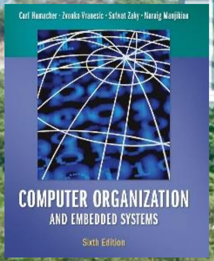
香港中文大學
The Chinese University of Hong Kong

CSCI2510 Computer Organization

Lecture 05: Program Execution

Ming-Chang YANG

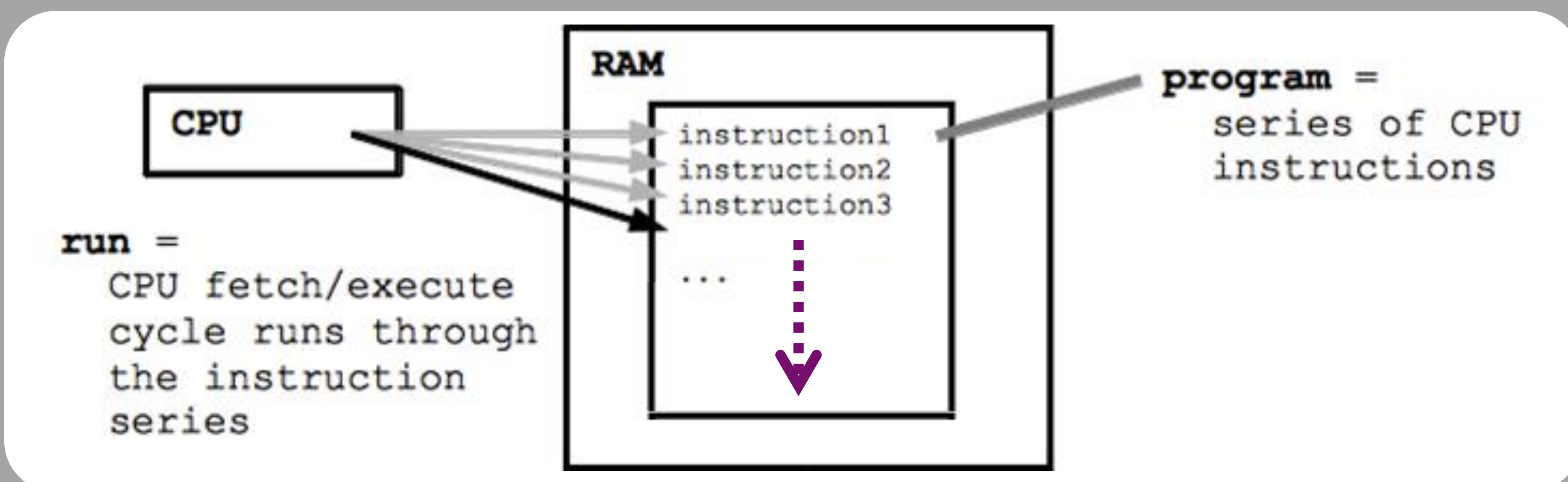
mcyang@cse.cuhk.edu.hk



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

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

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
```

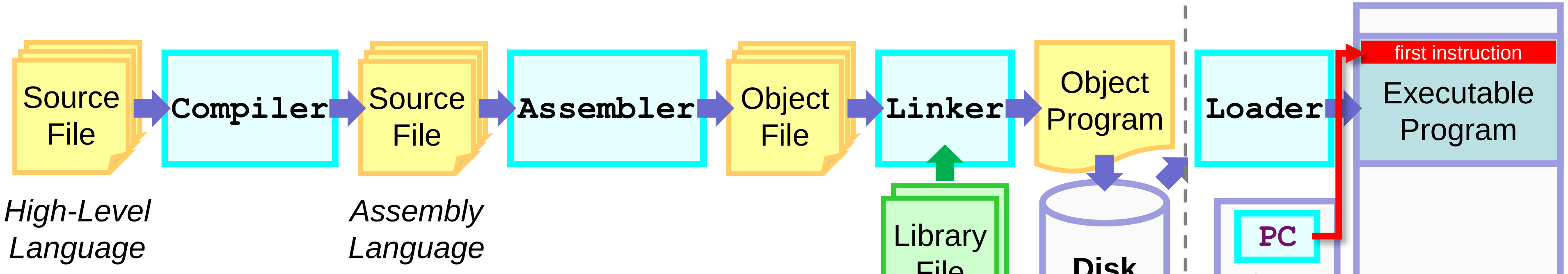
https://gerardnico.com/code/lang/machine
https://clip2art.com/explore/Boy%20clipart%20teacher/

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

---

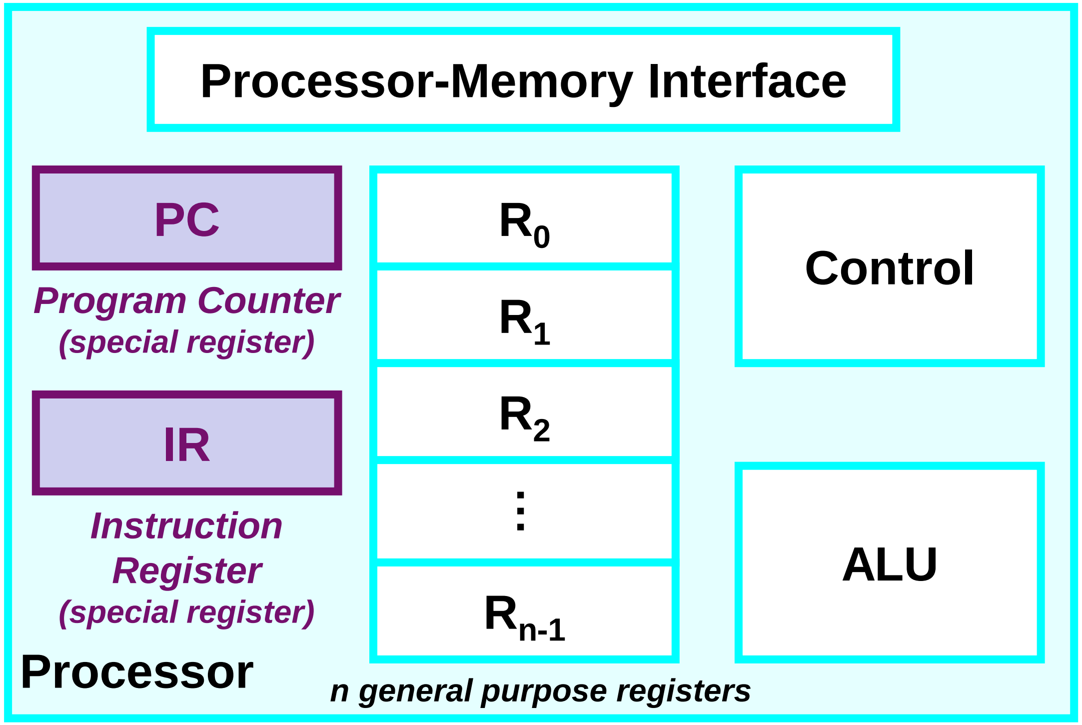
- **Loader**: Load the program into memory and load the address of the first instruction into program counter (PC)



Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

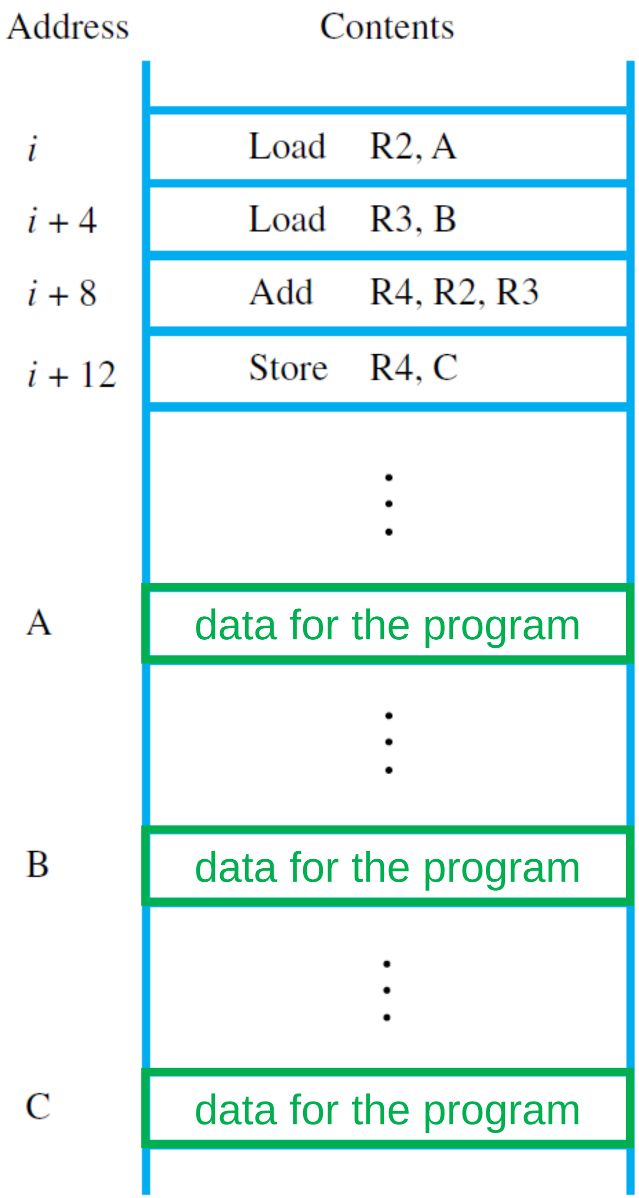
Program Counter & Instruction Register



- To direct the instruction execution and sequencing, two special registers are needed:
  - **Program Counter (PC)** contains the memory address of the **NEXT** instruction to be fetched and executed.
  - **Instruction Register (IR)** holds the **CURRENT** instruction that is being executed.

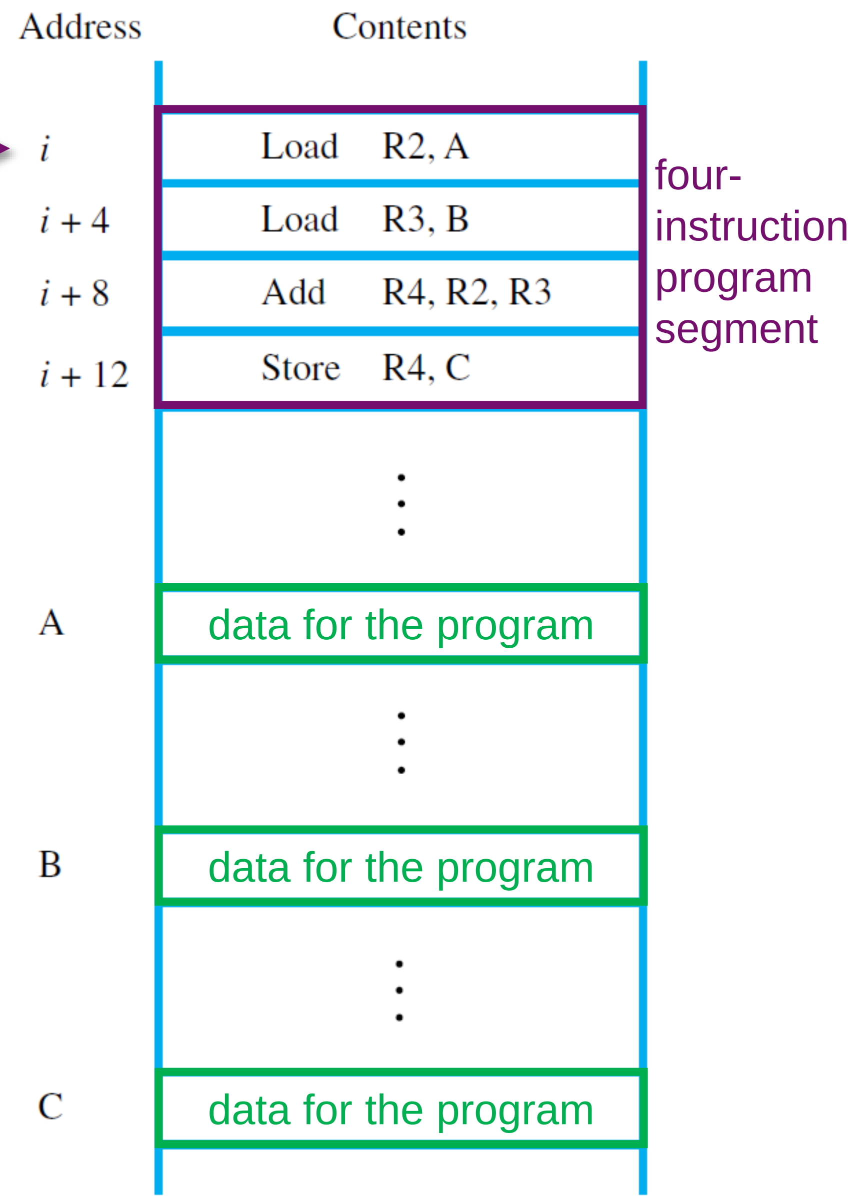
Instruction Execution & Sequencing (1/3)

- 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.



Instruction Execution & Sequencing (3/3)

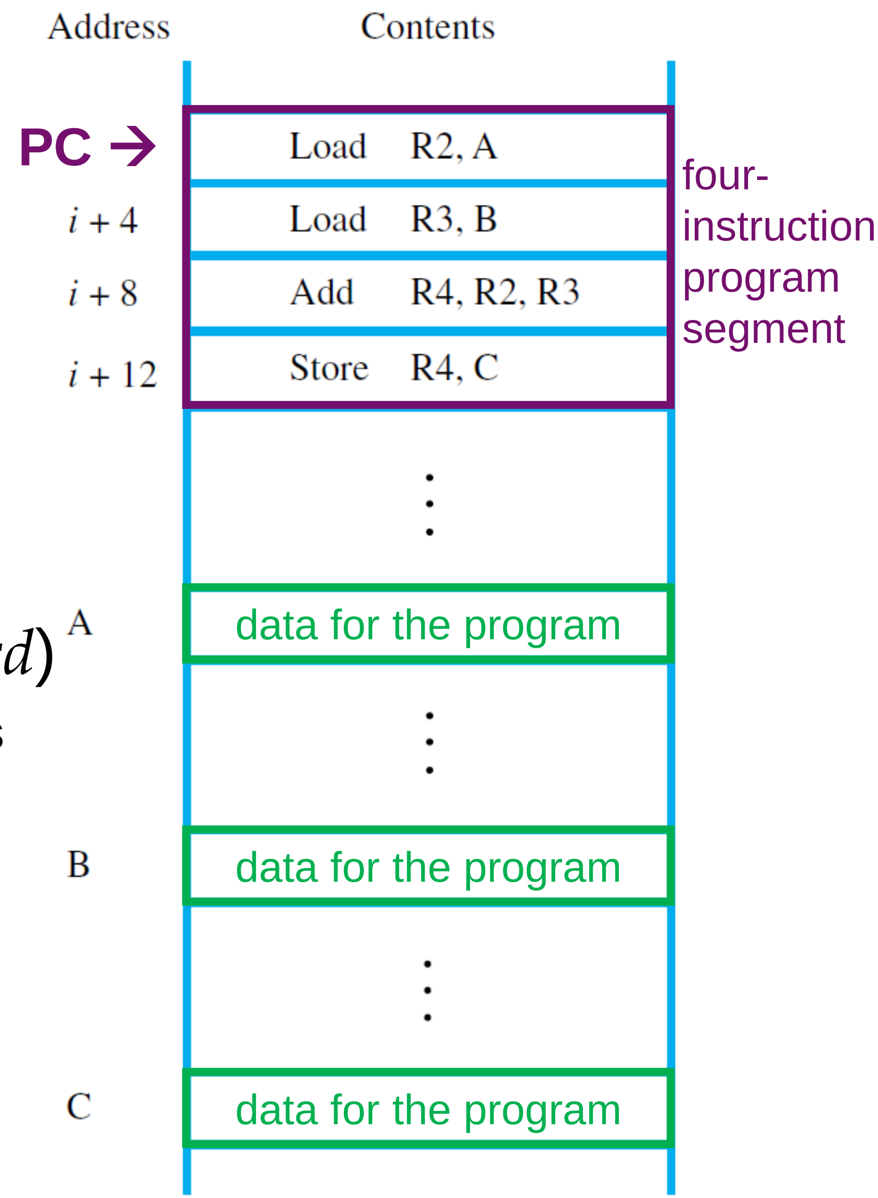
- **Straight-Line Sequencing:**
  - CPU fetches and executes instructions indicated by PC, one at a time, in the order of increasing addresses.

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:

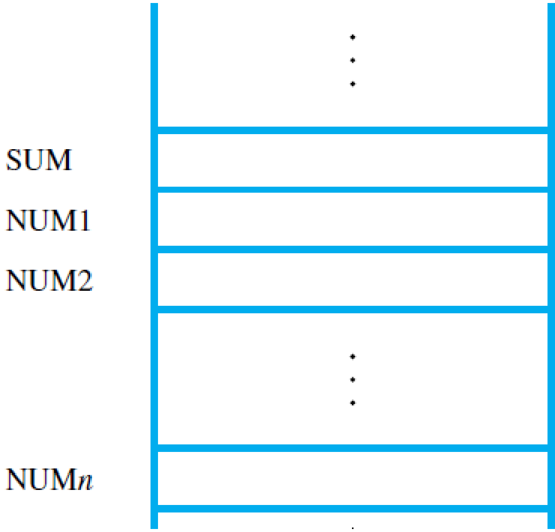
- Interpret (or decode) IR
- Perform the operation



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

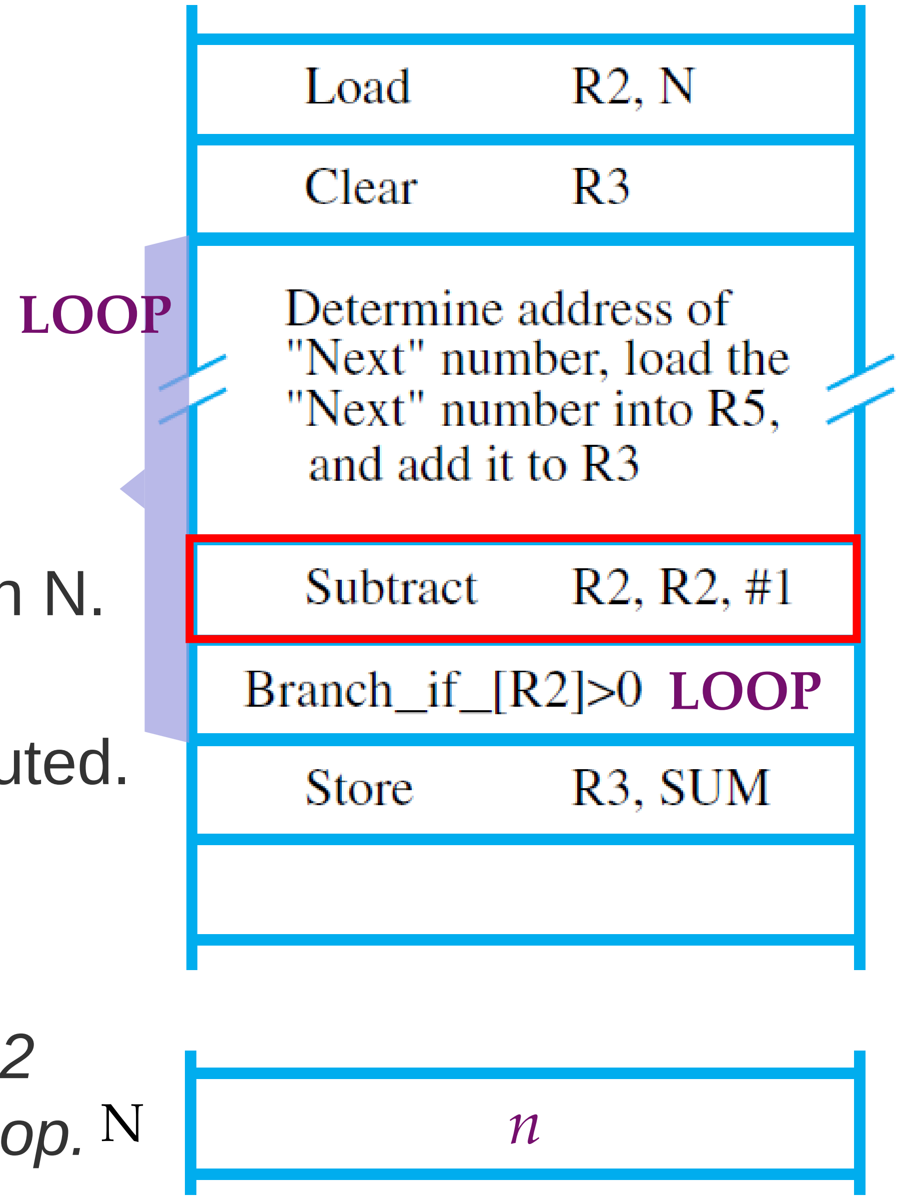
- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

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,

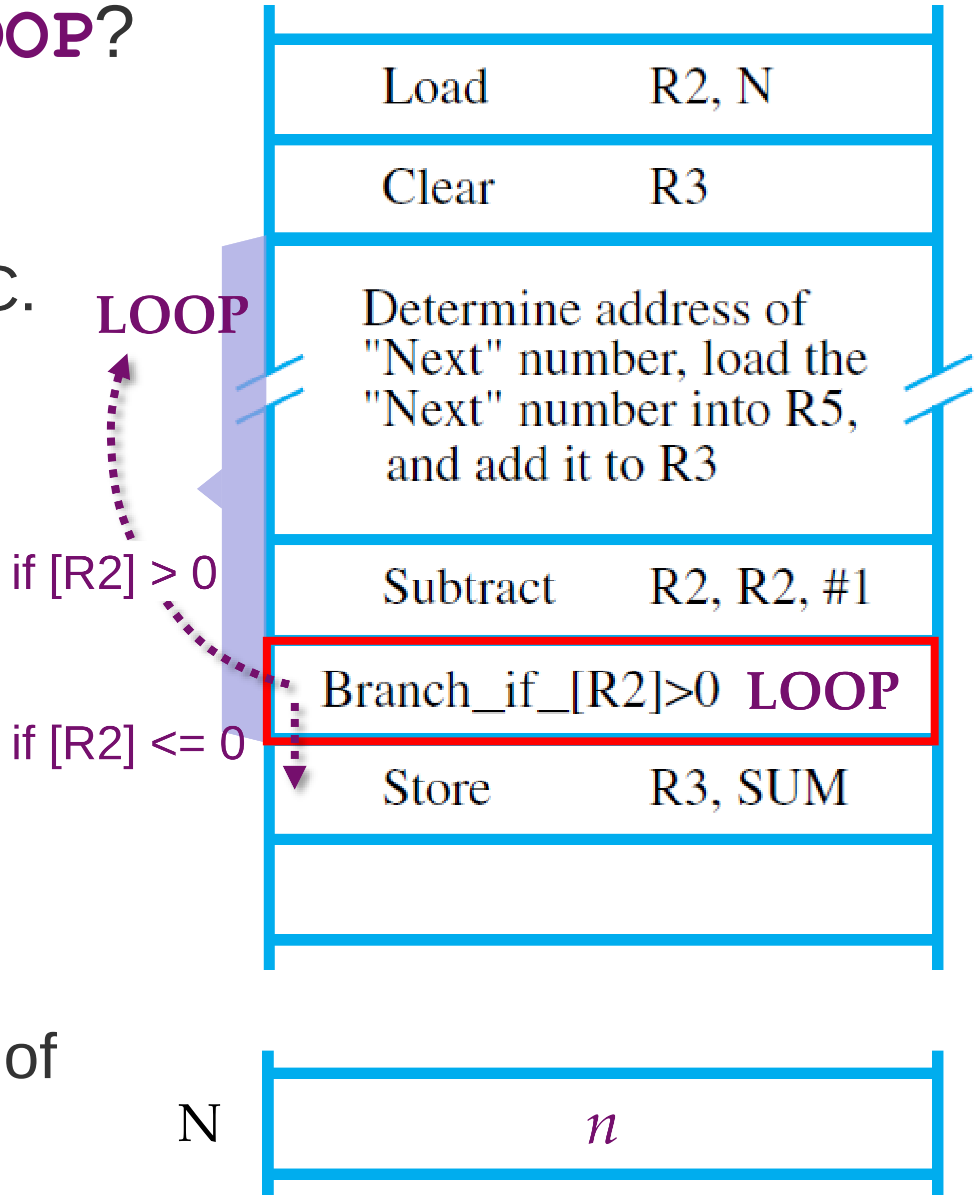
`Subtract R2, R2, #1`

  - *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

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

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: ______________________________

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

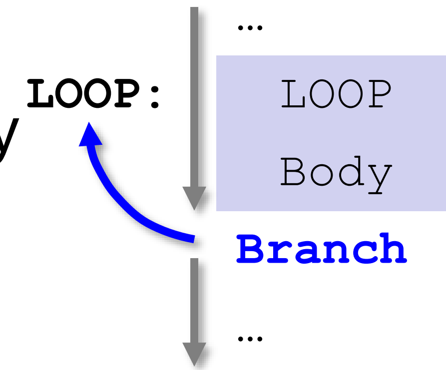
  Answer: ______________________________

Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

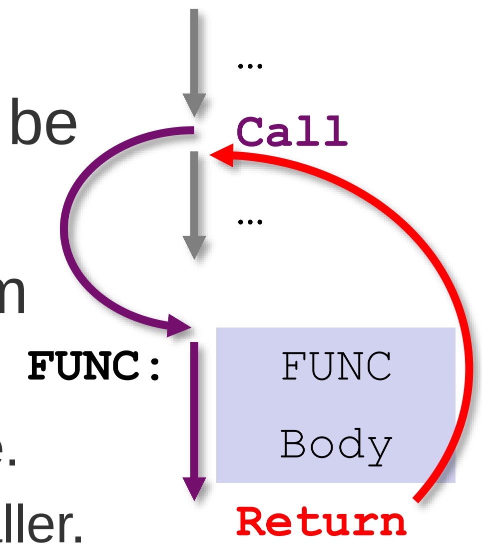
Branch vs. Subroutine

- **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.

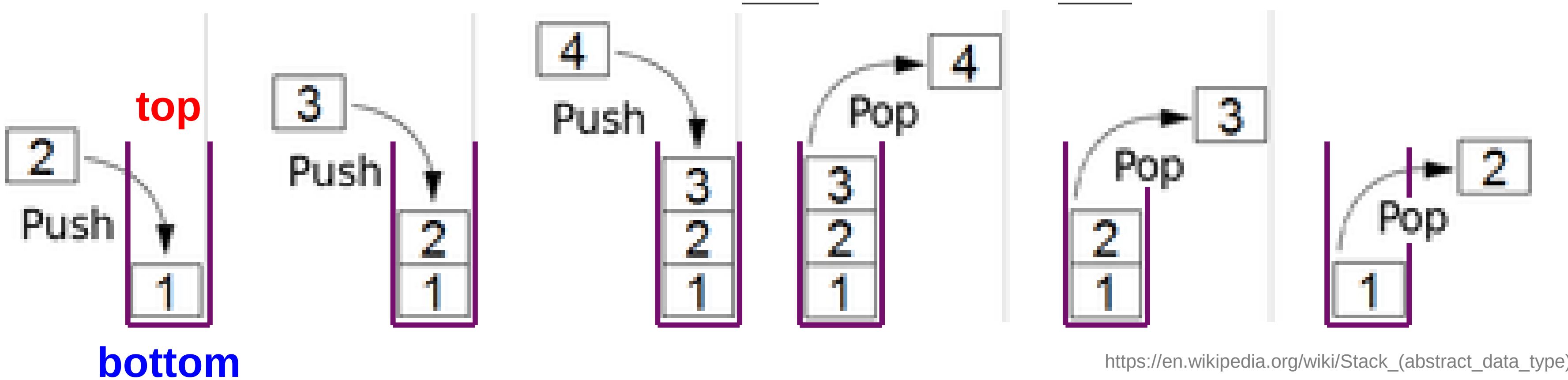


Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

Stack

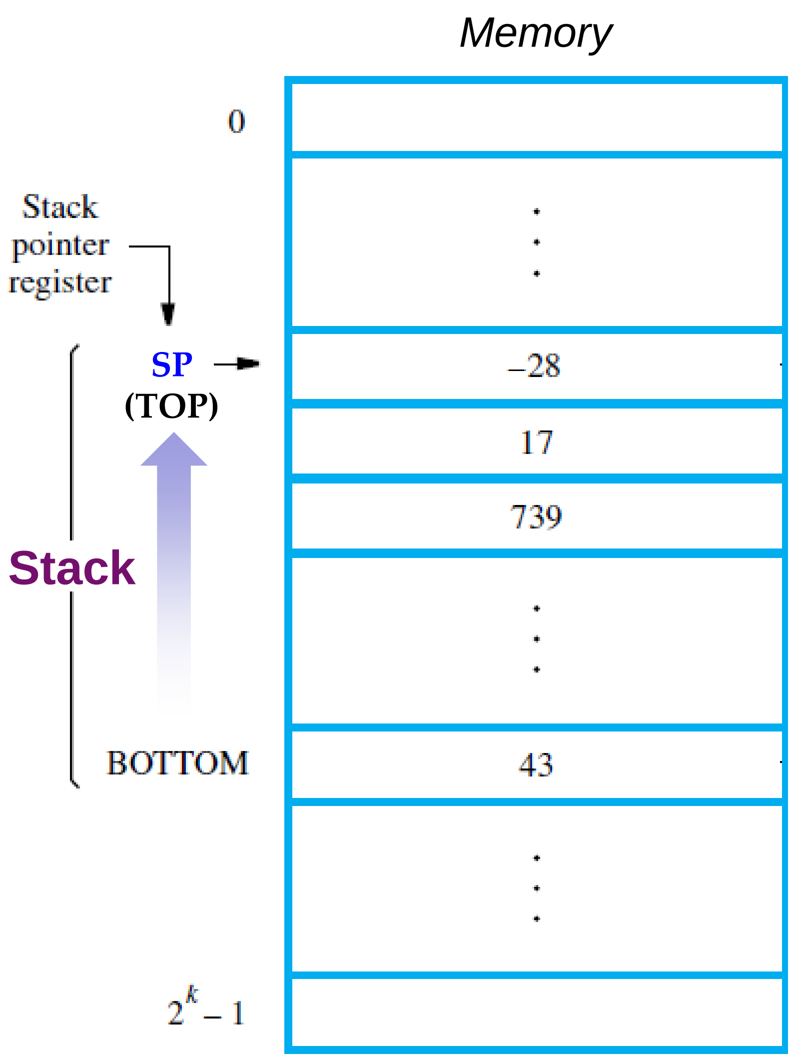
- **Stack** is a list of data elements (usually words):
  - *Elements can only be removed at one end of the list.*
    - This end is called the **top**, and the other end is called the **bottom**.
    - Examples: a stack of coins, plates on a tray, a pile of books, etc.
  - **Push**: Placing a new item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

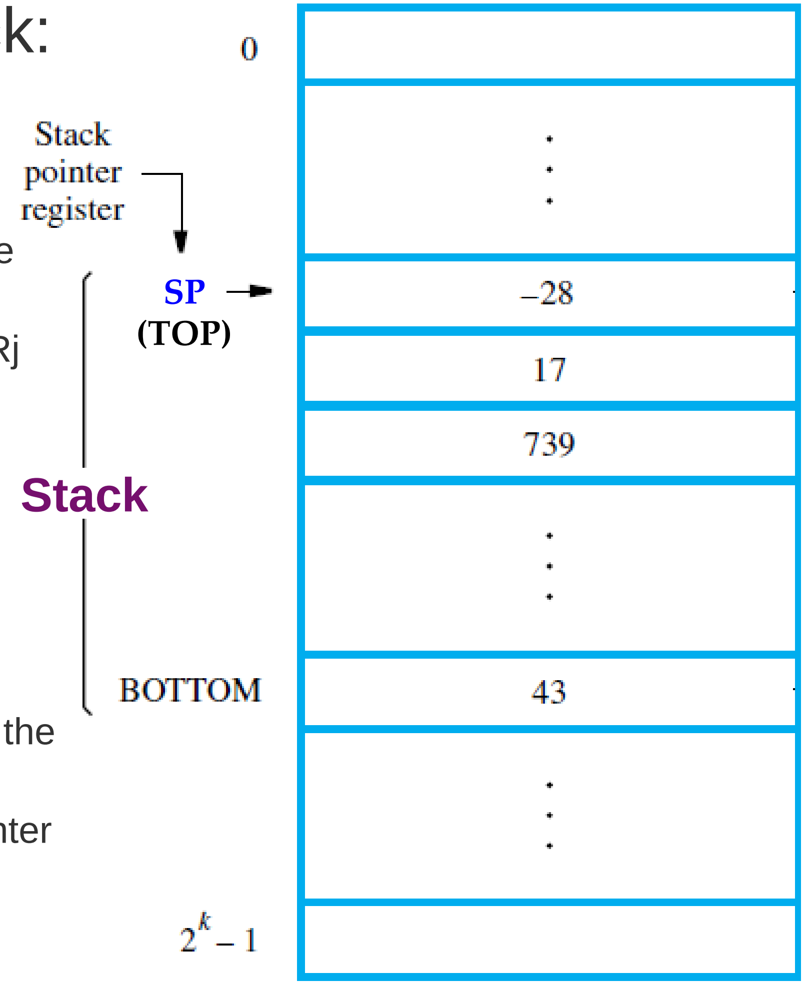
```
Subtract    SP, SP, #4
Store       Rj, (SP)
```

    - 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

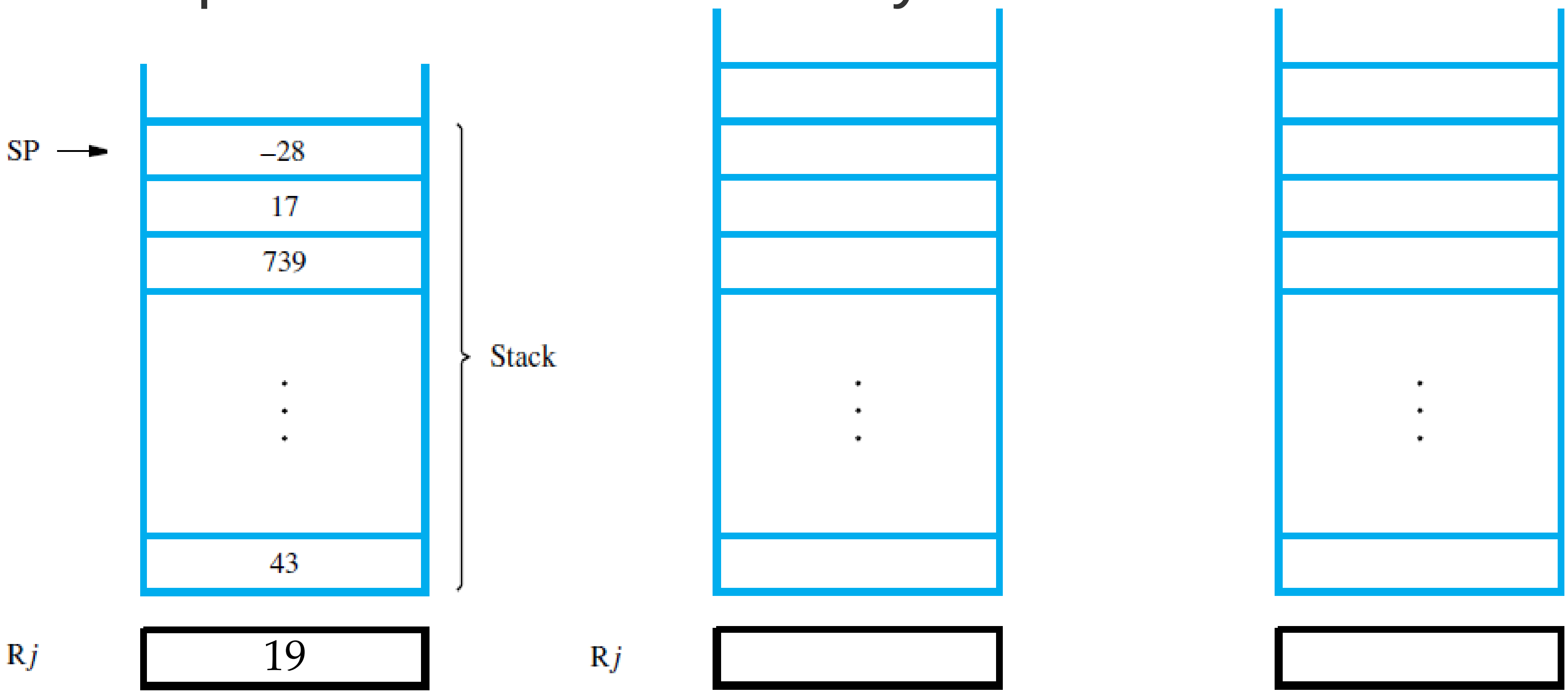
- Most CISC processors have all of the five basic addressing modes—Immediate, Register, Absolute, Indirect, and Index.
- Three additional addressing modes are often found in CISC processors:

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.



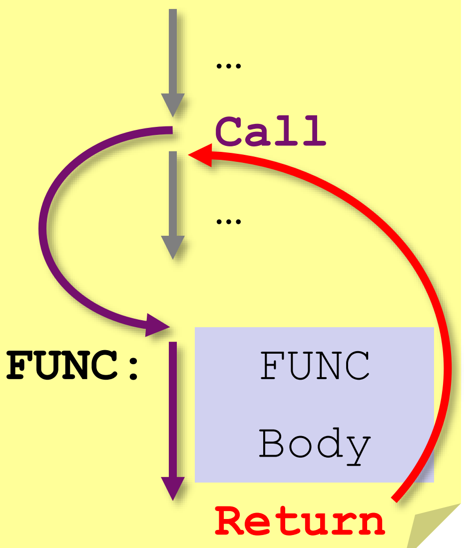
(a) Before **Push & Pop** (b) After **Push** (c) After **Pop**

Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

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.



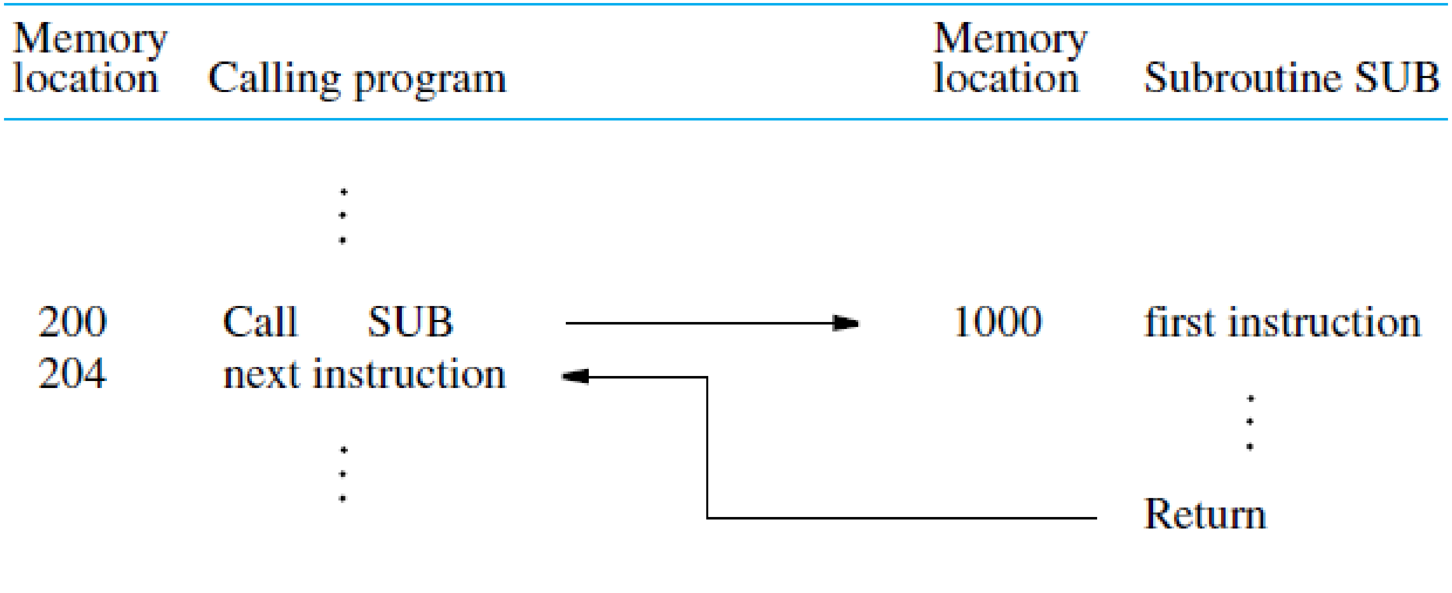
- However, the subroutine may be called from <u>any places</u> in a calling program.
- Thus, provision must be made for **returning** to the appropriate location.
  - That is, the <u>content of the PC</u> must be saved by the Call instruction to enable correct return to the calling program.

Subroutine Linkage

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

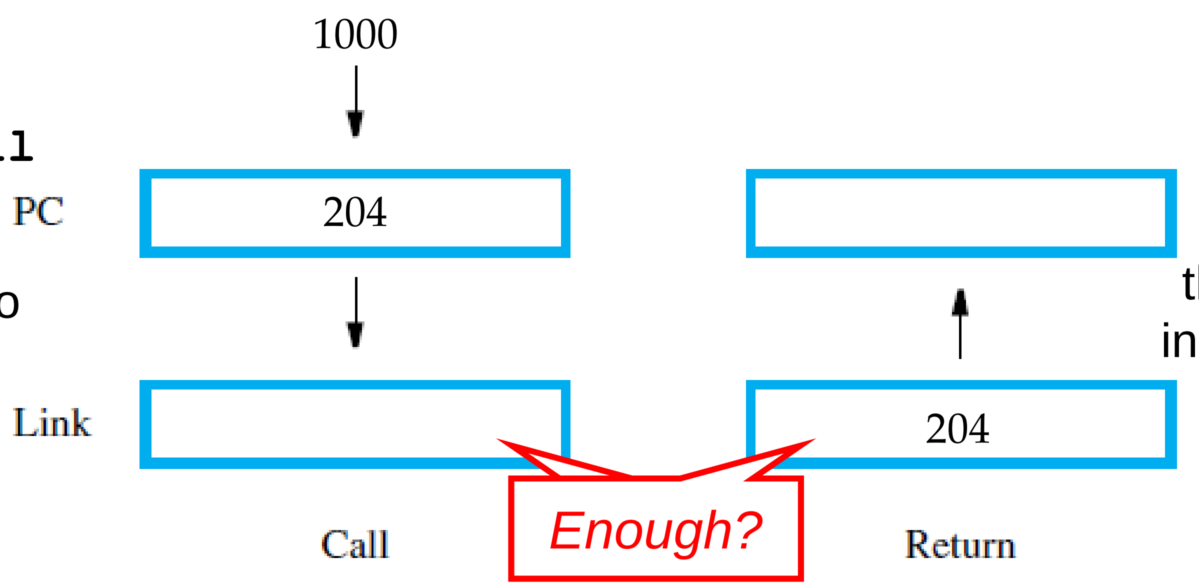
- The simplest method: saving the return address in a special processor register called the link register.
  - The **Call** instruction can be implemented as a special *branch* instruction:
    - ① Keep the content of the PC in the link register.
    - ② Branch to the target address specified by `Call`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



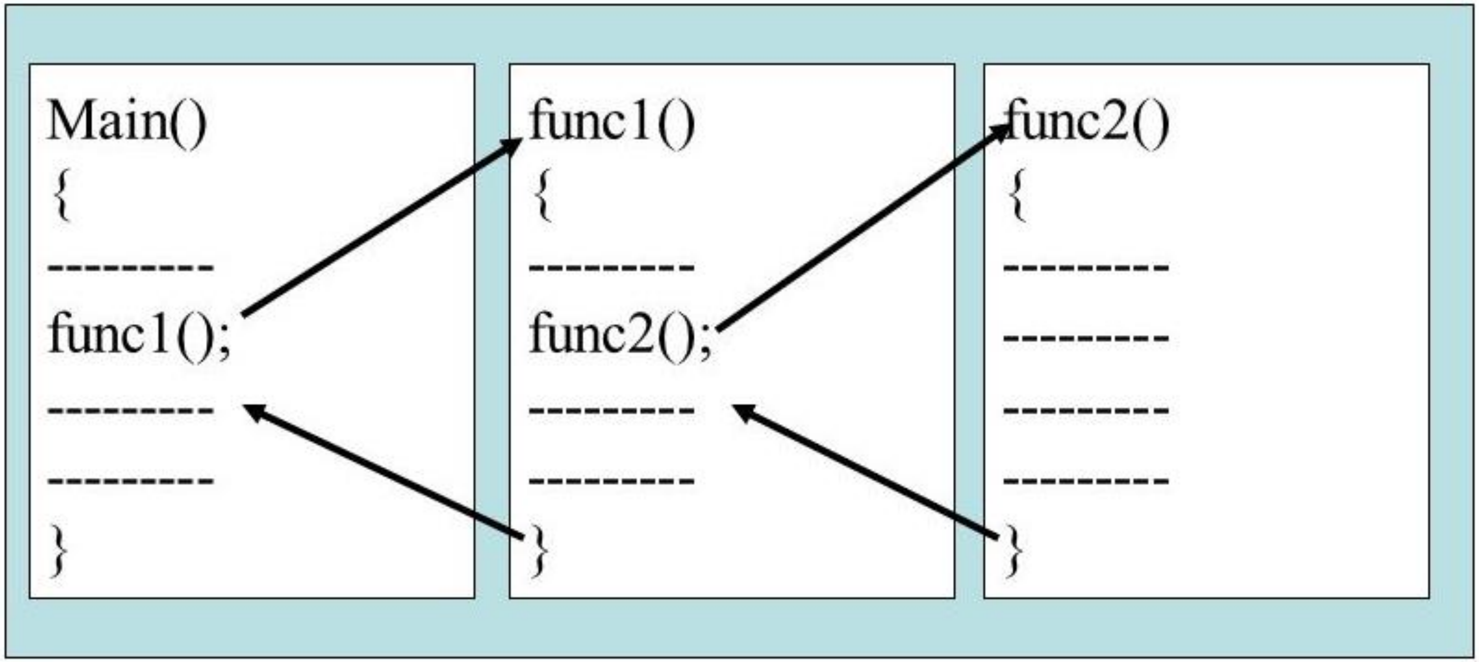
Branch back to the address kept in the link register by **`Return`**

Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

Subroutine Nesting (1/3)

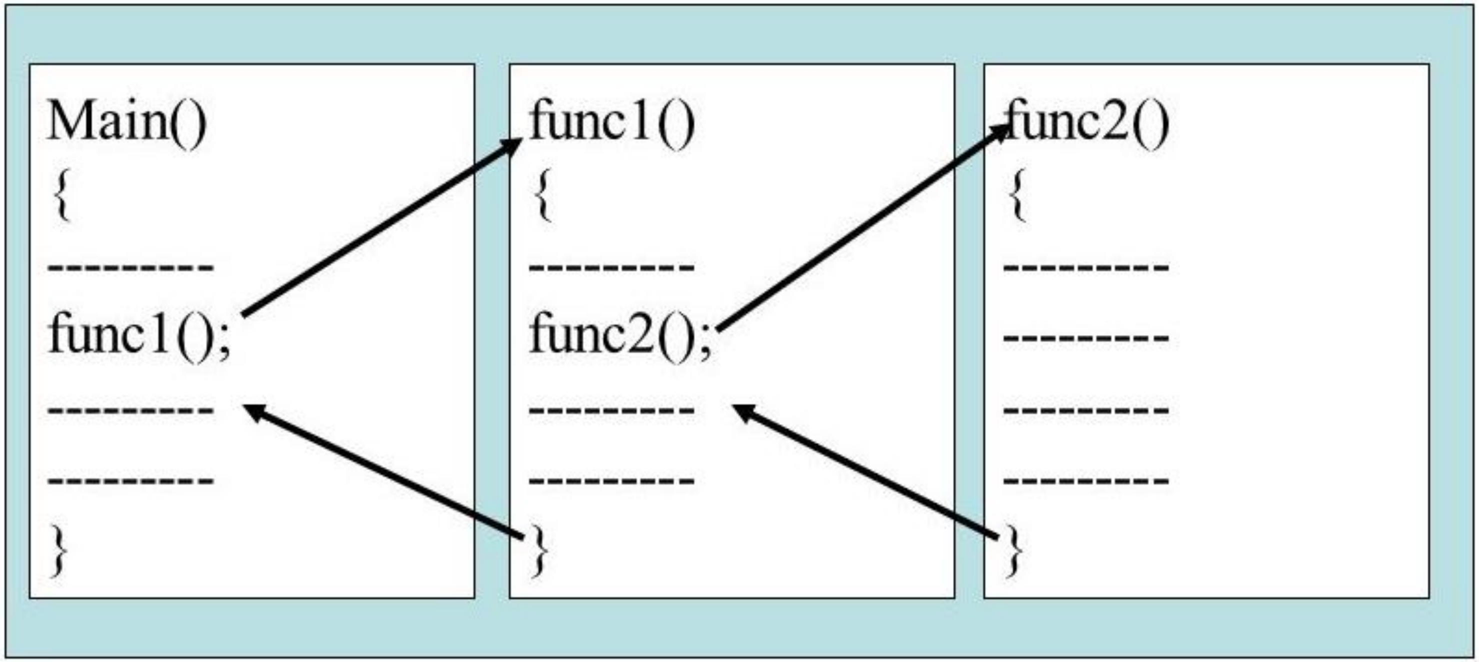
- **Subroutine Nesting**: One subroutine calls another subroutine or itself (i.e., recursion).
  - If the return address of the second call is also stored in the link register, the first return address will be lost … ERROR!
  - Subroutine nesting can be carried out to ANY DEPTH …



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

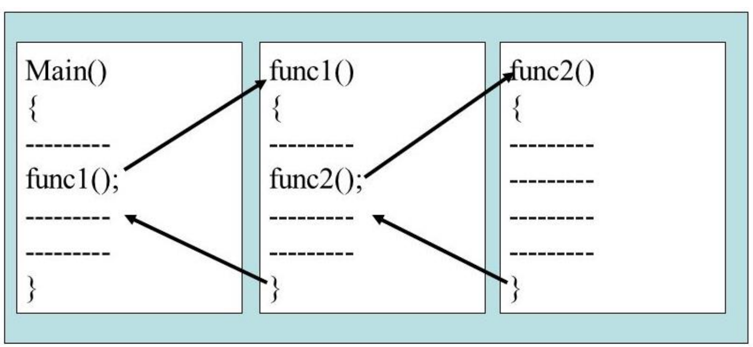
Subroutine Nesting (2/3)

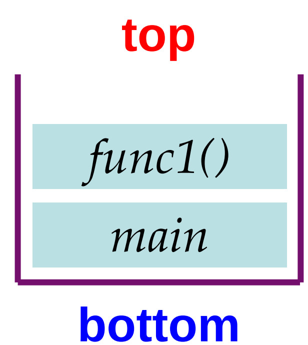
- *Observation*: The return address needed for the first return is the last one generated in the nested calls.
  - That is, return addresses are generated and used in a last-in–first-out (LIFO) order.



Subroutine Nesting (3/3)

- Processor stack is useful to store subroutine linkage:
  - **`Call`** instruction:
    - ① Keep the content of the PC in the link register.
    - ② Branch to the target address specified by **`Call`** instruction.
    - **NEW →** ③ Push the contents of the link register to the processor stack.
  - **`Return`** instruction:
    - **NEW →** ① Pop out the saved subroutine linkage from the processor stack to restore the link register.
    - ② Branch to the address kept in the link register by **`Return`** instruction.



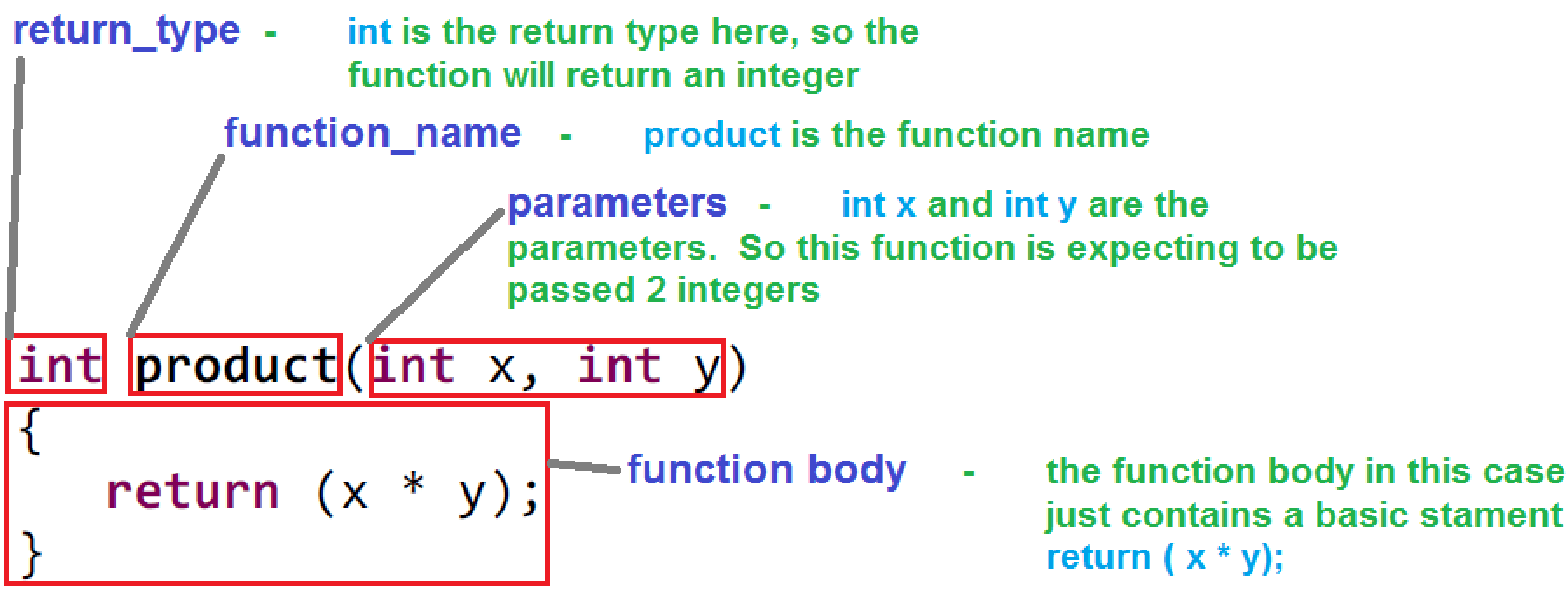


Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  – Condition Codes
- Subroutines
  – Stack
  – Subroutine Linkage
  – Subroutine Nesting
  – Parameter Passing
  – The Stack Frame

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.
  - Let's revisit the program that adds up a list of numbers:
    - **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 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 5.5

- 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.

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.
  ⋮
```

- Are **`N`** and **`NUM1`** passed as <u>values</u> or <u>references</u>?

Issues of Para. Passing via Registers?

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.

- What if the subroutine is going to use R2 and R4, or any other registers that contain useful information to the calling program?
- What if the subroutine calls itself (i.e., recursion)?
- What if there are more parameters than #registers?

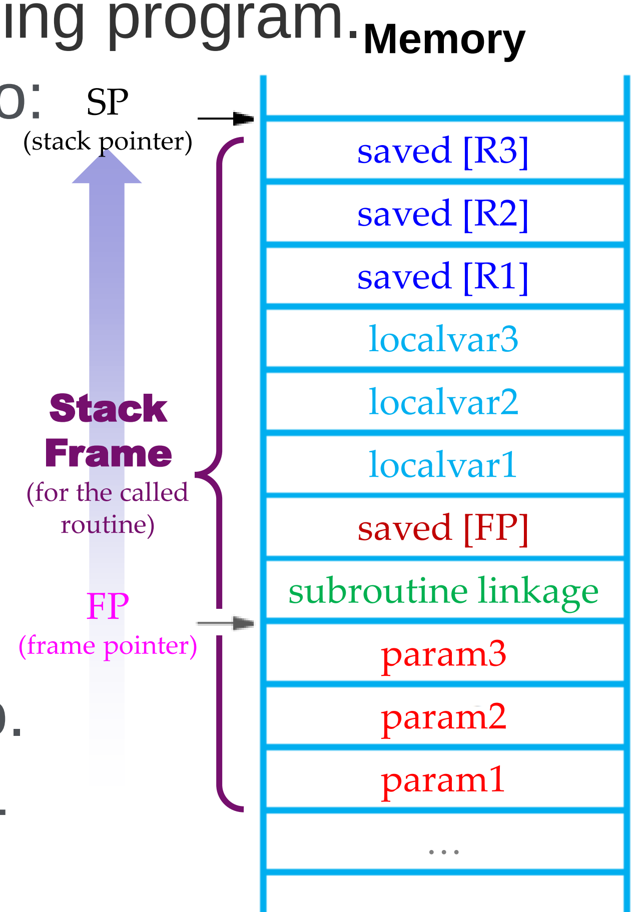
Processor stack can, once again, help with these issues!

Outline

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

The Stack Frame

- **Stack Frame**: a private workspace (in the processor stack) for each of the called subroutine.
  - It is allocated when subroutine is entered and deallocated when the subroutine returns to the calling program.
  - It is multi-functional and can be used to:
    - Pass parameters (and the results);
    - Keep the subroutine linkage;
    - Accommodate local variables;
    - Backup the contents of registers (which will be used by the subroutine).
  - It is also useful to have a general-purpose register, called frame pointer (FP), for easy access to the saved info.
    - E.g., for parameters: (FP), 4(FP), 8(FP), …
    - E.g., for subroutine linkage: -4(FP)

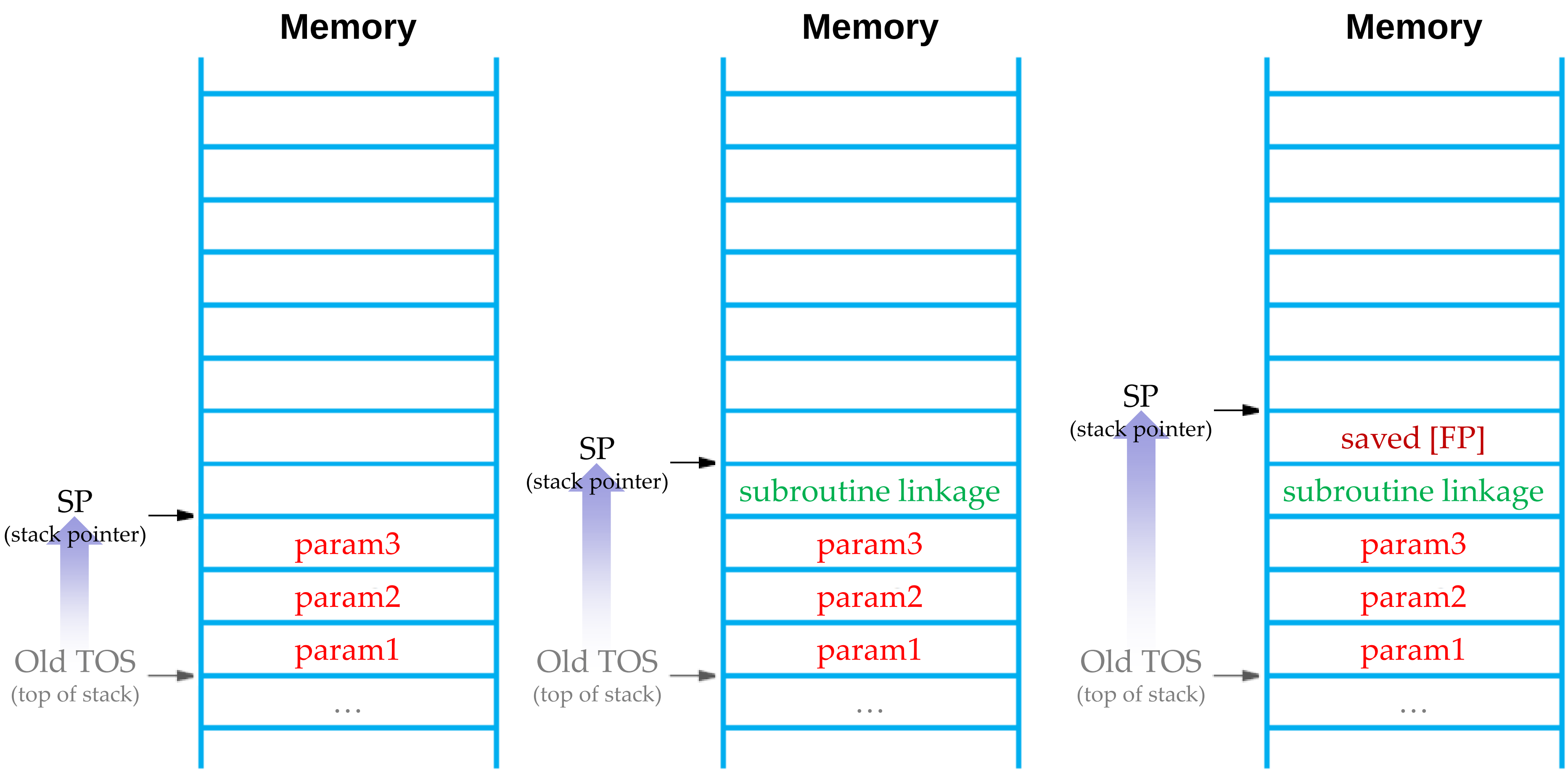


The Stack Frame: Allocation (1/2)

① Calling program pushes param. and calls the sub.;

② The subroutine saves the sub. linkage (from link reg.);

③ The subroutine saves the FP (which may contain info of use to the calling program);

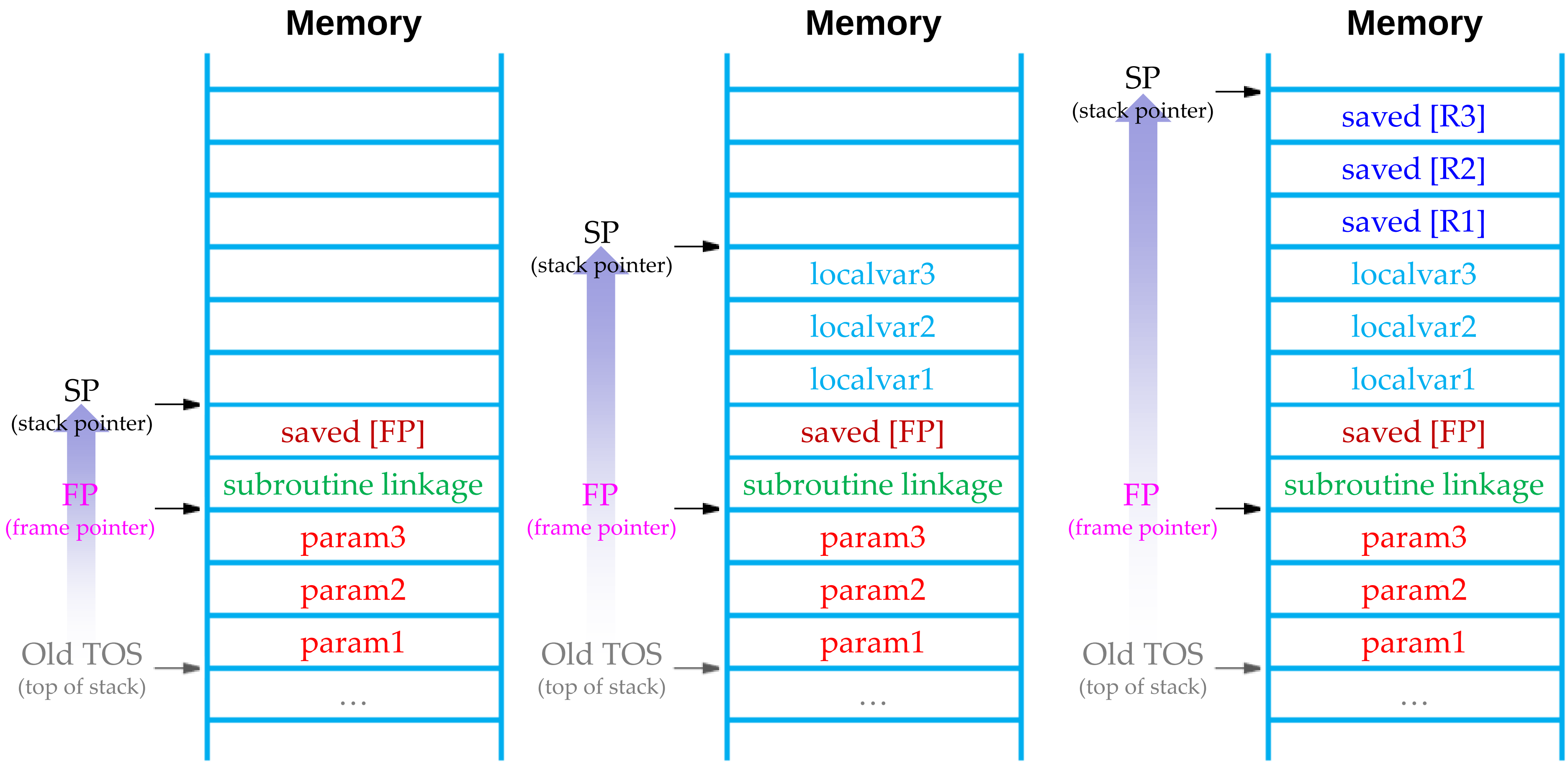


The Stack Frame: Allocation (2/2)

④ The subroutine updates FP;
(Why here? It is about "calling convention"!)

⑤ The subroutine creates local variables;

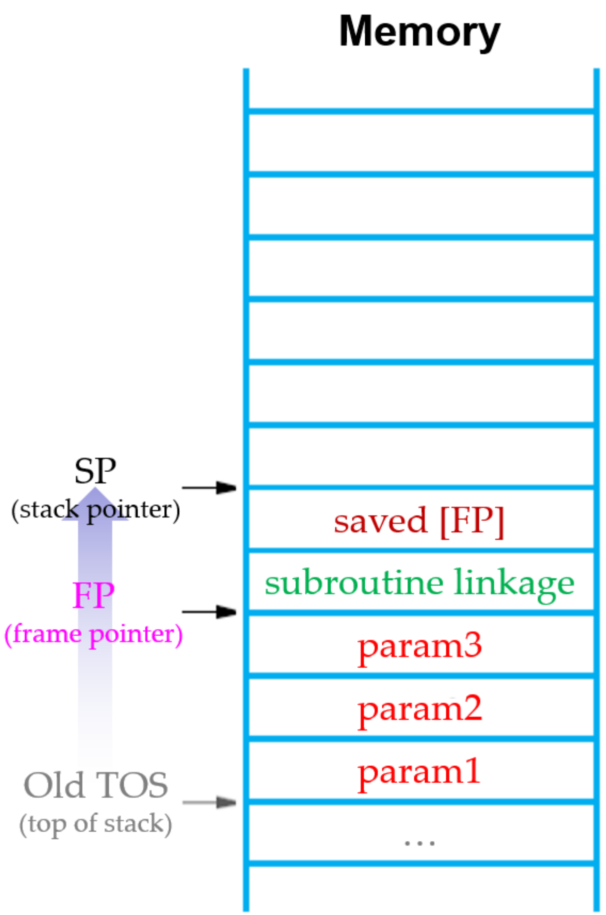
⑥ The subroutine saves the to-be-used registers.



Class Exercise 5.6

- During the allocation of the stack frame, the frame pointer (FP) is updated in Step ④.
- Can we have the FP updated earlier (say, as Step ① or ②)?

④ The subroutine updates FP;

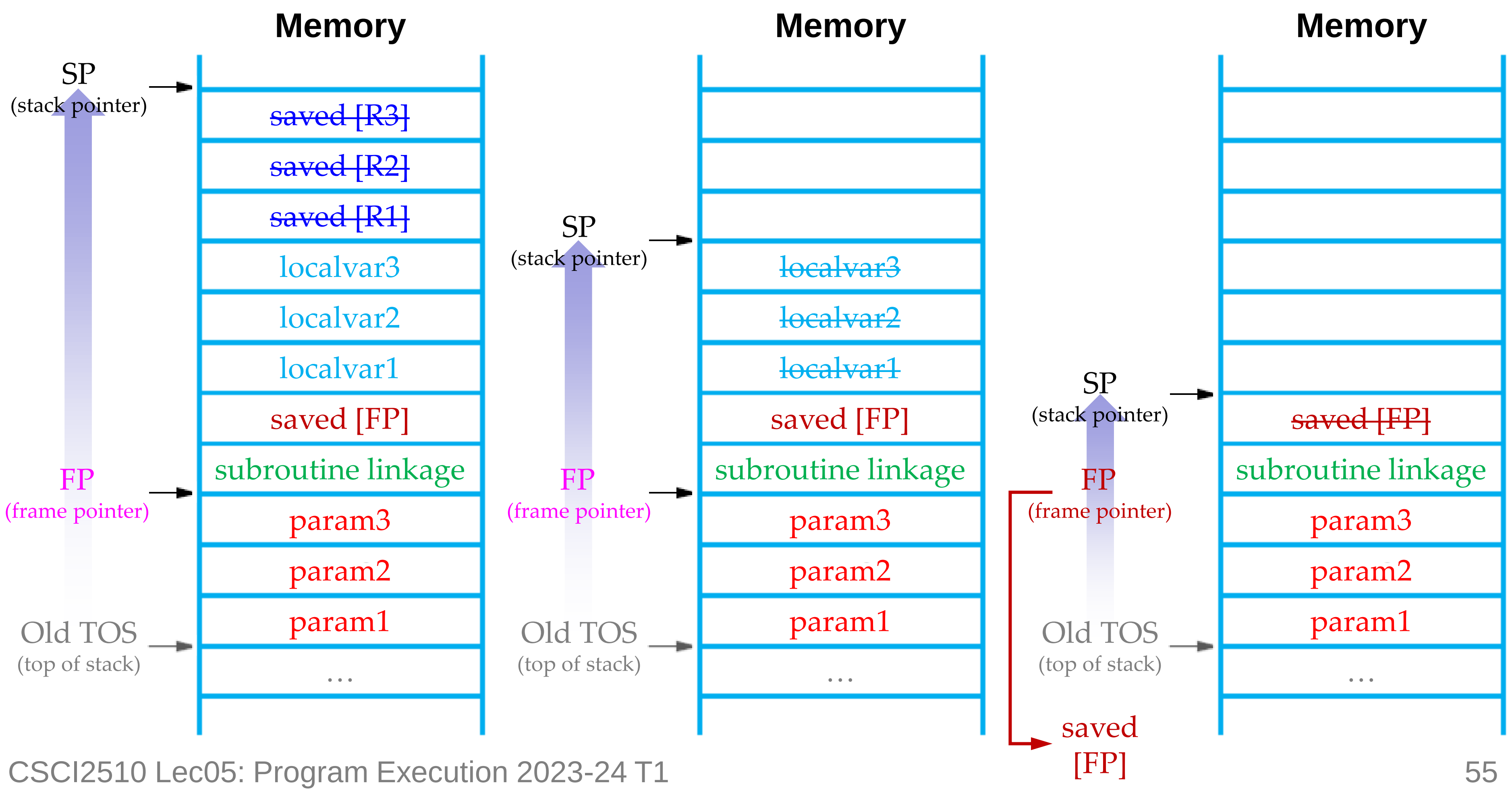


The Stack Frame: Deallocation (1/2)

① The subroutine restores the "used" registers;

② The subroutine deletes the local variables;

③ The subroutine restores the FP with saved [FP];

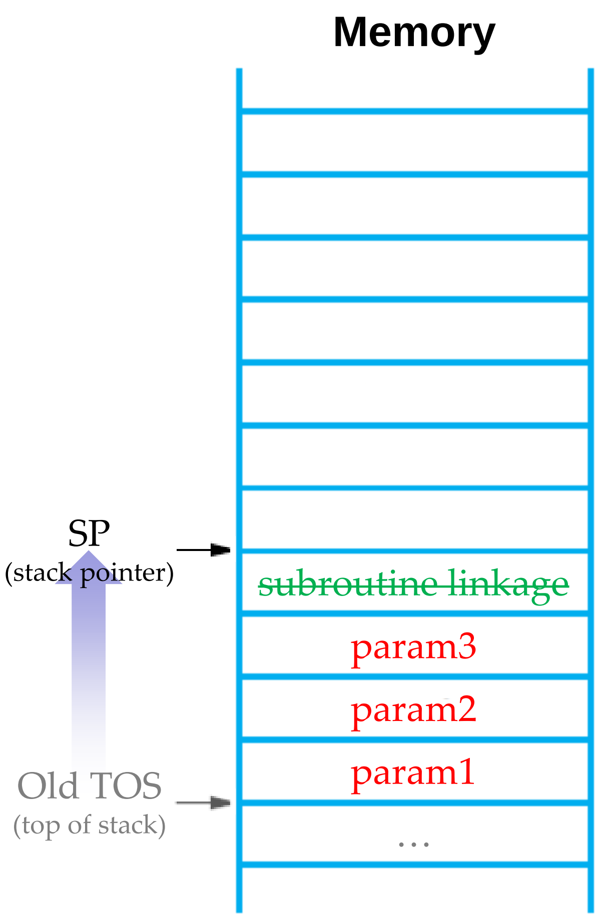


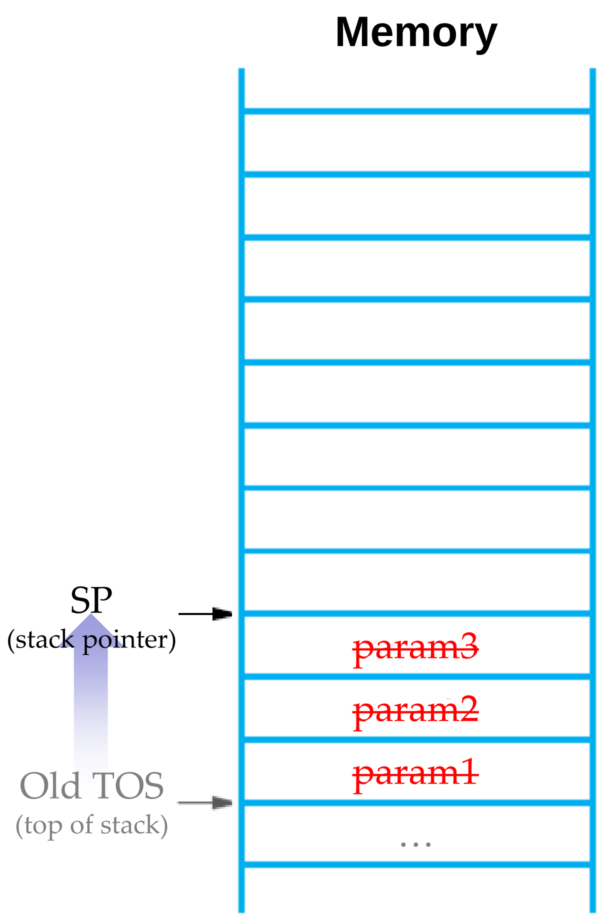
The Stack Frame: Deallocation (2/2)

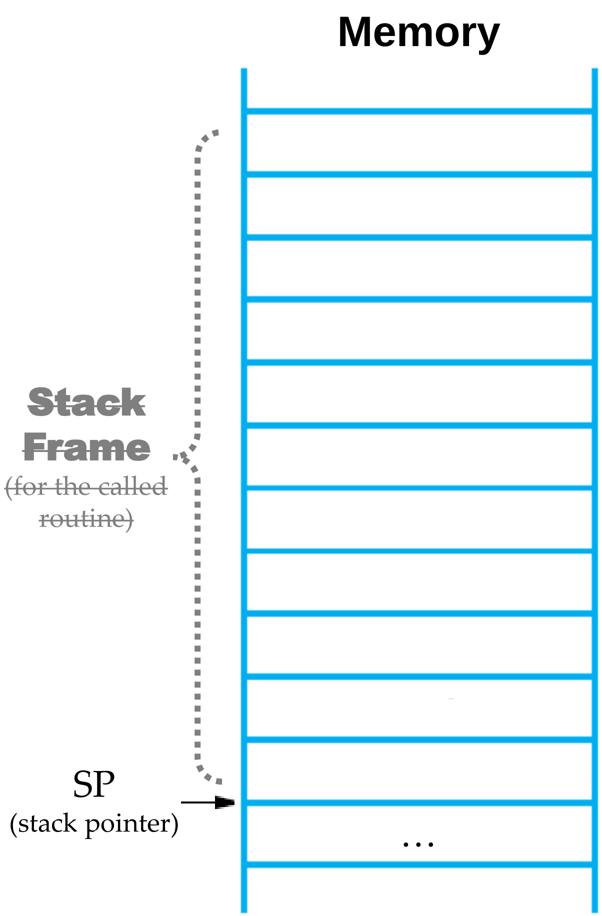
④ The subroutine returns to the callee (how?);

⑤ Calling program pops out param. (and get results, if any).

⑥ The stack frame is deallocated entirely.

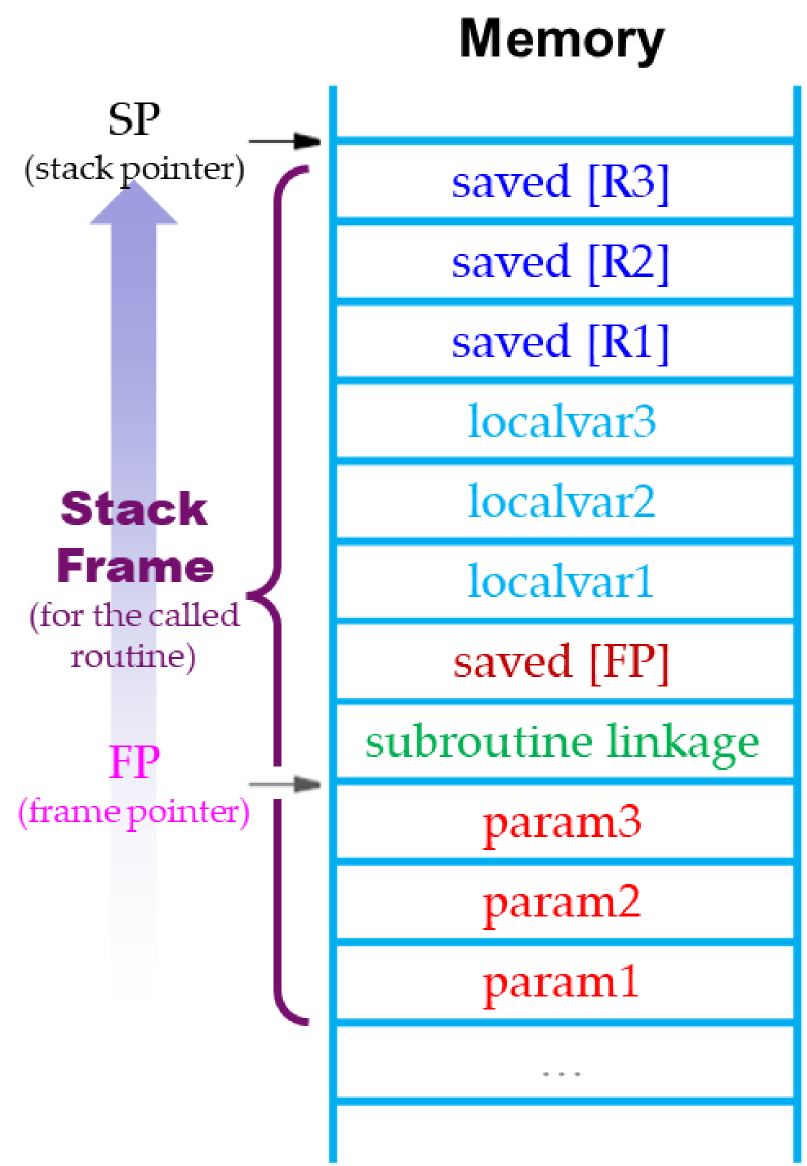






Class Exercise 5.7

- We have demonstrated how the parameters can be passed to the subroutine via the stack frame.
- Can you think of a way to return the computed results to the calling program via the stack frame?



Calling Convention

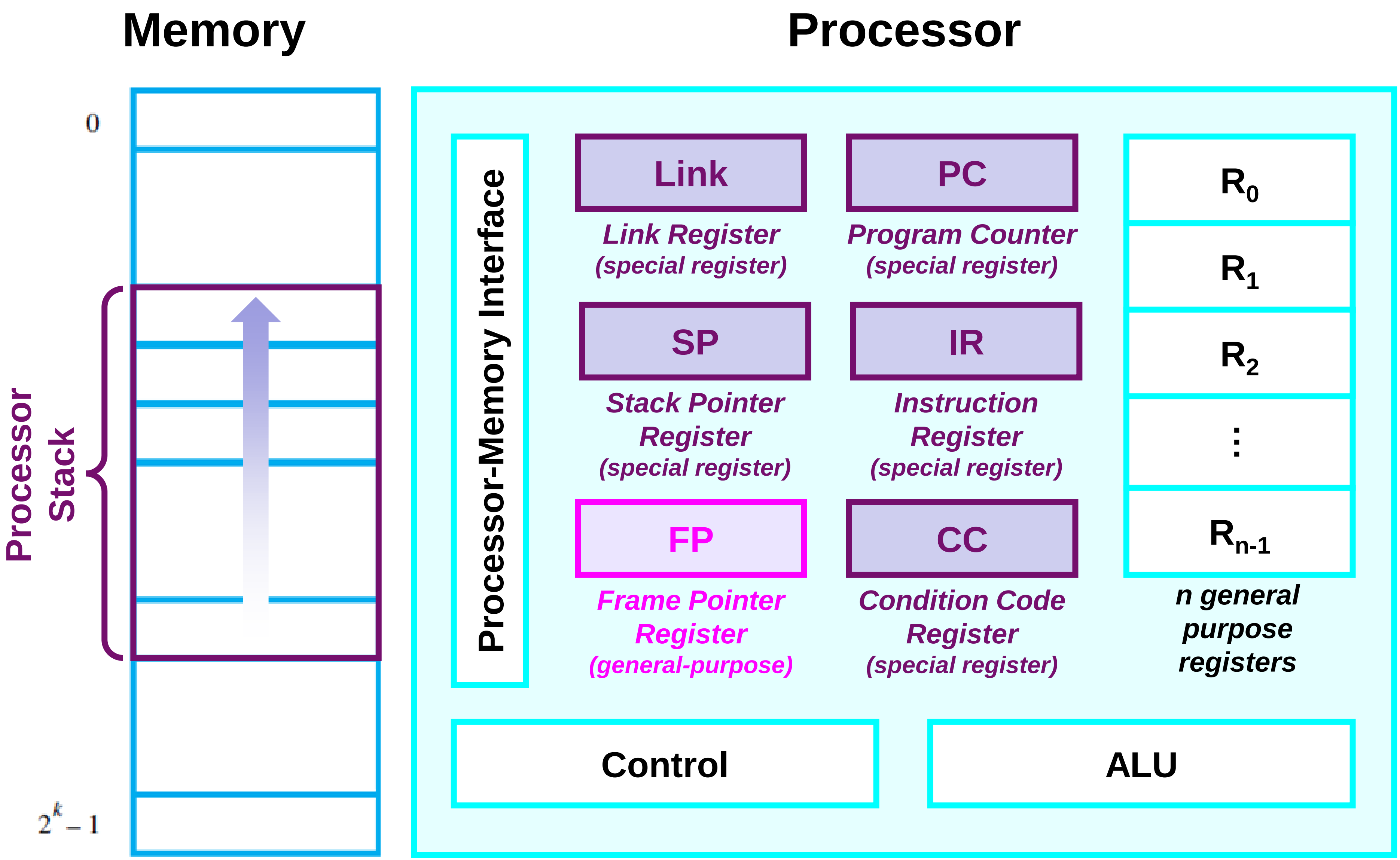
- **Calling convention** is an implementation-level scheme about:
  - How subroutines receive parameters from their caller and how they return a result;
  - How the caller and the callee (i.e., the subroutine) cooperate to prepare and restore the environment (e.g., the stack frame).
- In practice, there are, for sure, different calling conventions.
- What we introduced in lectures and tutorials is mainly based on RISC-V calling convention.

You may also have your own calling convention!

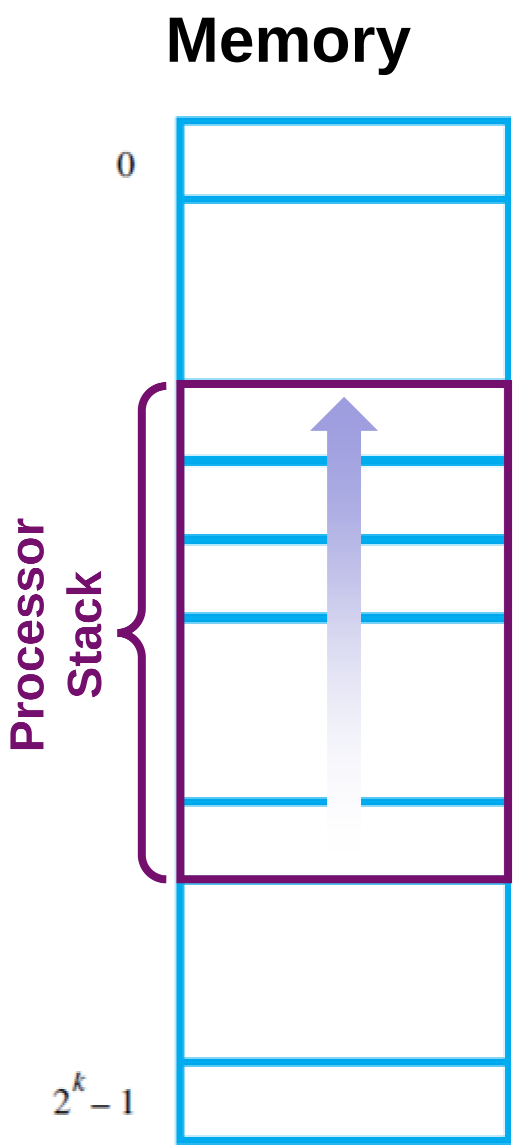
Summary (1/2)

- Flow for Generating/Executing a Program
- Instruction Execution and Sequencing
- Branching
  - Condition Codes
- Subroutines
  - Stack
  - Subroutine Linkage
  - Subroutine Nesting
  - Parameter Passing
  - The Stack Frame

Summary (2/2)


Memory
Processor
0
$2^k - 1$
Processor Stack
Processor-Memory Interface
Link
Link Register
(special register)
PC
Program Counter
(special register)
SP
Stack Pointer Register
(special register)
IR
Instruction Register
(special register)
FP
Frame Pointer Register
(general-purpose)
CC
Condition Code Register
(special register)
R_0
R_1
R_2
⋮
R_{n-1}
n general purpose registers
Control
ALU

Remark: Another Convention



vs.