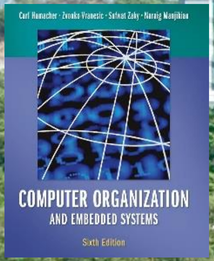
香港中文大學
The Chinese University of Hong Kong

CSCI2510 Computer Organization

Lecture 05: Program Execution

Ming-Chang YANG

mcyang@cse.cuhk.edu.hk



Reading: Chap. 2.3~2.7, 2.10, 4

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Recall: Language Translation

High-level Language

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
TEMP = V(k);
V(k) = V(k+1);
V(k+1) = TEMP;
```

C/Java Compiler

Fortran Compiler

Assembly Language

`lw:` loads a word from **memory** into a register

`sw:` saves a word from a register into **RAM**

`0($2):` treats the value of register $2 + 0 bytes as a location

`4($2):` treats the value of register $2 + 4 bytes as a location

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
Sw $t0, 4($2)
```

MIPS Assembler

Machine Language

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

https://gerardnico.com/code/lang/machine
https://clip2art.com/explore/Boy%20clipart%20teacher/

Assembly Language

- Machine instructions are represented by 0s and 1s.

→ Such patterns are *awkward* to deal with by humans!

→ We use symbolic names to represent 0/1 patterns!

- **Assembly Language**: a complete set of such symbolic names and rules for their use constitutes a programming language
  - **Syntax**: the set of rules for using the *mnemonics* or *notations* and for specifying complete instructions/programs
  - **Mnemonics**: acronyms to represent instruction operations
    - E.g. Load → **`LD`**, Store → **`ST`**, Add → **`ADD`**, etc.
  - **Notations**: shorthand for registers or memory locations
    - E.g. register 3 → **R3**, a particular memory location → **LOC**

Assembly Language Syntax

- **Three-operand Instruction**:

  `operation dest, src1, src2`

  - E.g. "**`Add A, B, C`**" means "A ← [B] + [C]"
    - Note: We use [X] to represent the content at location X.

- **Two-operand Instruction**:

  `operation dest, src`

  - E.g. "**`Move A, B`**" means "A ← [B]"
  - E.g. "**`Add A, B`**" means "A ← [A] + [B]"
    - Note: Operand A is like both the source and the destination.

Some machines may put destination last:

`operation src, dest`

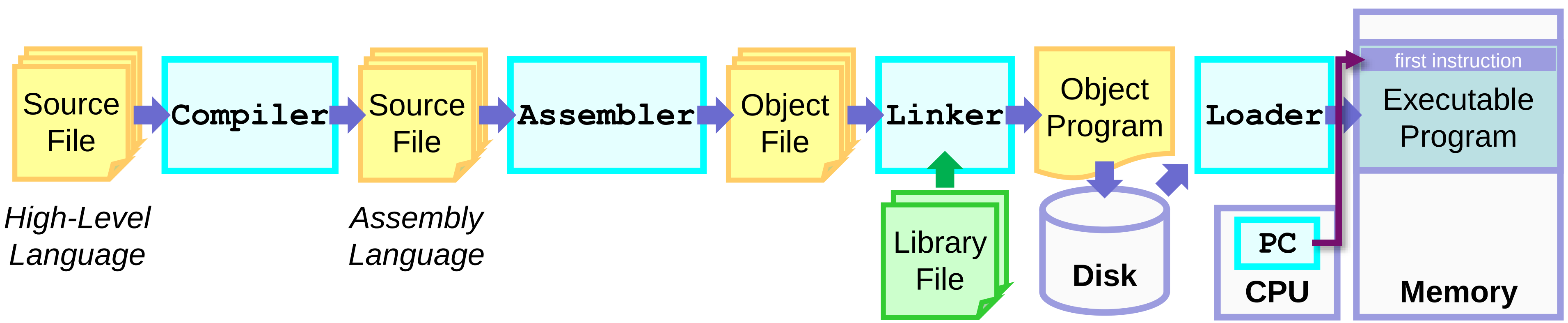
- **One-operand Instruction**:
  - Some machines have an register called accumulator (ACC)
    - E.g. "**`Add B`**" means "ACC ← ACC + [B]"
    - E.g. "**`Load B`**" means "ACC ← [B]"
    - E.g. "**`Store B`**" means "B ← ACC"

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Generating/Executing an Program

- **Compiler**: Translate a high-level language source programs into assembly language source programs
- **Assembler**: Translate assembly language source programs into object files of machine instructions
- **Linker**: Combine the contents of object files and library files into one object/executable program
  - **Library File**: Collect useful subroutines of application programs
- **Loader**: Load the program from disk into memory & load the address of the first instruction into program counter



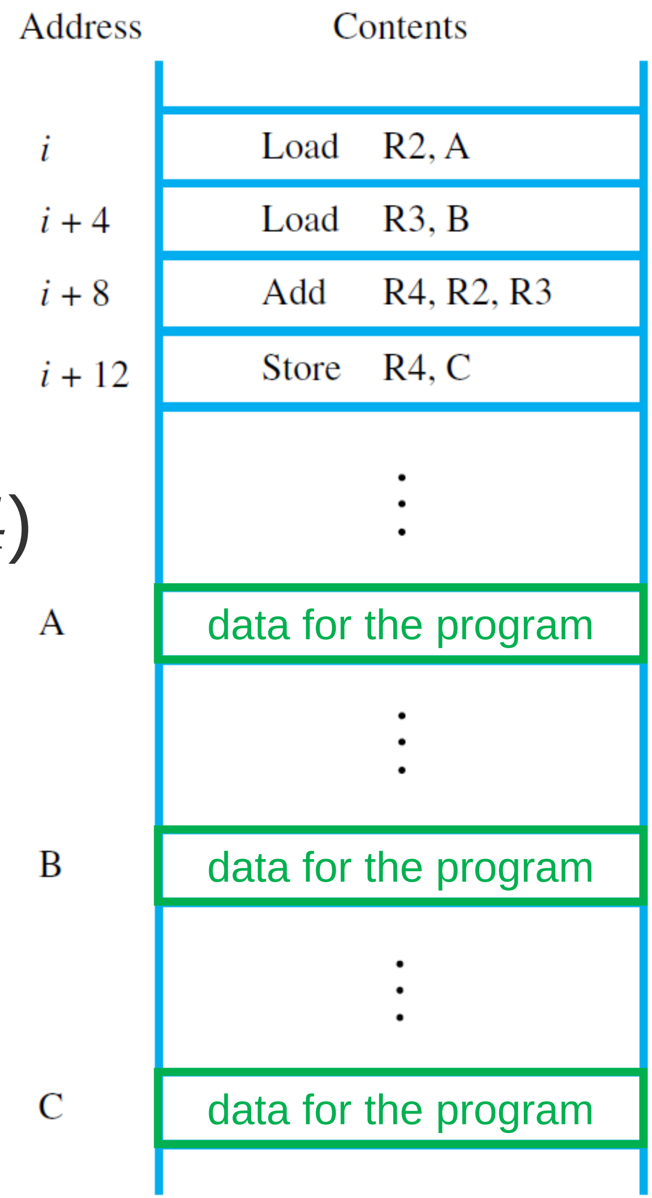
Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Instruction Execution & Sequencing (1/3)

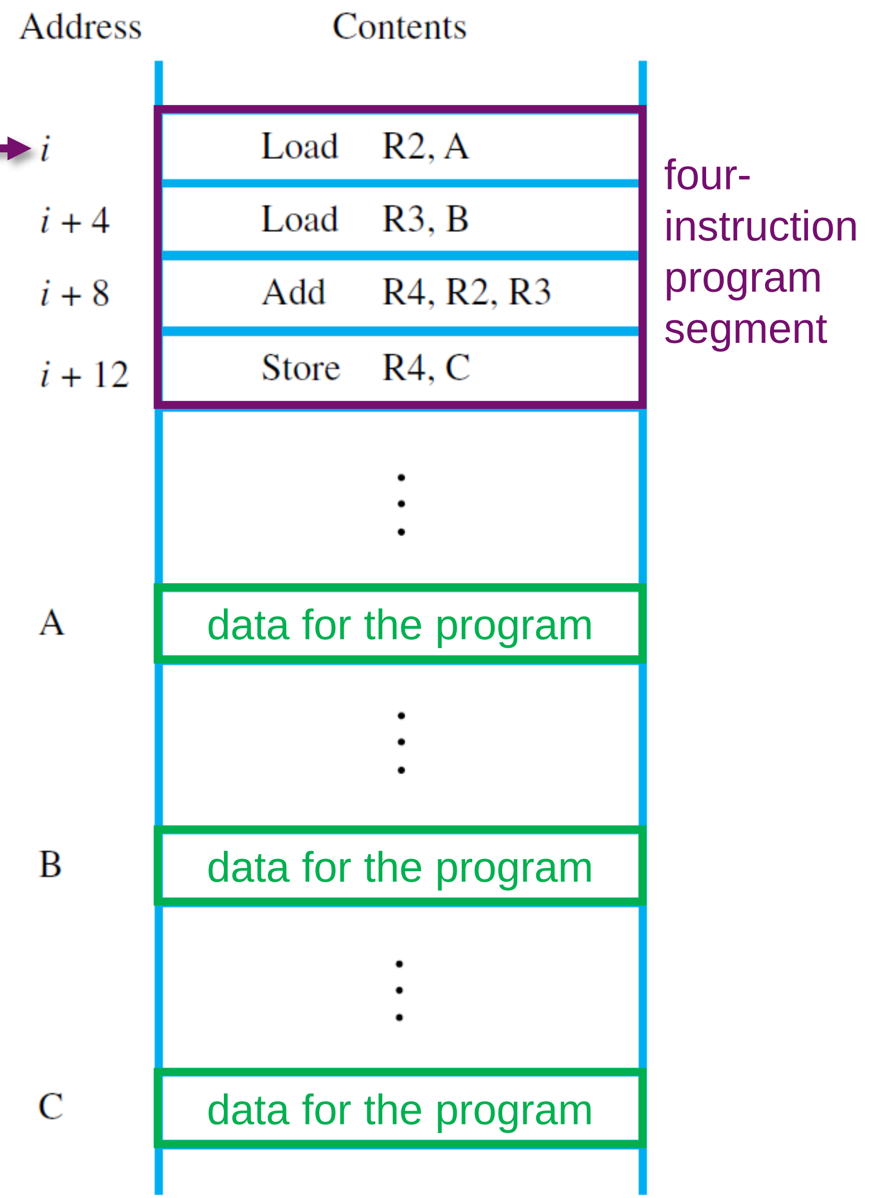
- Consider a machine:
  - RISC instruction set
  - 32-bit word, 32-bit instruction
  - Byte-addressable memory

- Given the task $C{=}A{+}B$ ($Lec04$)
  - Implemented as C ← [A] + [B]
  - Possible program segment:
    - **`Load R2, A`**
    - **`Load R3, B`**
    - **`Add R4, R2, R3`**
    - **`Store R4, C`**



Instruction Execution & Sequencing (2/3)

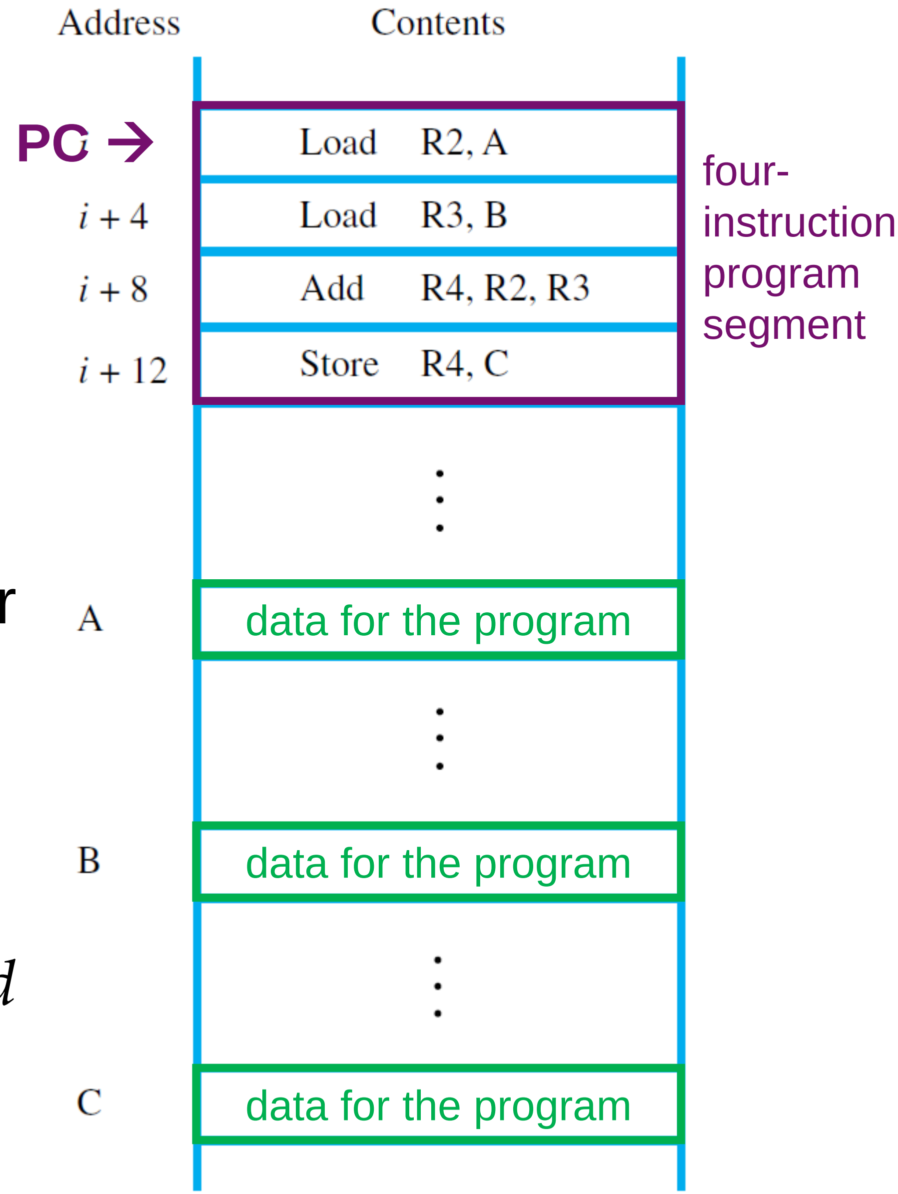
- Assume the 4 instructions are loaded in successive memory locations:
  - Starting at location i
  - The 2nd, 3rd, 4th instructions are at $i + 4$, $i + 8$, and $i + 12$
    - Each instruction is 4 bytes
- To execute this program
  - The program counter (PC) register in the processor should be loaded with the address of the 1st instruction.
    - **PC**: holds the address of *the next instruction* to be executed.



Instruction Execution & Sequencing (3/3)

- CPU fetch and execute instruction indicated by PC
  - **Instruction Fetch**:
    - IR ← [PC]
    - PC = PC + 4 (*32-bit word*)
  - **Instruction Execute**:
    - Check **I**nstruction **R**egister
      - **IR**: a register in CPU for placing instruction
    - Perform the operation

- *<u>Straight-line sequencing</u>: Fetch and execute instructions, one at a time, in the order of increasing addresses*



Class Exercise 5.1

Student ID: ____________ Date:
Name: __________________ ________

- Consider a task of adding *n* num:
  - The symbolic memory addresses of the *n* numbers: NUM1, NUM2, …, NUMn
  - The result is in memory location SUM.
- Please write the program segment to add n num into R2.
- Answer:

Outline

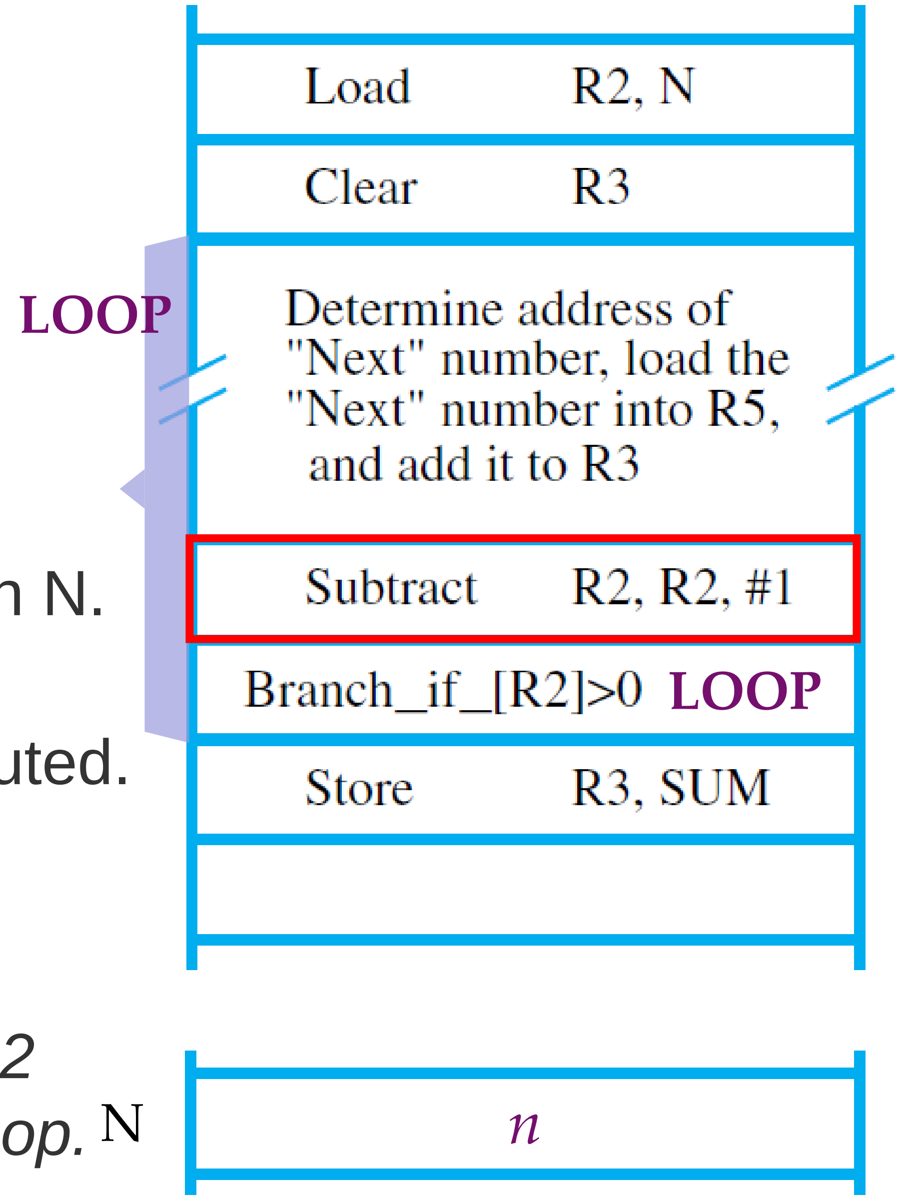
- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Branching: Implementing a Loop (1/2)

- The body of the loop:
  - **Start**: at location **LOOP**
  - **Body:** the repeated task
    - E.g. "**Load-Add**" instructions
  - **End**: at Branch_if_[R2]>0
- Assume that
  - n is stored in memory location N.
  - R2 represents the number of times (i.e. n) the loop is executed.
- Within the body of the loop,

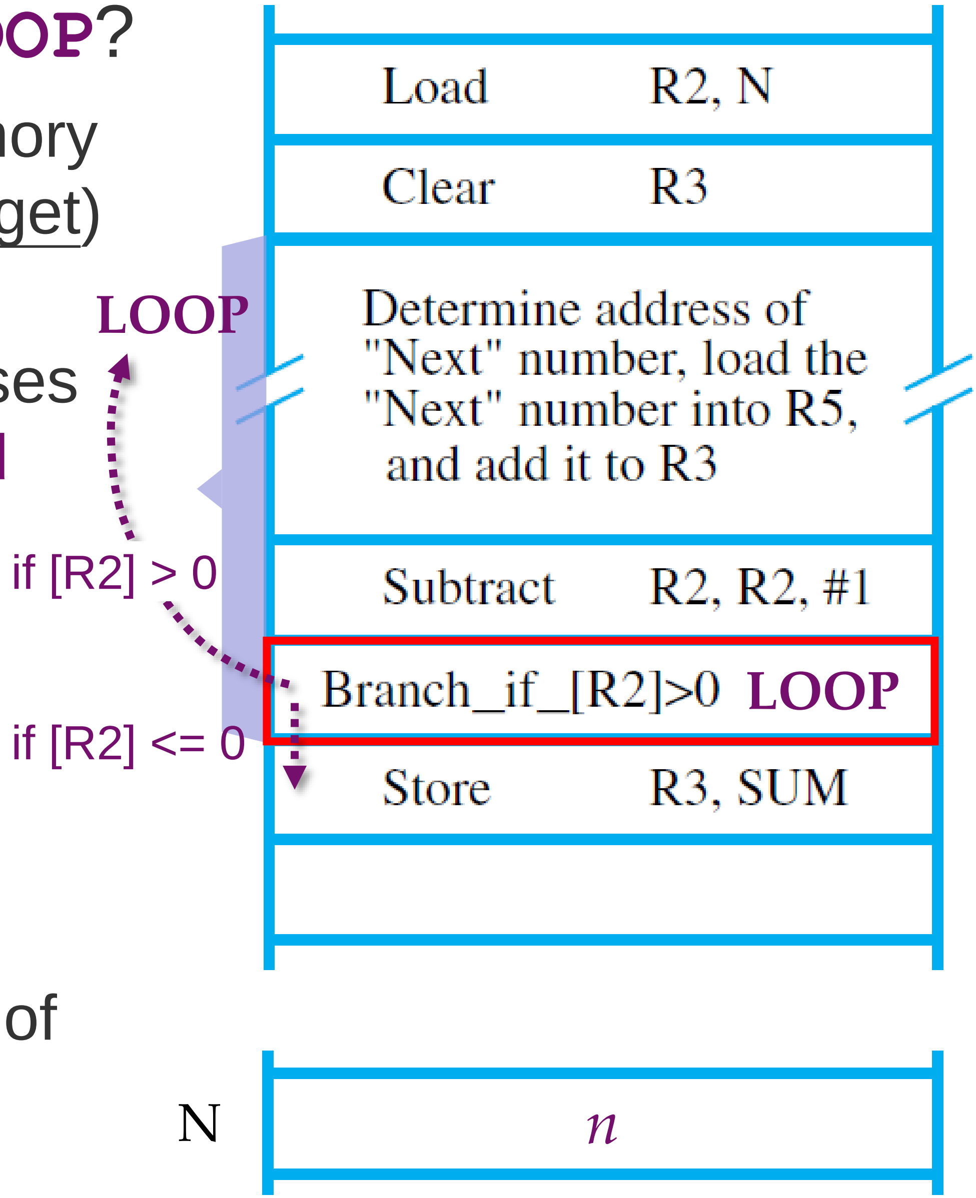
  Subtract R2, R2, #1

  - *Decreasing the contents of R2 by 1 each time through the loop.*



Branching: Implementing a Loop (2/2)

- How to "jump back" to **`LOOP`**?
  - **Branch**: loads a new memory address (called <u>branch target</u>) into the PC.
  - **Conditional Branch**: causes a branch only if a specified condition is satisfied.
- Branch_if_[R2]>0 **`LOOP`**
  - A **conditional branch** instruction that causes branch to location LOOP.
  - **Condition**: If the contents of R2 are <u>greater than zero</u>.



Class Exercise 5.2

- The program for adding a list of *n* numbers can be derived as follows. In which, the indirect addressing is used to access successive numbers in the list.
- Please fill in the blank comment fields below:

LABEL	OPCODE	OPERAND	COMMENT
	`Load`	`R2, N`	*Load the size of the list.*
	`Clear`	`R3`	*Initialize sum to 0.*
	`Move`	`R4, #NUM1`	
`LOOP:`	`Load`	`R5, (R4)`	
	`Add`	`R3, R3, R5`	
	`Add`	`R4, R4, #4`	
	`Subtract`	`R2, R2, #1`	
	`Branch_if_[R2]>0`	`LOOP`	
	`Store`	`R3, SUM`	Store the final sum.

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Condition Codes (1/2)

- Operations performed by the processor typically generate number results of *positive*, *negative*, or *zero*.
  - E.g. `Subtract R2, R2, #1` (in the Loop program)
- **Condition Code Flags**: keep the information about the results for subsequent conditional branch (if any).
  - **Condition Code Register** (or **Status Register**): groups and stores these flags in a special register in the processor.
- Four common flags:

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out occurs; otherwise, cleared to 0

Condition Codes (2/2)

- Consider the Conditional Branch example:
  - If condition codes are used, the branch could be simplified:

    Branch_if_[R2]>0 LOOP → Branch>0 LOOP

    without indicating the register involved in the test.

  - The new instruction causes a branch if neither N nor Z is 1.
    - The Subtract instruction would cause both N and Z flags to be cleared to 0 if R2 is still greater than 0.

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out occurs; otherwise, cleared to 0

Class Exercise 5.3

- Given two 4-bit registers R1 and R2 storing signed integers in 2's-complement format. Please specify the condition flags that will be affected by **`Add R2, R1`**:

$$if\ R1 = (2)_{10} = (0010)_2,\ \ R2 = (-5)_{10} = (1011)_2$$

Answer: ______________________________

$$if\ R1 = (2)_{10} = (0010)_2,\ \ R2 = (-2)_{10} = (1110)_2$$

Answer: ______________________________

$$if\ R1 = (7)_{10} = (0111)_2,\ R2 = (1)_{10} = (0001)_2$$

Answer: ______________________________

$$if\ R1 = (5)_{10} = (0101)_2,\ R2 = (-2)_{10} = (1110)_2$$

Answer: ______________________________

Recall: Signed Integer Representation

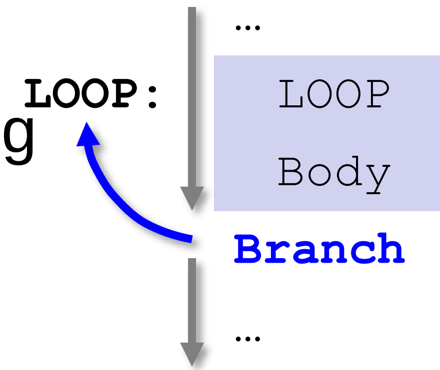
B	Values Represented		
$b_3b_2b_1b_0$	Sign-and-magnitude	1's-complement	2's-complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Outline

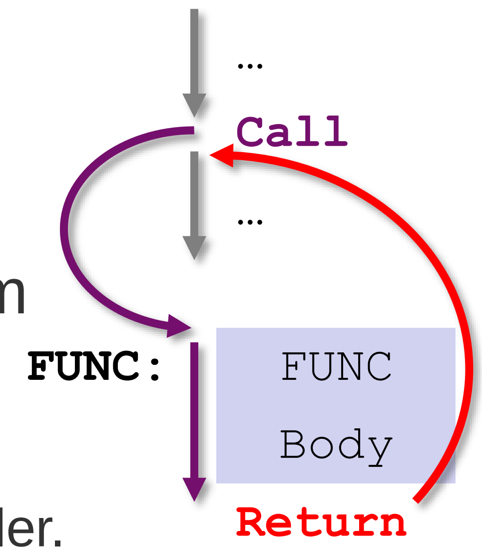
- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Branch vs. Subroutine

- **Branch**:
  - Jumping to a particular instruction by loading its memory address into PC.



- It's also common to <u>perform a particular task many times</u> on different values.
- **Subroutine/Function Call**
  - **Subroutine**: a <u>block of instructions</u> that will be executed each time when calling.
  - **Subroutine/Function Call**: when a program *branches* to (back from) a subroutine.
    - **Call**: the instruction performing the branch.
    - **Return**: the instruction branching back to the caller.
  - "**Stack**" is essential for subroutine calls.

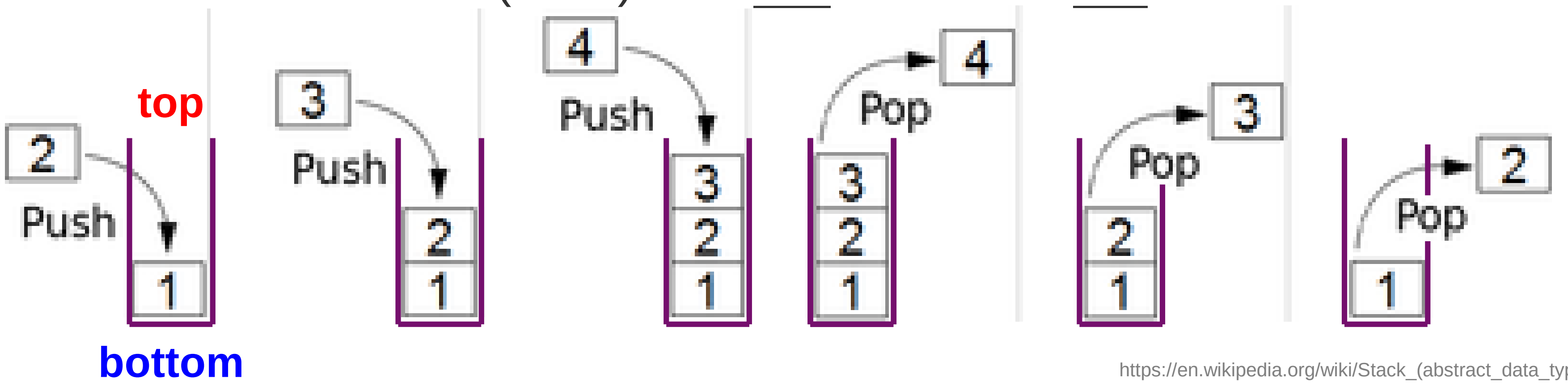


Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Stacks

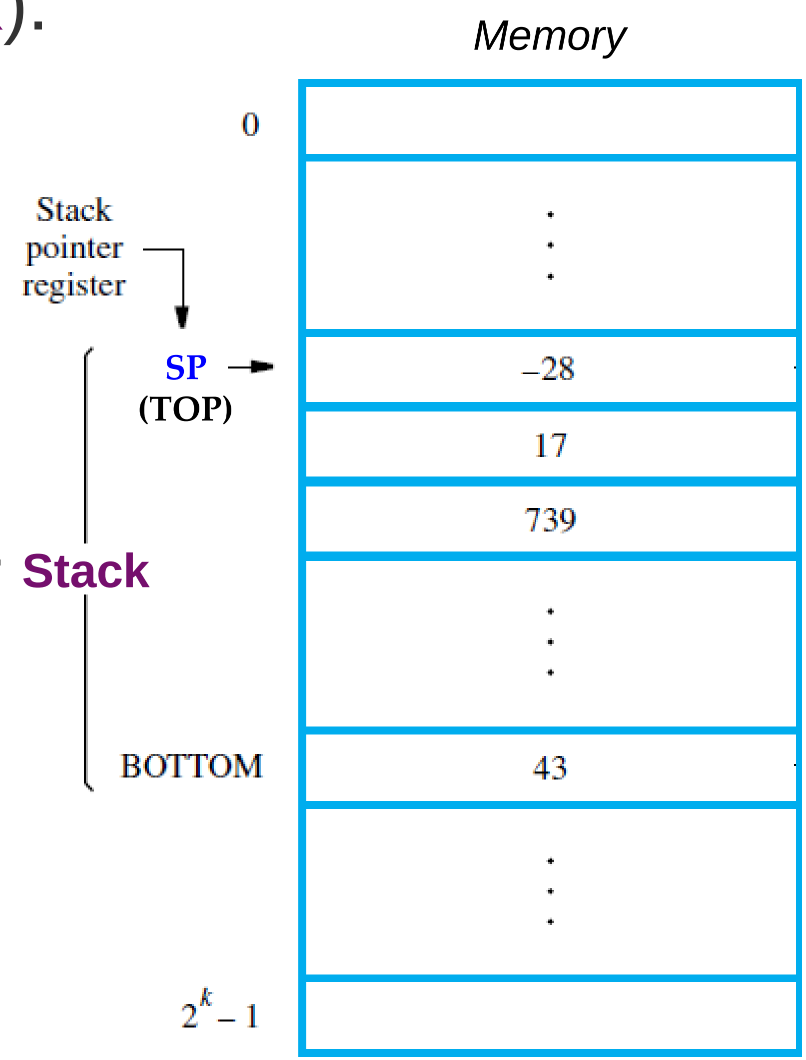
- **Stack** is a list of data elements (usually words):
  - *Elements can only be removed at one end of the list.*
    - This end is called the **top**, and the other end is called the **bottom**.
    - Examples: a stack of coins, plates on a tray, a pile of books, etc.
  - **Push**: Placing a new item at the top end of a stack
  - **Pop**: Removing the top item from a stack
  - Stack is often called LIFO or FILO stack:
    - *Last-In-First-Out* (LIFO): The last item is the first one to be removed.
    - *First-In-Last-Out* (FILO): The first item is the last one to be removed.



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Processor Stacks (1/2)

- Modern processors usually provide native support to stacks (called processor stack).
  - A processor stack can be implemented by using a portion of the main memory.
    - Data elements of a stack occupy successive memory locations.
    - The first element is placed in location BOTTOM (*larger address*).
    - The new elements are pushed onto the TOP of the stack.
  - **Stack Pointer (SP)**: a special processor register to keep track of the address of the TOP item of processor stack.



Processor Stacks (2/2)

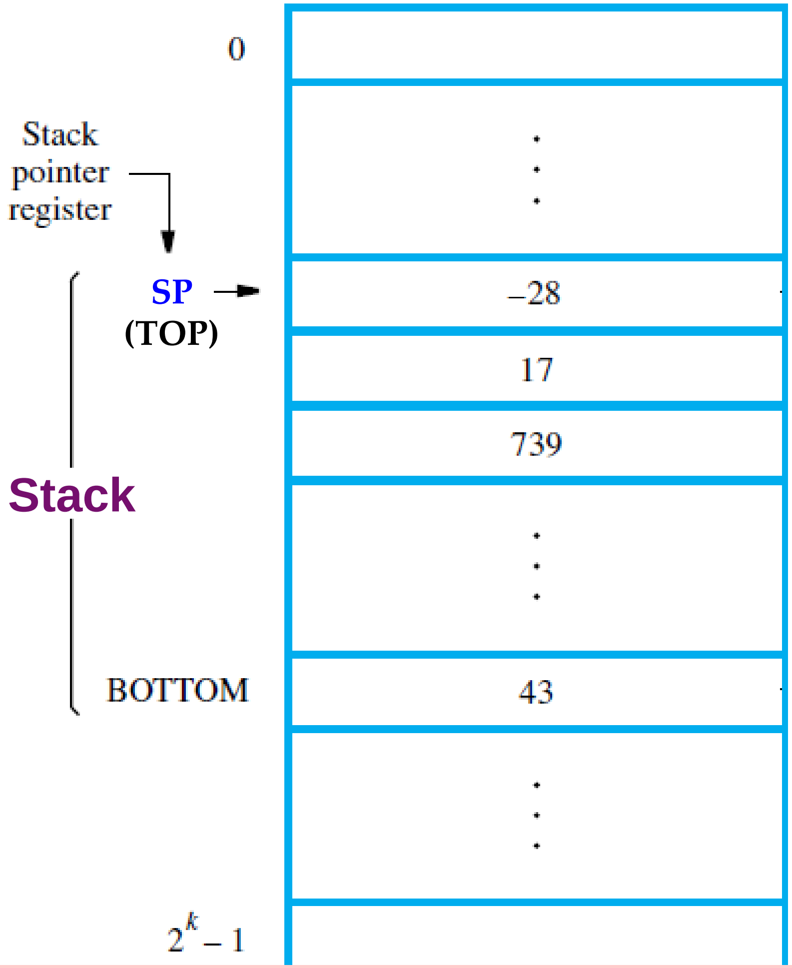
- Given a stack of word data items, and consider a byte-addressable memory with a 32-bit word:
  - **Push** a item in Rj onto the stack:

```
Subtract     SP, SP, #4
Store        Rj, (SP)
```

    - The `Subtract` instruction first subtracts 4 from the contents of SP and places the result in SP.
    - The `Store` instruction then places the content of Rj onto the stack.
  - **Pop** the top item into Rj

```
Load         Rj, (SP)
Add          SP, SP, #4
```

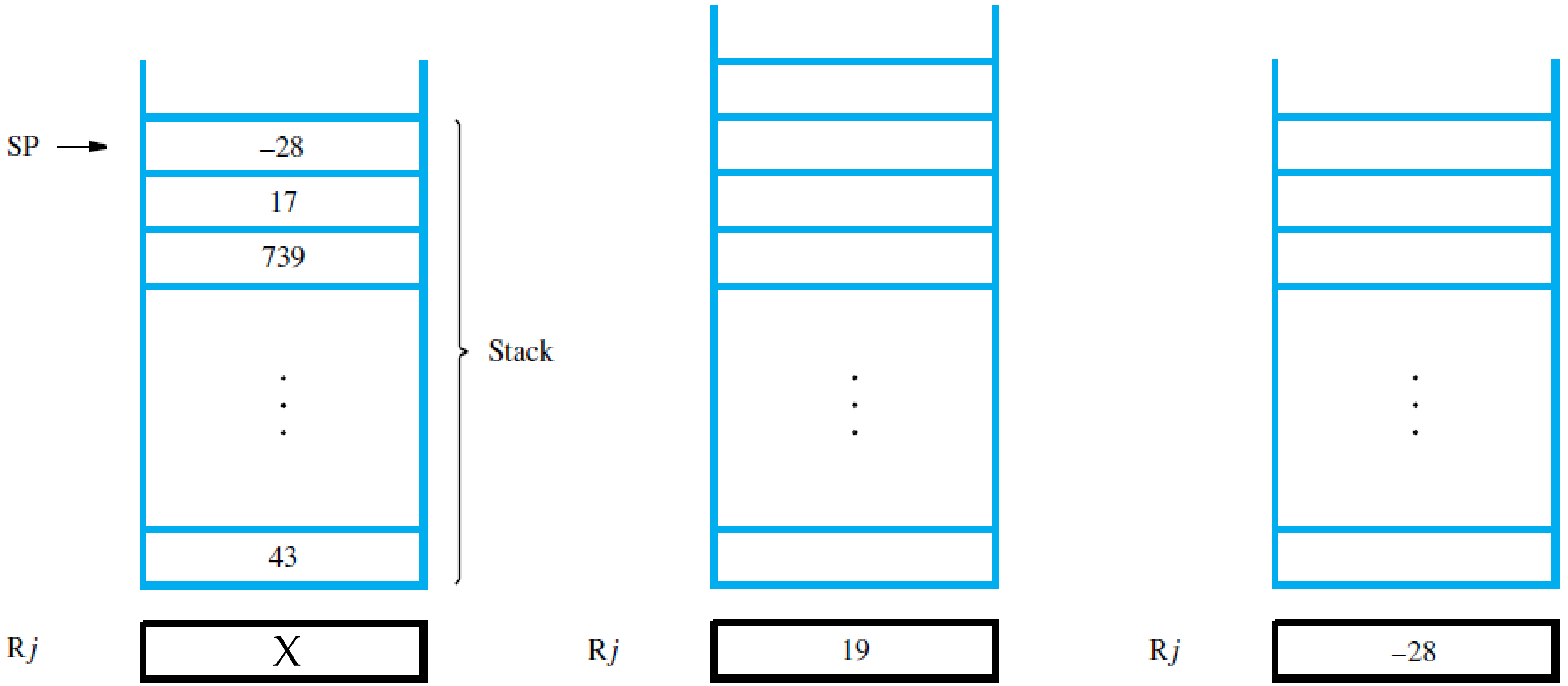
    - The `Load` instruction first loads the top value from the stack into register Rj
    - The `Add` instruction then increments the stack pointer by 4.



Questions: How to use Autoincrement and Autodecrement addressing modes to simplify?

Class Exercise 5.4

- Fill in the contents of the stack and the register Rj, denote the location of SP, and specify the range of the stack, after the push/pop operation is performed:



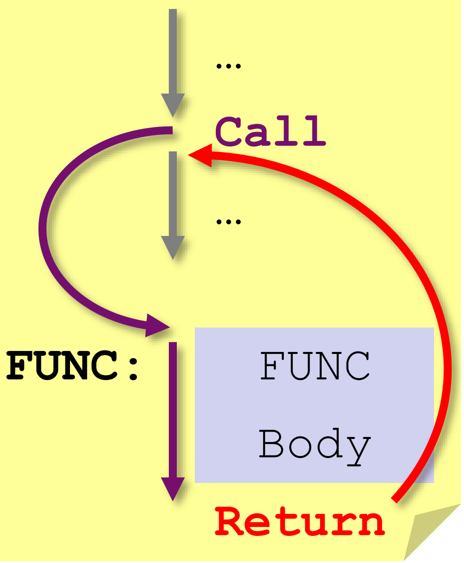
(a) Before **Push/Pop** (b) After **Push** (c) After **Pop**

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Revisit: Subroutine

- Recall:
  - When a program branches to a subroutine we say that it is **calling** the subroutine.
  - After a subroutine calling, the subroutine is said to **return** to the program that called it.
    - Continuing immediately after the instruction that called the subroutine.

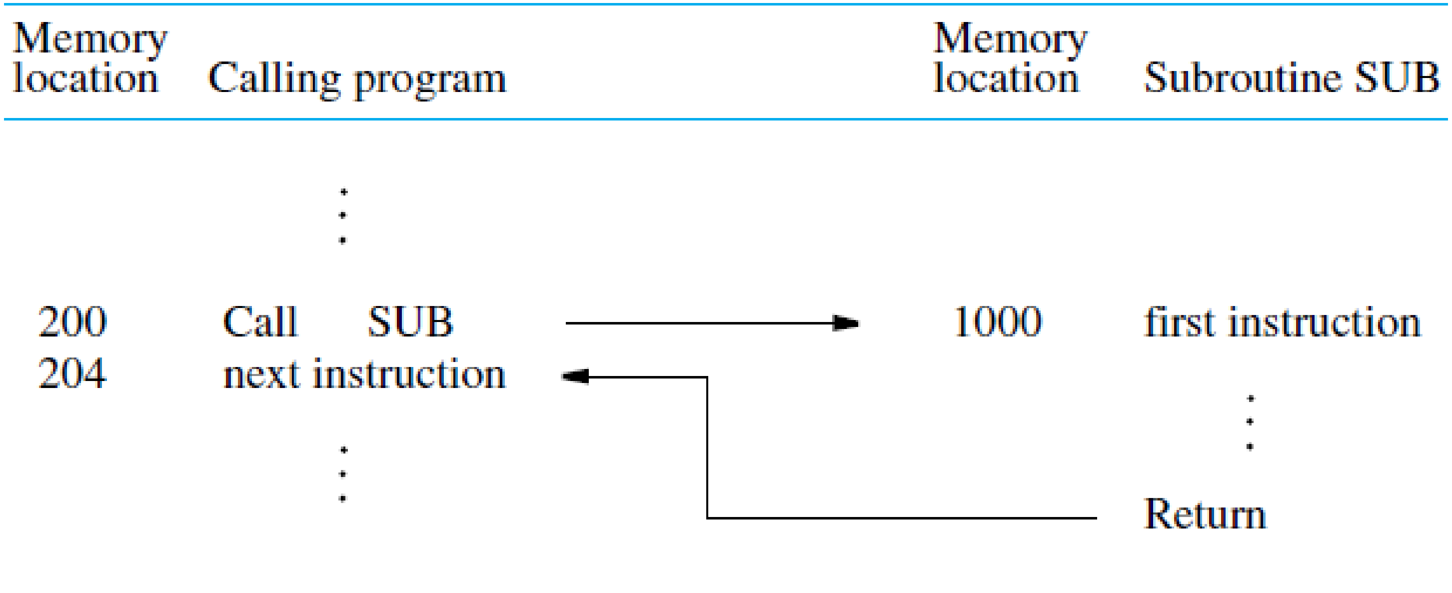


- However, the subroutine may be called from different places in a calling program.
- Thus, provision must be made for **returning** to the appropriate location.
  - That is, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

Subroutine Linkage

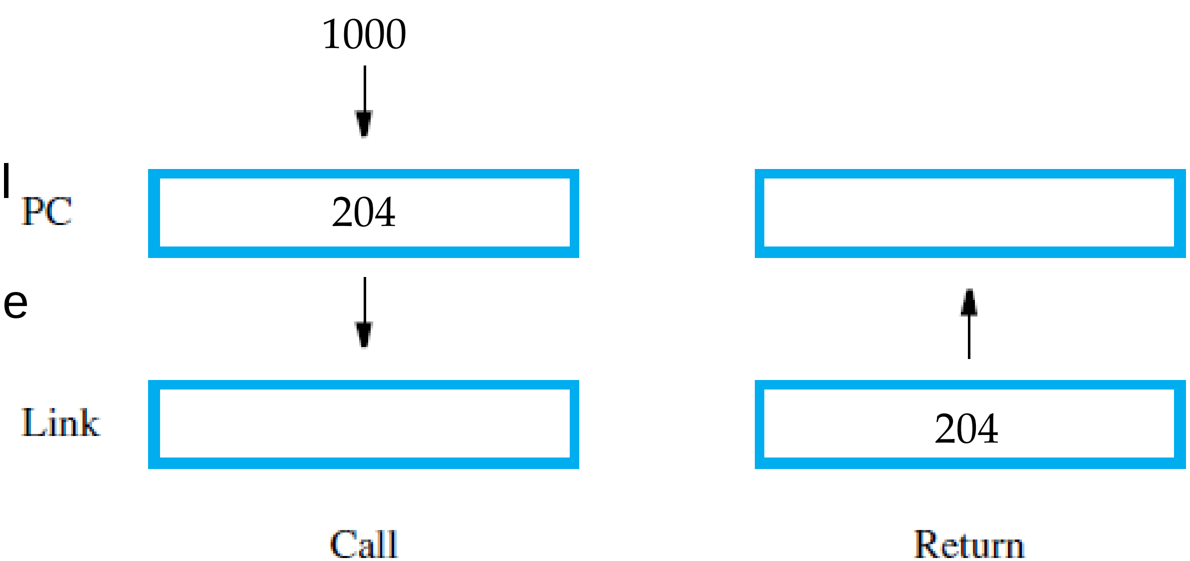
- **Subroutine Linkage** method: the way makes it possible to call and return from subroutines.
  - The simplest method: saving the return address in a special processor register called the link register.
- With the help of link register,
  - The **Call** instruction can be implemented as a special *branch* instruction:
    - Store the contents of the PC in the link register.
    - Branch to the target address specified by the Call instruction.
  - The **Return** instruction can be implemented as a special *branch* instruction as well:
    - Branch to the address contained in the link register.

Example of Subroutine Linkage



Branch to the target address specified by Call

Store PC into the link register.



Branch back to the address contained in the link register.

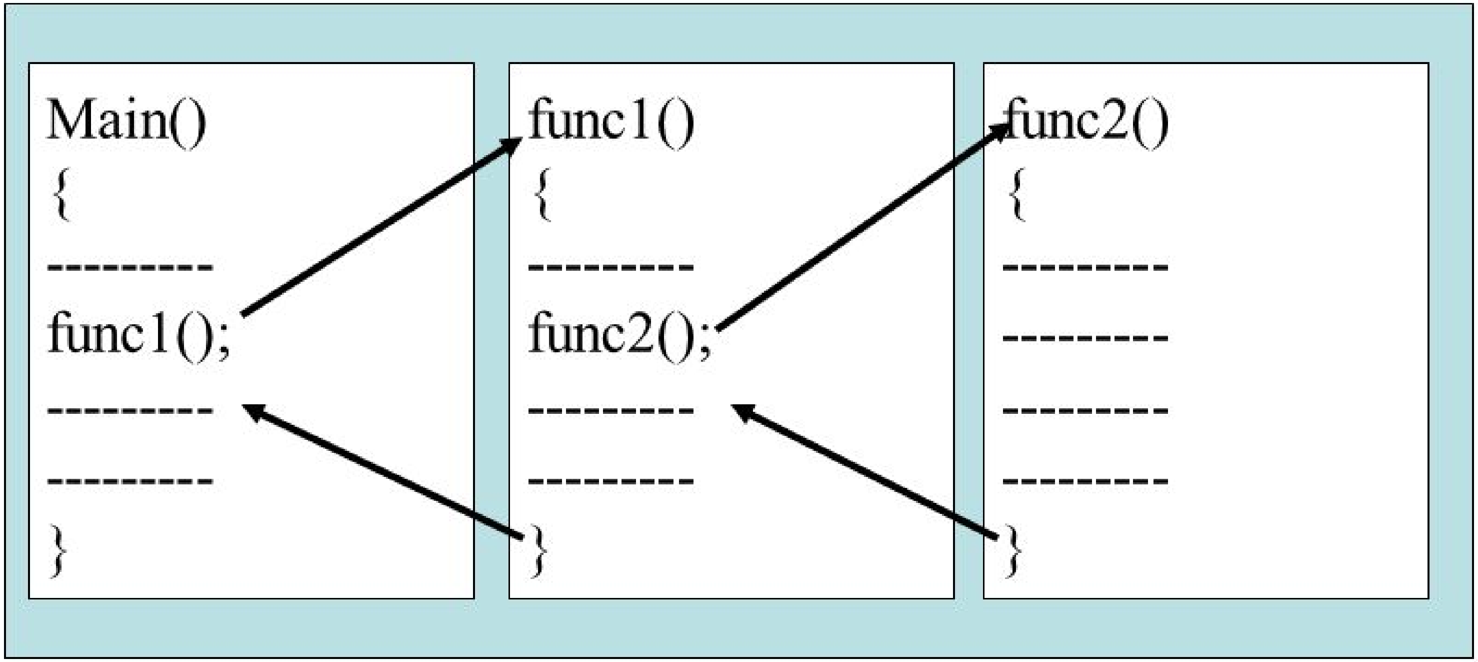
Question: Is one link register enough for all cases?

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Subroutine Nesting (1/3)

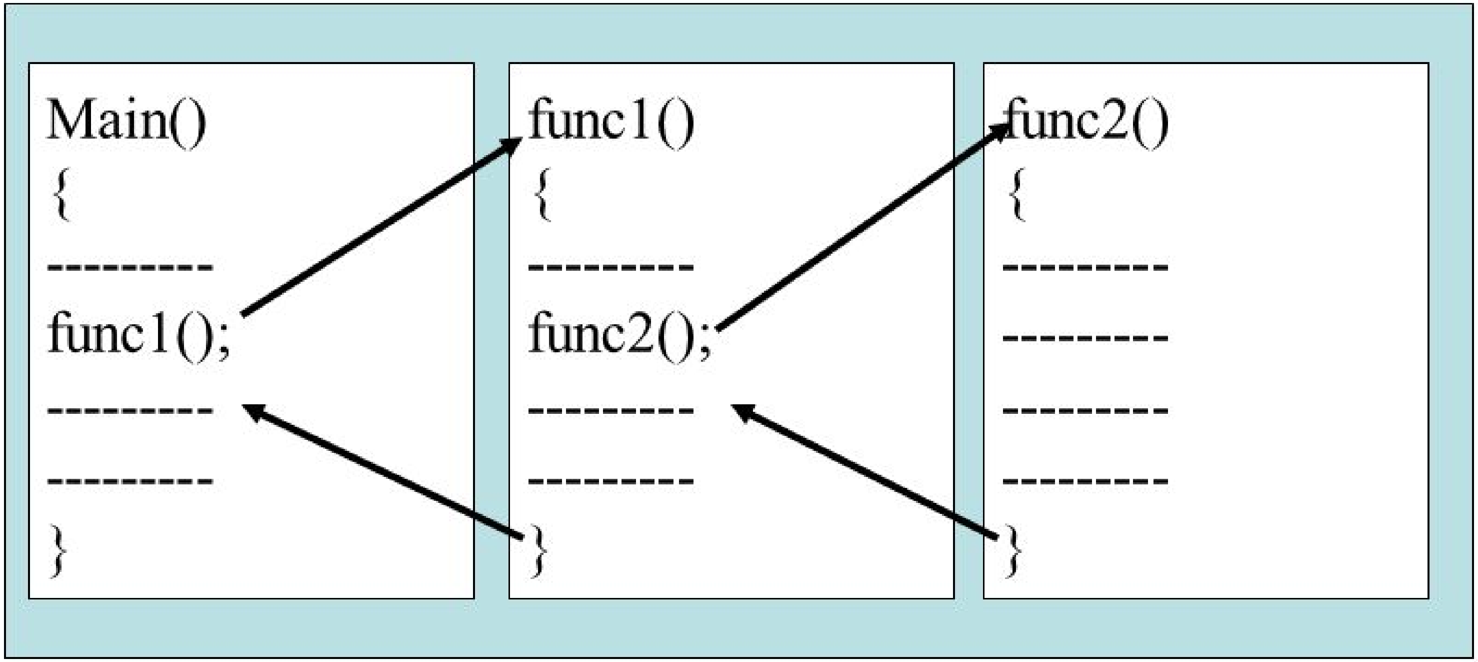
- **Subroutine Nesting**: One subroutine calls another subroutine or itself (i.e. recursion).
  - If the return address of the second call is also stored in the link register, the first return address will be lost … ERROR!
  - Subroutine nesting can be carried out to any depth …



https://slideplayer.com/slide/7603076/

Subroutine Nesting (2/3)

- *Observation*: The return address needed for the first return is the last one generated in the nested calls.
  - That is, return addresses are generated and used in a last-in–first-out (LIFO) order.



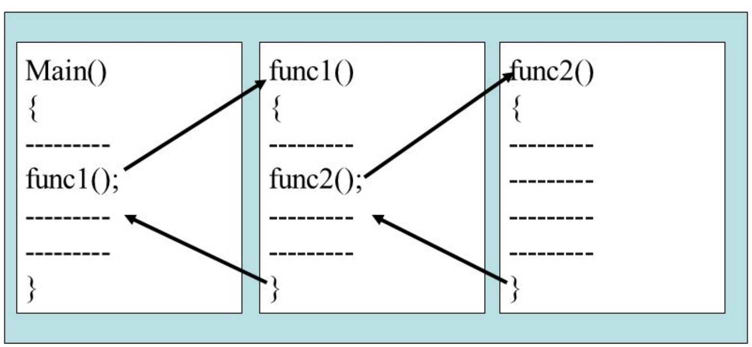
Subroutine Nesting (3/3)

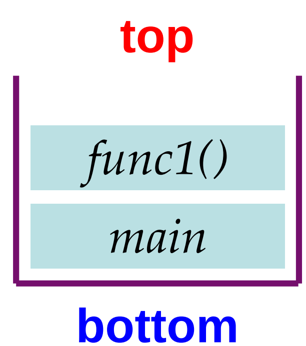
- Processor stack is useful to store subroutine linkage:
  - The **Call** instruction:
    - ~~Store the contents of the PC in the link register~~

    → Push the contents of the PC to the processor stack
    - Branch to the target address specified by the Call instruction.

    → (*Unchanged*)
  - The **Return** instruction:
    - ~~Branch to the address contained in the link register~~

    → Branch to the address popped out from the processor stack



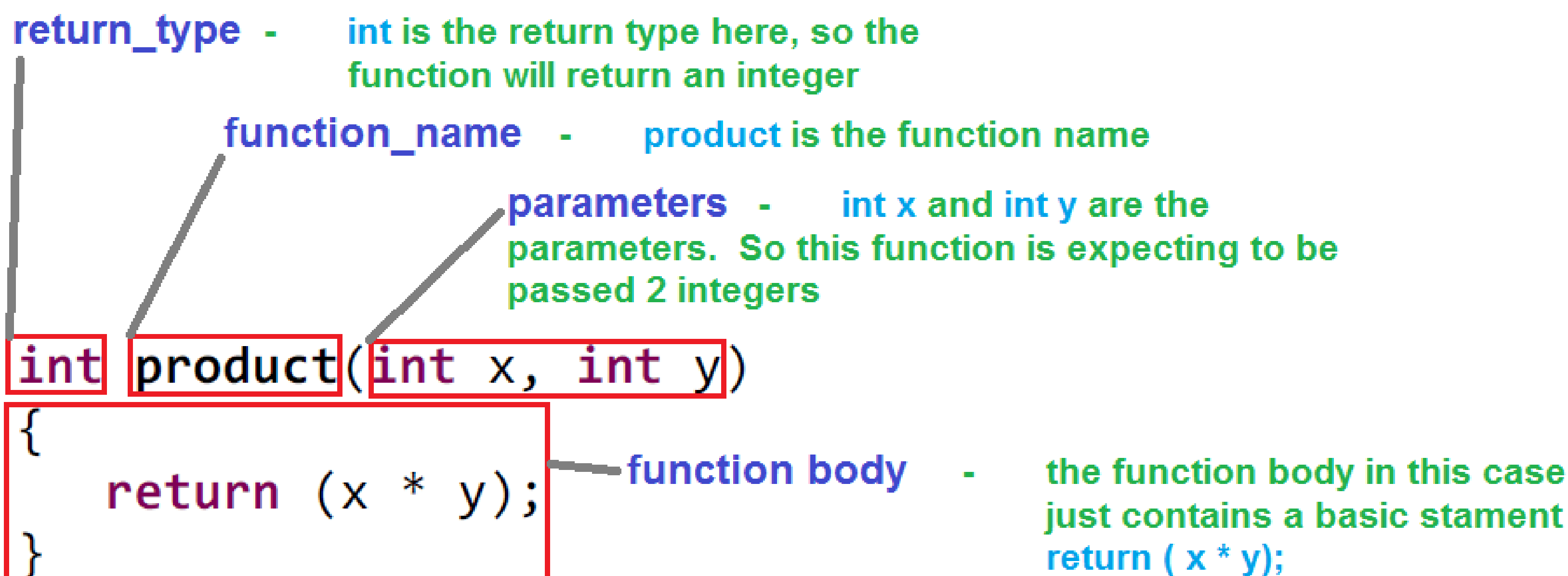


Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

Parameter Passing

- **Parameter Passing**: The exchange of information between a calling program and a subroutine.
  - When calling a subroutine, a program must provide the parameters (i.e. operands or their addresses) to be used.
  - Later, the subroutine returns other parameters, which are the results of the computation.



http://coder-tronics.com/c-programming-functions-pt1/

Parameter Passing via Registers

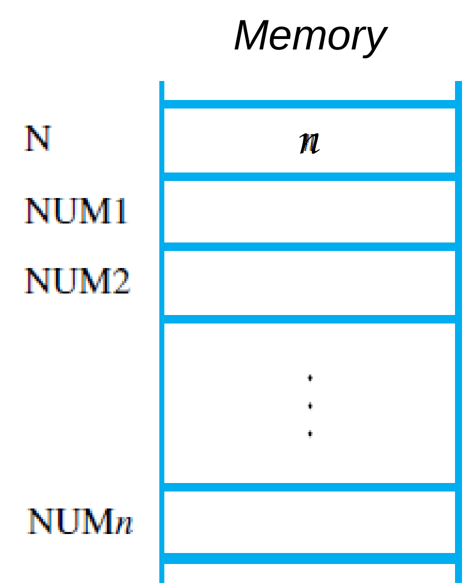
- The simplest way is placing parameters in registers.
- Recall the program for adding a list of numbers.
- The program can be implemented as a subroutine with
  - R2 & R4 are used to pass the size of list & the address of the first num,
  - R3 is used to pass back the sum computed by the subroutine.

Calling Program

```
          Load               R2, N         Parameter 1 is list size.
          Move               R4, #NUM1     Parameter 2 is list location.
          Call               LISTADD       Call subroutine.
          Store              R3, SUM       Save result.
          ⋮
```

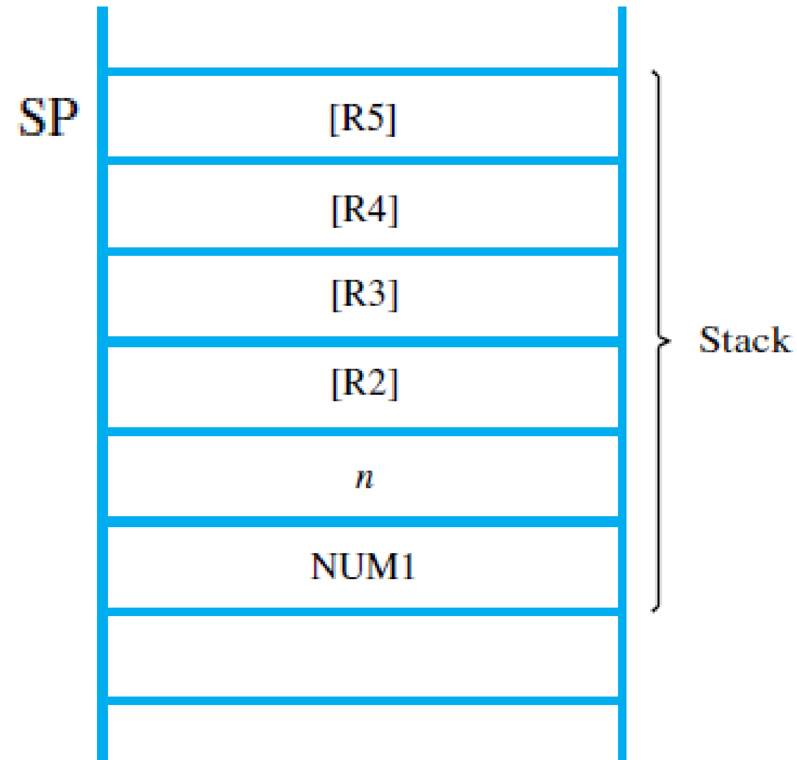
Subroutine

```
LISTADD:  Subtract           SP, SP, #4    Save the contents of
          Store              R5, (SP)        R5 on the stack.
          Clear              R3            Initialize sum to 0.
LOOP:     Load               R5, (R4)      Get the next number.
          Add                R3, R3, R5    Add this number to sum.
          Add                R4, R4, #4    Increment the pointer by 4.
          Subtract           R2, R2, #1    Decrement the counter.
          Branch_if_[R2]>0   LOOP
          Load               R5, (SP)      Restore the contents of R5.
          Add                SP, SP, #4
          Return                           Return to calling program.
```



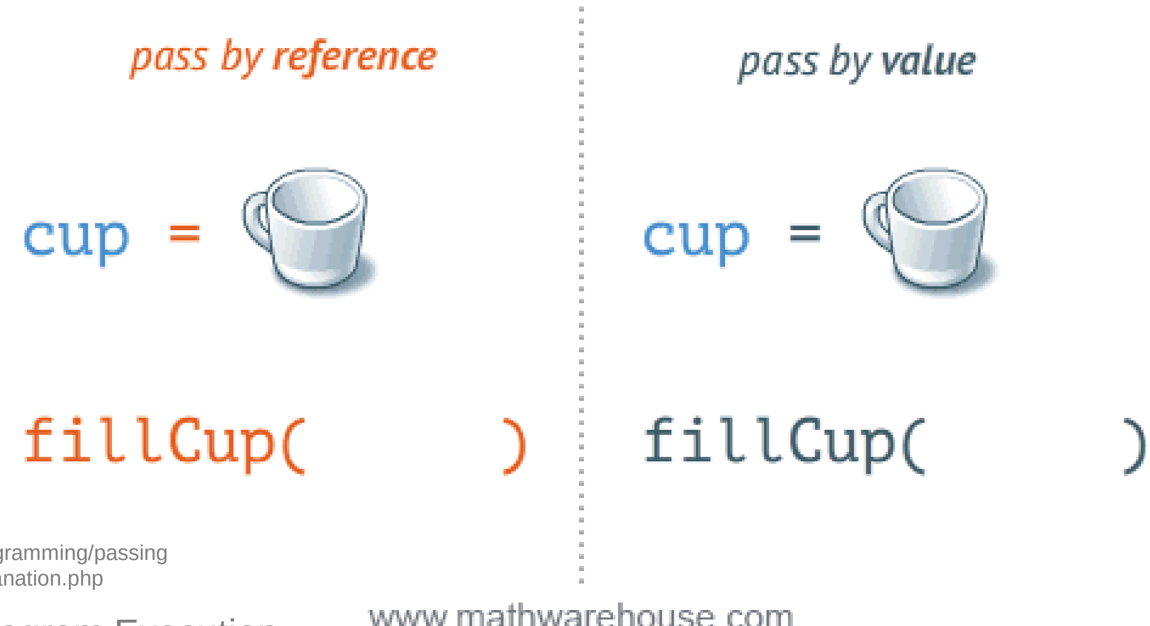
Parameter Passing on Stack

- What if there are more parameters than registers?
- What if the subroutine calls itself (recursion)?
- The processor stack, again, provides a good scheme to pass an arbitrary number of parameters.
- What we can pass via stack?
  – We can push *all parameters to be computed* onto the stack.
  – We can push *the contents of all "to-be-used" registers* onto the stack.
  – We can also push *the computed sum* before the return to the calling program.



Parameter Passing by Value / Reference

- What kind of parameters can we pass?
- **Passing by Value**
  - The actual number N is passed by an immediate value.
- **Passing by Reference** (more powerful, be careful!)
  - Instead of passing the actual values in the list, the routine passes the starting address (i.e. reference) of the NUM list.



https://www.mathwarehouse.com/programming/passing-by-value-vs-by-reference-visual-explanation.php

Summary

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing an Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stacks
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing