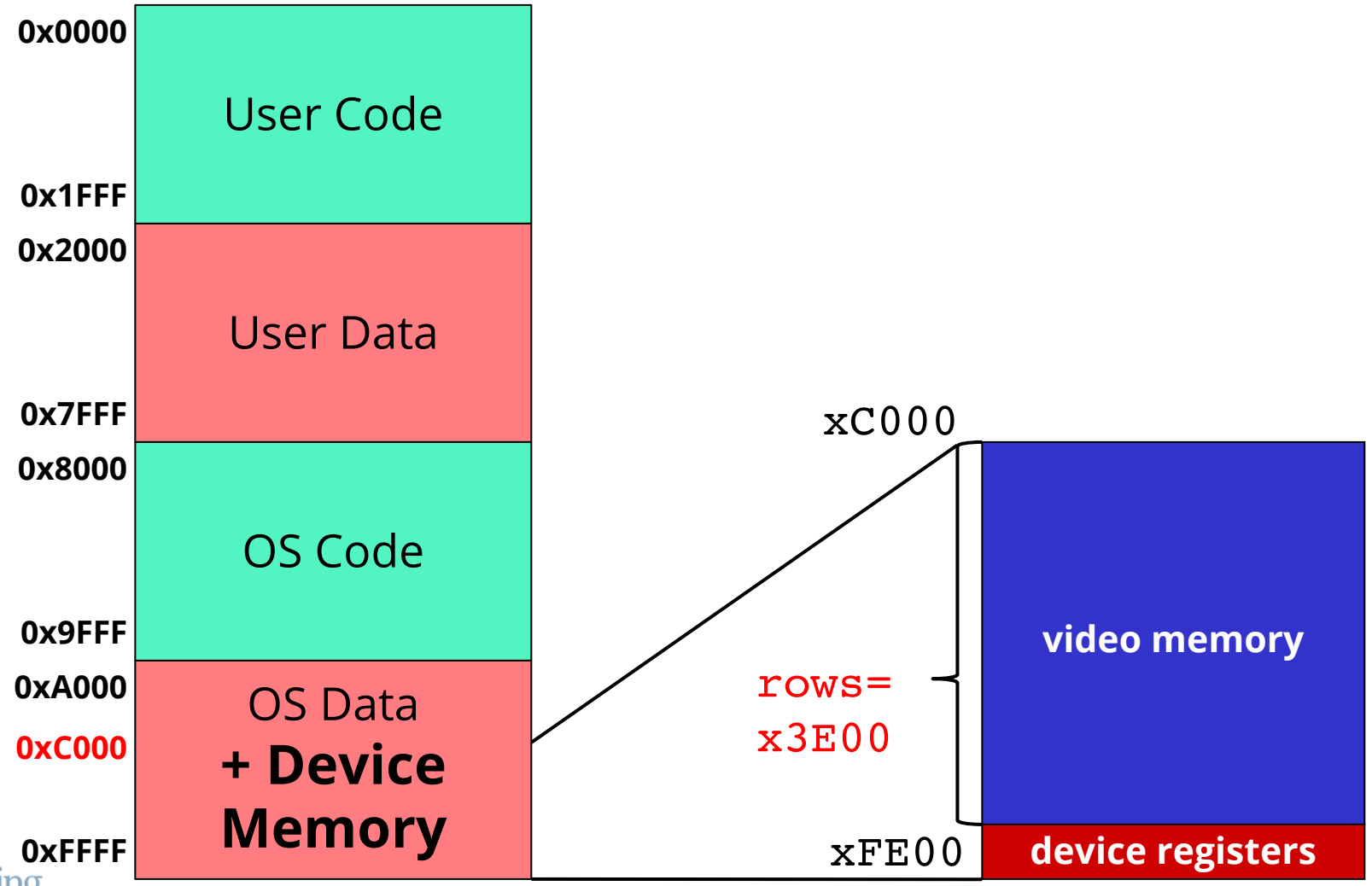
- Works like an egg timer, set desired time in TIR,
  - Then poll/check TSR to see if time has expired

VIDEO:

- Video display **control** register (**VDCR**): `xFE0C`
- Video display's many **data** registers: `xC000–xFDFF`
  - *We'll talk about how to work with video next!*

DEVICE 4: VIDEO – HOW TO CALCULATE PIXELS

Where Does Video Memory Live?



`0x3E00` converts to =15,872 row / drawers

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

One row of video memory per pixel on video display

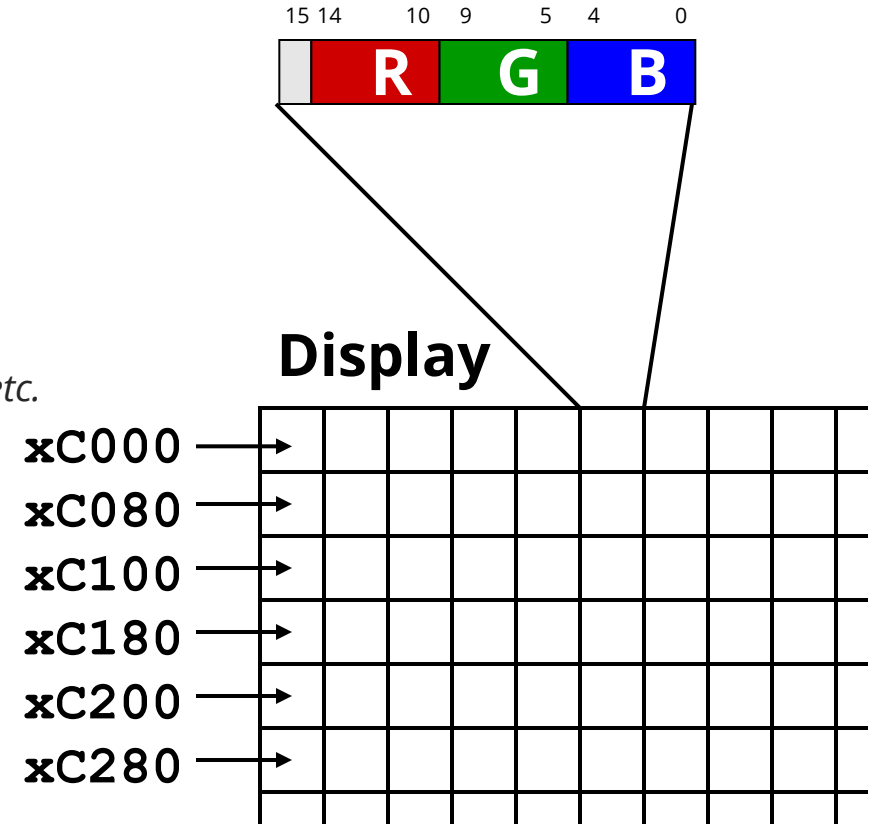
LC4 Pixel-Based Video Display

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



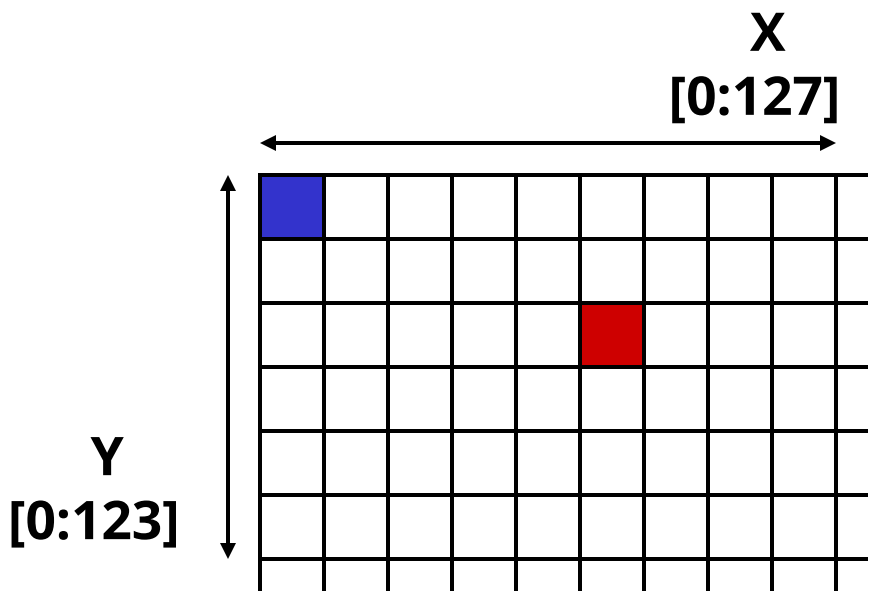
Addressing a Pixel

What's the difficult thing here?

- Computing memory address that corresponds to `vmem[y][x]`

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

- Logically 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



USER MEMORY VS. OS MEMORY

LC-4 Memory Map

Recall the full LC-4 Memory Map

- We have 2 Program Memories
- And 2 Data Memories

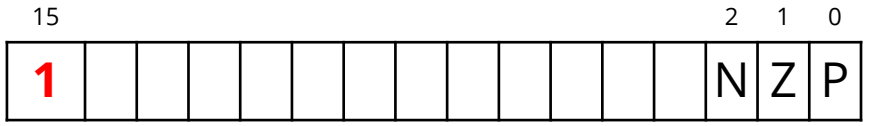
User Region

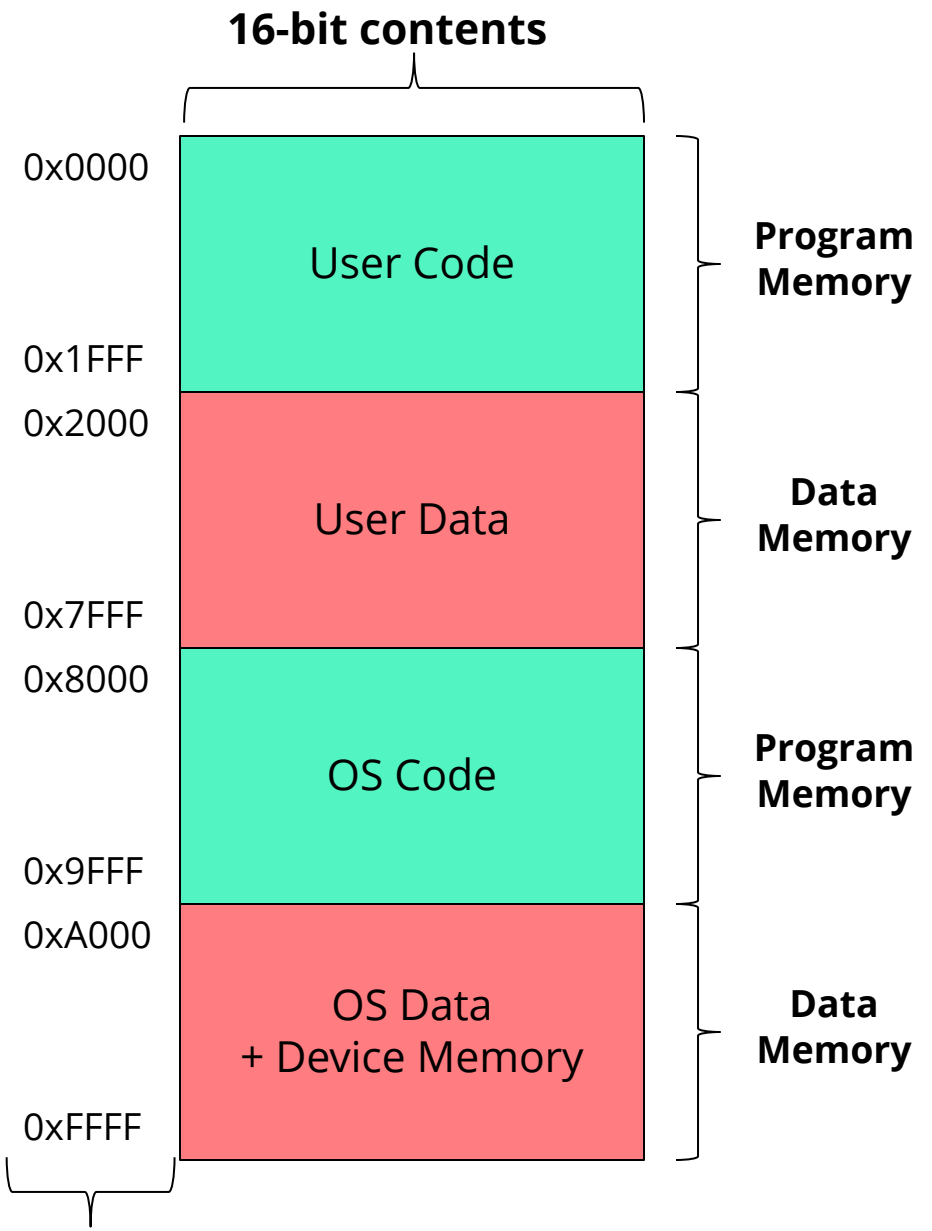
- Programs run by users (e.g.: factorial program)
- Processes run in user mode have **PSR[15]=0**
- ***NOT allowed*** to access OS locations in memory

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





Installing GETC in OS Program Memory

```
; 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
   LDR R0, R0, #0
   BRzp GETC

   LC R0, OS_KBDR_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*

Installing GETC in OS Program Memory

```
; 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]=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 from User Program Memory

When a program is running in User Program Memory, PSR[15] = 0

- We know we can't LDR/STR to device memory
- This is why we installed our GETC subroutine in OS program memory, so that PSR[15]=1
- But what if we are running a program in USER program memory (PSR[15]=0)…
  - ➢ *and we want to JSR to a subroutine in OS program memory (to perform some I/O)?*
  - ➢ JSR won't allow it! Recall JSR semantics: PC = (PC & 0x8000) | (IMM11 << 4)
  - ➢ JMP won't allow it either…
    - – Why?? The LC4 will kill any program that attempts to jump into OS memory, since its PSR[15]=0
- What can we do?
  - ➢ There 1 instruction in the ISA that can change PSR[15] from 0 to 1…do you recall it?
    - – **The TRAP instruction!**

WHAT IS A TRAP?

TRAP Instruction vs. JSR 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 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 form 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
    - – *Fancy name for subroutines in OS? You guessed it: TRAPS*

RTI Instruction vs. RET Pseudo-Instruction

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 RET instruction:

- Allow a subroutine running in the OS program memory
  - ➢ to return back to a caller in the USER program memory

Installing GETC in OS Program Memory

```
; 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       ; gets character from keyboard, saves it in R0
   RTI                  ; sets PC = R7 and restores PSR[15]=0
```

User Mode vs OS Mode – An example

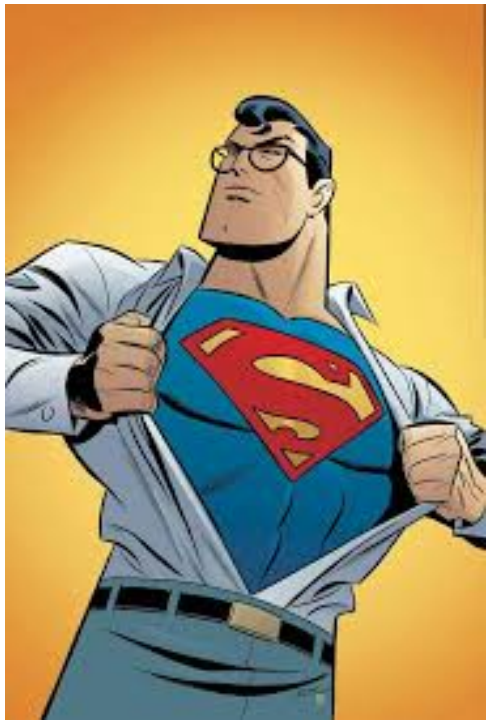
User Mode

Mild mannered reporter

Decent typist

PSR[15]=0…No powers

SuperUser Mode



All powerful!

PSR[15]=1

THE TRAP VECTOR TABLE

The Limits of the TRAP Instruction

Mnemonic	Semantics	Encoding
TRAP UIMM8	R7 = PC+1, PC = (x8000 | UIMM8), PSR[15] = 1	1111----UUUUUUUU

The TRAP instruction is limited:

- It does not allow the user to jump just anywhere in the OS
- In only let's the user jump into the first 256 rows of the OS
- Why? So that the user doesn't have free range in OS program memory

How it limits the user:

- In the semantics: PC = (x8000 | UIMM8)
  - ➢ What is the largest 8-bit number you can make? 256, 255 if you start from 0
  - ➢ e.g.: PC = x8000 | xFF = x80FF

Installing TRAPs Properly in OS Program Memory

```
; 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
```

- Because of TRAPs limitation,
  - *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?
  - Say our TRAP is > 256 lines long,
  - we couldn't have more than 1 TRAP

Installing TRAPs Properly in OS Program Memory

Since TRAP instruction can only jump to the first 256 rows of OS Program Memory…

- We'll make the first 256 rows JMP instructions!
- They will serve as a jumping off point to TRAPS deeper in the OS Program Memory

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

The first 256 lines of OS Program Memory

- *called the:* ***TRAP VECTOR TABLE***

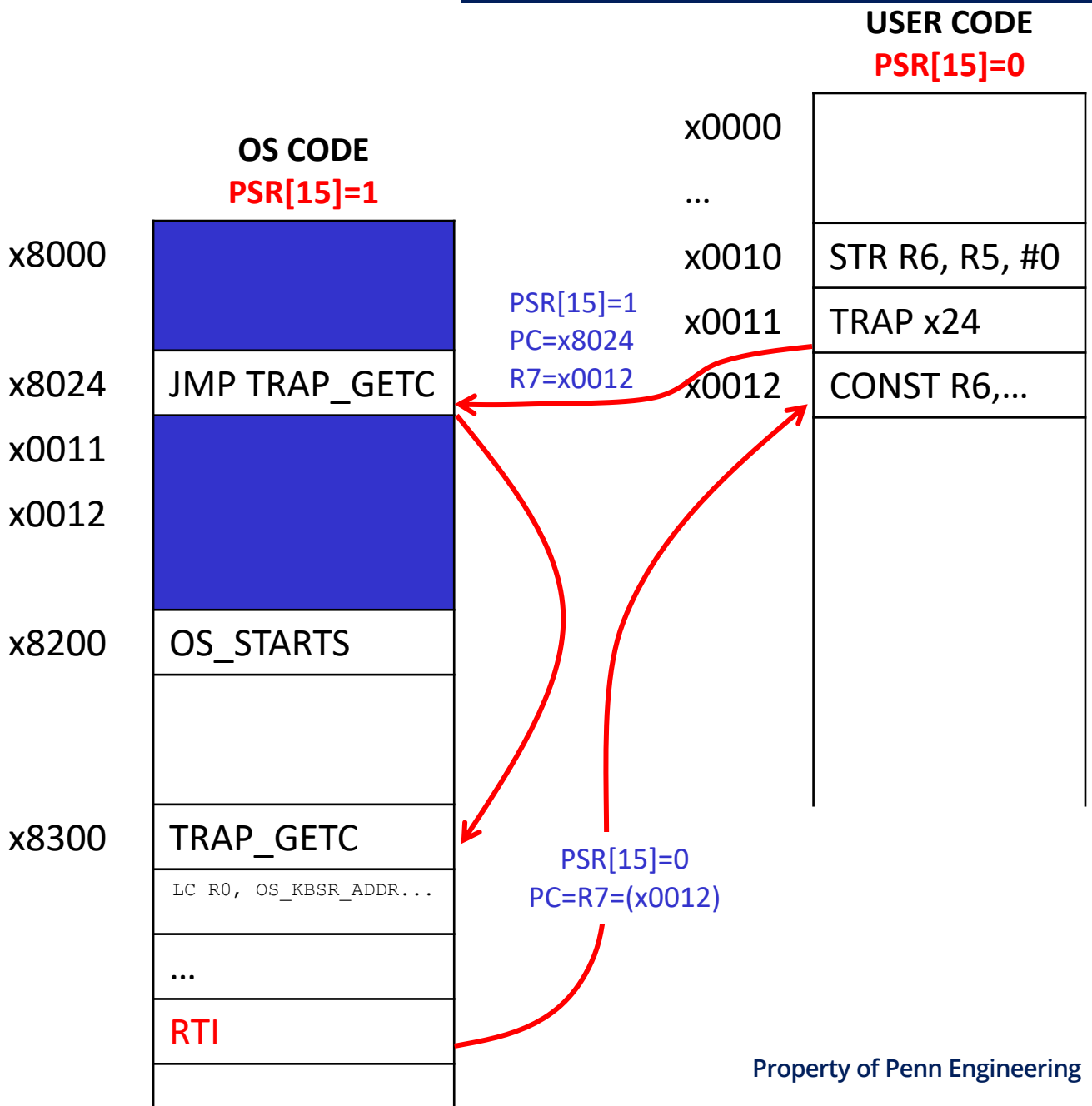
- We publish this list to the user:
  - user can call the TRAPS by number:
  - *e.g.: TRAP x01, will call TRAP: PUTC*
  - *the table listing helps them map # to TRAP*

Installing TRAPs Properly in OS Program Memory

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

Anatomy of a Trap

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



A Word On What We've Created…

This system of TRAP calling, vector table organization, protection of the hardware

- This **system** of handling I/O…this way we are **operating**…
- Becomes known as the **Operating System** for the LC4
- The Operating System essentially serves as the means to handle I/O
- In the next segment, we'll abstract some of these ideas…
  - But I wanted you to know why we call this **OS** Program Memory!

PASSING DATA TO/FROM TRAPS

Things to Note About TRAP Calls:

1) The TRAP subroutine uses register file too!

- You may be using register file in your code in user program memory
  - ➢ The TRAP may overwrite it!
  - ➢ Don't expect register file to be in the same shape as when you called the TRAP

2) Two ways to pass arguments to TRAPS:

- Use a register in the register file

  Or

- Use data memory and pass a starting address in data memory to the TRAP

Backing Up the Register File

The register file will be used inside a TRAP

- *It will likely overwrite everything in the REGFILE*
- BEFORE you call a TRAP, 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]
TRAP x03                ; call the TRAP
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 TRAP

Restore content of REGFILE AFTER you return from TRAP

Examples of Passing Data to/from Traps

TRAP_GETC

- Uses register file's R0 to return data to the calling program

TRAP_PUTC

- Uses register file's R1 to get data from the calling program

TRAP_PUTS

- For your HW
- Before you call the trap, you place a "string" or ASCII characters in user data memory:
  - Ex: x4000 = 'T'
    x4001 = 'O'
    x4002= 'M'
- Next, set register file's R0 = x4000 (starting address in data mem)
- Call the trap, passing in R0
- Trap does an LDR from R0, getting 'T' from user data mem
  - ➢ Continues looping loading characters until data = 0

OPERATING SYSTEMS – LOGICAL DISCUSSION

Operating Systems (OSes)

First job of an OS:

- Handle I/O
- OSes virtualize the hardware for user applications

In real systems, only the operating system (OS) does I/O

- "User" programs *ask* OS to perform I/O on their behalf
- 3 reasons for this setup:

1) Standardization

- I/O device interfaces are nasty, and there are many of them
- Think of disk interfaces: S-ATA, iSCSI, IDE
- User programs shouldn't have to deal with these interfaces
  - ➢ In fact, even OS doesn't have to deal with most of them
  - ➢ Most are buried in "device drivers"

Operating Systems (OSes)

2) Raise the level of **abstraction**

- Wrap nasty physical interfaces with nice logical ones
  - ➢ Wrap disk layout in file system interface

3) Enforce **isolation** (usually with help from hardware)

- Each user program thinks it has the hardware to itself
  - ➢ User programs unaware of other programs or (mostly) OS
- Makes programs much easier to write
- Makes the whole system more stable and secure
  - ➢ A can't mess with B if it doesn't even know B exists

In the Real World

- This system call/TRAP mechanism is commonly used in actual systems
- As an example the BIOS (Basic Input Output System) on many Intel PCs provided precisely this functionality to allow programs to access basic input and output devices, keyboards, displays, timers etc.
    - More modern systems use EFI (Extensible Firmware Interface) which is a more sophisticated version of the same thing.

WHAT IS SYNCHRONIZATION

What Is Synchronization?

Two schemes for interacting with I/O devices:

1) Way have seen so far is **polling**

- "Are we there yet? Are we there yet? Are we there yet?"
- CPU keeps checking status register in a loop
- Very inefficient, CPU has better things to do

2) Alternative scheme is called **interrupts**

- "Wake me when we get there."
- Device sends special signal to CPU when status changes
- CPU stops current program, saves its state
- CPU "handles interrupt": checks status, moves data
- CPU resumes stopped program, as if nothing happened!!!!!!!!!

LC4 Simulator Does Not Support Interrupts

Although LC-3 ISA does

- Presumably LC4 does too, but haven't tried
- And could be tricky without PC-relative LD/ST

Why not?

- Suppose program wants to read from the keyboard
- Interrupts make sense if CPU could do something else meantime
- Program itself can't do anything, it's waiting for the key …
- …so another program has to run while it's waiting
- And LC4 doesn't support multi-programming
- So there is no real point ☺

How Interrupts Work

CPU in user mode

Doing boring user work partying etc. etc.

Holy smokes

A device is ready for input or output

Better raise the interrupt!

CPU switches to superhero mode:

Lays a smackdown on the I/O device.

When done return to regular life partying like nothing happened!

What Is Synchronization?

Interrupt driven I/O is asynchronous

- Highly overloaded word, here means “takes unbounded time”

Asynchronous programming is difficult to say the least!

- Suspend, find something else to do, resume on signal

Better leave this to the OS

- Suspends you transparently
- Runs another program while you wait (it knows about these)
- Resumes you transparently on signal

To you, I/O looks synchronous

- Similarly overloaded, here means “happens immediately”