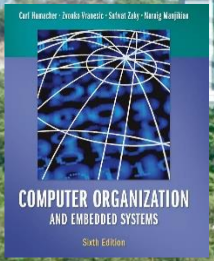
香港中文大學
The Chinese University of Hong Kong

CSCI2510 Computer Organization
Lecture 05: Program Execution

Ming-Chang YANG

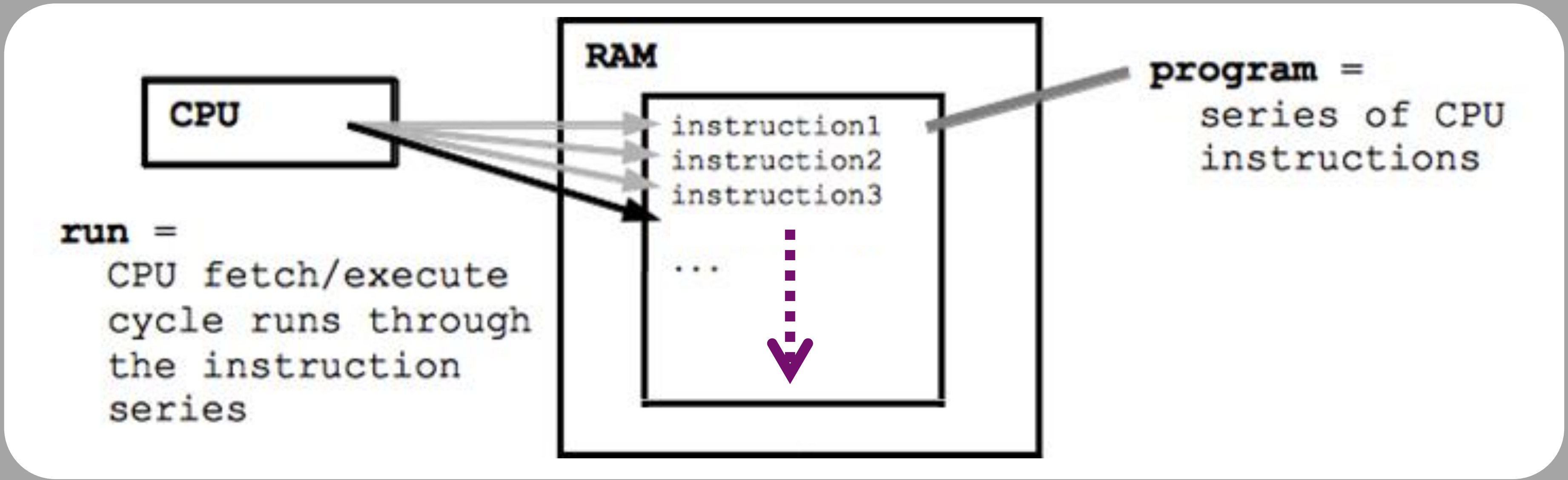
mcyang@cse.cuhk.edu.hk



Reading: Chap. 2.3~2.7, 2.10, 4

Recall: Program Execution

- A computer is governed by instructions.
  - To perform a given task, a program consisting of a list of machine instructions is stored in the memory.
    - Data to be used as operands are also stored in the memory.
  - Individual instructions are brought from the memory into the processor, one after another, in a sequential way (normally).
  - The processor executes the specified operation/instruction.



Outline

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

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 PCs have a special 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 a Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stack
    - 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 $0, 0($2)
lw $1, 4($2)
sw $1, 0($2)
sw $0, 4($2)
```

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

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

`$0,$1,$2`: registers

`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

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

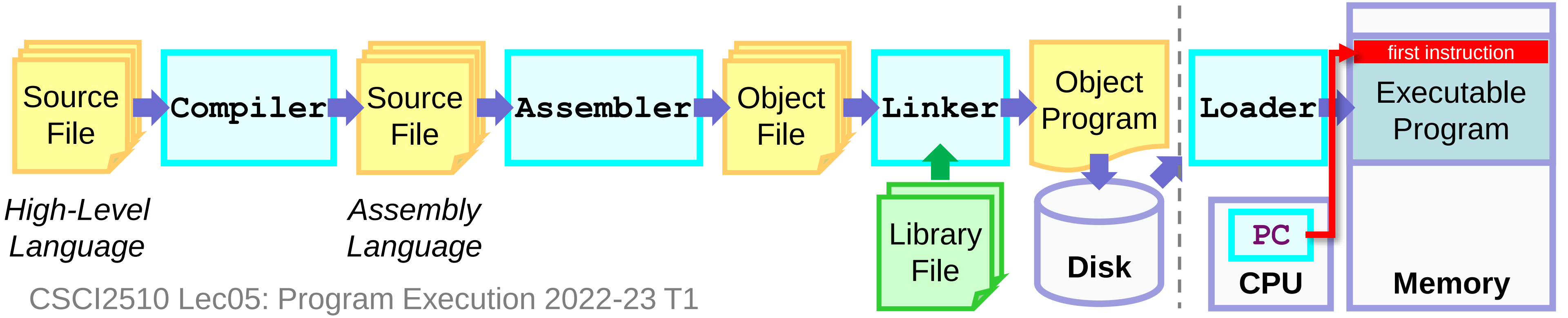
https://clip2art.com/explore/Boy%20clipart%20teacher/

Generating/Executing a 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

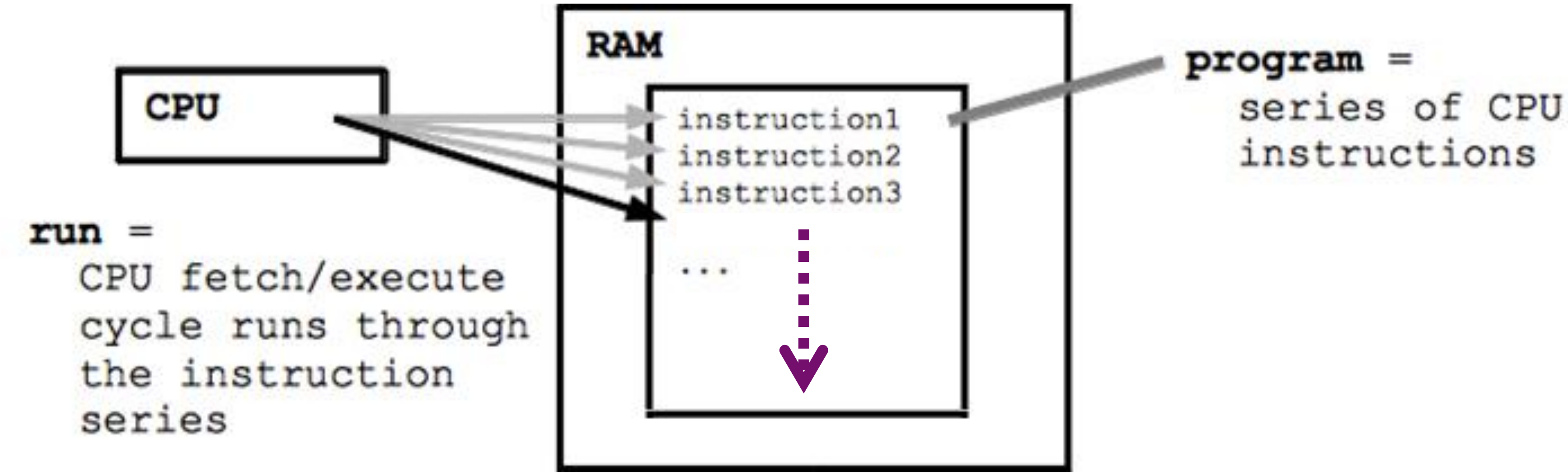
---

- **Loader**: Load the program into memory and load the address of the first instruction into program counter (PC)



Activities in a Computer: Instructions

- A computer is governed by instructions.
  - To perform a given task, a program consisting of a list of machine instructions is stored in the memory.
    - Data to be used as operands are also stored in the memory.
  - Individual instructions are brought from the memory into the processor, one after another, in a sequential way (normally).
  - The processor executes the specified operation/instruction.



An Example of Program Execution

- Considering a program of 3 instructions:

PC → I_0: Load R0, LOC

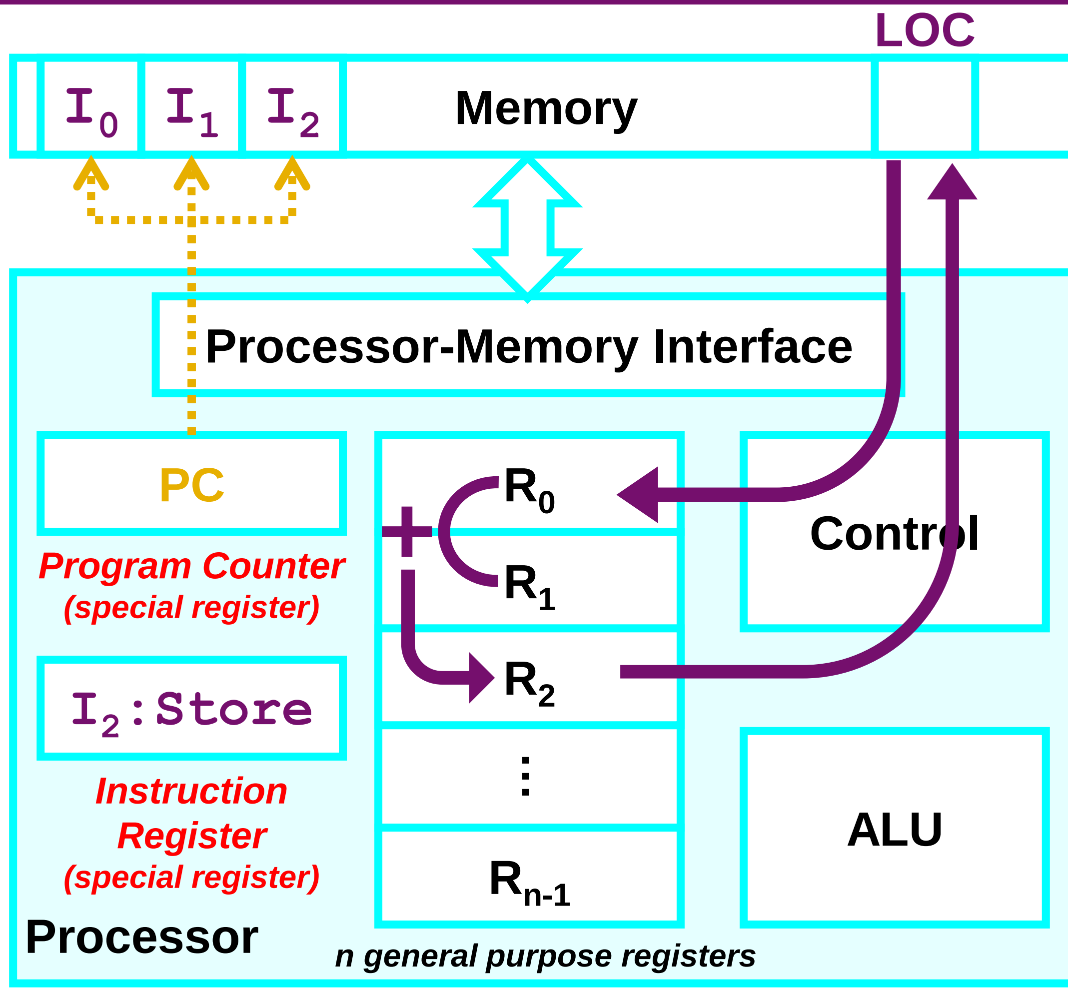
- Reads the contents of a memory location LOC
- Loads them into processor register R0

– **I_1: Add R2, R0, R1**

- Adds the contents of registers R0 and R1
- Places their sum into register R2

– **I_2: Store R2, LOC**

- Copies the operand in register R2 to memory location LOC



PC: contains the memory address of the NEXT instruction to be fetched and executed.
IR: holds the CURRENT instruction that is being executed.
R_0~R_{n-1}: n general-purpose registers.

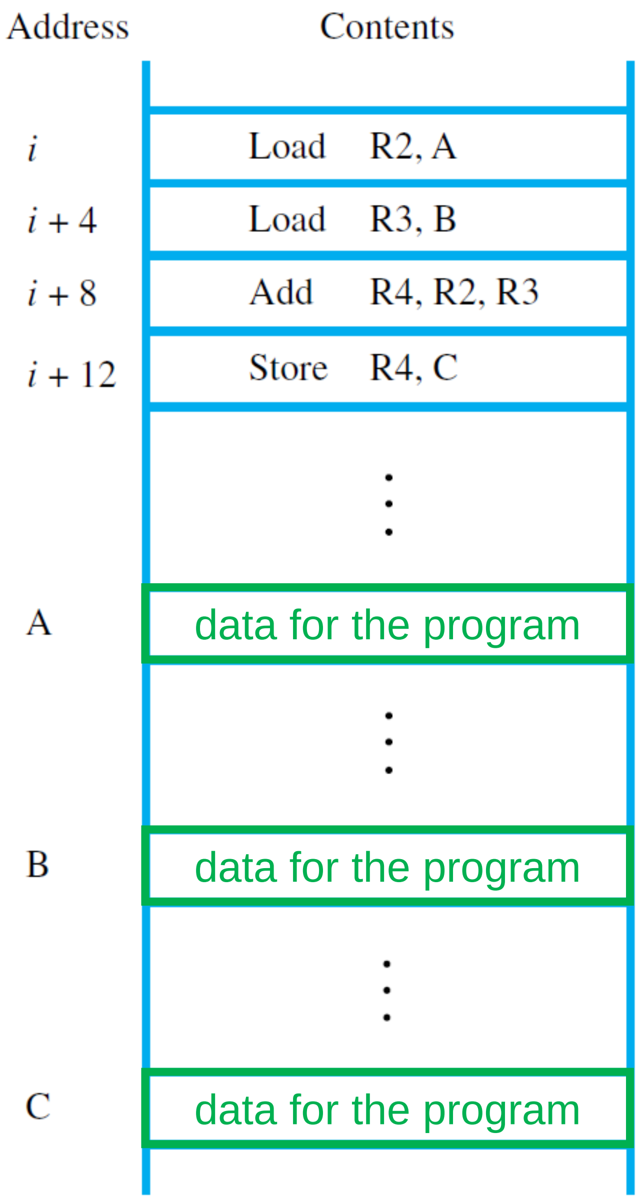
Outline

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

Instruction Execution & Sequencing (1/3)

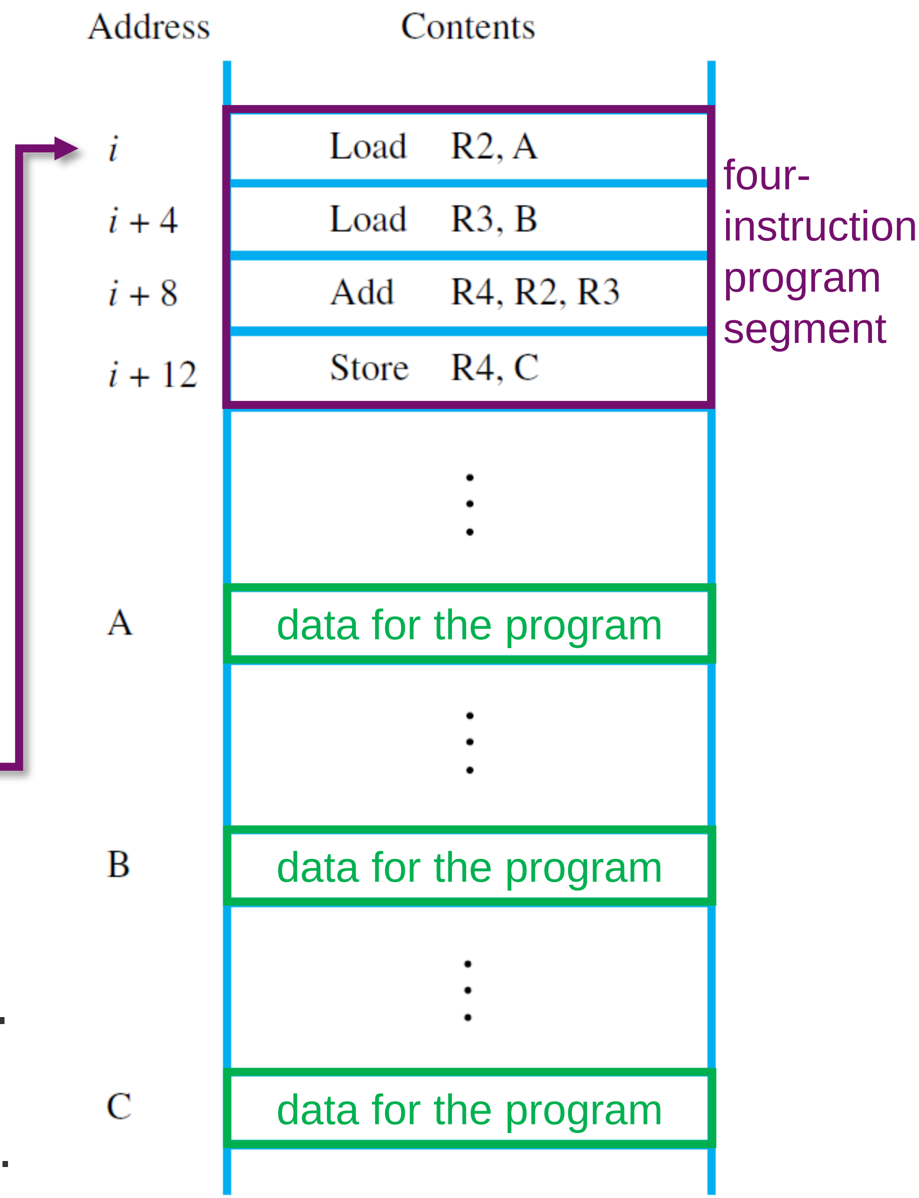
- Consider a machine:
  - RISC instruction set
  - 32-bit word, 32-bit instruction
  - Byte-addressable memory

- Given the task $C=A+B$ (*Lec04*)
  - Implemented as C ← [A] + [B]
  - Possible RISC-style 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.



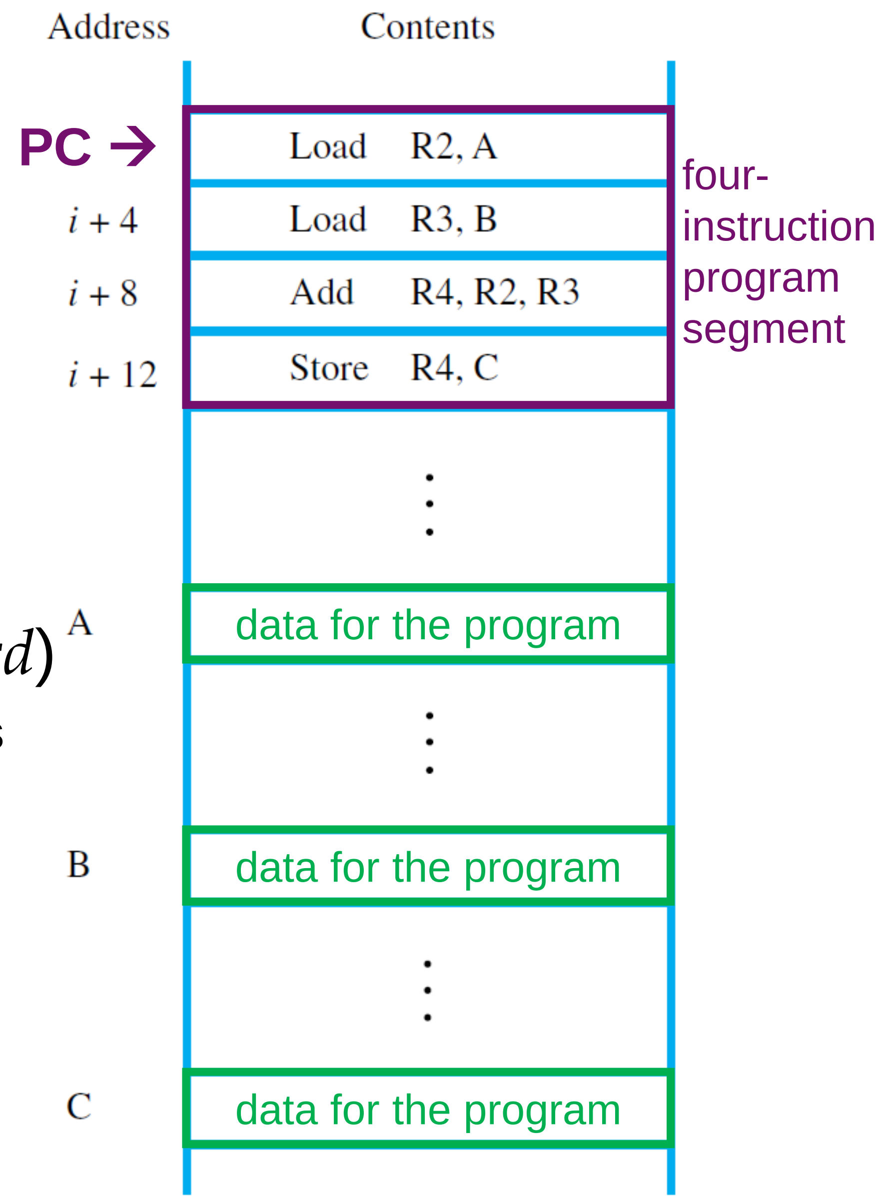
- **Straight-Line Sequencing:**
  - CPU fetches and executes instructions indicated by PC, one at a time, in the order of increasing addresses.

1) Instruction Fetch:

- `IR ← [[PC]]`
- `PC ← [PC] + 4` (*32-bit word*)
  - ✓ **PC** contains the memory address of the next instruction.
  - ✓ **IR** holds the current instruction.

2) Instruction Execute:

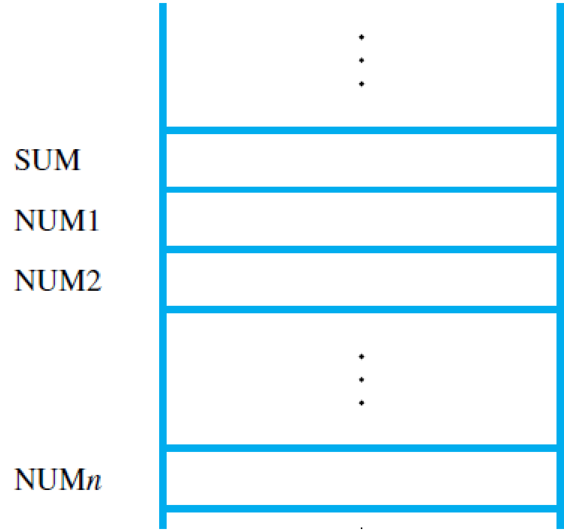
- Interpret (or decode) IR
- Perform the operation



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

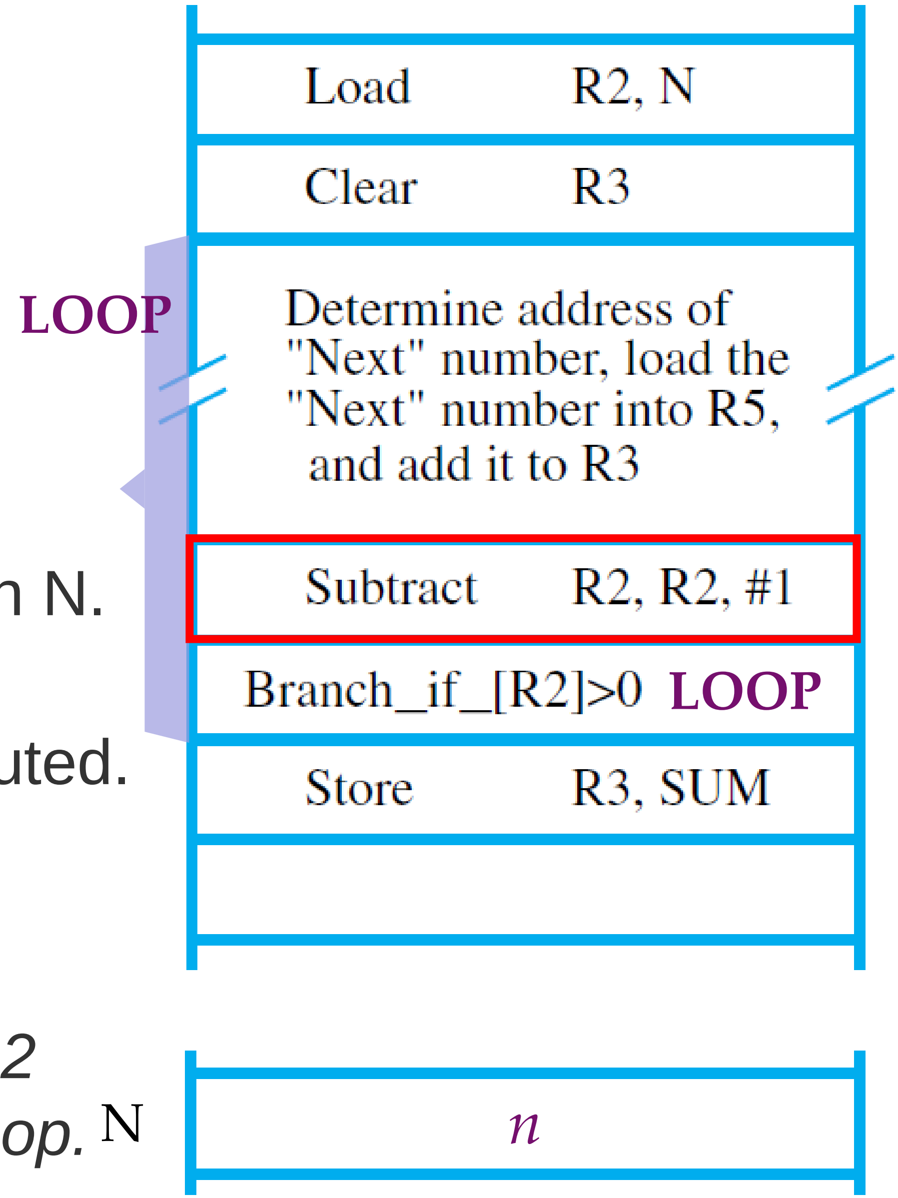
- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing a Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stack
    - Subroutine Linkage
    - Subroutine Nesting
    - Parameter Passing

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,

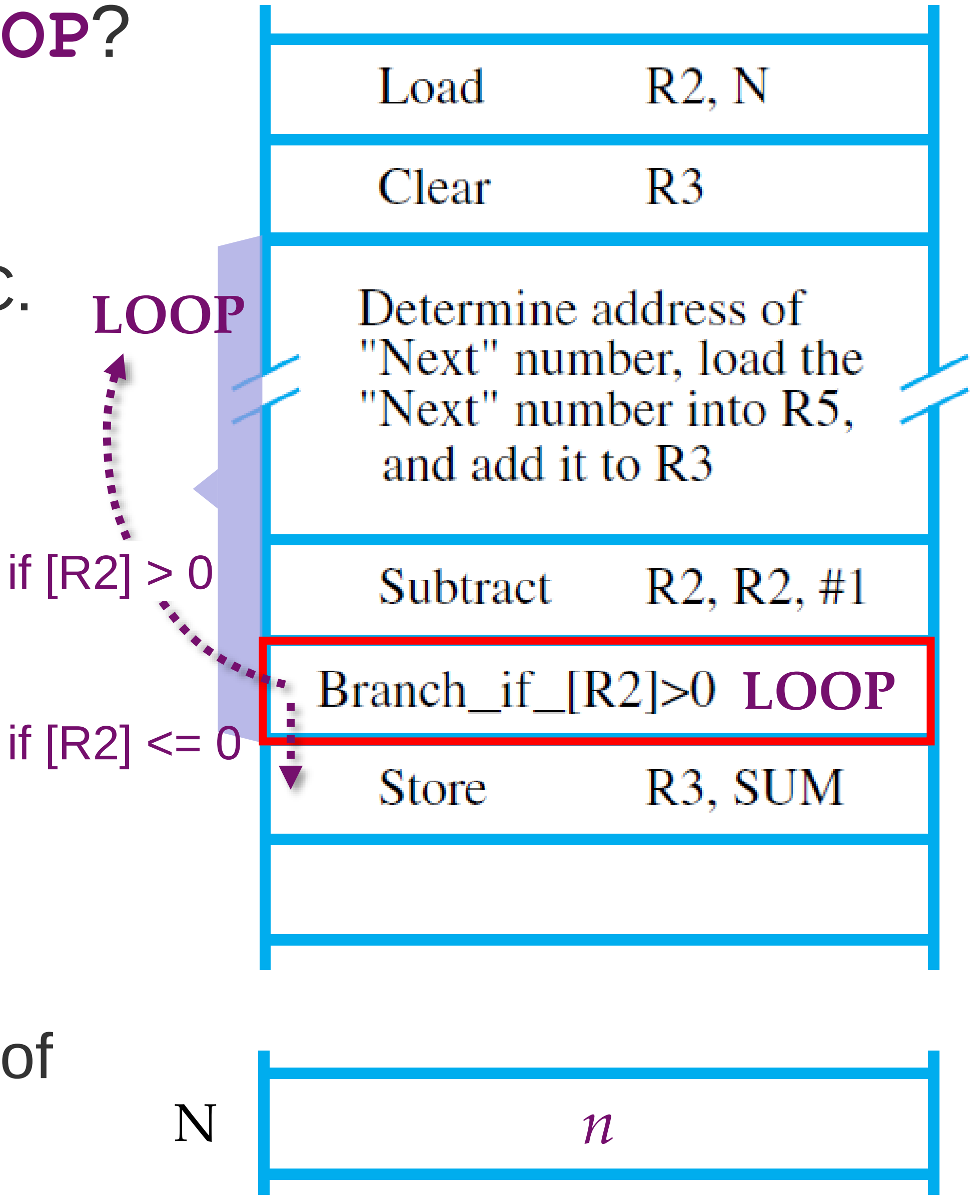
```
Subtract R2, R2, #1
```

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



Branching: Implementing a Loop (2/2)

- How to "jump back" to **LOOP**?
  - ① **Branch**: loads a new memory address (called branch target) into the PC.
  - ② **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 greater than zero.



Class Exercise 5.2

- The below program intends to adding a list of n numbers. In which, we want to use the indirect addressing to access successive numbers in the list.
- Please fill in the blank field below:

LABEL	OPCODE	OPERAND	COMMENT
	`Load`	`R2, N`	*Load the size of the list.*
	`Clear`	`R3`	*Initialize sum to 0.*
	`Move`	`R4, addr NUM1`	*Get address of the first number.*
`LOOP:`	`Load`		*Get the next number.*
	`Add`	`R3, R3, R5`	*Add this number to sum.*
	`Add`	`R4, R4, #4`	*Increment the pointer to the list.*
	`Subtract`	`R2, R2, #1`	*Decrement the counter.*
	`Branch_if_[R2]>0`	`LOOP`	*Branch back if not finished.*
	`Store`	`R3, SUM`	*Store the final sum.*

An Example of Nested Loops

	Move	R2, **addr T**	R2 points to string T.
	Move	R3, **addr P**	R3 points to string P.
	Load	R4, N	Get the value n.
	Load	R5, M	Get the value m.
	Subtract	R4, R4, R5	Compute $n - m$.
	Add	R4, R2, R4	The address of $T(n - m)$.
	Add	R5, R3, R5	The address of $P(m)$.
LOOP1:	Move	R6, R2	Use R6 to scan through string T.
	Move	R7, R3	Use R7 to scan through string P.
LOOP2:	LoadByte	R8, (R6)	Compare a pair of
	LoadByte	R9, (R7)	characters in
	Branch_if_[R8]≠[R9]	NOMATCH	strings T and P.
	Add	R6, R6, #1	Point to next character in T.
	Add	R7, R7, #1	Point to next character in P.
	Branch_if_[R5] > [R7]	**LOOP2**	Loop again if not done.
	Store	R2, RESULT	Store the address of $T(i)$.
	Branch	DONE	
NOMATCH:	Add	R2, R2, #1	Point to next character in T.
	Branch_if_[R4] ≥ [R2]	**LOOP1**	Loop again if not done.
	Move	R8, #–1	Write –1 to indicate that
	Store	R8, RESULT	no match was found.
DONE:	next instruction		

Chap. 2.12.2, Computer Organization and Embedded Systems (6th Ed.)

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing a Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stack
    - 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 of the "most recent" instruction.
  - The subsequent instruction may use it for different purposes.

Common Condition 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 Code Register** (or **Status Register**): groups and stores these flags in a special register in the processor.

Condition Codes (2/2)

- Consider the Conditional Branch example:
  - If condition codes are used, the **`branch`** instruction (**`Branch_if_[R2]>0 LOOP`**) could be simplified as:

    `Branch>0 LOOP`

    without indicating the register involved in the test.
  - This 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.

Common Condition 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

Recall: Overflow in Integer Arithmetic

- **Overflow**: The result of an arithmetic operation does not fall within the representable range.
  - In **Unsigned** Number Arithmetic:
    - *Rule*: A carry-out of 1 from the MSB-bit always indicates an overflow.
      - E.g. $(1111)_2 + (0001)_2 = (\mathbf{1}\ 0000)_2$ ← *overflowed*
      - E.g. $(0111)_2 + (0001)_2 = (\mathbf{0}\ 1000)_2$ ← *no overflow*
  - In **2's-complement Signed** Number Arithmetic:
    - The carry-out bit from the sign-bit is not an indicator of overflow.
      - E.g. $(+7)_{10} + (+4)_{10} = (0111)_2 + (0100)_2 = (\mathbf{0}\ 1011)_2 = (-5)_{10}$
      - E.g. $(-4)_{10} + (-6)_{10} = (1100)_2 + (1010)_2 = (\mathbf{1}\ 0110)_2 = (+6)_{10}$
    - *Observation*: Addition of opposite sign numbers *never* causes overflow.
      - E.g. $(+7)_{10} + (-6)_{10} = (\mathbf{0}111)_2 + (\mathbf{1}010)_2 = (\mathbf{0}001)_2 = (+1)_{10}$ ← *no overflow*
    - *Rule*: If the two numbers are the same sign and the result is the opposite sign, we say that an overflow has occurred.
      - E.g. $(+7)_{10} + (+4)_{10} = (\mathbf{0}111)_2 + (\mathbf{0}100)_2 = (\mathbf{1}011)_2 = (-5)_{10}$ ← *overflowed*
      - E.g. $(-4)_{10} + (-6)_{10} = (\mathbf{1}100)_2 + (\mathbf{1}010)_2 = (\mathbf{0}110)_2 = (+6)_{10}$ ← *overflowed*

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: ______________________________

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

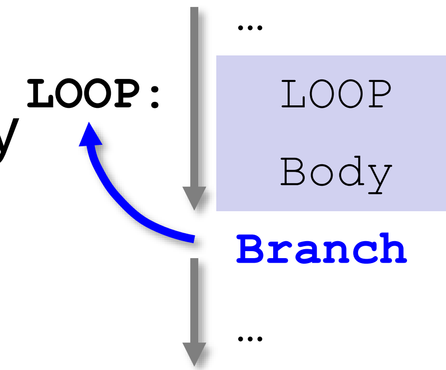
Answer: ______________________________

Outline

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

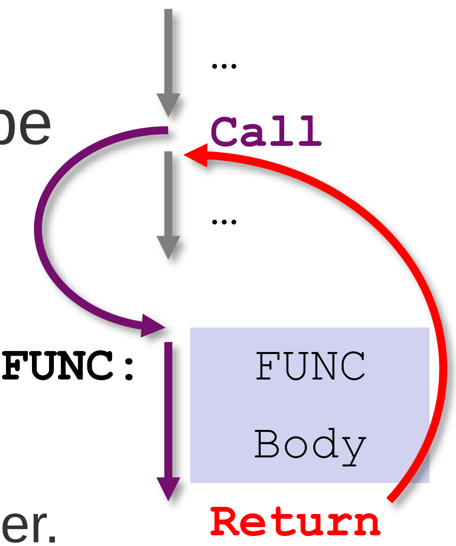
Branch vs. Subroutine

- **Branch:**
  - The instruction jumping to any instruction by loading its memory address into PC.



- It's also common to perform a particular task many times on different values.

- **Subroutine/Function Call**
  - **Subroutine**: a block of instructions that will be executed each time when calling.
  - **Subroutine/Function Call**: when a program *branches* to and back from a subroutine.
    - `Call`: the instruction branching to the subroutine.
    - `Return`: the instruction branching back to the caller.
  - "**Stack**" is essential for subroutine calls.

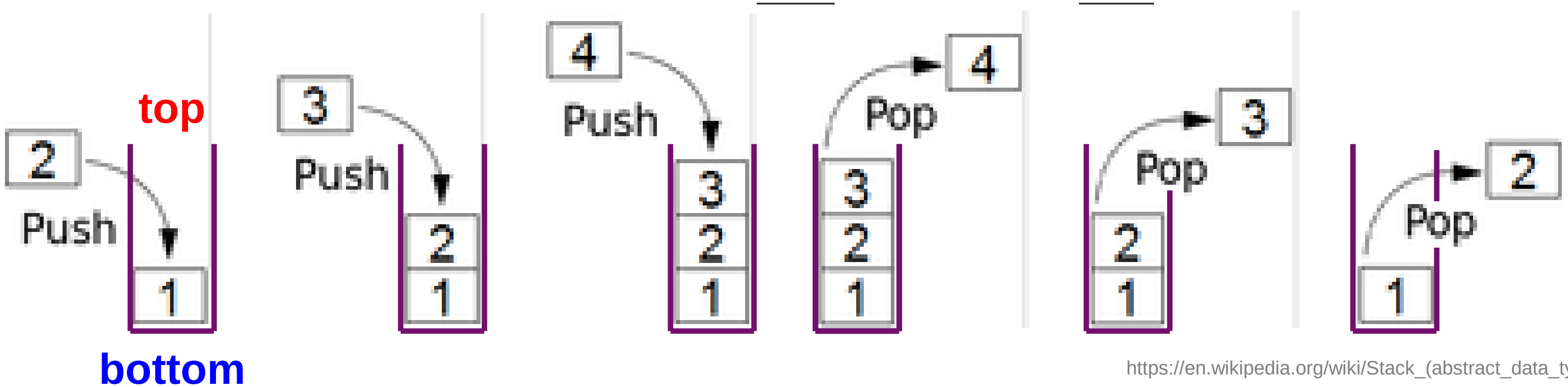


Outline

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

Stack

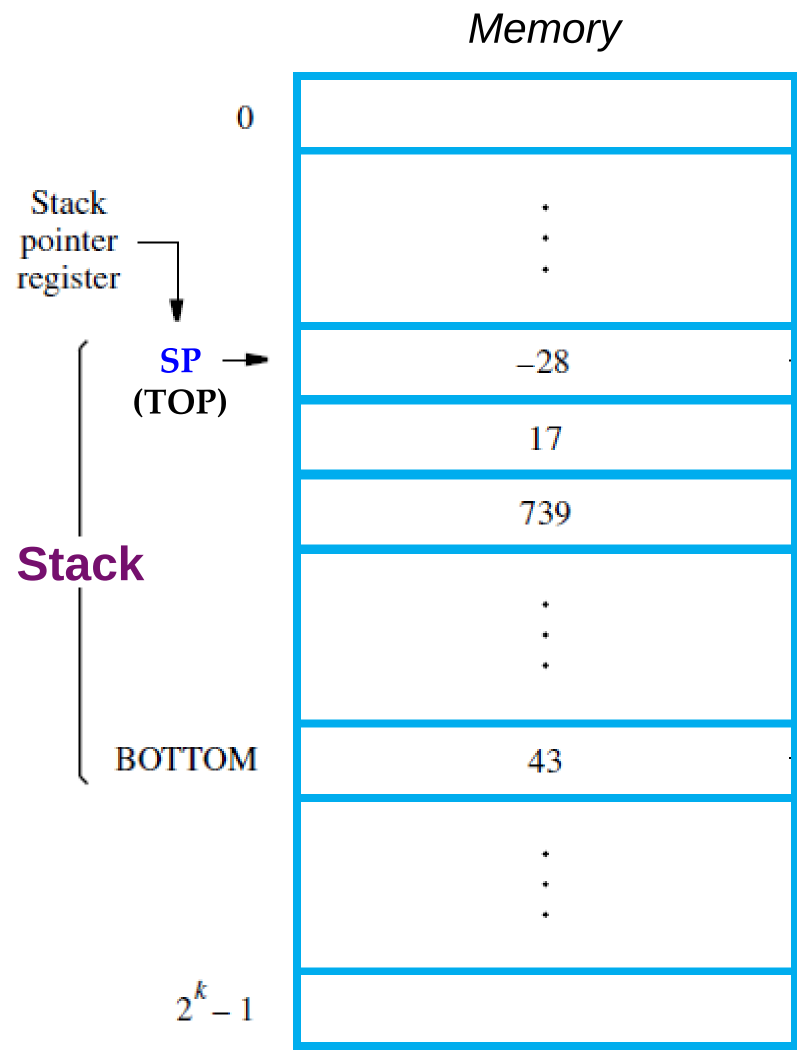
- **Stack** is a list of data elements (usually words):
  - *Elements can only be removed at one end of the list.*
    - This end is called the **top**, and the other end is called the **bottom**.
    - Examples: a stack of coins, plates on a tray, a pile of books, etc.
  - **Push**: Placing a new item 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 Stack (1/2)

- Modern processors usually provide native support to stack (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 Stack (2/2)

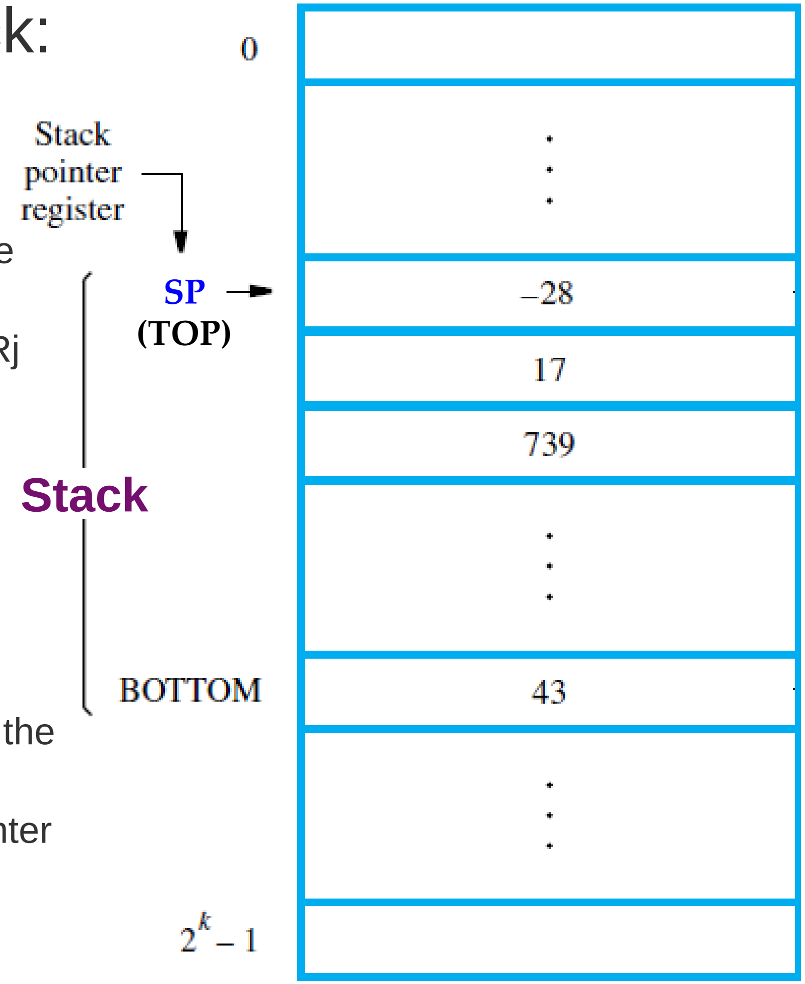
- Given a stack of word data items, and consider a byte-addressable memory with a 32-bit word:
  - **Push** an item in Rj onto the stack:

```
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.



Recall: Additional Addressing Modes

- Most CISC processors have all of the five basic addressing modes—Immediate, Register, Absolute, Indirect, and Index.

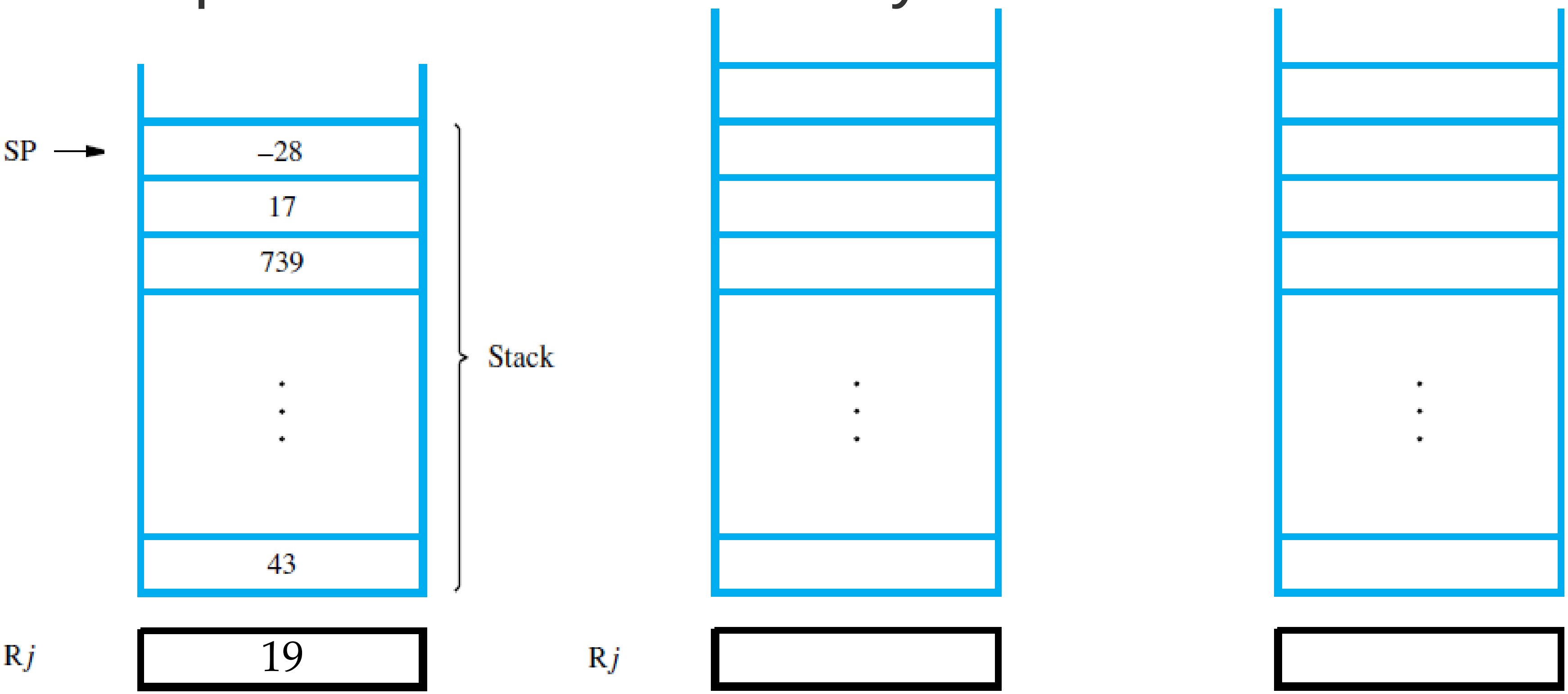
- Three additional addressing modes are often found in CISC processors:

Address Mode	Assembler Syntax	Addressing Function
1*) Autoincrement	$(Ri)+$	$EA = [Ri]$ $Ri = Ri + S$
2*) Autodecrement	$-(Ri)$	$Ri = Ri - S$ $EA = [Ri]$
3*) Relative	$X(PC)$	$EA = [PC] + X$

EA: effective address
X: index value
S: increment/decrement step

Class Exercise 5.4

- Given the contents of the stack and the register Rj as below. Specify the location of SP and the content of register Rj after one ***push*** and one ***pop*** operations are performed consecutively.



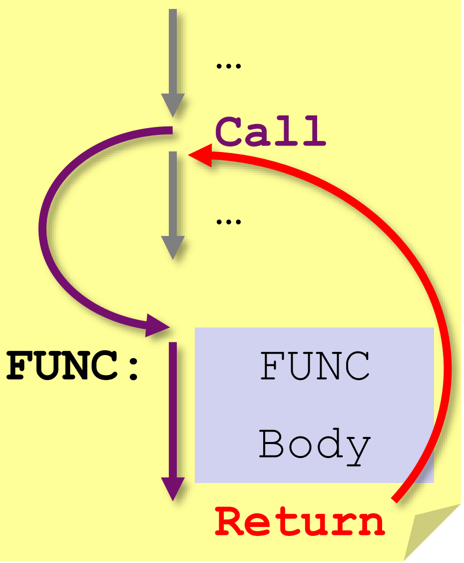
(a) Before **Push & Pop** (b) After **Push** (c) After **Pop**

Outline

- Revisit: Assembly Language Basics
- Program Execution
  - Flow for Generating/Executing a Program
  - Instruction Execution and Sequencing
  - Branching
    - Condition Codes
  - Subroutines
    - Stack
    - 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.



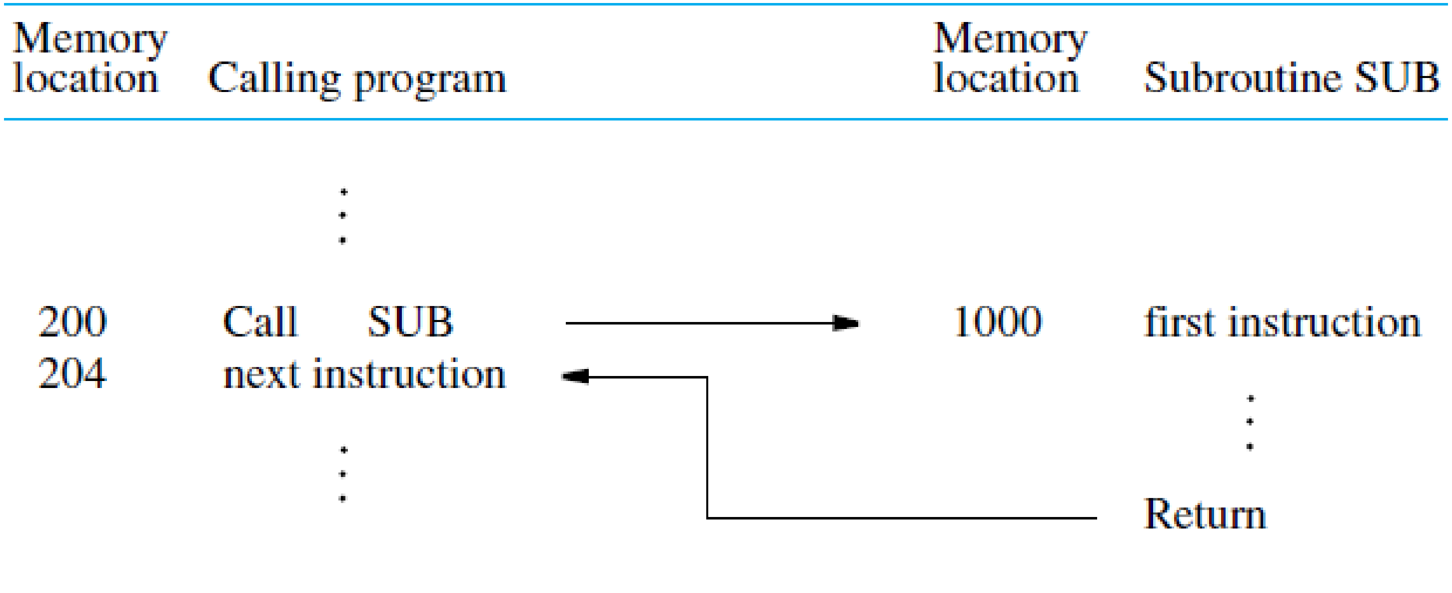
- However, the subroutine may be called from <u>any places</u> in a calling program.
- Thus, provision must be made for **returning** to the appropriate location.
  - That is, the <u>content of the PC</u> must be saved by the Call instruction to enable correct return to the calling program.

Subroutine Linkage

- **Subroutine Linkage** method: the way makes it possible to `Call` and `Return` from subroutines.

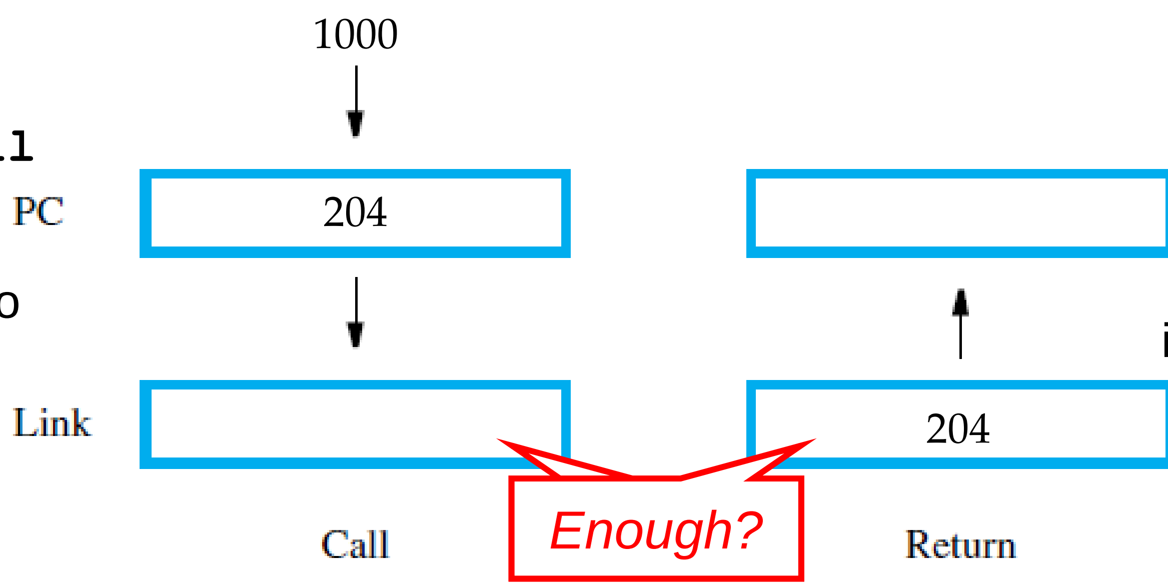
- The simplest method: saving the return address in a special processor register called the link register.
  - The **Call** instruction can be implemented as a special *branch* instruction:
    - ① Keep the content of the PC in the link register.
    - ② Branch to the target address specified by `Call` instruction.
  - The **Return** instruction can be implemented as a special *branch* instruction as well:
    - Branch to the address kept in the link register by `Return` instruction.

Example of Subroutine Linkage



② Branch to the target address specified by **`Call`**

① Keep [PC] into the link register.



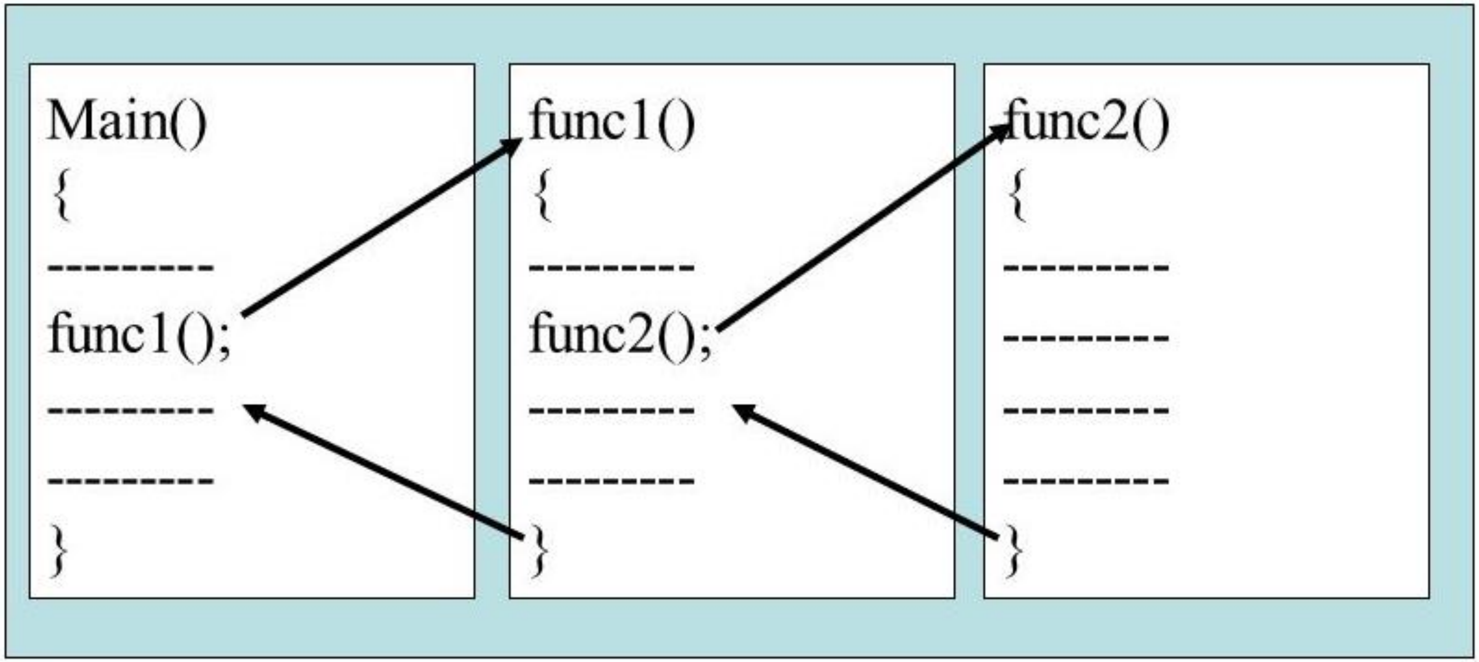
Branch back to the address kept in the link register by **`Return`**

Outline

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

Subroutine Nesting (1/3)

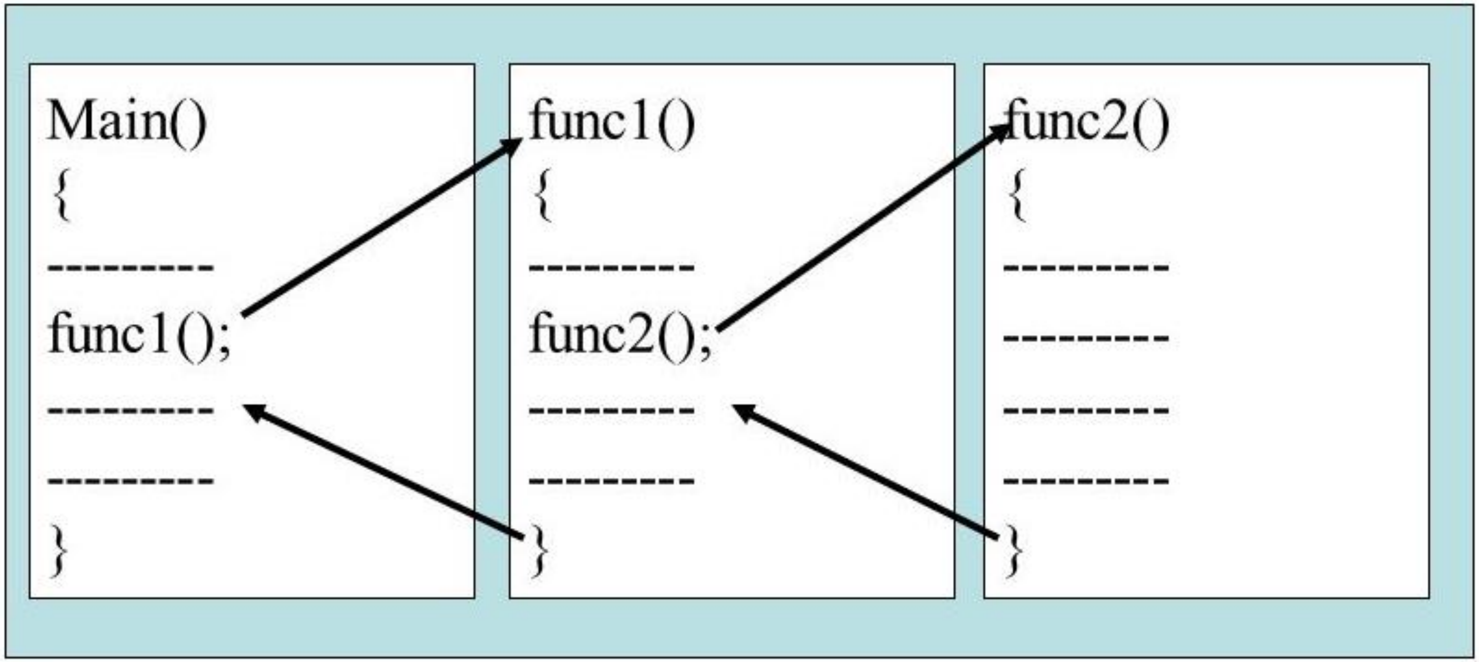
- **Subroutine Nesting**: One subroutine calls another subroutine or itself (i.e. recursion).
  - If the return address of the second call is also stored in the link register, the first return address will be lost … ERROR!
  - Subroutine nesting can be carried out to ANY DEPTH …



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

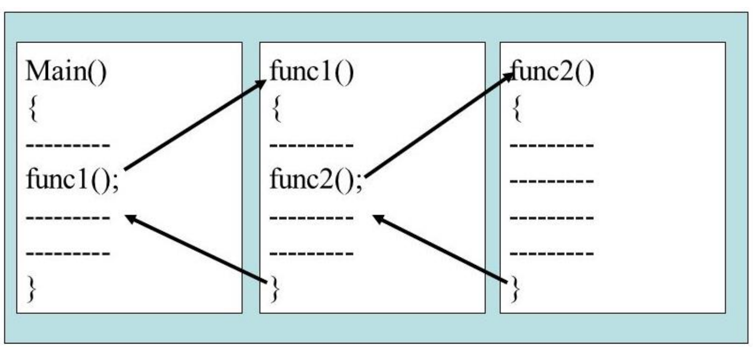
Subroutine Nesting (2/3)

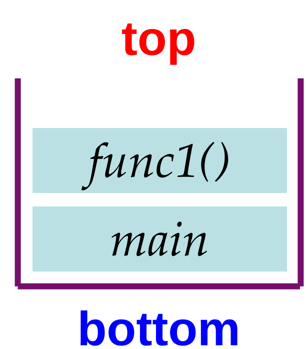
- *Observation*: The return address needed for the first return is the last one generated in the nested calls.
  - That is, return addresses are generated and used in a last-in–first-out (LIFO) order.



Subroutine Nesting (3/3)

- Processor stack is useful to store subroutine linkage:
  - **`Call`** instruction:
    - ~~① Store the contents of the PC in the link register~~
    - → ❶ Push the contents of the PC to the processor stack
    - ② Branch to the target address specified by **`Call`** instruction.
    - → ❷ (*Unchanged*)
  - **`Return`** instruction:
    - ~~• Branch to the address contained in the link register~~
    - → Branch to the address popped out from the processor stack



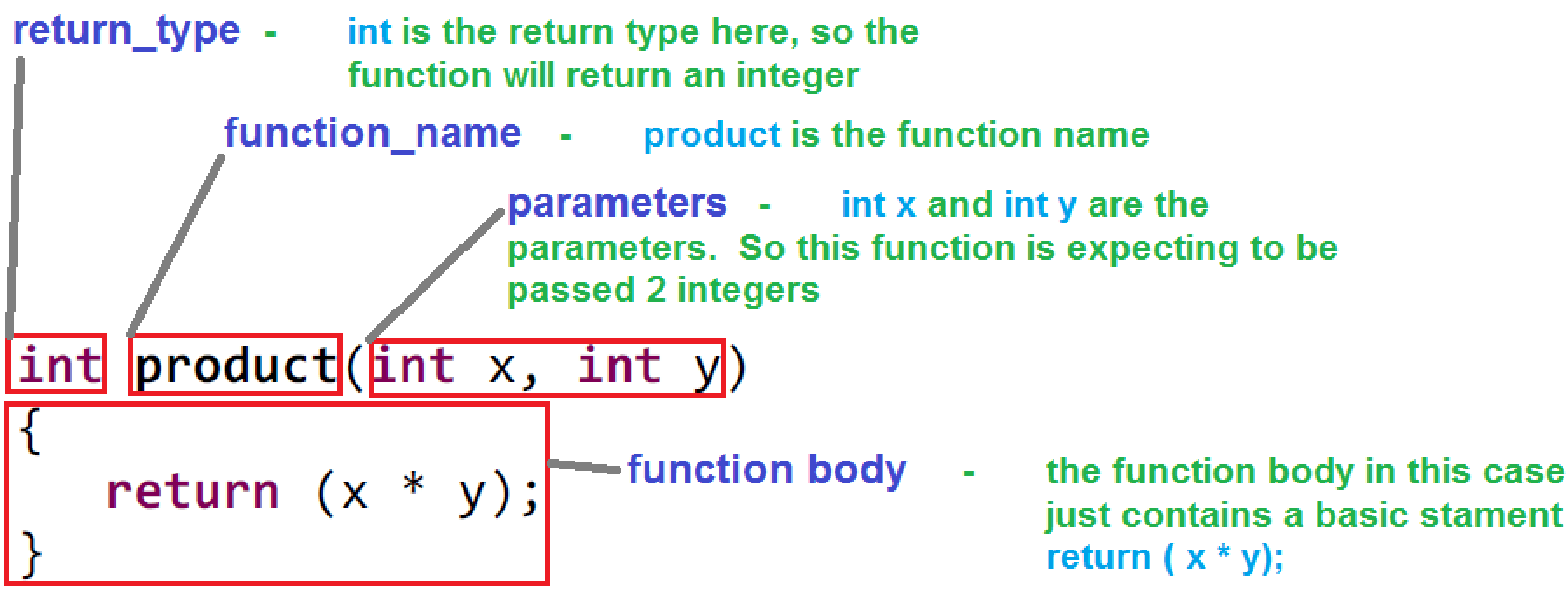


Outline

- Revisit: Assembly Language Basics
- Program Execution
  – Flow for Generating/Executing a Program
  – Instruction Execution and Sequencing
  – Branching
    - Condition Codes
  – Subroutines
    - Stack
    - 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.
- This program can be implemented as a subroutine:
  - R2 & R4 pass the list size & the address of the first num;
  - R3 passes back the sum computed by the subroutine.

Calling Program

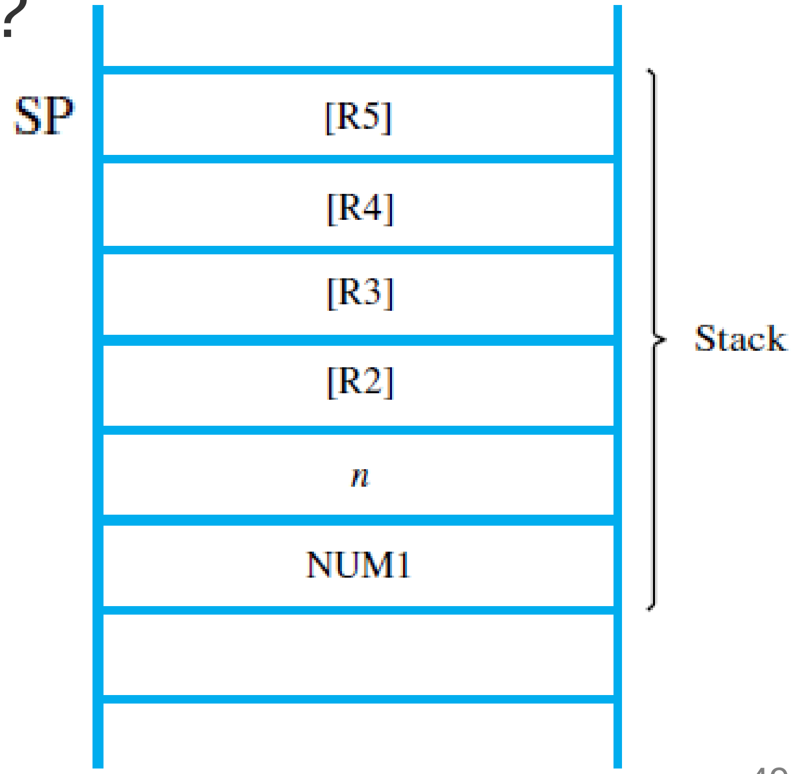
	Load	R2, N	Parameter 1 is list size.
	Move	R4, addr NUM1	Parameter 2 is list location.
	Call	LISTADD	Call subroutine.
	Store	R3, SUM	Save result.
	⋮		

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 via Stack (1/3)

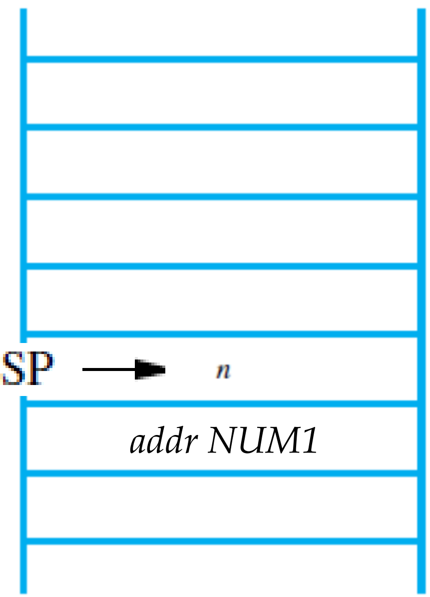
- What if there are more parameters than registers?
- What if the subroutine calls itself (recursion)?
- The processor stack once again provides a good scheme to pass (an arbitrary number of) parameters.
- What can we push onto stack?
  ① We shall **push** *all parameters to be computed* onto stack.
  ② We shall also **push** *the contents of all "to-be-used" registers* onto the stack.
  ③ We may also **push** *the computed result* before the return to the calling program.



Parameter Passing via Stack (2/3)

- Consider the program that adds a list of *n* numbers. It now uses the processor stack for parameter passing.

Calling Program



Move	R2, **addr NUM1**	Push parameters onto stack.
Subtract	SP, SP, #4	
Store	R2, (SP)	
Load	R2, N	
Subtract	SP, SP, #4	
Store	R2, (SP)	
Call	LISTADD	Call subroutine
Load	R2, 4(SP)	Get the result from the stack
Store	R2, SUM	and save it in SUM.
Add	SP, SP, #8	Restore top of stack

① **push** *all parameters to be computed* onto stack

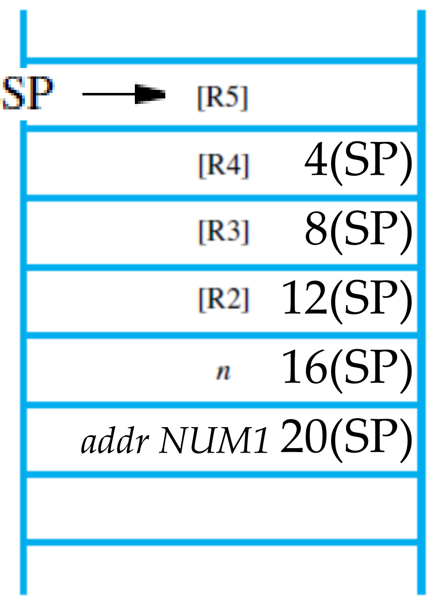
Parameter Passing via Stack (3/3)

Subroutine

② **push** *"to-be-used" registers*

③ **push** *the computed result*

```
LISTADD: Subtract          SP, SP, #16    Save registers
         Store             R2, 12(SP)
         Store             R3, 8(SP)
         Store             R4, 4(SP)
         Store             R5, (SP)
         Load              R2, 16(SP)     Initialize counter to n.
         Load              R4, 20(SP)     Initialize pointer to the list.
         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
         Store             R3, 20(SP)     Put result in the stack.
         Load              R5, (SP)       Restore registers.
         Load              R4, 4(SP)
         Load              R3, 8(SP)
         Load              R2, 12(SP)
         Add               SP, SP, #16
         Return                           Return to calling program.
```



Class Exercise 5.5

- In the example program that uses the processor stack for parameter passing, the result is passed back to the calling program by **`Store R3, 20(SP)`**. To pass back the result, can we use the instruction **`Store R3, 16(SP)`** instead?

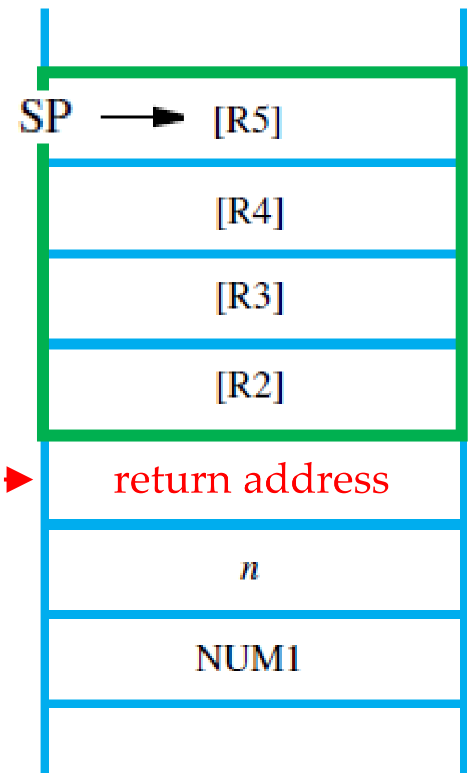
Subroutine Linkage & Para. Passing

- Recall: Processor stack is also useful to store subroutine linkage (i.e., return address).
  - [PC] is pushed onto stack.

Calling Program

```
Move        R2, addr NUM1
Subtract    SP, SP, #4
Store       R2, (SP)
Load        R2, N
Subtract    SP, SP, #4
Store       R2, (SP)
Call        LISTADD
Load        R2, 4(SP)
Store       R2, SUM
Add         SP, SP, #8
```

Any problems?



Subroutine

```
LISTADD:  Subtract    SP, SP, #16
          Store       R2, 12(SP)
          Store       R3, 8(SP)
          Store       R4, 4(SP)
          Store       R5, (SP)
          Load        R2, 16(SP)
          Load        R4, 20(SP)
          Clear       R3
LOOP:     Load        R5, (R4)
          Add         R3, R3, R5
          Add         R4, R4, #4
          Subtract    R2, R2, #1
          Branch>0    LOOP
          Store       R3, 20(SP)
          Load        R5, (SP)
          Load        R4, 4(SP)
          Load        R3, 8(SP)
          Load        R2, 12(SP)
          Add         SP, SP, #16
          Return
```

Any problems?

Parameter Passing by Value / Reference

- What kind of parameters can we pass?
- **Passing by Value**
  - The actual number 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 number.

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

Class Exercise 5.6

- Consider the calling program that calls the subroutine `LISTADD` to add a list of n numbers, in which
  - The size n is stored in memory location/address **`N`**, and
  - **`NUM1`** is the memory address for the first number.
- Are **`N`** and **`NUM1`** passed as values or references?

LABEL	OPCODE	OPERAND	COMMENT
	`Move`	`R2, addr NUM1`	*Push parameters onto stack*
	`Subtract`	`SP, SP, #4`	
	`Store`	`R2, (SP)`	
	`Store`	`R2, N`	
	`Subtract`	`SP, SP, #4`	
	`Store`	`R2, (SP)`	
	`Call`	`LISTADD`	*Call subroutine*
	`Load`	`R2, 4(SP)`	*Get the result from the stack*
	`Store`	`R2, SUM`	*Store the result in SUM*
	`Add`	`SP, SP, #8`	*Restore top of stack*

Summary

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