CS061: Machine Organization & Assembly Language Lab 1

Agenda

1. Presentation:
    a. Intro to Assembly Language
    b. Basics of LC-3 Programming
    c. Setting up LC-3 Tools
    d. Lab Descriptions

2. Work Time / Questions / Demos

What is Assembly Language?

- **Assembly Language**: Human-readable representation of instructions that correspond to CPU instructions.

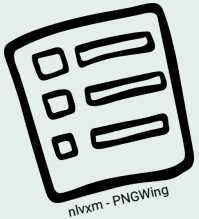
- Formal Definitions:
  - Oxford: “A low-level symbolic code converted by an assembler.”
  - Merriam Webster: “a programming language that consists of instructions that are mnemonic codes for corresponding machine language instructions.”

Let's Back Up A Little

- **CPU** (central processing unit): Responsible for executing a set of instructions that change the state of the system.

- CPU only understands limited set of instructions.
  - Think of it like a dog!
  - Dogs can't understand full human conversation, but understand commands like "sit" or "stay".
  - CPU can't understand a raw C++ program. It only understands "add x to y" or "set value at memory address x to value y", etc.

- Assembly language is the readable representation of the raw instructions the CPU understands.

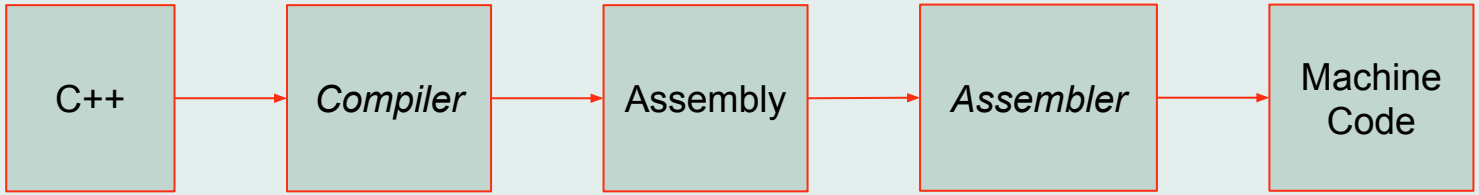


Assembler

- Computers can only understand 1s and 0s.
- **Assembler:** Converts assembly language (human-readable representation) to machine code (1s and 0s).

Running Hello World

- How do we get from a full C++ program to something that the CPU understands?
- Compiler takes C++ source code and converts it into assembly language *(*if using gcc)*.
- Then an assembler converts the assembly language to machine code!



But...Why?

- Why learn how to read/write assembly language?
- Careers:
  - Computer/Software Security
  - Reverse Engineering
  - Embedded Systems Developers
  - Compiler Writer

- Understanding how computers work under the hood.
  - OS Development
  - High Performance Computing
  - Software Testing - Static Analysis
  - Debugging

LC-3

- **LC-3** (Little Computing 3): Educational assembly language.
  - Easier to learn/understand than actual assembly language!

- **ISA** (Instruction-Set Architecture):
  - **15 instructions**.
  - Basic things like addition, bitwise NOT, bitwise AND, save value at memory, load value from memory, etc.
  - Still missing some basic things: no subtraction instruction or multiply instruction.

- **8 registers (16-bits)**
  - Registers are like global integer variables.
  - Can store/read some value at them.
  - *R0 - R7*

- LC-3 program consists of pseudo-ops, instructions, and data.

Show me code!!!

```
;---------------------------
; Name: Doe, Sam
; Email: sdoe013@ucr.edu
;
; Lab: Lab 1, Ex 1
; Section: 021
; TA: Sanchit Goel, Westin Montano
;
; ----------------------------

.ORIG x3000
; Program Code Here

AND R1, R1, #0

LD R1, DEC_6

DO_WHILE_LOOP
    ADD R1, R1, #-1
    BRp DO_WHILE_LOOP
END_DO_WHILE_LOOP


HALT
; Local Data
DEC_6   .FILL #6

.END
```

What does it mean?

```
;---------------------------
; Name: Doe, Sam
; Email: sdoe013@ucr.edu
;
; Lab: Lab 1, Ex 1
; Section: 021
; TA: Sanchit Goel, Westin Montano
;
; ----------------------------
```

- Comments start with ';'
- Header section - Put this in all your exercises!

```
.ORIG x3000
; Program Code Here

AND R1, R1, #0

LD R1, DEC_6

DO_WHILE_LOOP
    ADD R1, R1, #-1
    BRp DO_WHILE_LOOP
END_DO_WHILE_LOOP
```

- **Pseudo-Op**: Not a CPU instruction, but an assembler directive.
  - Executes at assembly time!
- All pseudo-ops start with "."
- ".ORIG" tells the assembler to put code/data starting at that memory address (x3000)

```
HALT
; Local Data
DEC_6   .FILL #6

.END
```

- Instruction that indicates the end of a program!

How do we math?

```
; Program Code Here
   Destination  Source  Operand
AND R1, R1, #0

LD R1, DEC_6

DO_WHILE_LOOP
    ADD R1, R1, #-1
    BRp DO_WHILE_LOOP
END_DO_WHILE_LOOP
```

- General Instruction Format:
- *Instruction, Destination Register, Source Register, Source Register 2 / Operand*

- AND instruction performs bitwise AND between source register (R1) and operand (#0) then stores it in destination register (R1).
- R1 <- R1 & 0 (*Register Transfer Notation*)
- Sets R1 to 0

- Subtracts R1 by 1.
- ADD adds source register (R1) with operand (#-1) and stores value in destination register (R1).
- R1 <- R1 - 1

Code is Memory

```
.ORIG x3000
; Program Code Here

x3000  AND R1, R1, #0

x3001  LD R1, DEC_6

       DO_WHILE_LOOP
x3002      ADD R1, R1, #-1
x3003      BRp DO_WHILE_LOOP
       END_DO_WHILE_LOOP


x3004  HALT
       ; Local Data
x3005  DEC_6   .FILL #6
```

- Think of memory as a contiguous block of values with an address representing location of value.
- Each instruction takes up one memory space.
- For reference, every memory space holds a 16-bit value.

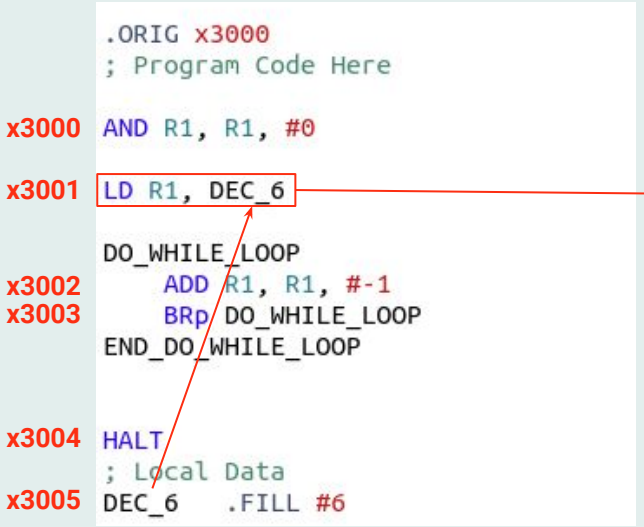
- **Labels**: Alias for a memory address!
  - DO_WHILE_LOOP refers to the address x3002 which corresponds to the ADD instruction.
  - END_DO_WHILE_LOOP refers to the address x3004.

- DEC_6 refers to the address at x3005.
- The .FILL pseudo-op sets the value at address x3005 to 6.

Code is Memory

Memory Address	Value (Hex)	Value (Decimal)	Instruction/Pseudo-Op
x3000	x5260	21088	*AND R1, R1, #0*
x3001	x2203	8707	*LD R1, DEC_6*
x3002	x127F	4735	*ADD R1, R1, #-1*
x3003	x03FE	1022	*BRp DO_WHILE_LOOP*
x3004	xF025	-4059	*HALT*
x3005	x0006	6	*DEC_6 .FILL #6*

Movin' Values



- Can we put a value at a memory address into a register, and vice-versa?
- LD (**Load Direct**) takes a value from memory and puts in the destination register (R1).
  - Operand (DEC_6) is a label.
  - DEC_6 refers to address x3005.
  - Value at x3005 is 6.
  - R1 <- Mem[DEC_6]
  - R1 will have 6 after instruction executes.
- 3 different types of loading/storing in LC-3.

On One Condition

```
        .ORIG x3000
        ; Program Code Here

x3000   AND R1, R1, #0

x3001   LD R1, DEC_6

        DO_WHILE_LOOP
x3002       ADD R1, R1, #-1
x3003       BRp DO_WHILE_LOOP
        END_DO_WHILE_LOOP


x3004   HALT
        ; Local Data
x3005   DEC_6   .FILL #6
```

- Sometimes we need to repeat a section of code (loops) or only run a section of code if something happens (conditions).
- BR - **Branch** jumps to a line of code based on a condition.
  - Condition specified by p (positive), n (negative), z (zero).
  - Condition set by any instruction that modifies registers.
  - The line to jump to is determined by the label operand.
- BRp will check if result of addition above is > 0.
  - If it is, jumps to address corresponding to DO_WHILE_LOOP (x3002).
  - If not, continues to next line.

From the Top

```
.ORIG x3000
; Program Code Here

AND R1, R1, #0

LD R1, DEC_6

DO_WHILE_LOOP
    ADD R1, R1, #-1
    BRp DO_WHILE_LOOP
END_DO_WHILE_LOOP


HALT
; Local Data
DEC_6    .FILL #6

.END
```

R1 <- R1 & 0 = Set R1 to 0

R1 <- Mem[DEC_6] = Load R1 with 6

R1 <- R1 - 1 = Subtract R1 by 1
Branch = Jump to DO_WHILE_LOOP if R1 > 0

Repeat 5 times (R1 = 6, then R1 = 5, …R1 = 0)

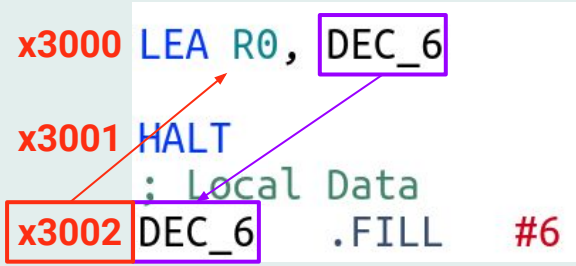
End Program

This program does nothing meaningful 😛

Load Effective Address (LEA)

- Sometimes need the memory address of a label in a register.
  - E.g. loading/storing, PUTS, etc

- LEA Rx, LABEL
  - Gets the memory address of the label and stores it in Rx.
  - E.g. LEA R0, DEC 6 : Gets the memory address of label DEC_6 (x3002) and stores it in R0.
    - R0 <- DEC_6.
    - R0 <- x3002



Stringz

- “.**STRINGZ**”
  - *Pseudo-op* to tell assembler to create a string starting at a memory address.
  - Each character in the string *occupies 1 memory space*.
  - End of a string is denoted by 0 (\0 - the null character).

- “**PUTS**” (Print)
  - MSG address is the start of a string.
  - PUTS prints out the string *starting* at the address specified by R0.
  - PUTS will *always* use R0 as the starting address.

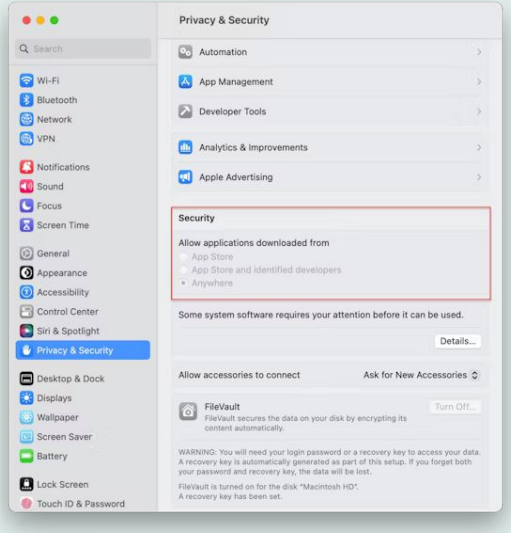
```
x3000 LEA R0, MSG
x3001 PUTS

x3002 HALT
      ; Local Data
x3003 MSG .STRINGZ     "WOW!"
```

Address	x3003	x3004	x3005	x3006	x3007
Value	‘W’	‘O’	‘W’	‘!’	0

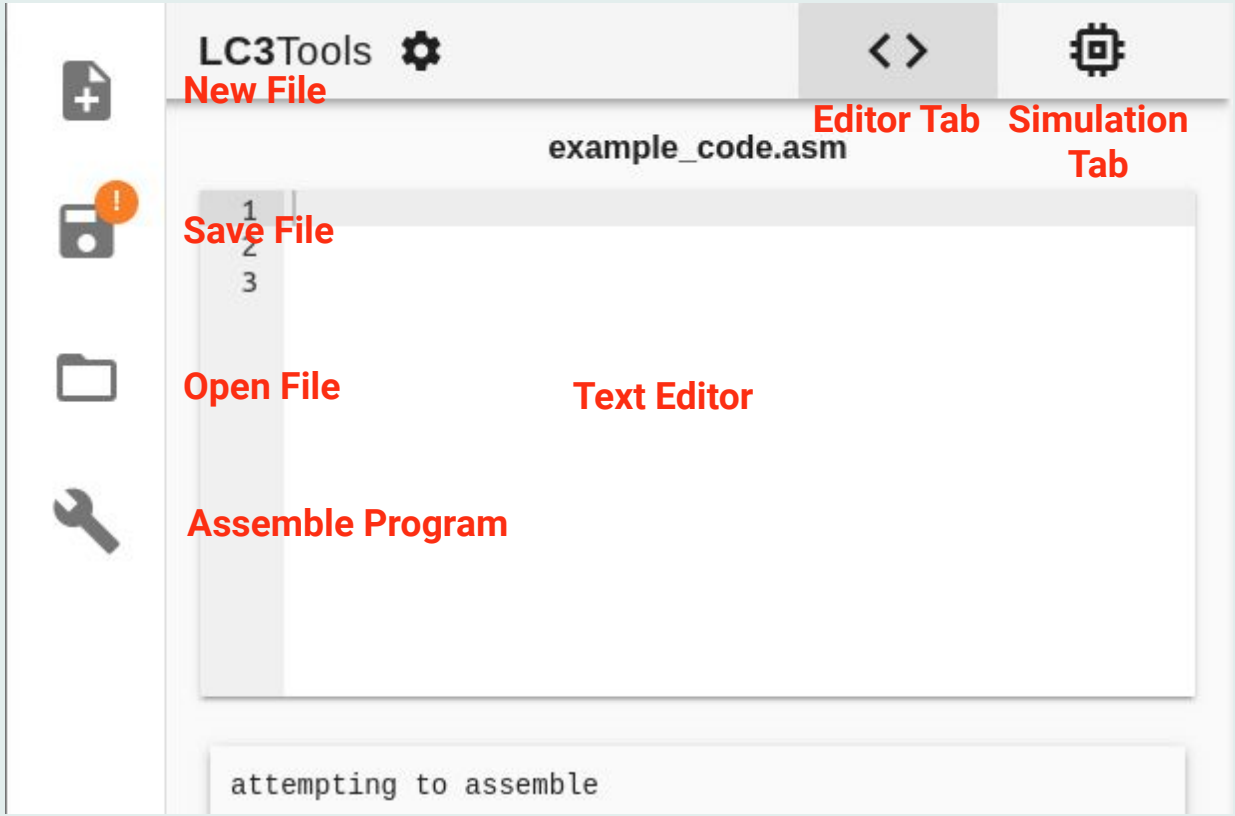
Your Turn!

- Download LC3 Tools from https://github.com/chiragsakhuja/lc3tools/releases/tag/v2.0.2
  - The Lab 1 manual has the link.
  - Download LC3Tools-2.0.2.exe for Windows
  - Download LC3Tools-2.0.2.dmg for Mac
  - Download LC3Tools-2.0.2.AppImage for Linux

- On Mac, make sure to allow opening apps from anywhere.
  - To enable this, go to System Settings -> Privacy & Security -> Security.
  - Select the Anywhere option.



Using LC-3 Tools



Using LC-3 Simulation

Play

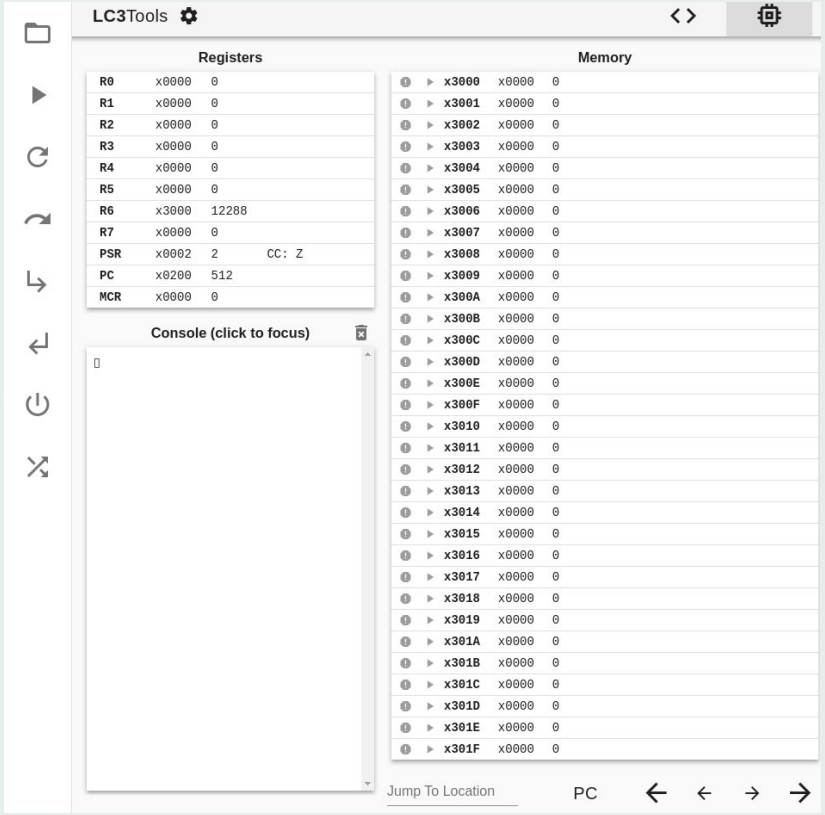
Reload Object File

Step Over

Step In

Step Over

Reset



Exercise 0

- Print “Hello World” to the Console (in the Simulation Tab).

- “**LEA**” (Load Effective Address)
  - LEA R0, MSG_TO_PRINT
  - Remember: label is alias for memory address.
  - LEA puts the memory address that the label (MSG_TO_PRINT) refers to into destination register (R0).
  - E.g. suppose MSG_TO_PRINT referred to address x3006 then after instruction runs, R0 = x3006.

- “.**STRINGZ**”
  - Pseudo-op to tell assembler to create a string starting at a memory address.
  - Each character in the string occupies 1 memory space.
  - End of a string is denoted by 0 (\0 - the null character).

- “**PUTS**” (Print)
  - MSG_TO_PRINT address is the start of a string.
  - PUTS prints out the string starting at the address specified by R0.
  - PUTS will *always* use R0 as the starting address.

Exercise 1

- Create a program that multiplies R1 by R2.
- Code given to you.
- Refer to previous slides to see how the assembly instructions work.

Demos

- When you complete all exercises in a lab, you must demo the lab to a TA or grader to receive credit.
- TAs & Graders will ask questions about your code and how it works.
- If you do not answer a question, you can always come back and re-demo (without penalty).
- Must demo before the next lab session to receive full credit for the lab!
  - May demo during any TA or Grader's office hours (TBA) but not the professor's.
  - **If demo before Friday, then will receive 1 extra point for the lab.**
  - If demo previous lab during the next lab session, there will be 3 point penalty.
  - 7 for demoing, 3 points for attendance = 10 points for full credit (11 is extra credit).

- To sign up for a demo, use the demo spreadsheet link that will be provided.
  - Net ID is the first part of your email. e.g. sdoe03 is the net ID if sdoe03@ucr.edu is the email.
  - Demos are usually taken in order.
  - **Please only sign up once you have completed all the exercises and fully understand your code.**