NIOS Processor Stack and Function Calls

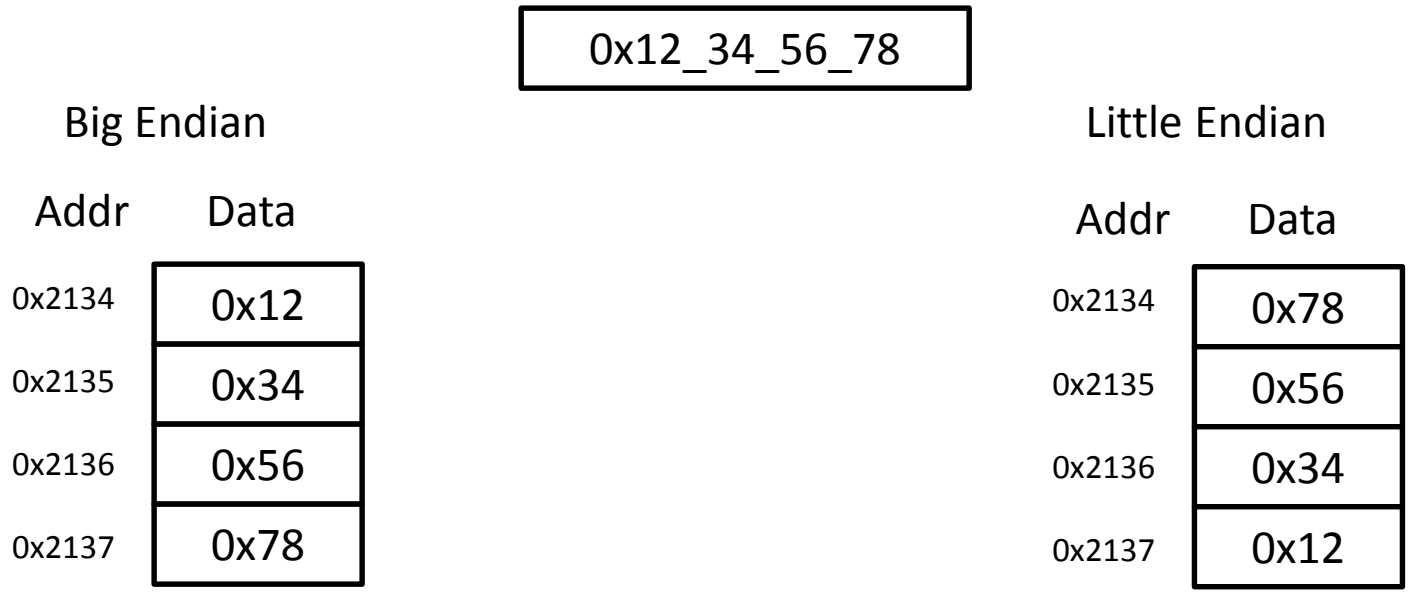
Engineering 304 Lab

Data Element Sizes in Memory

- NIOS processors use byte-addressable memory
- Byte = 8 bits (ldb, stb)
  - Requires one address location
- Half word = 16 bits = 2 bytes (ldh, sth)
  - Requires two address locations – referenced by an even-numbered address
- Word = 32 bits = 4 bytes (ldw, stw)
  - Requires four address locations – referenced by an address which is a multiple of four

Big vs Little Endian

- Example: Storing 0x12345678 into address 0x2134
- Motorola/Freescale: big endian; Intel: little endian



- NIOS is little endian

0x000084c0	00000000
0x000084d0	12345678
0x000084e0	00000000

Big vs Little Endian

	+0x0
0x000084c0	00 00 00 00
0x000084d0	78 56 34 12
0x000084e0	00 00 00 00
0x000084f0	00 00 00 00

- Example: Storing 0x12345678 into address 0x2134

0x12_34_56_78

Big Endian

Addr	Data
0x2134	0x12
0x2135	0x34
0x2136	0x56
0x2137	0x78

Little Endian

Addr	Data
0x2134	0x78
0x2135	0x56
0x2136	0x34
0x2137	0x12

- NIOS is little endian (compare byte vs word)

Procedural Programming

- Code is broken down into functions (procedures)
- Each function executes a specific operation
  - Arguments are passed to the function
  - The calculation is completed
  - A return value is sent back to the calling function
    - Actual value returned or an error number (0=success, others are error codes)

Review Recursion

- Recursion is when one function calls itself
- The function must clearly define a recursion stopping point
- Same instructions in memory are repeatedly executed (must manage the data well!)
- Factorial calculation example

Registers used for function calls

- Arguments 1-4 are placed in r4, r5, r6, & r7 in that order as needed
- Return values 1 and 2 are placed in R2 & R3 in that order as needed
- C-code Example

  Z = my_Function(A, B, C);

  Steps:

  A is placed in r4, B in r5, and C in r6

  The assembly instruction "`call my_Function`" is then used

  {my_Function does its work and then puts the value of Z in r2 and then uses the "`ret`" instruction}

  The calling function finds Z in r2

Function Call Registers (cont)

- Stack Pointer (sp)
  - Always points to the last used entry in the stack
  - Should be initialized to the address of the last byte in memory + 1
    - E.g. memory from 0x1000 – 0x17ff, then SP initialized to 0x17ff+1 = 0x1800
- Return Address (ra)
  - Holds the address of the instruction immediately after the "`call`" instruction when the "`call`" is executed ("breadcrumb")
  - Used by the "`ret`" instruction to return to the calling function's instruction immediately after the "`call`"

Pushes and Pops (stack)

- Push = the placing of an item on a stack
  - E.g. adding napkins to a spring-loaded napkin dispenser
- Pop = the taking of an item off a stack
  - E.g. removing a napkin from a spring-loaded dispenser
- On NIOS, all pushes and pops are memory ⇔ registers (`ldw/stw`) and are of size=word (4 bytes)
- Stack Pointer points at last used stack entry and must be adjusted for each push/pop

Pushing and Popping on NIOS

1-element Push:

```
addi sp, sp, -4
stw rX, 0(sp)
```

1-element Pop:

```
ldw rX, 0(sp)
addi sp, sp, 4
```

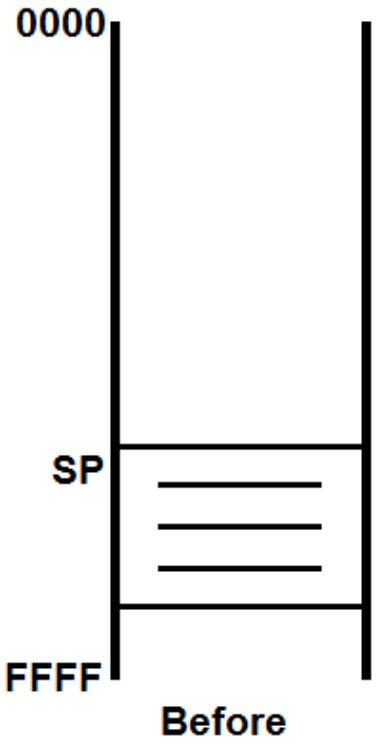
2-element Push:

```
addi sp, sp, -8
stw rX, 4(sp)
stw rY, 0(sp)
```

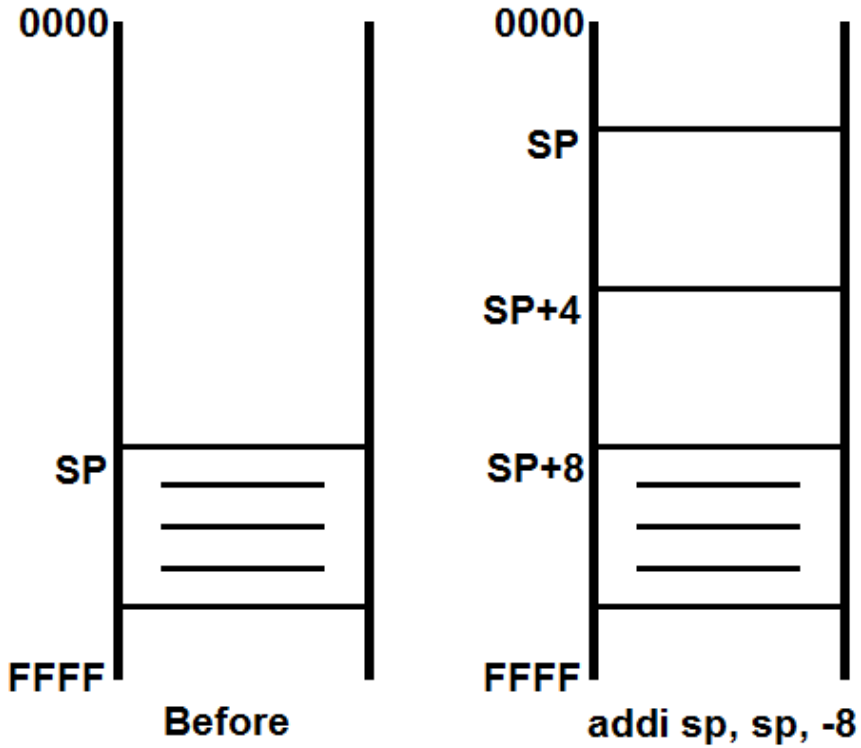
2-element Pop:

```
ldw rY, 0(sp)
ldw rX, 4(sp)
addi sp, sp, 8
```

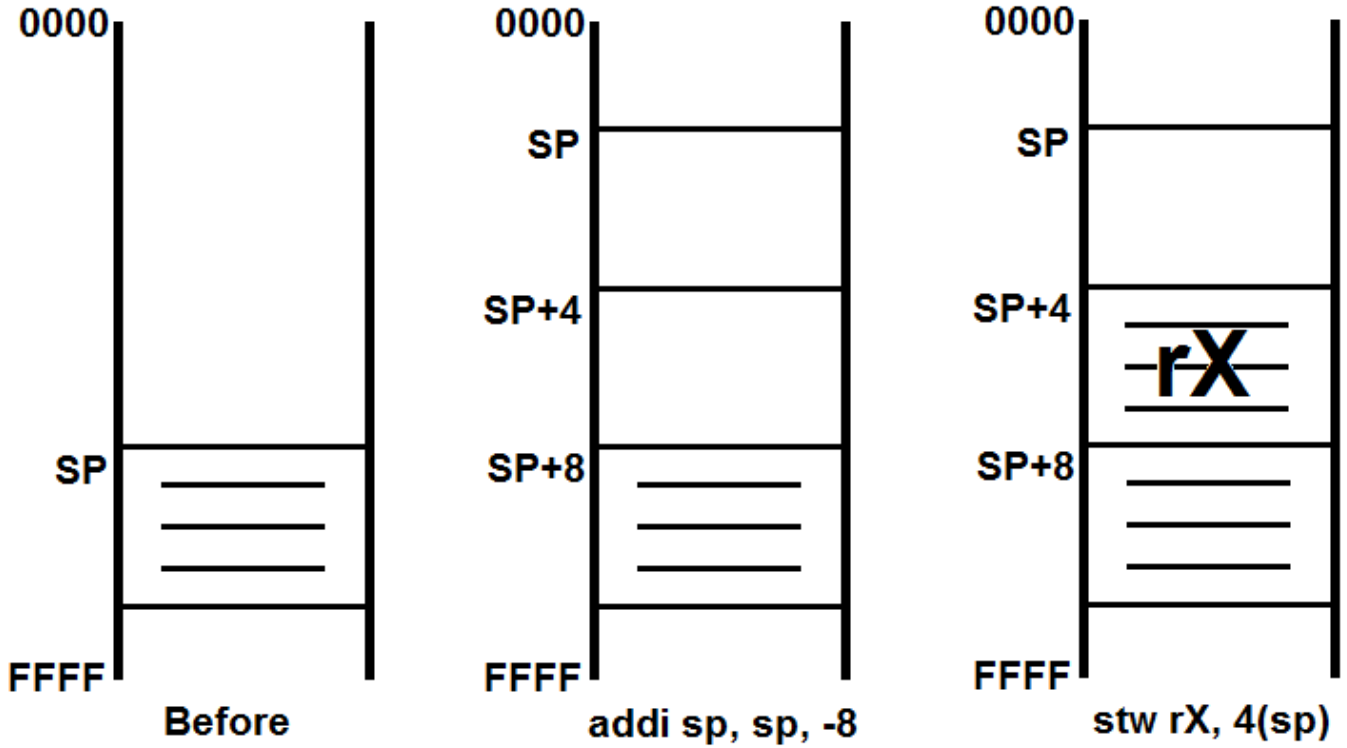
Before Performing 2 Pushes



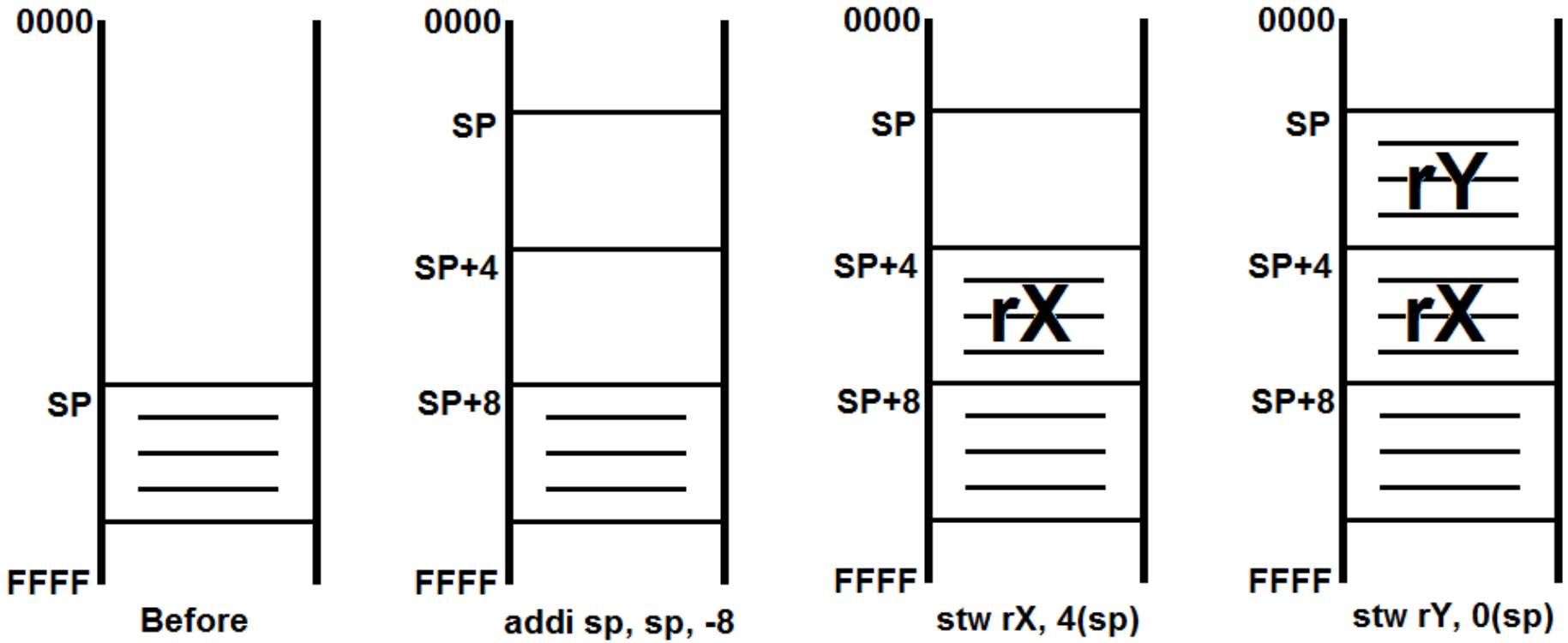
Push: After Adjusting the Stack Pointer



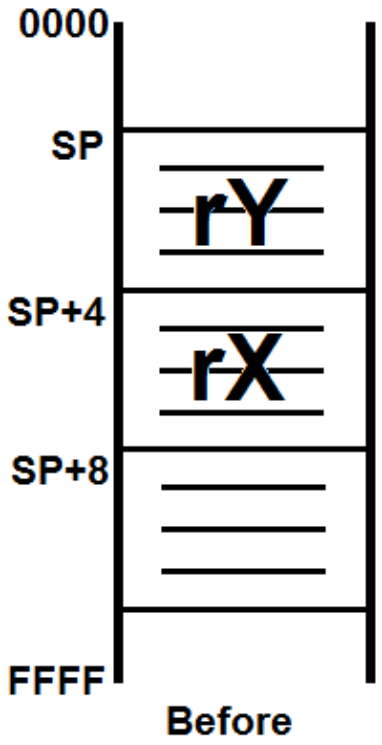
Push: After Saving rX



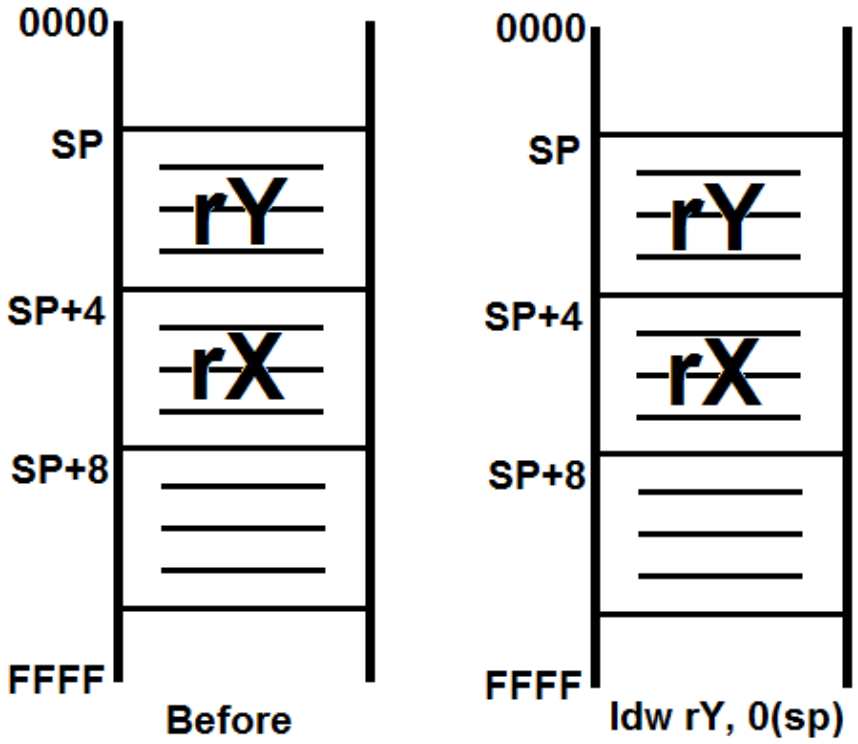
Push: After Saving rY



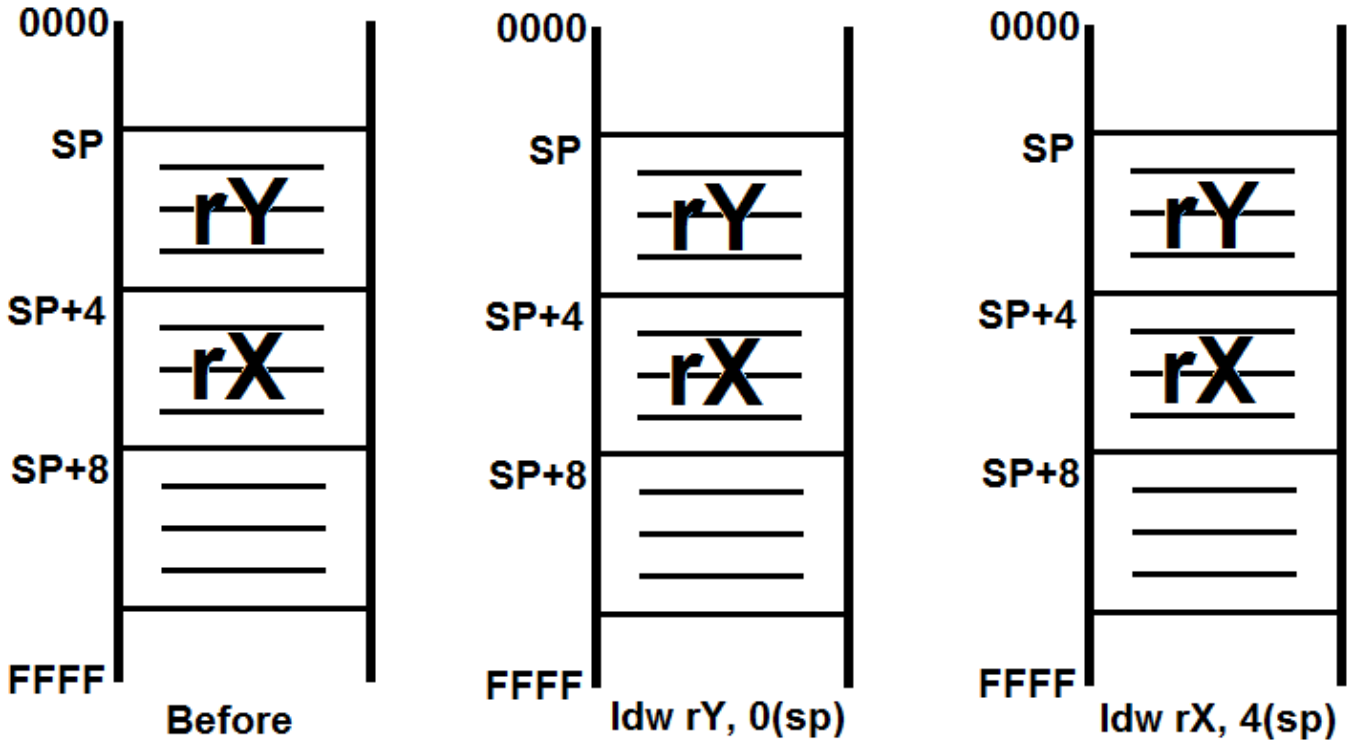
Before Performing 2 Pops



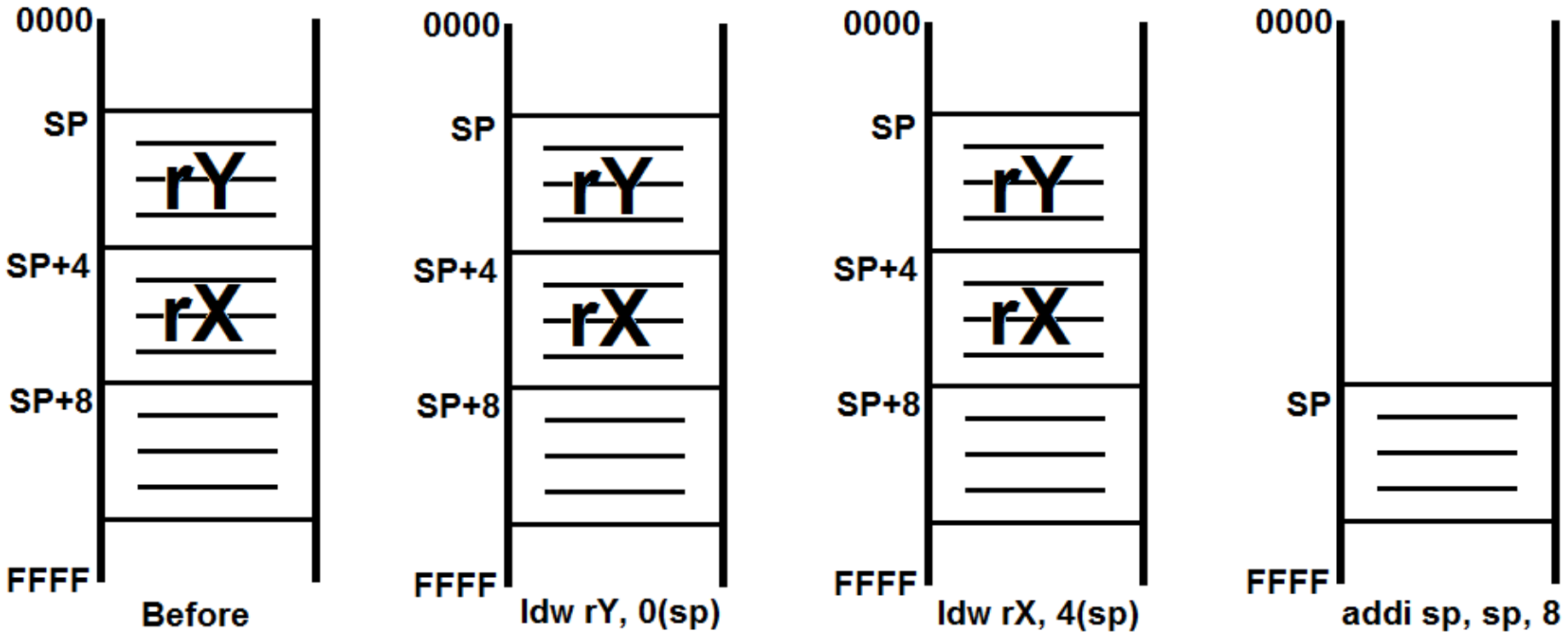
Pop: After Reading rY from Stack



Pop: After Reading rX from Stack



Pop: After Adjusting the Stack Pointer



Function Calls

- The following describes the standard assembly program structure for making function calls, saving registers, and restoring registers. Actions to perform at various points in the program are shown in {}s.
- Calling a subfunction uses the "`call label`" instruction (label = name of subfunction)
- To return to the calling function, use the "`ret`" instruction at the end of the subfunction

_start function

```
_start:
```
{setup stack pointer to the highest memory address +1}

…

```
Pre_call:
```
{Push caller-saved registers}

```
 call Sub1
Post_call:
```
{reverse-pop caller-saved registers}

…

```
End_start:
  br End_start /* infinite loop here */
```

Subfunction (includes “main()”)

```
Sub1:
Prologue:
 {Push ra, callee-saved registers}
 …
Pre_call:
 {Push caller-saved registers}
call Subfunction
Post_call:
 {reverse-pop caller-saved registers}
 …
Epilogue:
 {reverse-pop callee-saved registers, ra}
End_Sub1:
ret
```

Leaf Routine (calls no other fcn)

```
SubLeaf:
/* no pushing or popping needed unless using callee-saved
registers */
  …
End_SubLeaf:
ret
```

Recommended Approach

- Ignore push/pop requirements and create code for each function, properly using the registers r4-r7 and r2-r3
- Identify all pre-call, post-call, prologue, and epilogues spots
- Add comments to list what is pushed/popped
- Add the necessary instructions to do the push and pop operations identified