

PennSim & LC4 Memory

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

🌐 When poll is active, respond at **pollev.com/tqm**

📱 Text **TQM** to **37607** once to join

What is your favourite editor for code?

Vim
Emacs
Nano
Notepad
Notepad++
Sublime
Atom
Visual Studio Code
A JetBrains editor (Intellij, CLion, PyCharm, etc)
MS Word or Powerpoint
MS Paint IDE
Other

Logistics

- ❖ HW03 Sequential Logic: **This Friday** 10/7 @ 11:59 pm
 - Written Homework, submitted to gradescope
 - **NO EXTENSIONS OVER 72 HOURS**
 - Should have everything you need
 - Practice in Recitations this week

- ❖ HW04 LC4 Programming: to be released this Friday
 - Programming assignment
 - May not have everything you need until Monday's lecture

- ❖ Check-in04 to be released tomorrow

In-Person Lecture Policies

- ❖ I ask that you wear a mask in lecture

- ❖ If you are using your electronics (outside of polls), please sit in the back
 - Having electronics out make it a lot easier to distracted by random notifications
 - Easy for people sitting nearby & behind you to get distracted by your distractions

Lecture Outline

- ❖ **ASM Files, Object Files, PennSim Demo**
- ❖ LC4 Program Design (if/while/for)
- ❖ LC4 Memory

More LC4 Syntax

- ❖ Integer immediates (CONST, HICONST, ADD, SLL, etc.) can be either in hexadecimal or in decimal form
 - Hexadecimal constants **0xFF** or **xFF**
 - Decimal constants: **#240**, **#-240**
- ❖ Comments
 - Comments in LC4 are preceded by a ;

- ❖ Example:

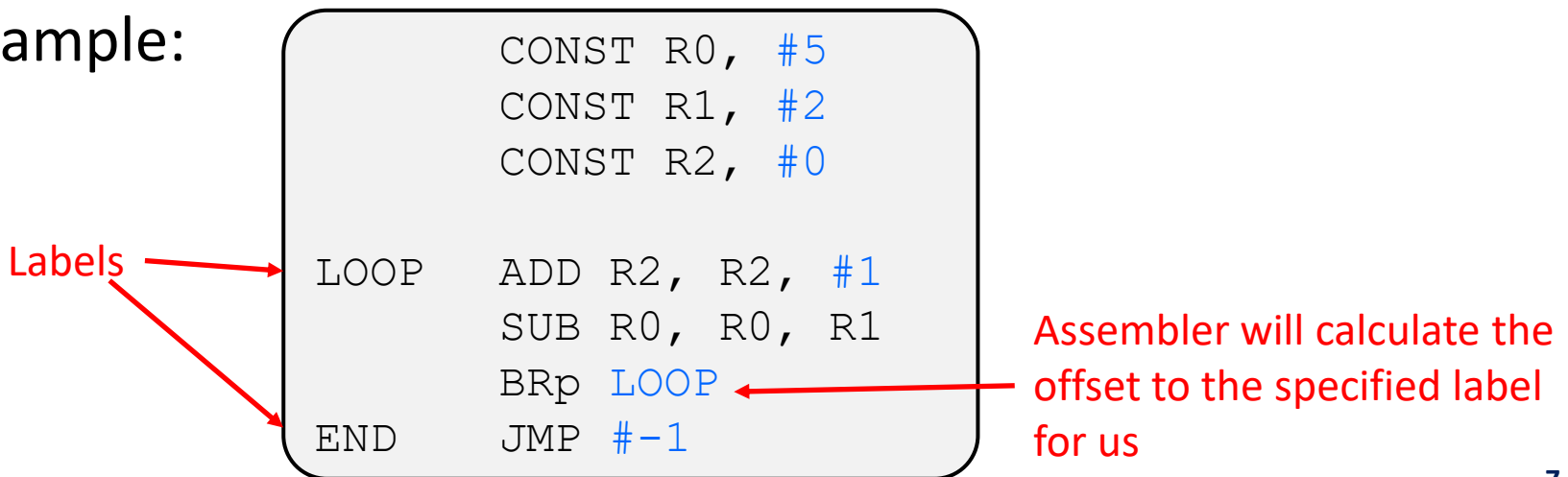
```
CONST R0, 0x20
CONST R1, x10
CONST R2, #64
; this is a comment

DIV R3, R2, R1 ; this is a comment too
ADD R3, R3, R0
```

LC4 Labels

- ❖ It can be cumbersome to calculate offsets for jumps.
- ❖ LC4 assembler allows us to put labels on memory. We can use labels for Jumps and Branch instructions to make our lives easier
 - A Label is just a “name” for a memory location. Like how we can refer to a memory location with an address.

- ❖ Example:



Code in Memory

- ❖ An instruction fits in 1 memory location (16 bits)
- ❖ These instructions are stored in memory and accessed sequentially
 - When we trace through the code, we are just accessing the next location in memory

```

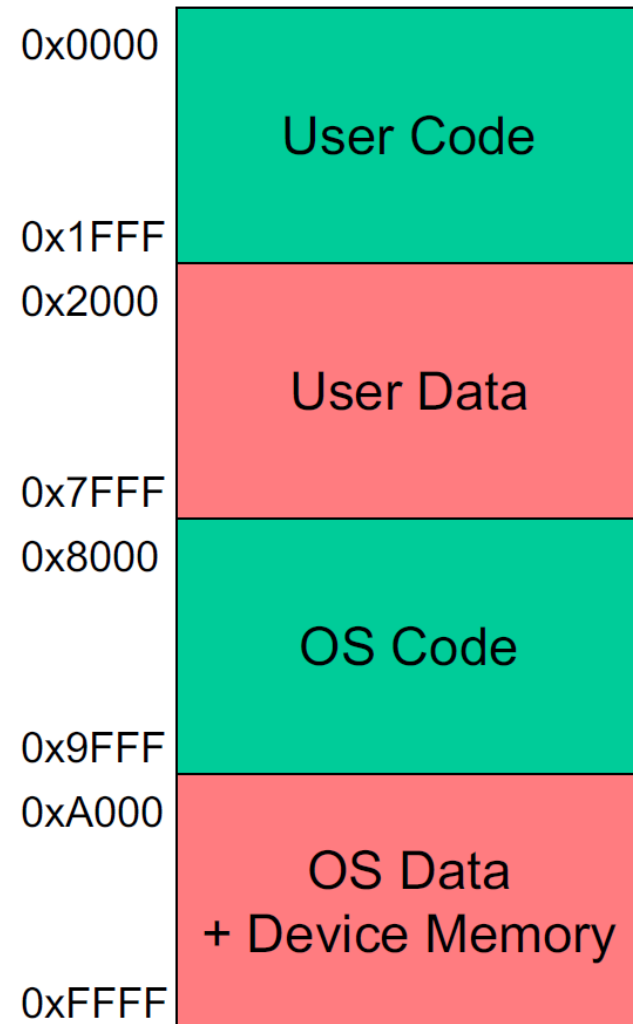
CONST R0, #32
CONST R1, #16
CONST R2, #64

DIV R3, R2, R1
ADD R3, R3, R0
SUB R0, R2, R3
    
```

Index # (Address)	Information (Data)
0	0x9020
1	0x9210
2	0x9440
3	0x1699
4	0x1681
5	0x1093
6	0x0102

LC4 Memory Layout

- ❖ The address space in LC4 is split into separate pieces for code and for data
- ❖ Separate regions for OS and User as well (more on OS in ~2 weeks)
- ❖ More to memory than this, but that will be discussed later



LC4 ASM Directives

- ❖ We can include directives to indicate where things in our ASM program should be loaded into memory
 - **.CODE**
 - Next instructions are in the CODE space
 - **.DATA**
 - Next values are in the DATA space
 - **.ADDR**
 - Set the current address to the specified value

- ❖ Other directives exist, more on those later

LC4 Example .asm file

```

;; Multiplication program
;; C = A*B
;; R0 = A, R1 = B, R2 = C
    .CODE           ; This is a code segment
    .ADDR 0x0000    ; Start filling in instructions at
                    ; address 0x00

CONST R2, #0      ; Initialize C to 0
LOOP
CMPI R1, #0       ; Compare B to 0

BRnz END          ; if (B <= 0) Branch to the end

ADD R2, R2, R0    ; C = C + A

ADD R1, R1, #-1   ; B = B - 1
BRnzp LOOP        ; Go back to the beginning of the loop
END

```

Directives to indicate this code starts at address 0 in memory

LC4 ASM Files

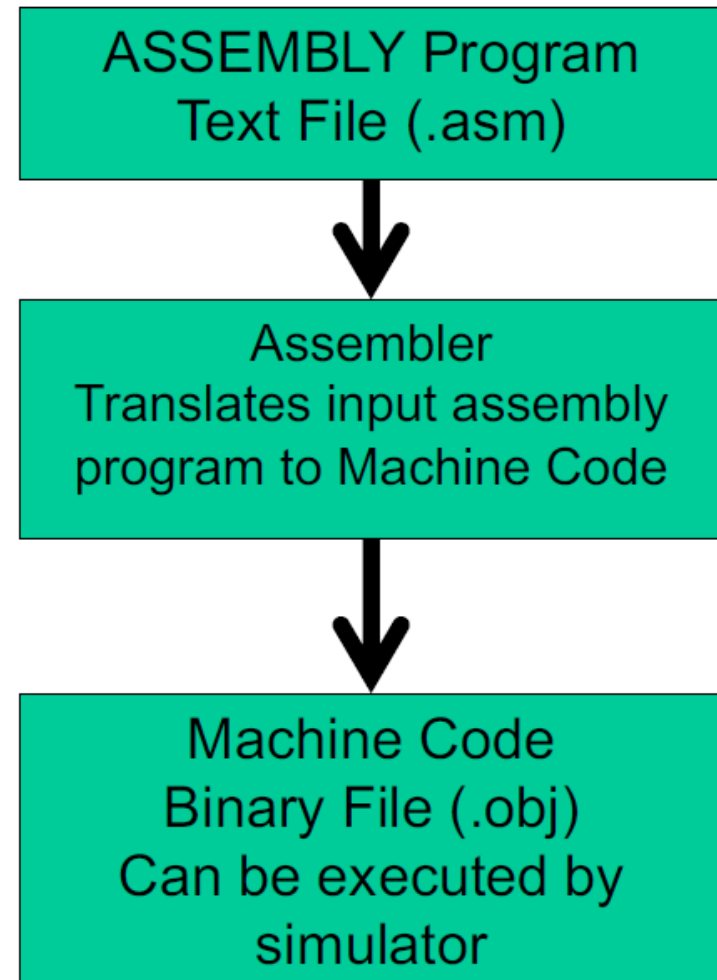
- ❖ LC4 Assembly Files are text files that contain a lot of conveniences for LC4 programmers
 - Instructions written as text (not as 16-bit patterns)
 - Comments
 - Initial values of some memory locations
 - Labels to refer to addresses and values
 - Directives
 - Pseudo Instructions
- ❖ ASM files are not directly run on the LC4 processor, this file needs to be processed into something machine-readable

Text Files

- ❖ .asm files and source code files for most languages are text files:
 - They can only contain ascii (or sometimes Unicode) characters.
 - Are not directly executed by the computer
- ❖ Text files are different from .docx and .pdf files
 - You cannot write text files in Microsoft Word
- ❖ Text files are created and edited by a text editor
 - Vim, Emacs, Notepad, Notepad++, Nano, Sublime, Atom, etc.

ASM -> OBJ Process

- ❖ ASM Files need to be processed by an assembler to become machine code
- ❖ Machine code can be executed directly by the computer hardware. (or a simulator in our case)
- ❖ Demo: multiply.obj vs multiply.asm



PennSim

- ❖ A Java program written to:
 - Convert LC4 Assembly to machine code
 - Simulate the operations of LC4 ISA
 - Provide debugging tools for LC4 ISA

- ❖ PennSim Demo
 - (See the lecture recording)

PennSim Commands Pt. 1

- ❖ PennSim has a command line at the top that you can type commands into
 - **as <outfile> <infile>**
 - Assembles a file from an assembly file into an object file .asm -> .obj
 - **ld <filename>**
 - Load an object file into memory
 - **set <register> <value>**
 - Loads a specific value into a register, works for special registers too (PC and PSR)
 - **help <command>**
 - Gives some helpful information on the specified command
 - **reset**
 - Resets the state of the simulator – clears memory and registers

PennSim Commands Pt. 2

- **clear**
 - Clears the output window
- **break <set | clear> label**
 - Set or clear a break point at the specific label
- **step**
 - Simulate the execution of the next instruction
- **continue**
 - Continue the simulation until a breakpoint or fatal error is encountered
- **script <filename>**
 - You can put a sequence of commands in a text file and then run them all at once using this command. Convenient and required for HW04.

Lecture Outline

- ❖ ASM Files, Object Files, PennSim Demo
- ❖ **LC4 Program Design (if/while/for)**
- ❖ LC4 Memory

LC4 Review: If & Loops in LC4

- ❖ Not all programming constructs have direct LC4 instructions

- ❖ How would we implement

```
if (R0 >= 3)
    R1 = R0;
```

```
START
    CMPI R0, #3
    BRn AFTER_IF
    ADD R1, R0, #0
AFTER_IF
    ; ...
```

LC4 Review: If & Loops in LC4

- ❖ Not all programming constructs have direct LC4 instructions
- ❖ How would we implement

```
if (R0 != R2) {  
    R1 = R2;  
} else {  
    R1 = 0;  
}
```

```
START  
    CMP R0, R2  
    BRz ELSE  
    ADD R1, R2, #0  
    JMP AFTER  
ELSE  CONST R1, #0  
AFTER  
    ; ...
```

LC4 Review: If & Loops in LC4

- ❖ Not all programming constructs have direct LC4 instructions

- ❖ How would we implement

```
while (R0 != 2) {  
    // ...  
}
```

```
START_LOOP  
    CMPI R0, #2  
    BRz AFTER_LOOP  
    ; ...  
    JMP START_LOOP  
AFTER_LOOP  
    ; ...
```

LC4 Review: If & Loops in LC4

- ❖ Not all programming constructs have direct LC4 instructions

- ❖ How would we implement

```
for (R0 = 0; R0 < R6; R0++) {  
    // ...  
}
```

```
        CONST R0, #0  
START_LOOP  
        CMP R0, R6  
        BRzp AFTER_LOOP  
        ; ...  
        ADD R0, R0, #1  
        JMP START_LOOP  
AFTER_LOOP  
        ; ...
```

Note On Labels

- ❖ When you are writing LC4 assembly, the labels you use must be unique.
 - If you use the same label more than once, the assembler will not know which location you are referring to with `JMP <LABEL>`

- ❖ To avoid name conflicts, it is common to number the labels or give more specific names.

- ❖ Instead of just using **LOOP**
 - **LOOP_1**
 - **LOOP_2**
 - **SUM_NUM_LOOP**
 - etc.

Assembly Programming Strategy

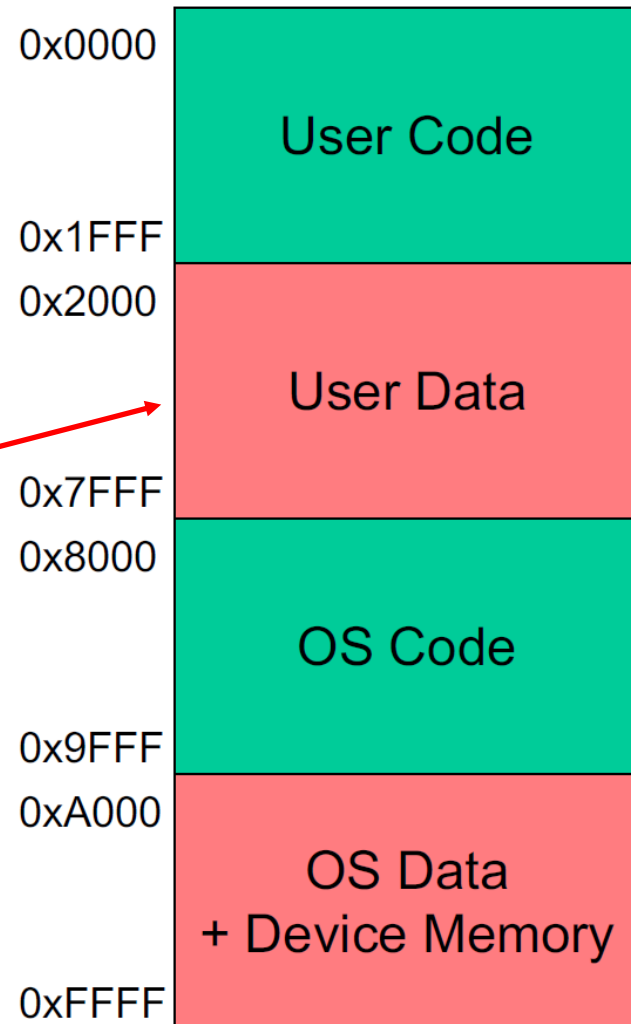
- ❖ One approach
 - Start by writing a pseudo code program
 - Try to keep code “simple”
 - don’t overuse variables, avoid recursion, etc
 - Comment while you do this
 - Translate each variable to a register
 - Translate each line/piece of code to Assembly
 - Test your assembly to make sure it works

Lecture Outline

- ❖ ASM Files, Object Files, PennSim Demo
- ❖ LC4 Program Design (if/while/for)
- ❖ **LC4 Memory**

Data Memory

- ❖ In LC4 we have 8 general purpose registers.
- ❖ Most programs need more than 8 variables
- ❖ User data is a portion of memory where we can store data that can't currently be in a register



Memory vs Registers

❖ Registers

- Quick access storage
- Can easily directly modify values in registers

❖ Memory:

- Memory can take longer to read/write
- Read/write memory requires its own instructions
- Need to read data from memory into a register before we can operate on it
- After data is updated, it needs to be stored back in memory for memory to be updated.

Pointers

- ❖ A **Pointer** is a variable (register in our case) that contains the address of a memory location.
 - We can use this pointer to read/write to that memory location
 - We can modify the address stored in the pointer through arithmetic to get a new address
- ❖ **Dereferencing** a pointer is when we take the address stored in the pointer and access that memory location for a read/write
- ❖ Pointers will be important for pretty much the rest of the course, especially for C programming.

LDR and STR

❖ LDR $Rd, Rs, IMM6$

- Action: $Rd = \text{memory}[Rs + \text{SEXT}(IMM6)]$
- Reads the data at address $Rs + \text{SEXT}(IMM6)$ in memory and loads it in Rd

❖ STR $Rt, Rs, IMM6$

- Action: $\text{memory}[Rs + \text{SEXT}(IMM6)] = Rt$
- Stores the value in Rt to the memory location at address $Rs + \text{SEXT}(IMM6)$

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What is we wanted to read a value stored at address 0x4020 in memory?

```

    → CONST    R0  x20
      HICONST  R0  x40
      LDR      R1, R0, #0
      ADD      R0, R0, #1
      LDR      R2, R0, #0
      LDR      R3, R0, #1
    
```

Set R0 to contain address 0x4020

Registers	Value
R0	--
R1	--
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What is we wanted to read a value stored at address 0x4020 in memory?

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
ADD      R0,   R0,   #1
LDR      R2,   R0,   #0
LDR      R3,   R0,   #1
  
```

Set R0 to contain address 0x4020

Registers	Value
R0	0x0020
R1	--
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What is we wanted to read a value stored at address 0x4020 in memory?

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
ADD      R0,   R0,   #1
LDR      R2,   R0,   #0
LDR      R3,   R0,   #1
    
```

Load the value at R0 + 0 into R1

R0 now "points to" the value stored at 0x4020

Registers	Value
R0	0x4020
R1	--
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What if we wanted to read a value stored at address 0x4020 in memory?

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
ADD      R0,   R0,   #1
LDR      R2,   R0,   #0
LDR      R3,   R0,   #1
  
```

R0 now “points to” the value stored at 0x4020

Registers	Value
R0	0x4020
R1	0xEF24
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What is we wanted to read a value stored at address 0x4020 in memory?

```
CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,  #0
ADD      R0,   R0,  #1
LDR      R2,   R0,  #0
LDR      R3,   R0,  #1
```

R0 is updated to “point to” the value stored at 0x4021

Registers	Value
R0	0x4021
R1	0xEF24
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What is we wanted to read a value stored at address 0x4020 in memory?

```
CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
ADD      R0,   R0,   #1
LDR      R2,   R0,   #0
LDR      R3,   R0,   #1
```

Registers	Value
R0	0x4021
R1	0xEF24
R2	0x9823
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

LDR Example

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ What is we wanted to read a value stored at address 0x4020 in memory?

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1, R0, #0
ADD      R0, R0, #1
LDR      R2, R0, #0
LDR      R3, R0, #1
    
```

→

Registers	Value
R0	0x4021
R1	0xEF24
R2	0x9823
R3	0x401E

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

```

→ CONST    R0 x20
   HCONST  R0 x40
   STR     R1, R0, #0
   ADD    R0, R0, #1
   ADD    R1, R1, #14
   STR    R1, R0, #0
   ADD    R1, R1, #9
   STR    R1, R0, #1
  
```

Set R0 to contain address 0x4020

Registers	Value
R0	--
R1	#24
R2	--
R3	--

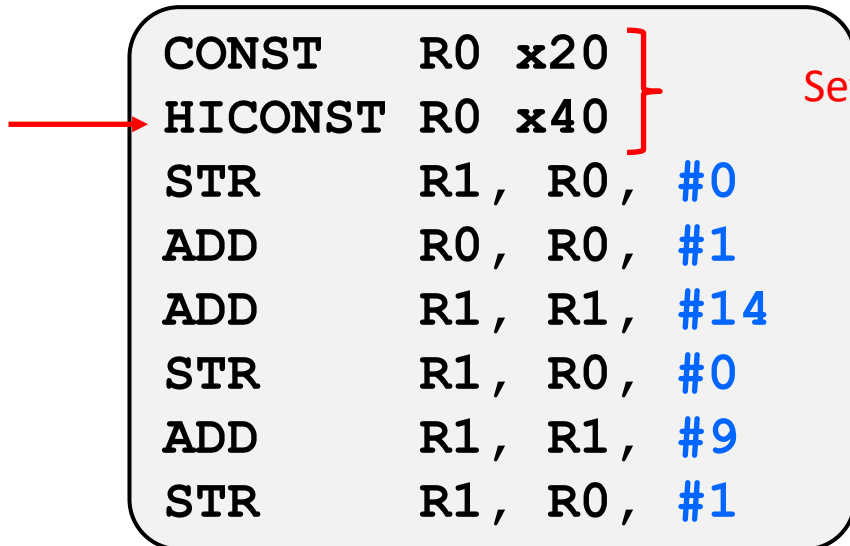
Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps



Registers	Value
R0	0x0020
R1	#24
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

```
CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
```

Registers	Value
R0	0x4020
R1	#24
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0xEF24
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

```

CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
  
```

Registers	Value
R0	0x4020
R1	#24
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0x0018
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

```
CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
```

Registers	Value
R0	0x4021
R1	#24
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0x0018
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

```
CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
```

Registers	Value
R0	0x4021
R1	#38
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0x0018
x4021	0x9823
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

```
CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
```

Registers	Value
R0	0x4021
R1	#38
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0x0018
x4021	0x0026
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- To store a value in memory, we follow similar steps

```

CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
  
```

Registers	Value
R0	0x4021
R1	#47
R2	--
R3	--

Memory


x401E	0xDEAD
x401F	0xF00D
x4020	0x0018
x4021	0x0026
x4022	0x401E
x4023	0x328F

STR

Red arrow is the Program Counter (PC) which points to the next instruction to execute

- ❖ To store a value in memory, we follow similar steps

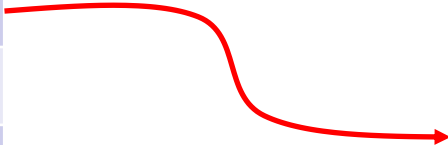
```
CONST    R0    x20
HICONST  R0    x40
STR      R1, R0, #0
ADD      R0, R0, #1
ADD      R1, R1, #14
STR      R1, R0, #0
ADD      R1, R1, #9
STR      R1, R0, #1
```



Registers	Value
R0	0x4021
R1	#47
R2	--
R3	--

Memory

x401E	0xDEAD
x401F	0xF00D
x4020	0x0018
x4021	0x0026
x4022	0x002F
x4023	0x328F



Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

- A. 110
- B. 210
- C. 240
- D. 320
- E. I'm not sure

```
CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
CONST    R2,   #110
ADD      R1,   R1,   R2
```

Registers	Value
R0	--
R1	--
R2	--

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333

Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

A. 110

B. 210

C. 240

D. 320

E. I'm not sure

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0, #0
CONST    R2,   #110
ADD      R1,   R1, R2
  
```

Registers	Value
R0	--
R1	--
R2	--

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333



Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

- A. 110
- B. 210
- C. 240
- D. 320
- E. I'm not sure

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
CONST    R2,   #110
ADD      R1,   R1,   R2
  
```

Registers	Value
R0	0x0020
R1	--
R2	--

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333



Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

- A. 110
- B. 210
- C. 240
- D. 320
- E. I'm not sure

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
CONST    R2,   #110
ADD      R1,   R1,   R2
  
```

Registers	Value
R0	0x4020
R1	--
R2	--

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333



Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

- A. 110
- B. 210
- C. 240
- D. 320
- E. I'm not sure

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
CONST    R2,   #110
ADD      R1,   R1,   R2
  
```

Registers	Value
R0	0x4020
R1	210
R2	--

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333



Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

- A. 110
- B. 210
- C. 240
- D. 320
- E. I'm not sure

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
CONST    R2,   #110
ADD      R1,   R1,   R2
  
```

Registers	Value
R0	0x4020
R1	210
R2	110

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333

Poll Everywhere

pollev.com/tqm

- ❖ What value is stored at address 0x4020 by the end of the program?

A. 110

B. 210

C. 240

D. 320

E. I'm not sure

```

CONST    R0    x20
HICONST  R0    x40
LDR      R1,   R0,   #0
CONST    R2,   #110
ADD      R1,   R1,   R2
  
```



Registers	Value
R0	0x4020
R1	320
R2	110

Memory

x401F	240
x4020	210
x4021	107
x4022	380
x4023	333

Value was not written back to memory with STR!
Memory is not changed

More LC4 Directives

- ❖ These assembly directives provide information on how various data/constants should be assembled
 - **.FALIGN**
 - Pad current address to the next multiple of 16. Useful since a subroutine must start on an address that is a multiple of 16
 - **.FILL IMM16**
 - Set value at the current address to the specified 16 bit value
 - **.BLKW UIMM16**
 - Reserve UIMM16 words at the current address
 - **.CONST IMM16**
 - Associate IMM16 with the preceding label
 - **.UCONST UIMM16**
 - Associate UIMM16 with the preceding label

LC4 Pseudo Instructions

- ❖ Pseudo instructions look like normal instructions, but the assembler translates them into real instructions.
 - Pseudo instructions act as convenient aliases to make code more readable for us
 - **RET**
 - Return from a subroutine
 - Actual implementation: **JMPR R7**
 - **LEA Rd, <LABEL>**
 - Load effective address associated with a label into a register
 - Actual implementation: a **CONST, HCONST** Pair
 - **LC Rd, <LABEL>**
 - Load a constant 16-bit value associate with a label via **.CONST** or **.UCONST** into a register
 - Actual implementation: a **CONST, HCONST** Pair

Arrays

- ❖ Arrays are a common programming structure that consists of a list of other data
 - (we will work with arrays of integers)
- ❖ Arrays in LC4
 - Since integers are 16-bits and registers are 16-bits, we can only fit one integer into a register.
 - We cannot “assign” a register to an array the same way we can do so for integers
 - Instead, store the array into data memory and have a register with the address of the beginning of the array and a register with the length of the array

sum_numbers.asm

- ❖ A sample program where we iterate over an array and get the sum of all integers in that array
 - Very useful to look at when working on HW04

- ❖ Demo

Strings & strlen.asm

- ❖ Strings can be thought of as similar to arrays, but as a sequence of characters.
- ❖ Instead of having both the address to the start and the length, we only have the address to the start
 - The end of the string is marked by having a null-terminator character (which is equal to 0) to mark the end of the string.
- ❖ strlen.asm
 - Demo that shows how to calculate the length of a string in LC4