# CS061:
# Machine Organization & Assembly Language
# Lab 4

# Agenda

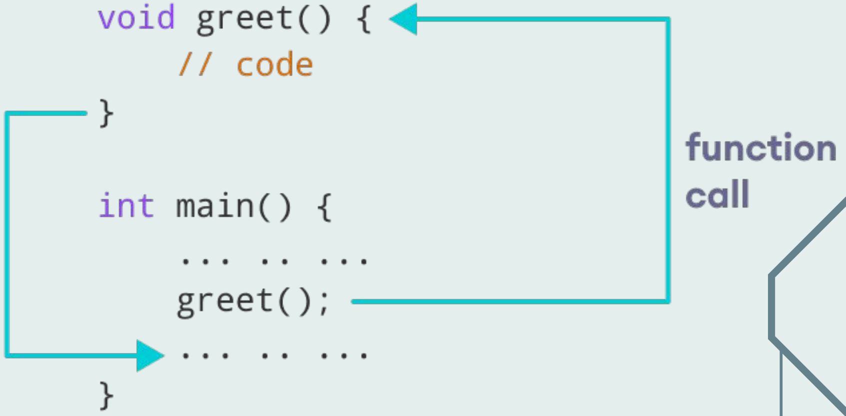
1. Presentation:
    a. Subroutines: Isolation
    b. Subroutines: Control Flow
    c. Subroutines: Parameters / Return Values
    d. Subroutines: Subroutine Data
    e. Subroutines: Headers
    f. Subroutines: Live Demo
    a. Lab Description

2. Work Time / Questions / Demos

# Sub-routine'd

- Sub-routines are similar to functions in C++ (and other programming languages).
  - Isolate code for re-use.
  - Takes in parameters.
  - Returns values.

- Similar execution flow:
  - Main program will make a call to a sub-routine.
  - Code in the sub-routine will execute.
  - Main program will resume.




https://www.programiz.com/cpp-programming/function

# Isolation

- Subroutines are usually placed x200 memory addresses apart.

- Why isolate sub-routines?

- Organizational:
  - Looks cleaner.
  - No messy intertwining with main program or other code.

- Sub-routines at fixed locations easier to access.

- Allow subroutine's code and data to expand without worrying about conflicting with other code / data

```
.ORIG x3000
; Main Program

HALT
; Local Data

.END

.ORIG x3200
; Function Code
.END
```

# Control Flow

- How can the program jump to code far away (e.g. in x3200)?
  - Does branch (BR) work?

- Suppose we run code at x3200, how to get back to main program at x3000?
  - Could we use labels to jump to and from?

```
.ORIG x3000
; Main Program

HALT
; Local Data

.END

.ORIG x3200
; Function Code
.END
```

# Control Flow

- What if we could jump to an address specified by a register?
  - Mashup between LDR and BR.

- **JMP** Rx: Jumps to address specified in register Rx.
  - Not actually used!

- Solves the problem of how to get there, but not how to get back?
  - Can execute stuff at x3200, but can't get back to the main program.

```
.ORIG x3000
; Main Program
; R5 = x3200
LD R5, FUNC_PTR
; Go to x3200
JMP R5

HALT
; Local Data
FUNC_PTR .FILL x3200
.END

.ORIG x3200
; Function Code
.END
```

# Control Flow

```
        ; Main Program
        ; R5 = x3200
x3000   LD R5, FUNC_PTR

                  Imaginary
x3001   ADD R7, PC, #0
x3002   ADD R7, R7, #3
        ; Go to x3200
x3003   JMP R5

x3004   ADD R1, R1, #0
```

- Imagine if we could access a register keeping track of the current line we're at in the program.
    - PC = Program Counter
    - Memory address of current instruction.

- PC = x3001
- R7 = PC = x3001

- R7 <- R7 + 3 = x3001 + 3
    - R7 = x3004

# Control Flow

```
.ORIG x3200
; Subroutine Code
; ...
; Go back to
; main program
JMP R7
.END
```

- Suppose executing the subroutine.
- R7 is still x3004.
  - x3004 is back at the main program.
- When finish executing subroutine, can just jump back to address in R7.

# Control Flow - Revealed

```
      .ORIG x3000
      ; Main Program
      ; R5 = x3200
x3000 LD R5, FUNC_PTR
x3001 JSRR R5

x3002 ADD R1, R1, #0
```

- **There is no way to access the PC register directly!**

**JSRR Rx:** Jump to subroutine specified by register Rx.

- 1. Stores address of *next* instruction (e.g. x3002) into R7.
- 2. Jumps to address specified in R5.

# Control Flow - Revealed

```
.ORIG x3200
; Subroutine Code
; ...
; Go back to
; main program
RET
.END
```

**RET:** Jumps back to the address specified in R7.

- Alias for: JMP R7
- E.g. R7 = x3002

# Basic Subroutine

```
.ORIG x3000
; Main Program
LD R5, SUB_PTR
JSRR R5
; Code after
; subroutine.
HALT
; Local Data
SUB_PTR .FILL x3200
.END
.ORIG x3200
; Subroutine Code
; ...
; Go back to
; main program
RET
.END
```

- Load subroutine address in register.
- Call subroutine.
- Jumps back to main program (address in R7)

# Parameters/Return Values

- **Parameters** are registers set *before* calling the subroutine for use *in* the subroutine.
- **Return Values** are registers set *in* the subroutine for use *after* the subroutine finishes.

```
;--------------------------------------------------------
; Subroutine: SUB_ADD_10_3200
; Parameter (R3): Value to add 10 to.
; Postcondition: 10 is added to R3.
; Return Value (R3): R3 had 10 added to it.
;--------------------------------------------------------
```

```
.ORIG x3000

; Define parameters here
; R1 = 5, R3 = 10

LD R5, SUB_ADDR
JSRR R5

; Use return values

; HALT, .END, etc
; ...

.ORIG x3200
; Use parameters

; Set return values
RET
.END
```

# Parameters/Return Values

```
; Main program
; R3 = 5
ADD R3, R3, #5

LD R5, SUB_ADDR
JSRR R5

ADD R3, R3, #2
```

```
.ORIG x3200
; Subroutine Code
ADD R3, R3, #10
RET
.END
```

- Registers are like global variables!
    - Persist throughout the entire program!
- R3 = 5 in the main program.
- R3 is still 15!
- R3 is now 17!
- R3 is still 5!
- R3 is now 15!

# Subroutine Data

- Subroutines can have their own data similar to local data in the main program.
  - Labels defined in the subroutine data must be unique for the ENTIRE program.
  - E.g. Can’t have 2 labels called “DEC_65” in the source file.

- *TIP:* Append the address of the subroutine to the subroutine data labels.
  - E.g. instead of just OFFSET, call it OFFSET_3200

```
.ORIG x3200
; Subroutine Code
; Assume R0 has 5 in it
LD R1, OFFSET_3200
; Add 48 to R0
ADD R0, R0, R1
; Print out '5'
OUT

RET
; Subroutine Data
OFFSET_3200  .FILL   #48
.END
```

# Subroutine Header

1. Subroutine Name: Starts with “SUB”, ends with address of subroutine.
2. Parameters: Registers used as arguments for the subroutine and description.
   a. E.g. (R1) Address of Array
3. Postcondition: What the subroutine does.
4. Return Value: Registers that hold return values.
   a. E.g. (R3) Elements in array.

```
;=============================================================
; Subroutine: SUB_intelligent_name_goes_here_3200
; Parameter: (Register you are “passing in”): [description of parameter]
; Postcondition: [a short description of what the subroutine accomplishes]
; Return Value: [which register (if any) has a return value and what it means]
;=============================================================

.orig x3200       ; use the starting address as part of the sub name
```

# Subroutine Live Demo

- Writing a subroutine that multiplies a value by 2.
- Using the simulator with subroutines.

```
;------------------------------------------------------------------------
; Subroutine: SUB_MULT_2_3200
; Parameter (R1): Value to multiply by 2.
; Postcondition: Multiply the value in R1 by 2 and store it in R3.
; Return Value (R3): R3 had 10 added to it.
;------------------------------------------------------------------------
```

# Exercise 1

- Create an array of size 10 in local data (the data of the main program).
- Create a new subroutine called “SUB_FILL_ARRAY_3200”
  - R1 should be a parameter with the address of the array from local data.
  - Programmatically populate the array with 0 through 9 (decimal numbers, not ASCII).
  - At the end of the subroutine, revert R1 to the original address of the array!
- In the main program, call the subroutine you created!

```
;-------------------------------------------------------------------------
; Subroutine: SUB_FILL_ARRAY
; Parameter (R1): The starting address of the array. This should be
unchanged at the end of the subroutine!
; Postcondition: The array has values from 0 through 9.
; Return Value (None)
;--------------------------------------------------------------------------
```

# Exercise 2

- Copy exercise 1 into exercise 2 file.

- Create a new subroutine at x3400!
    - Subroutine should load each element from the array, convert it to an ASCII character, and store it back into the array.

- Call the subroutine in your main program (placed after the call to your previous subroutine)

```
;---------------------------------------------------------------------------
; Subroutine: SUB_CONVERT_ARRAY
; Parameter (R1): The starting address of the array. This should be
unchanged at the end of the subroutine!
; Postcondition: Each element (number) in the array should be represented as
a character. E.g. 0 -> ‘0’
; Return Value (None)
;---------------------------------------------------------------------------
```

# Exercise 3

- Copy the exercise 2 code into your exercise 3 file.
- Create a new subroutine at x3600 called “SUB_PRINT_ARRAY_3600”
    - Load each element from the array, and print it out using OUT.
- Call the subroutine in the main program after calls to your previous subroutine.
- Ensure that the program runs correctly such that 0 through 9 prints out to the console (e.g. “0123456789”).

```
;------------------------------------------------------------------------
; Subroutine: SUB_PRINT_ARRAY
; Parameter (R1): The starting address of the array. This should be unchanged at the
end of the subroutine!
; Postcondition: Each element (character) in the array is printed out to the
console.
; Return Value (None)
;-------------------------------------------------------------------------
```

# Exercise 4

- Copy the exercise 3 code into your exercise 4 file.

- Create a new subroutine at x3800 called “SUB_PRETTY_PRINT_ARRAY_3800”
  - Print out “=====” (5 equal signs) before and after printing out the array.
  - **Call your “SUB_PRINT_ARRAY” subroutine. Do not copy the print array logic!**
- Call the subroutine in the main program after calls to your previous subroutines.

- **This will not work properly!** Step through the program and answer the questions in the lab manual.

```
;-------------------------------------------------------------------------
; Subroutine: SUB_PRETTY_PRINT_ARRAY
; Parameter (R1): The starting address of the array. This should be unchanged
at the end of the subroutine!
; Postcondition: Prints out “=====” (5 equal signs), prints out the array,
and after prints out “=====” again.
; Return Value (None)
;--------------------------------------------------------------------------
```

# Demo Info

- **Lab Grade Breakdown:**
  - 3 points for attendance.
  - 7 points for demoing (+1 bonus point demo'd before/during Friday).
  - 3 point penalty if lab is demo'd during the next lab session.

- **Tips before you demo:**
  - ***Understand your code!*** (Know what each line does & the input/output)
  - ***Test your code!*** (Check for correct output and that there are no errors)