COMP2300-COMP6300-ENGN2219
Computer Organization & Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

Functions

Functions

- Program fragments that are written once and invoked multiple times within the **same or a different** program

C Code

```
void main()
{
  int y1, y2;
  y1 = sum(42, 7);
  y2 = diffofsums(12, 7, 10, 1);
}

int diffofsums(int f, int g, int h, int i) {
   int result;
   result = sum(f, g) - (h + i);
   return result;
}
```

```
int sum(int a, int b)
{
  return (a + b);
}
```

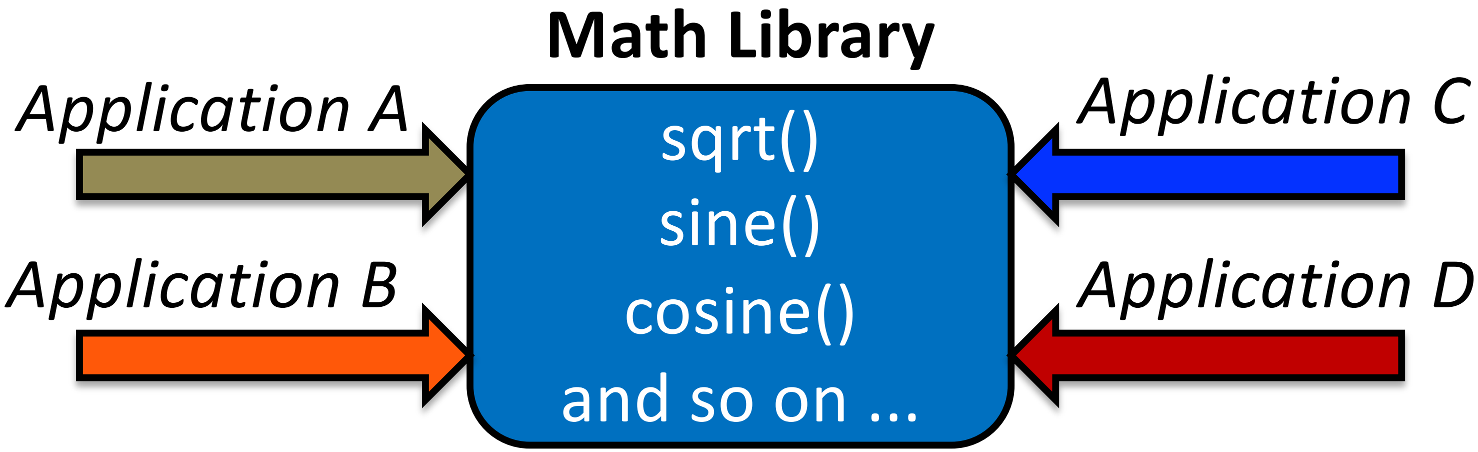
- One software engineer writes `sum(int a, int b)` and many others can reuse it without specifying the details

Libraries of Pre-Existing Functions

- One might require a **fragment** that has been supplied by the manufacturer or by some independent software supplier
- It is almost always the case that collections of such fragments are available to programmers to **free** them from having to write their own
- These collections are called libraries
- Example: Math library provides square root, sine, cosine, arctangent
- Programmers do not need to reinvent the wheel

API

- **A**pplication **P**rogramming **I**nterface (**API**)
  - Defines the interfaces by which one software program communicates with another **at the source code level**



- **API defines the interface only**
  - The user of the API can ignore the implementation
  - Many implementations of the same API
  - C standard library hides many low-level details of the system

Usefulness of Functions

- High-level languages offer functions to enable
  - Abstraction & Modularity
  - Code reuse
  - Readability
  - Testability & validation
  - Maintainability

Functions

- Functions are also called **procedures** or **subroutines**

- Functions are ubiquitous which encourages ISA support
  - Special jump instructions
  - Special (isolated) space in memory to store function-related data
  - “Mechanism” to reduce interference between nested functions

What we will cover

- Architectural support for functions
  - Branch and Link instruction (BL)
  - Stack Pointer (SP)
  - Link Register (LR)

- Microarchitecture-level impact of programming styles
  - Iteration vs. Recursion

- Get a deeper understanding of hardware/software interaction and tradeoffs

Functions in C

- `main()` is the caller
  - It calls another function
  - Returns nothing (void)
  - Takes no input arguments

- `sum()` is the callee
  - Being called by some function
  - Takes two input arguments of type int: a and b
  - It returns an integer value
  - Computes the sum of a and b

C Code

```c
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Functions in C

- The caller provides the input arguments
  - 42 and 7 in this example

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

- The distinction between caller and callee depends on the context
  - What if someone calls the `main` function?

Leaf versus Non-Leaf Functions

- `sum()` is a **leaf** function
  - It does not call another function

- `main()` is a non-leaf function
  - It calls another function

- Non-leaf functions are **more complicated** especially at the assembly level

- `sum()` can be called from many locations in program
  - Motivation: code reuse

Functions as Detectives

- Assigned a secret mission (function call)
- Acquires necessary resources (acquire memory)
- Perform the mission (execute instructions)
- Leaves no trace (clean up memory)
- Returns safely to point of origin (function return)

Breaking Down Function Execution

- Caller **stores** arguments in registers or memory
- Function call: Caller **transfers** flow control to the callee
- Callee **acquires/allocates** memory for doing work
- Callee **executes** the function body
- Callee **stores** the result in “some” register
- Callee **deallocates** memory
- Function return: Callee **returns** control the caller

Instruction for Function Call

- It is usually the case that ISAs provide a special variant of the branch instruction for making the function call
  - MIPS : **jal**
  - ARM : **BL**
  - Intel x86 : **call**
  - RISC-V : **jal**
  - QuAC: No architectural support for functions

ARM Function Call

- `BL` (Branch and Link)
- CPU branches to the label specified by `BL`
- CPU stores the **return address** in the link register (`LR`)
- Return address is the address of the next instruction after the function call
- **How should we return from the function to the caller?**

Returning to Callee

- Returning from function requires updating the PC
  - Move the link register into `PC`
  - `MOV PC, LR`
- How should we pass arguments to the function?
- Where should we return the value?

Passing Arguments

- Passing arguments (convention)
  - `R0, R1, R2, R3`

- Returning value (convention)
  - `R0`

ARM Register Set

Name	Use
`R0`	Argument / return value / temporary variable
`R1-R3`	Argument / temporary variables
`R4-R11`	Saved variables
`R12`	Temporary variable
`R13 (SP)`	Stack Pointer
`R14 (LR)`	Link Register
`R15 (PC)`	Program Counter

Example of a Function Call

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

Memory Address	ARM Assembly Code		
0x00000200	MAIN	**BL**	**SIMPLE**
0x00000204		ADD	R4, R5, R6
...			
0x00401020	SIMPLE	**MOV**	**PC, LR**

- **BL** branches to SIMPLE
  LR = PC + 4 = **0x00000204**
- **MOV PC, LR** sets PC = LR
  (the next instruction executed is at **0x00000200**)

- MAIN and SIMPLE are labels (memory addresses) in assembly
- BL transfers flow to SIMPLE and stores the *return address* in LR
- The function returns after MOV, and the next instruction (ADD) is executed

Another Example: Difference of Sums

C code:

```
int main() {
   int y;
   ...
   y = diffofsums(2, 3, 4, 5);
   ...
}

int diffofsums(int f, int g, int h, int i) {
   int result;
   result = (f + g) – (h + i);
   return result;
}
```

ARM Assembly Code

```
; R4 = y

MAIN
   ...
   MOV R0, #2            ; argument 0 = 2
   MOV R1, #3            ; argument 1 = 3
   MOV R2, #4            ; argument 2 = 4
   MOV R3, #5            ; argument 3 = 5
   BL  DIFFOFSUMS        ; call function
   MOV R4, R0            ; y = returned value
   ...

; R4 = result
DIFFOFSUMS
   ADD R8, R0, R1        ; R8 = f + g
   ADD R9, R2, R3        ; R9 = h + i
   SUB R4, R8, R9        ; result = (f + g) - (h + i)
   MOV R0, R4            ; put return value in R0
   MOV PC, LR            ; return to caller
```

Questions

- How can we pass more than 4 function arguments?

- How can we ensure that registers in use by the caller are not corrupted?
  - `DIFFOFSUMS` overwrites `R4, R8, R9`
  - `MAIN` may have **"live"** values in these registers

- Answer: **The Stack**
  - A special area in memory used across function calls to preserve registers and store temporary values that overflow available registers

Uses of Stack

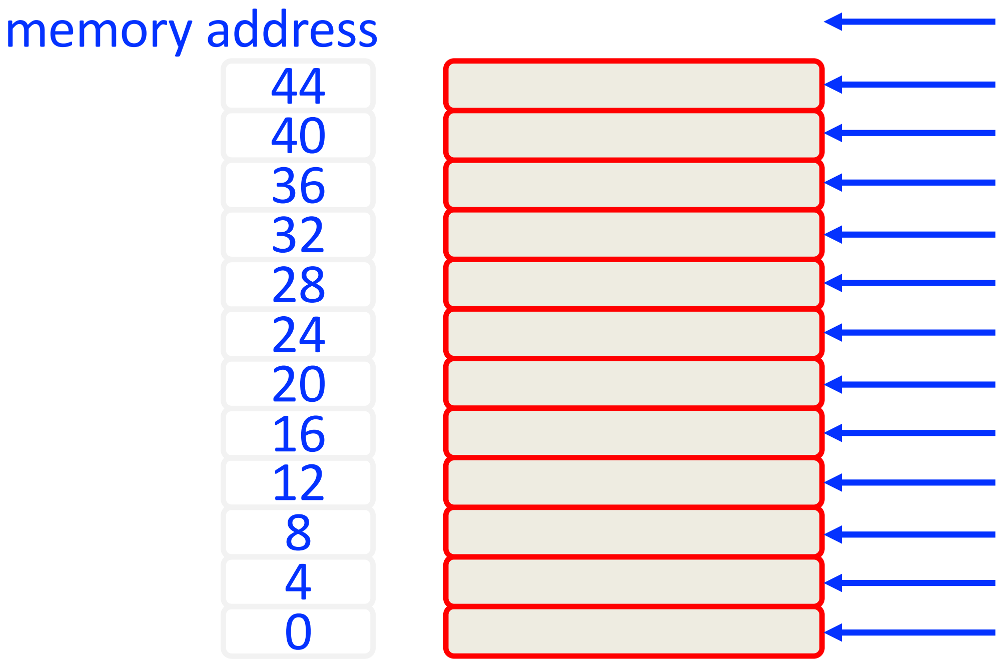
- Preserving and saving registers

- Passing extra arguments

- Temporary memory space for Storing function-local variables

The Stack

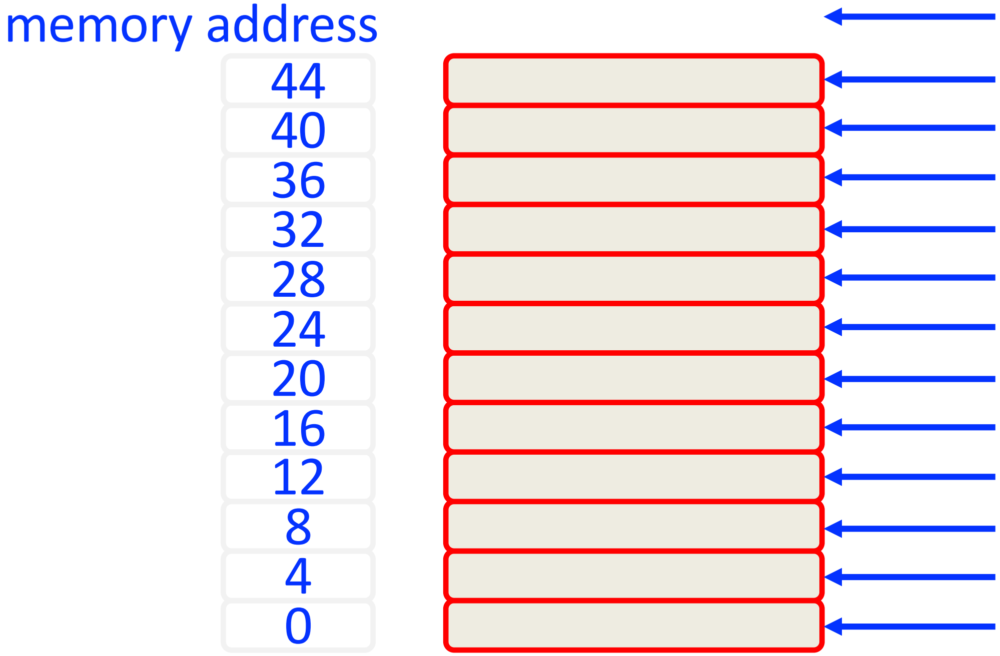
- A stack is like a Last In First Out (LIFO) Queue



- Stack expands and contracts as items are added and removed

The Stack

- A stack is like a Last In First Out (LIFO) Queue



- Stack expands and contracts as items are added and removed

Implementing a Stack

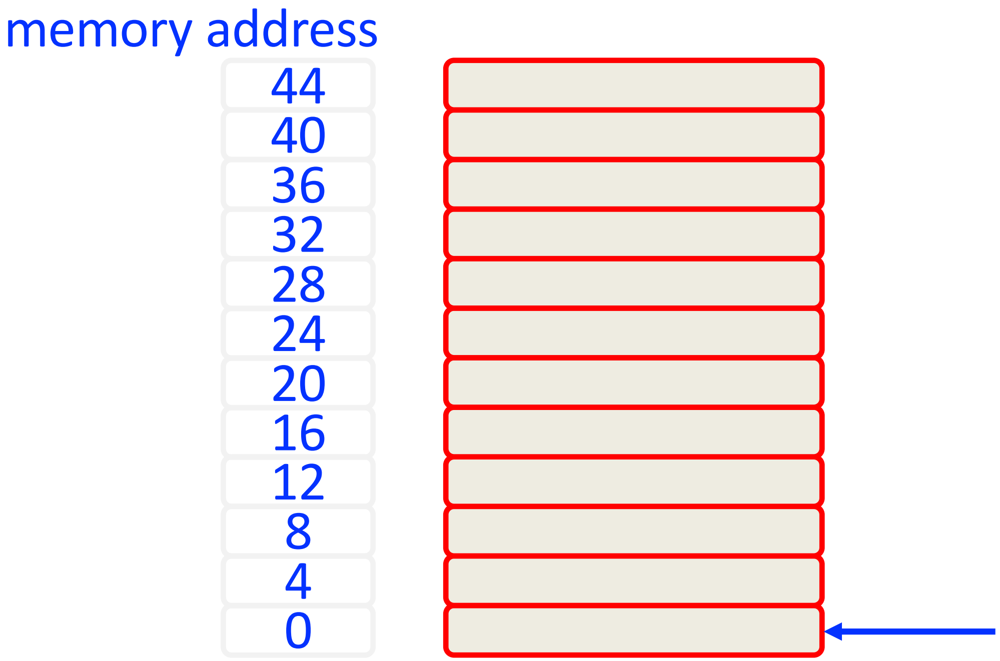
- What do we need to implement a stack?

- We need "some memory" for stack data (items)
  - Do we have memory? Yes, we can use data memory

- We need "an arrow" to point to the top of the stack
  - What does an arrow represent in comp. architecture?
  - It represents a pointer to a memory location
  - In other words, a register containing the memory address
  - Do we have a register? Yes, we can use an architectural register
  - **Stack Pointer (SP):** An architectural register that is **by convention** dedicated to storing the top of the stack

Implementing a Stack

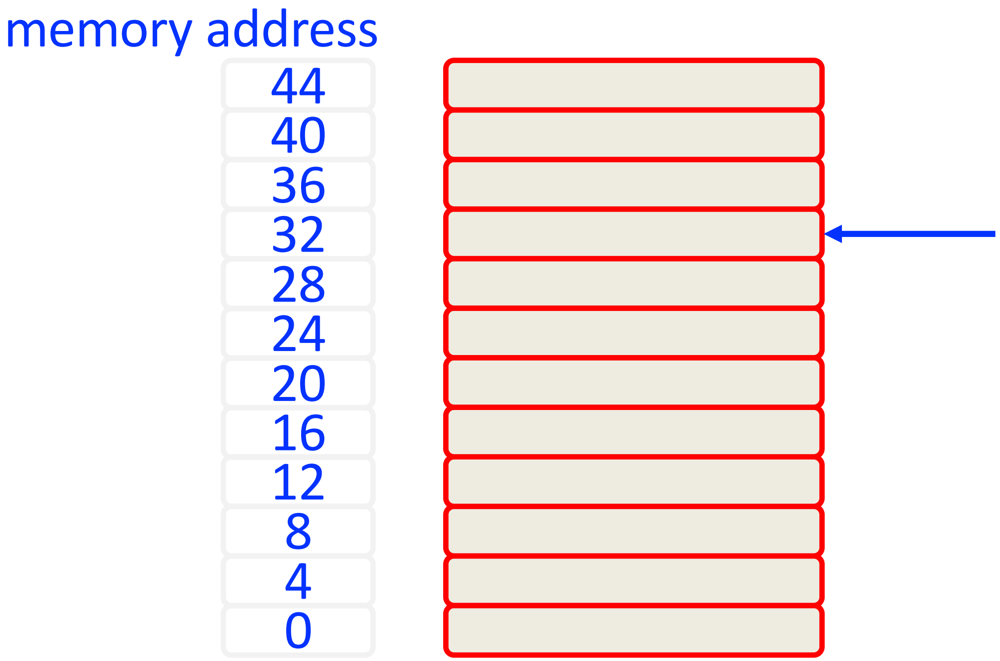
- Suppose we have the stack pointer initialized to 0. How do we make space for (reserve) **8 items** on the stack. Word size = 4 bytes



- Add 32 to the stack pointer: SP = SP + 32

Implementing a Stack

- Suppose we have the stack pointer initialized to 0. How do we make space for (reserve) **8 items** on the stack. Word size = 4 bytes



- Add 32 to the stack pointer: SP = SP + 32

Growing and Shrinking the Stack

- `push`
  - Put a new item on top of the stack and adjust the stack pointer accordingly (`SP = SP + 4`)

- `pop`
  - Remove an item from top of the stack and subtract 4 from the stack pointer

Push and Pop Operations

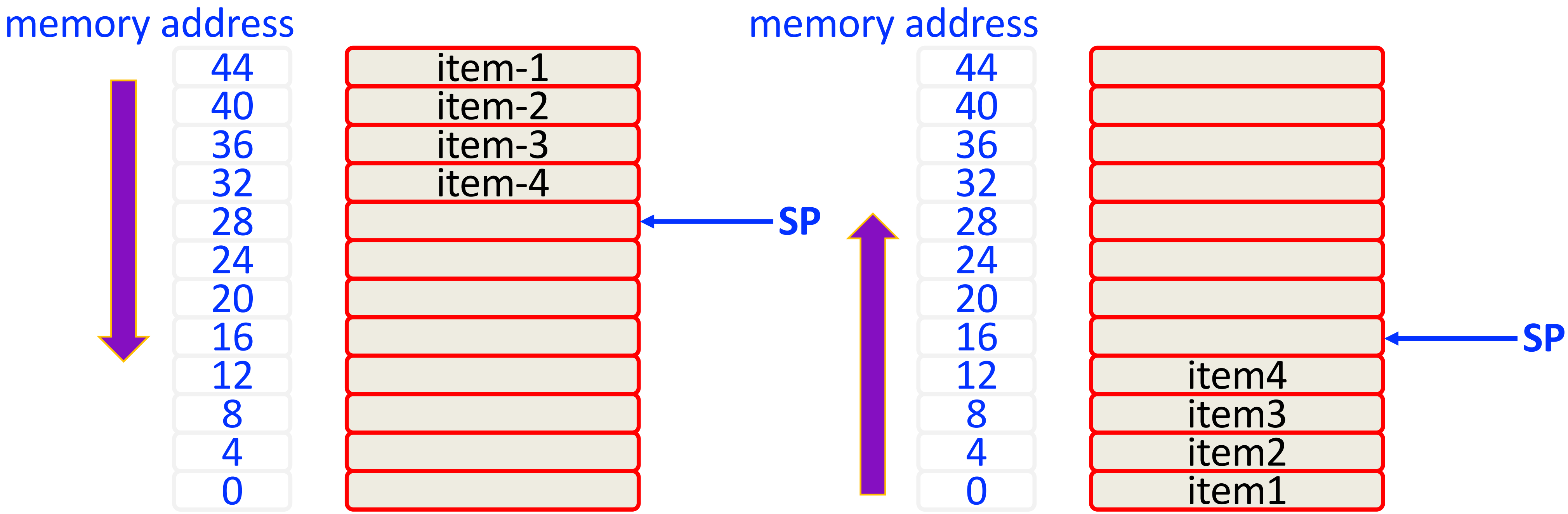
- We store the stack at “some” arbitrary address in memory
  - Details of how the address is chosen are not important

- `push {R0}`
  - Store `R0` onto the stack

- `pop {R0}`
  - Restore `R0` with whatever is at the top of the stack

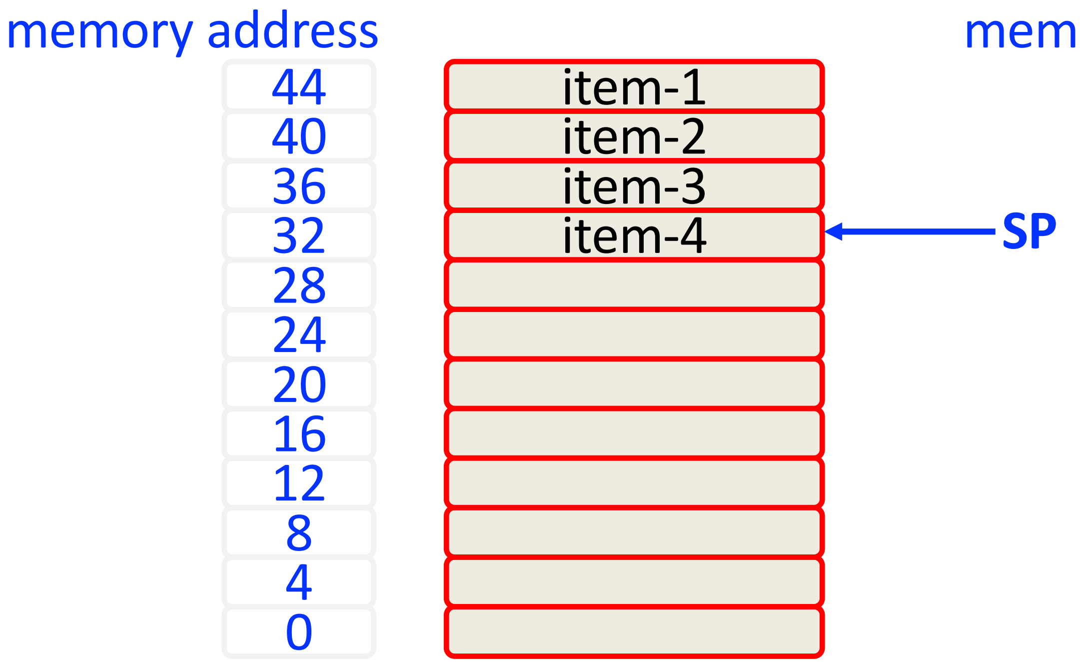
Different Ways to Manage Stack

- **Descending** Stack
  - Grows downward
  - SP points to the lowest address
- **Ascending** Stack
  - Grows upward
  - SP points to highest address

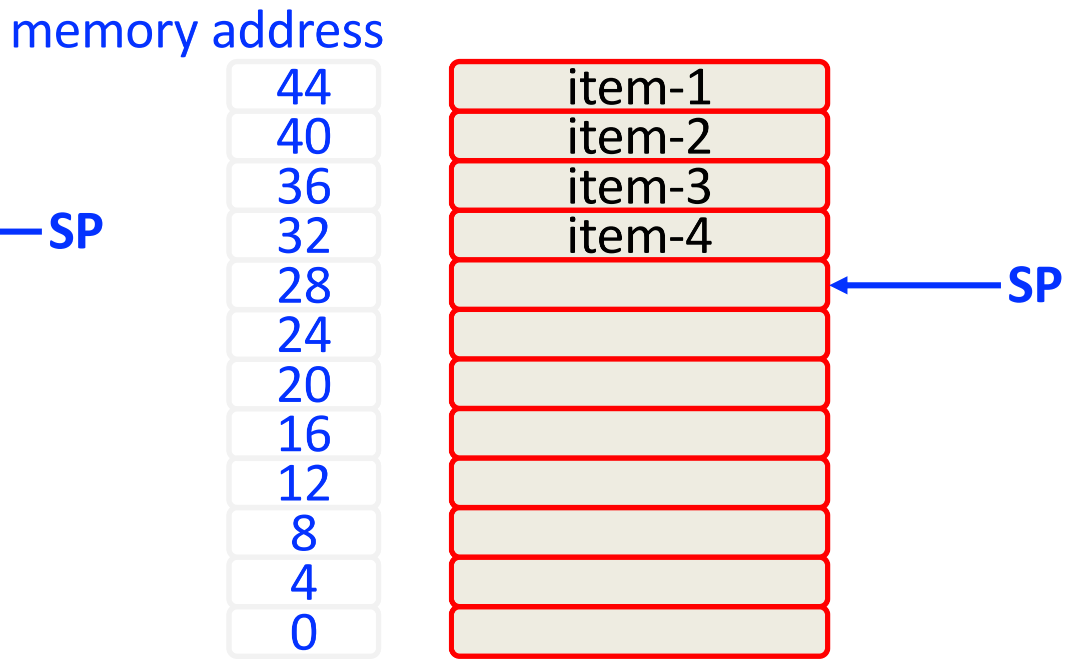


Further Classification

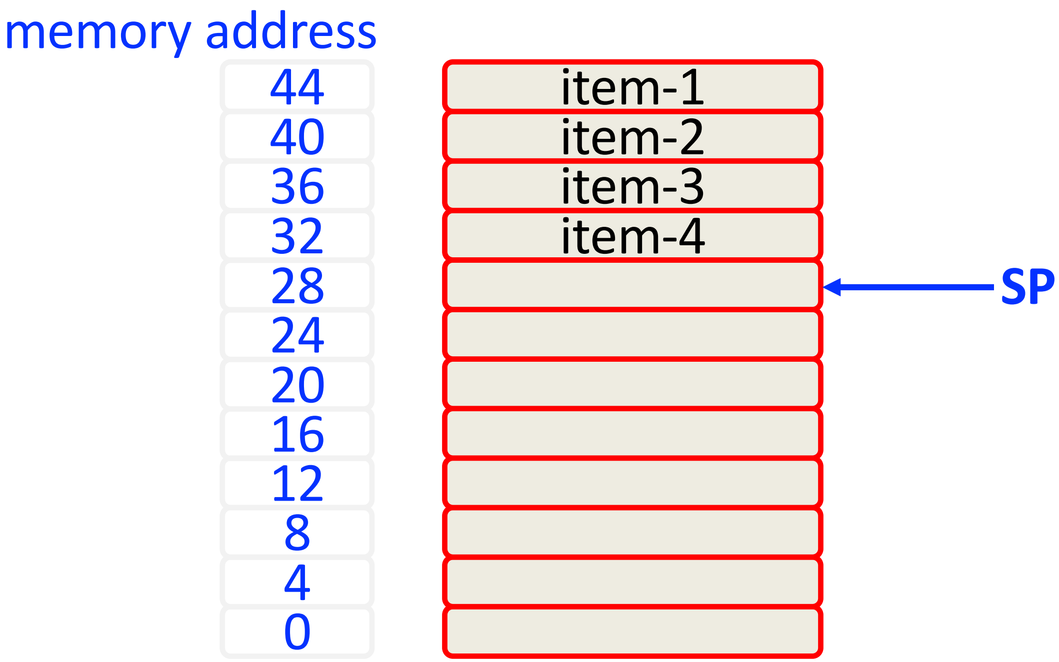
- **Full** Stack
  - SP **points to** the last allocated space on the stack (top)
  - Last item pushed



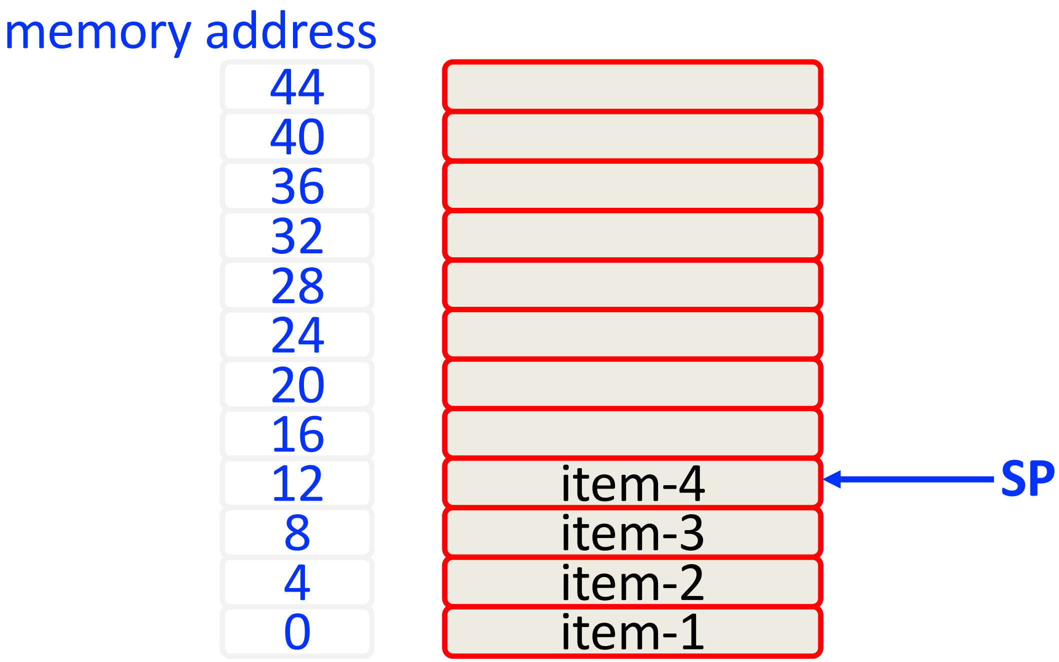
- **Empty** Stack
  - SP **points to one word beyond the top of stack**



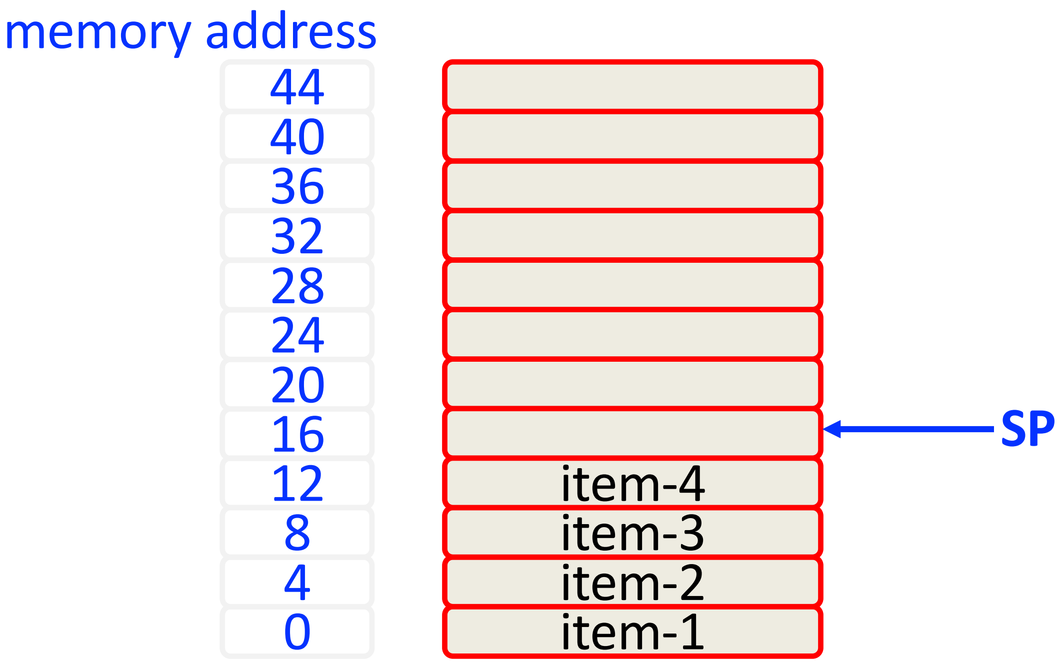
Empty Descending Stack



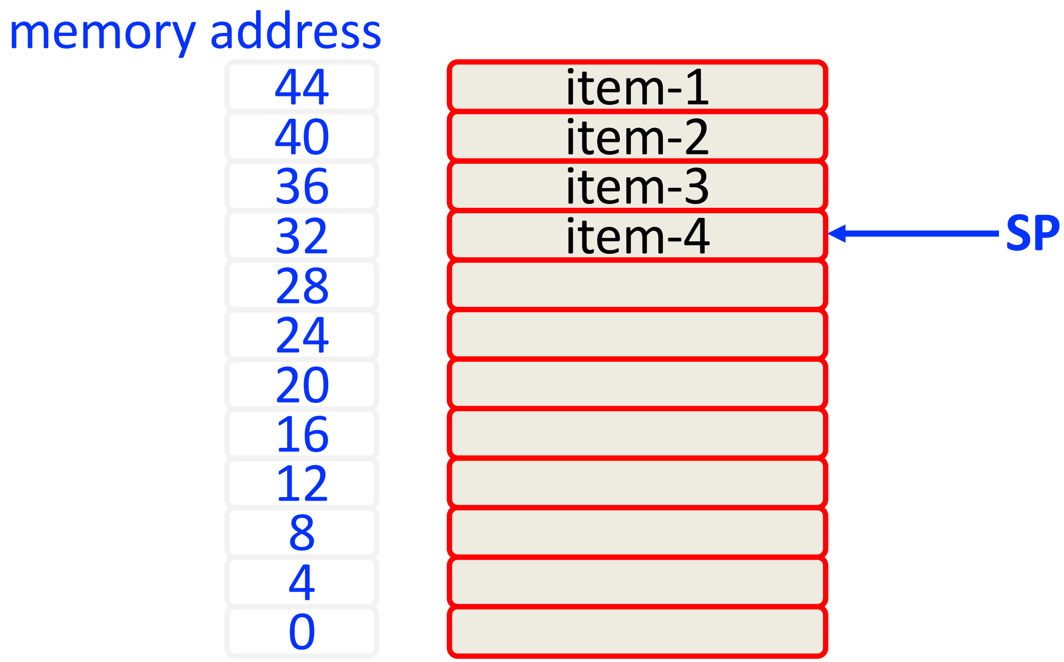
Full Ascending Stack



Empty Ascending Stack



Full Descending Stack



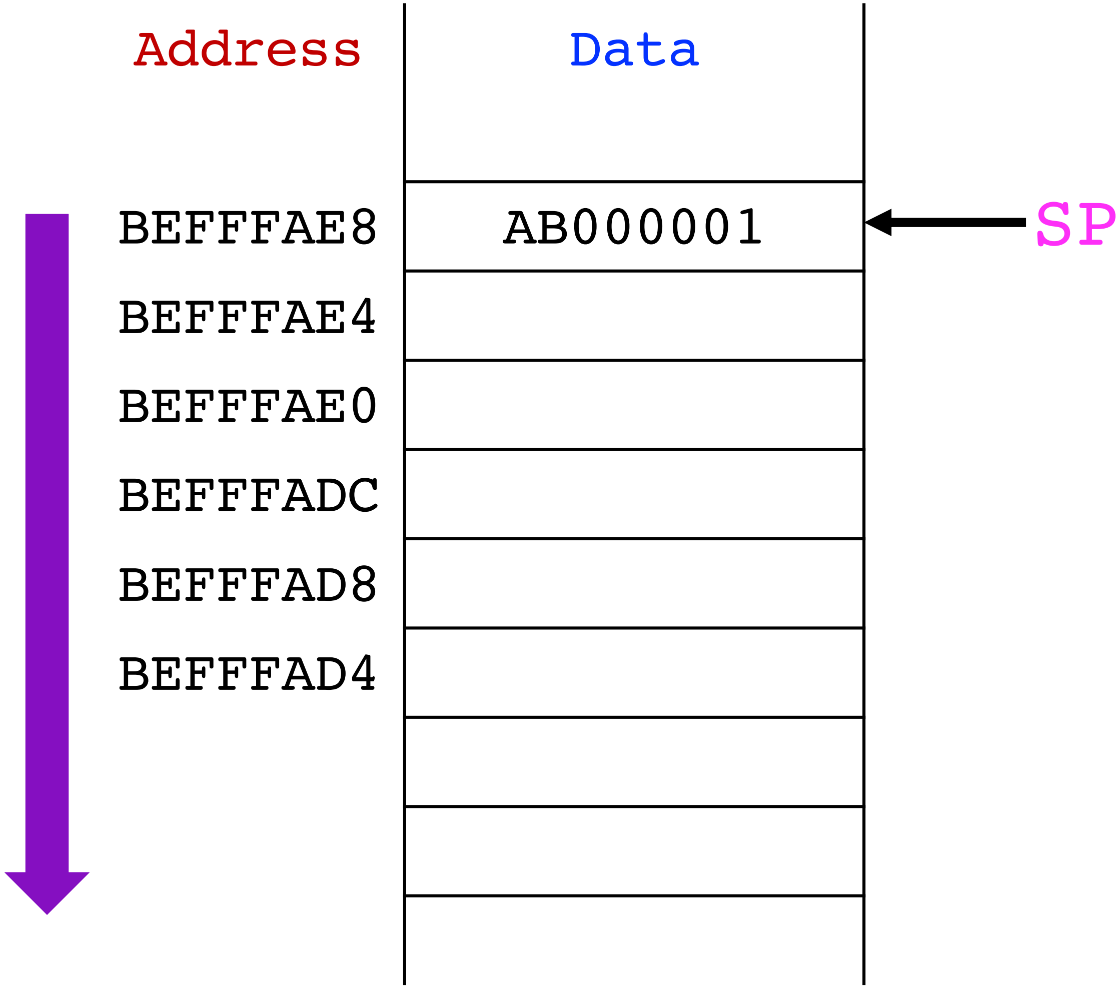
- ARM specifies a full descending stack, which we will assume in this course

The Stack

- ARM stack grows down in memory
- Stack Pointer (SP) points to the **top of the stack**
- SP register holds the address of (points to) the **top of the stack**

contents of stack pointer

SP 0xBEFFFAE8

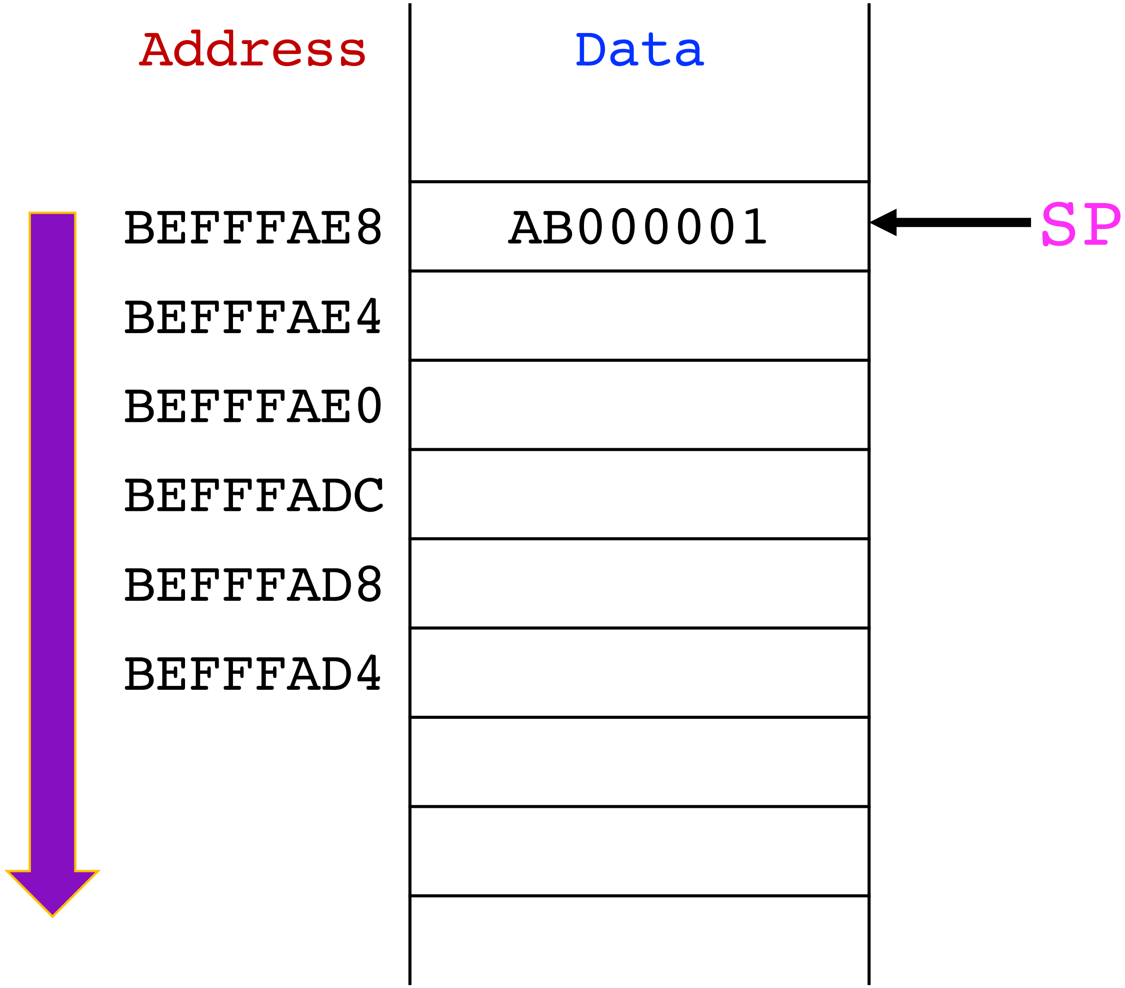


Growing the Stack

- Let us push two items on the stack
  - 0x12345678
  - 0xFFFFDDCC
- Where does the SP points to now?
- How does the stack look?

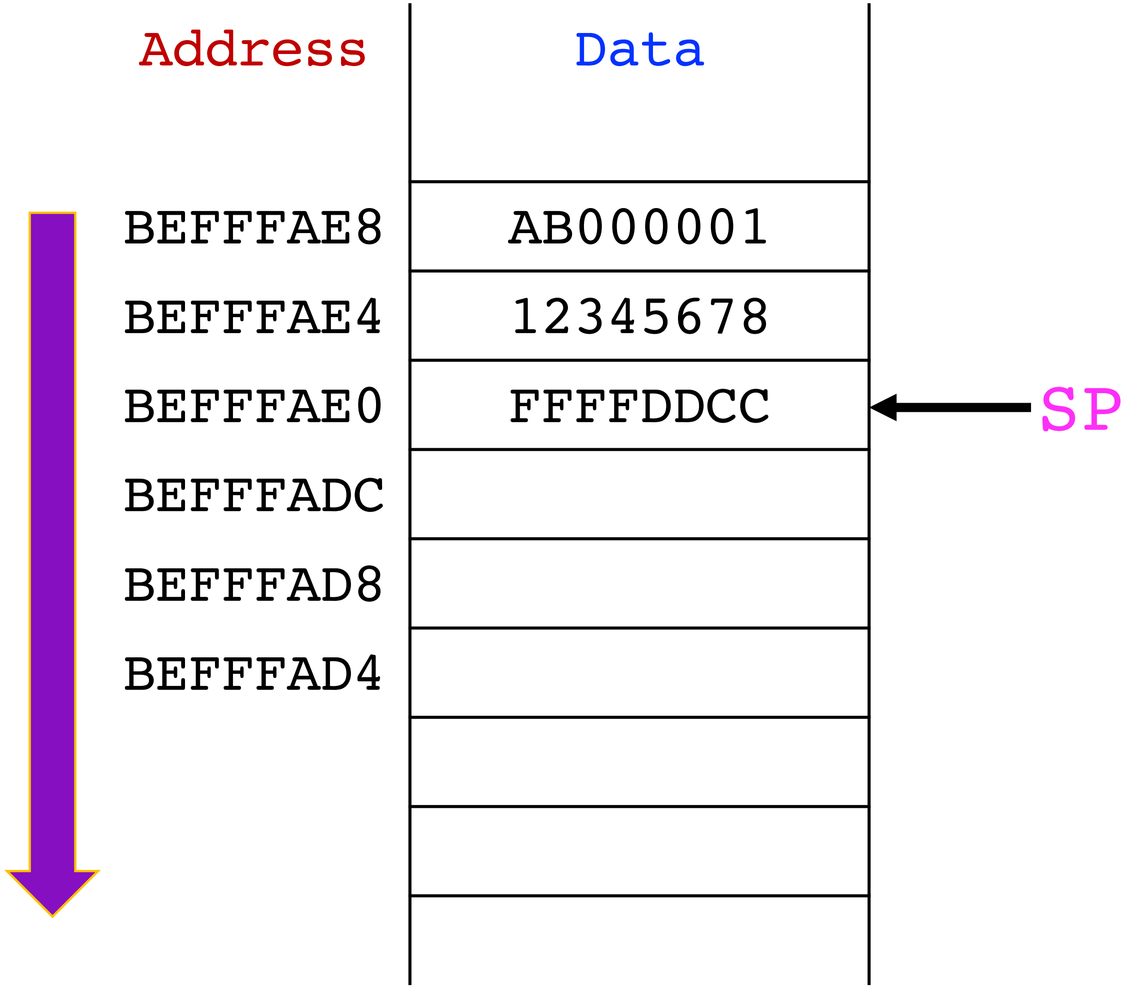
contents of stack pointer

SP 0xBEFFFAE8



Growing the Stack

- SP points to the most recently pushed item on the stack
- SP decrements by 8 to make space for two words



contents of stack pointer

SP `0xBEFFFAE0`

Saving and Restoring Registers

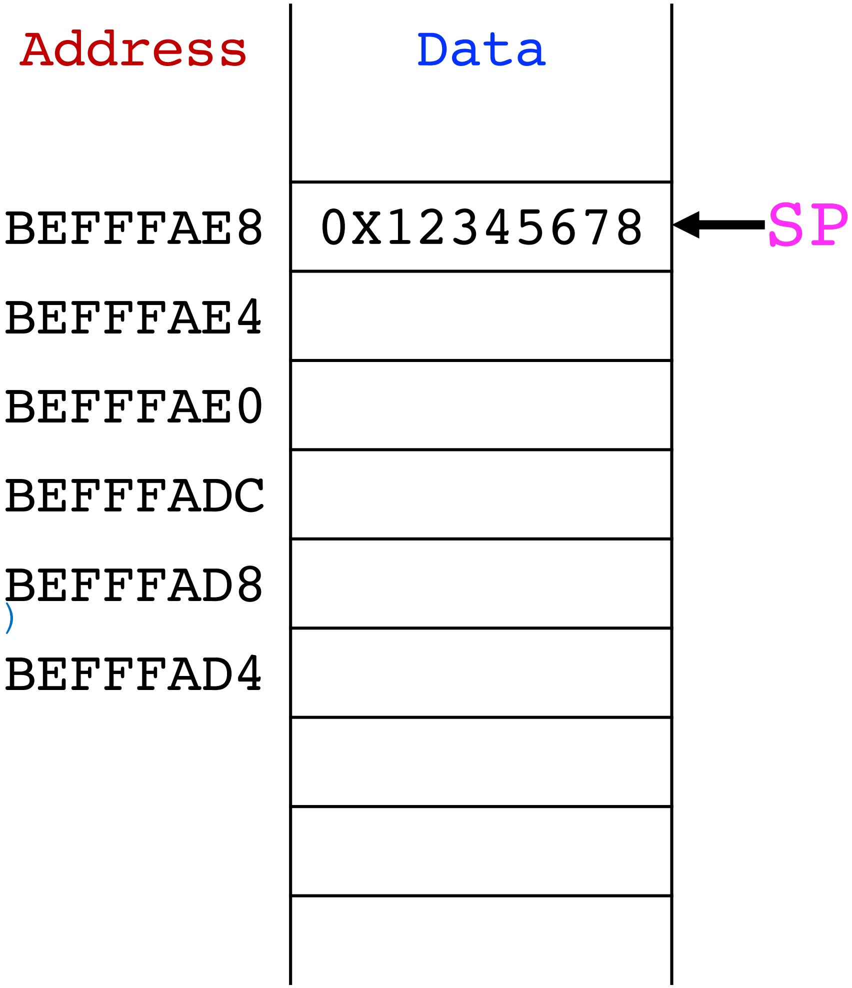
- `DIFFOFSUMS` **corrupts** three registers
  - Recall: Spy must not reveal their actions
  - No unintended side-effects (except leaving result in `R0`)
  - Callee should not corrupt caller’s execution

Saving and Restoring Registers

- Functions use the stack for saving/restoring registers
  - Allocate space on the stack (`SP = SP - 12`)
  - Store registers in use by the caller on the stack
  - Execute the function
  - Restore the registers from the stack
  - Deallocate space on the stack (`SP = SP + 12`)

Improved DIFFOFSUMS

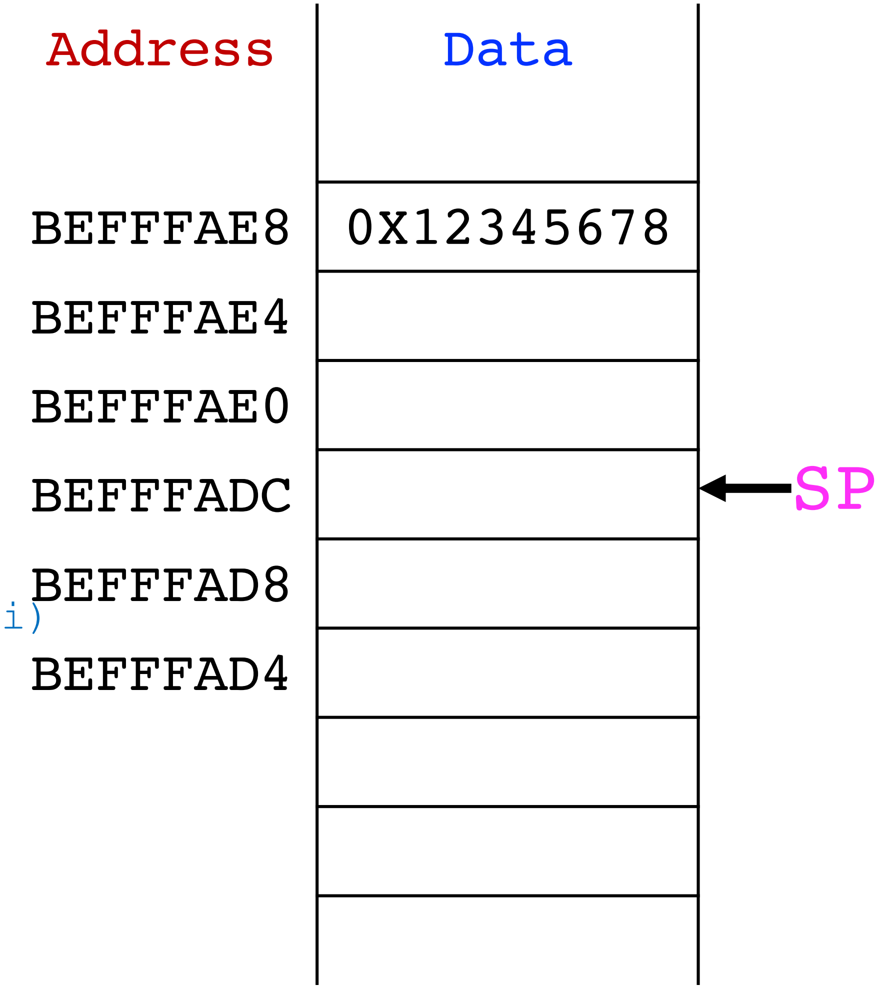
```
ARM Assembly Code
; R0 = result
DIFFOFSUMS
  SUB SP, SP, #12        ; make space on stack
                         ; for 3 registers
  STR R9, [SP, #8]       ; save R9 on stack
  STR R8, [SP, #4]       ; save R8 on stack
  STR R4, [SP]           ; save R4 on stack
  ADD R8, R0, R1         ; R8 = f + g
  ADD R9, R2, R3         ; R9 = h + i
  SUB R4, R8, R9         ; result = (f + g) - (h + i)
  MOV R0, R4             ; put return value in R0
  LDR R4, [SP]           ; restore R4 from stack
  LDR R8, [SP, #4]       ; restore R8 from stack
  LDR R9, [SP, #8]       ; restore R9 from stack
  ADD SP, SP, #12        ; deallocate stack space
  MOV PC, LR             ; return to caller
```



Improved DIFFOFSUMS

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12       ; make space on stack
                        ; for 3 registers
  STR R9, [SP, #8]      ; save R9 on stack
  STR R8, [SP, #4]      ; save R8 on stack
  STR R4, [SP]          ; save R4 on stack
  ADD R8, R0, R1        ; R8 = f + g
  ADD R9, R2, R3        ; R9 = h + i
  SUB R4, R8, R9        ; result = (f + g) - (h + i)
  MOV R0, R4            ; put return value in R0
  LDR R4, [SP]          ; restore R4 from stack
  LDR R8, [SP, #4]      ; restore R8 from stack
  LDR R9, [SP, #8]      ; restore R9 from stack
  ADD SP, SP, #12       ; deallocate stack space
  MOV PC, LR            ; return to caller
```



Improved DIFFOFSUMS

```
ARM Assembly Code
; R2 = result
DIFFOFSUMS
   SUB SP, SP, #12       ; make space on stack
                         ; for 3 registers
   STR R9, [SP, #8]      ; save R9 on stack
   STR R8, [SP, #4]      ; save R8 on stack
   STR R4, [SP]          ; save R4 on stack
   ADD R8, R0, R1        ; R8 = f + g
   ADD R9, R2, R3        ; R9 = h + i
   SUB R4, R8, R9        ; result = (f + g) - (h + i)
   MOV R0, R4            ; put return value in R0
   LDR R4, [SP]          ; restore R4 from stack
   LDR R8, [SP, #4]      ; restore R8 from stack
   LDR R9, [SP, #8]      ; restore R9 from stack
   ADD SP, SP, #12       ; deallocate stack space
   MOV PC, LR            ; return to caller
```

Address	Data	
BEFFFAE8	0X12345678	
BEFFFAE4	R9	
BEFFFAE0	R8	
BEFFFADC	R4	← SP
BEFFFAD8		
BEFFFAD4		

Improved DIFFOFSUMS

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12        ; make space on stack
                         ; for 3 registers
  STR R9, [SP, #8]       ; save R9 on stack
  STR R8, [SP, #4]       ; save R8 on stack
  STR R4, [SP]           ; save R4 on stack
  ADD R8, R0, R1         ; R8 = f + g
  ADD R9, R2, R3         ; R9 = h + i
  SUB R4, R8, R9         ; result = (f + g) - (h + i)
  MOV R0, R4             ; put return value in R0
  LDR R4, [SP]           ; restore R4 from stack
  LDR R8, [SP, #4]       ; restore R8 from stack
  LDR R9, [SP, #8]       ; restore R9 from stack
  ADD SP, SP, #12        ; deallocate stack space
  MOV PC, LR             ; return to caller
```

Address	Data	
BEFFFAE8	0X12345678	← SP
BEFFFAE4	R9	
BEFFFAE0	R8	
BEFFFADC	R4	
BEFFFAD8		
BEFFFAD4		

Calling Convention

- Preserving every register that a function uses is wasteful
  - DIFFOFSUMS preserves R4, R8, R9, but the caller may not be using R8 or R9

- Calling convention is a contract that callers and callees must follow

Calling Convention

- With a convention in place
  - Functions written by different programmers can interoperate
  - Functions compiled by two different compilers can interoperate
  - A library function written by third party can safely be used without worrying about corruption due to misplaced arguments and return value

ARM Calling Convention

- Preserved Registers
  - Registers that are preserved across function calls
  - Caller can expect these registers to appear as if a function call was never made
  - Callee must save and restore preserved registers

- Nonpreserved Registers
  - Caller must save these registers before making the function call
  - Their preservation is not the callee's responsibility

ARM Calling Convention

Preserved	Nonpreserved
Saved registers: `R4 – R11`	Temporary register: `R12`
Stack pointer: `SP (R13)`	Argument registers: `R0 – R3`
Return address: `LR (R14)`	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

- `SP` and `LR` are fancy names for `R13` and `R14`
- Stack above the stack pointer is preserved if the callee does not mess with the caller's stack space (a.k.a. stack frame)
- Stack pointer is preserved, because the caller deallocates the space it uses on the stack before returning

Rules for Caller and Callee

- Caller save rule: The caller must save any non-preserved registers that it needs after the call. After the call, it must restore these registers
- Callee save rule: Before a callee disturbs any of the preserved registers, it must save these registers. Before the return, it must restore these registers

PUSH and POP Instructions

- **PUSH:** Saves registers on the stack
  - `PUSH {R4}` stores R4 on to the stack and **adds 4 to** SP
- **POP:** Restores registers from the stack
  - `POP {R4}` stores [SP] in R4 and **subtracts 4 from** SP
- Can store multiple registers on the stack in a single PUSH
  - `PUSH {R4, R8, LR}`

R13 stored at highest memory address

lowest-numbered reg stored at lowest memory address

C Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);

  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH  {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH  {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP   {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP   {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR} ; save regs
  ADD   R5,  R0, R1  ; x = (a+b)
  SUB   R12, R0, R1  ; temp = (a-b)
  MUL   R5,  R5, R12 ; x = x*temp
  MOV   R4,  #0      ; i = 0
FOR
  CMP   R4, R0       ; i < a?
  BGE   RETURN       ; no: exit loop
  PUSH {R0, R1}      ; save regs
  ADD   R0, R1, R4   ; arg is b+i
  BL    F2           ; call f2(b+i)
  ADD   R5, R5, R0   ; x = x+f2(b+i)
  POP  {R0, R1}      ; restore regs
  ADD   R4, R4, #1   ; i++
  B     FOR          ; repeat loop
RETURN
  MOV   R0, R5       ; return x
  POP  {R4, R5, LR}  ; restore regs
  MOV   PC, LR       ; return
```

```
; R0=p, R4=r
F2
  PUSH {R4}          ; save regs
  ADD   R4, R0, 5    ; r = p+5
  ADD   R0, R4, R0   ; return r+p
  POP  {R4}          ; restore regs
  MOV   PC, LR       ; return
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH   {R4,  R5, LR}
   ADD    R5,   R0, R1
   SUB    R12, R0, R1
   MUL    R5,   R5, R12
   MOV    R4,   #0
FOR
   CMP    R4, R0
   BGE    RETURN
   PUSH   {R0, R1}
   ADD    R0, R1, R4
   BL     F2
   ADD    R5, R5, R0
   POP    {R0, R1}
   ADD    R4, R4, #1
   B      FOR
RETURN
   MOV    R0, R5
   POP    {R4, R5, LR}
   MOV    PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD   R4, R0, 5
   ADD   R0, R4, R0
   POP  {R4}
   MOV   PC, LR
```

Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	
BEFFFADC	R1	
BEFFFAD8	R0	← SP
BEFFFAD4		

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH   {R4,  R5, LR}
   ADD    R5,   R0, R1
   SUB    R12, R0, R1
   MUL    R5,   R5, R12
   MOV    R4,   #0
FOR
   CMP    R4, R0
   BGE    RETURN
   PUSH   {R0, R1}
   ADD    R0, R1, R4
   BL     F2
   ADD    R5, R5, R0
   POP    {R0, R1}
   ADD    R4, R4, #1
   B      FOR
RETURN
   MOV    R0, R5
   POP    {R4, R5, LR}
   MOV    PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD    R4, R0, 5
   ADD    R0, R4, R0
   POP    {R4}
   MOV    PC, LR
```

Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	
BEFFFADC	R1	
BEFFFAD8	R0	← SP
BEFFFAD4	R4	

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH   {R4,  R5, LR}
   ADD    R5,   R0, R1
   SUB    R12, R0, R1
   MUL    R5,   R5, R12
   MOV    R4,   #0
FOR
   CMP    R4, R0
   BGE    RETURN
   PUSH   {R0, R1}
   ADD    R0, R1, R4
   BL     F2
   ADD    R5, R5, R0
   POP    {R0, R1}
   ADD    R4, R4, #1
   B      FOR
RETURN
   MOV    R0, R5
   POP    {R4, R5, LR}
   MOV    PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD    R4, R0, 5
   ADD    R0, R4, R0
   POP   {R4}
   MOV    PC, LR
```

Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	← SP
BEFFFADC	R1	
BEFFFAD8	R0	
BEFFFAD4	R4	

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH   {R4,  R5, LR}
   ADD    R5,   R0, R1
   SUB    R12,  R0, R1
   MUL    R5,   R5, R12
   MOV    R4,   #0
FOR
   CMP    R4, R0
   BGE    RETURN
   PUSH   {R0, R1}
   ADD    R0, R1, R4
   BL     F2
   ADD    R5, R5, R0
   POP    {R0, R1}
   ADD    R4, R4, #1
   B      FOR
RETURN
   MOV    R0, R5
   POP    {R4, R5, LR}
   MOV    PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD    R4, R0, 5
   ADD    R0, R4, R0
   POP  {R4}
   MOV    PC, LR
```

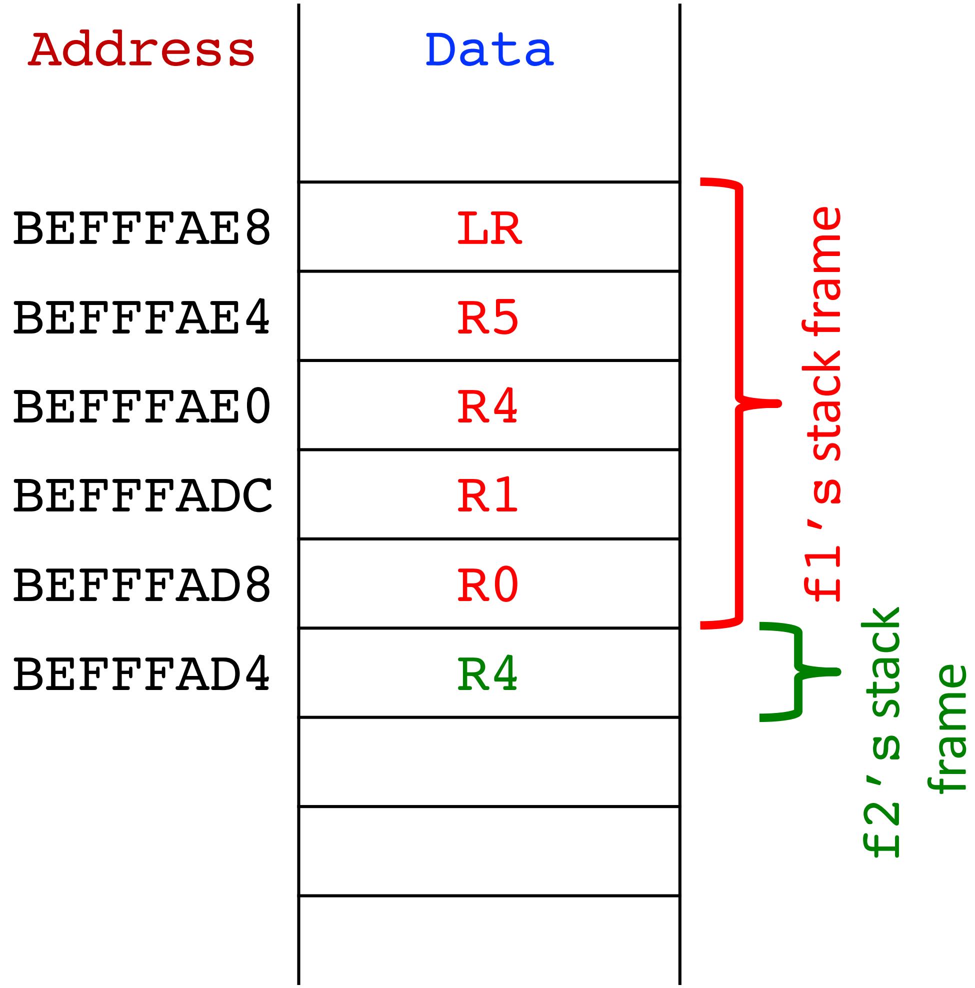
Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	SP
BEFFFADC	R1	
BEFFFAD8	R0	
BEFFFAD4	R4	

Exercise

- Provide two optimizations that reduce the stack space consumed by the previous program without impacting its correctness.

Stack Frame

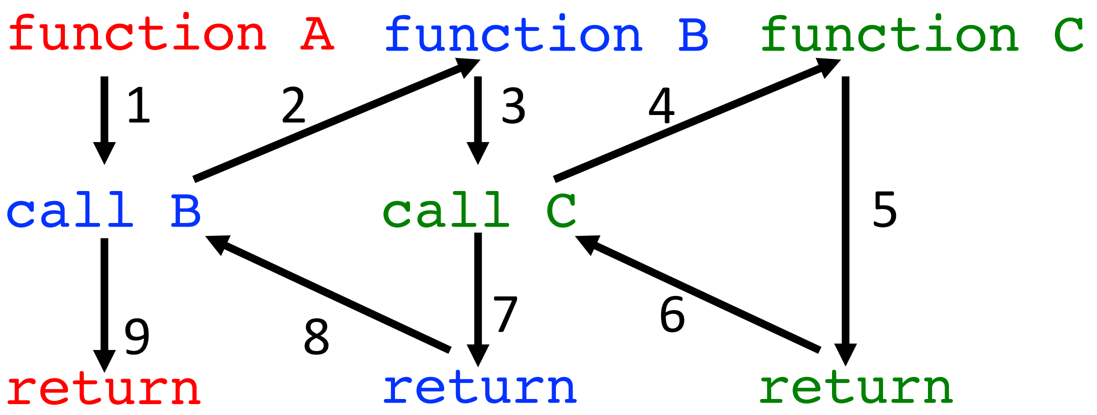
- The space that a function allocates on the stack is called its stack frame
  - Also called "activation record"
- **Execution Environment of function:** Stack frame, PC, preserved registers
- Caller's **execution env** must be preserved b/w call & return
- Callee's **execution env** must be installed on function invocation/activation

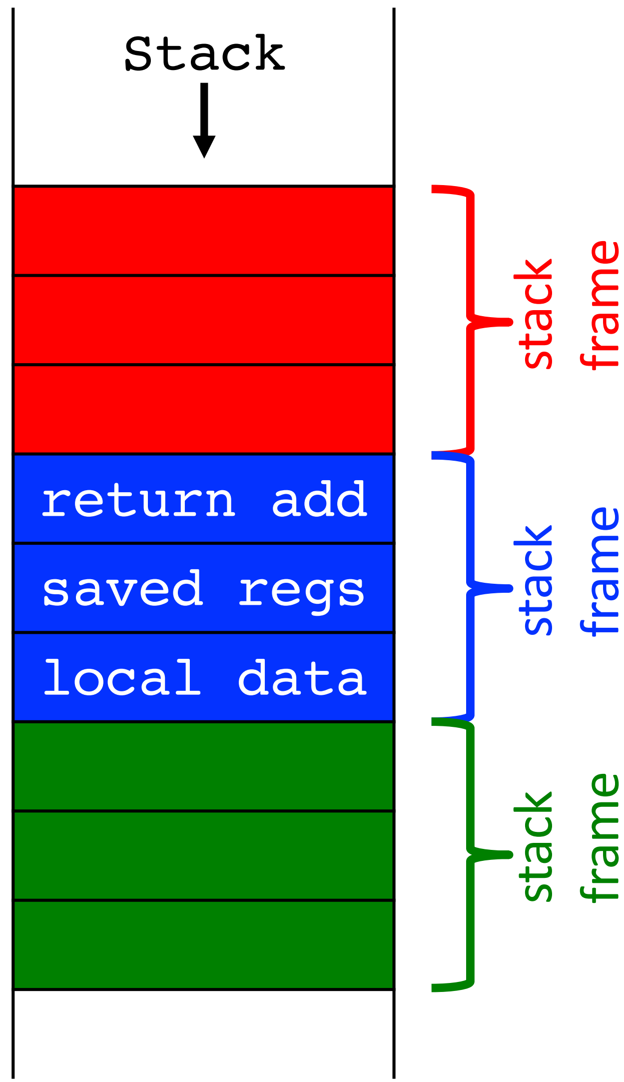


Stack Frame

- Many active frames during program execution

- **We call it the program's call stack**

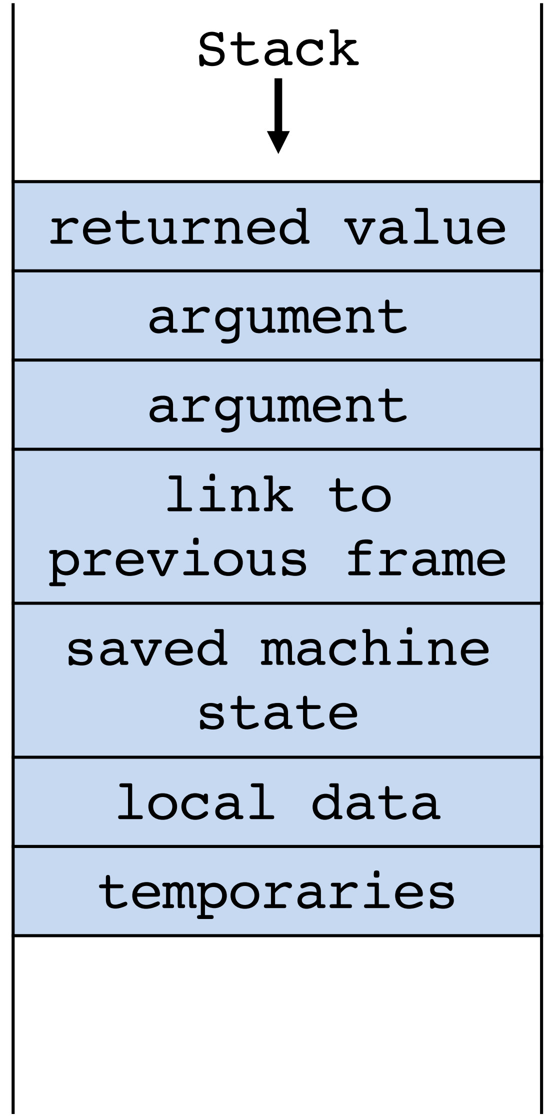




Things to Remember

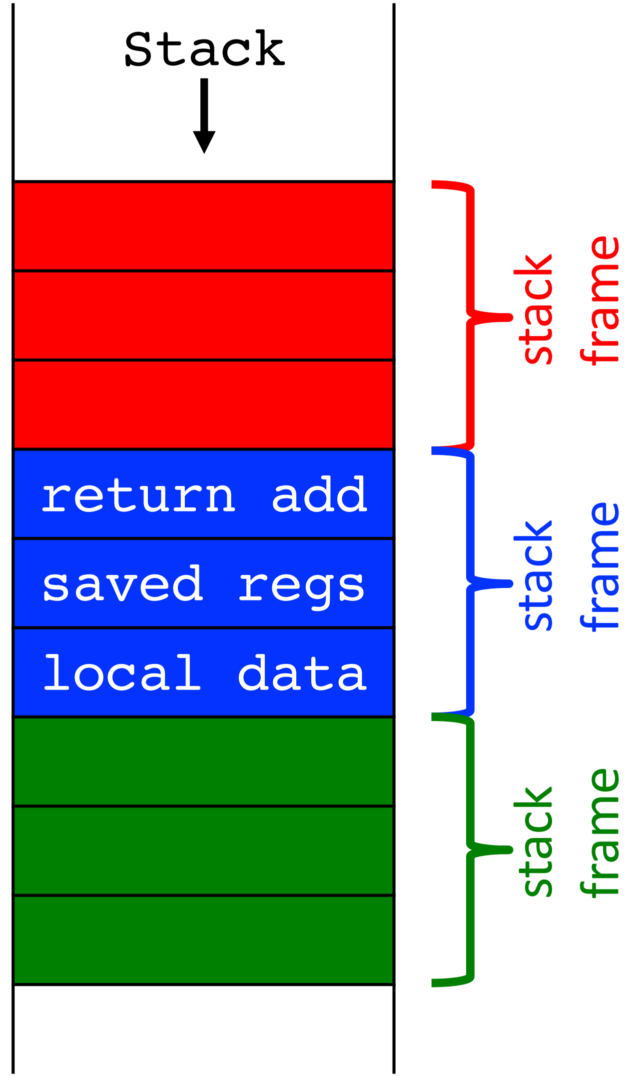
- The precise nature & layout of call stack depends on the compiler and architecture
- Stack is **not a hardware component**
- We **set aside** an area in memory and treat it as a stack

  - A different (more generic) <u>stack frame</u> is shown to the right



Group of Stack Frames

- Many names for **the call stack**
  - Execution Stack
  - Program Stack
  - Run-time Stack
  - Control Stack
  - Machine Stack
  - Activation Stack



Summary

- **Caller**
  - Puts arguments in `R0-R3`
  - Saves any needed registers (`R0-R3, R12`)
  - Call function: `BL CALLEE`
  - Restores registers
  - Looks for result in `R0`

- **Callee**
  - Saves registers that might be disturbed (`R4-R11, LR`)
  - Executes the function body (a.k.a. performs the function)
  - Puts the result in `R0`
  - Restores registers
  - Returns: `MOV PC, LR`

Recursion

- Recursion is a **powerful programming technique**
  - Clarity, simplicity, and convenience
- A recursive function is a non-leaf that calls itself
  - Both caller and callee at the same time

```
n = 0, factorial(0) = 1
n = 1, factorial(1) = 1
n = 2, factorial(2) = 2
n = 3, factorial(3) = 6
n = 4, factorial(4) = 24
n = 5, factorial(5) = 120
n = 6, factorial(6) = 720
and so on ....
```

C Code

```c
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

factorial(3)

C Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

```
n = 3, factorial(3) = 3 * factorial(2)
                    = 3 * 2 * factorial(1)
                    = 3 * 2 * 1 * factorial(0)
                    = 3 * 2 * 1 * 1
                    = 6
```

Recursion

ARM Assembly Code

```
0x8500 FACTORIAL  PUSH  {R0, LR}      ;Push n and LR on stack
0x8504            CMP   R0, #1        ;R0 <= 1?
0x8508            BGT   ELSE          ;no: branch to else
0x850C            MOV   R0, #1        ;otherwise, return 1
0x8510            ADD   SP, SP, #8    ;restore SP
0x8514            MOV   PC, LR        ;return
0x8518 ELSE       SUB   R0, R0, #1    ;n = n - 1
0x851C            BL    FACTORIAL     ;recursive call
0x8520            POP   {R1, LR}      ;pop n (into R1) and LR
0x8524            MUL   R0, R1, R0    ;R0 = n*factorial(n-1)
0x8528            MOV   PC, LR        ;return
```

factorial(3)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x1000

R0 0x0003

Address	Data	
BEFFFAE8		← SP
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(3)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x1000

R0 0x0003

Address	Data
BEFFFAE8	LR (0x1000)
BEFFFAE4	R0 (3) ← SP
BEFFFAE0	
BEFFFADC	
BEFFFAD8	
BEFFFAD4	
BEFFFAD4	
BEFFFAD4	
BEFFFAD4	

factorial(2)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x8520

R0 0x0002

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	← SP
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(2)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x8520

R0 0x0002

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	← SP
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(1)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	← SP
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(1)

ARM Assembly Code

```
0x8500 FACTORIAL   PUSH  {R0, LR}
0x8504             CMP   R0, #1
0x8508             BGT   ELSE
0x850C             MOV   R0, #1
0x8510             ADD   SP, SP, #8
0x8514             MOV   PC, LR
0x8518 ELSE        SUB   R0, R0, #1
0x851C             BL    FACTORIAL
0x8520             POP   {R1, LR}
0x8524             MUL   R0, R1, R0
0x8528             MOV   PC, LR
```

LR 0x8520

R0 0x0001

Address	Data
BEFFFAE8	LR (0x1000)
BEFFFAE4	R0 (3)
BEFFFAE0	LR (0x8520)
BEFFFADC	R0 (2)
BEFFFAD8	LR (0x8520)
BEFFFAD4	R0 (1) ← SP
BEFFFAD4	
BEFFFAD4	
BEFFFAD4	

factorial(1)

ARM Assembly Code

```
0x8500 FACTORIAL   PUSH  {R0, LR}
0x8504             CMP   R0, #1
0x8508             BGT   ELSE
0x850C             MOV   R0, #1
0x8510             ADD   SP, SP, #8
0x8514             MOV   PC, LR
0x8518 ELSE        SUB   R0, R0, #1
0x851C             BL    FACTORIAL
0x8520             POP   {R1, LR}
0x8524             MUL   R0, R1, R0
0x8528             MOV   PC, LR
```

LR 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	← SP
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

R0 = 1

ARM Assembly Code

```
0x8500 FACTORIAL   PUSH  {R0, LR}
0x8504             CMP   R0, #1
0x8508             BGT   ELSE
0x850C             MOV   R0, #1
0x8510             ADD   SP, SP, #8
0x8514             MOV   PC, LR
0x8518 ELSE        SUB   R0, R0, #1
0x851C             BL    FACTORIAL
0x8520             POP   {R1, LR}
0x8524             MUL   R0, R1, R0
0x8528             MOV   PC, LR
```

LR 0x8520 PC 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	← SP
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

R0 = 2 x 1

ARM Assembly Code

```
0x8500 FACTORIAL   PUSH  {R0, LR}
0x8504             CMP   R0, #1
0x8508             BGT   ELSE
0x850C             MOV   R0, #1
0x8510             ADD   SP, SP, #8
0x8514             MOV   PC, LR
0x8518 ELSE        SUB   R0, R0, #1
0x851C             BL    FACTORIAL
0x8520             POP   {R1, LR}
0x8524             MUL   R0, R1, R0
0x8528             MOV   PC, LR
```

LR	0x8520
PC	0x8520
R0	0x0002
R1	0x0002

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	← SP
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

R0 = 3 X 2 = 6

ARM Assembly Code

```
0x8500 FACTORIAL   PUSH  {R0, LR}
0x8504             CMP   R0, #1
0x8508             BGT   ELSE
0x850C             MOV   R0, #1
0x8510             ADD   SP, SP, #8
0x8514             MOV   PC, LR
0x8518 ELSE        SUB   R0, R0, #1
0x851C             BL    FACTORIAL
0x8520             POP   {R1, LR}
0x8524             MUL   R0, R1, R0
0x8528             MOV   PC, LR
```

LR	0x1000
PC	0x1000
R0	0x0006
R1	0x0003

Address	Data
	← SP
BEFFFAE8	LR (0x1000)
BEFFFAE4	R0 (3)
BEFFFAE0	LR (0x8520)
BEFFFADC	R0 (2)
BEFFFAD8	LR (0x8520)
BEFFFAD4	R0 (1)
BEFFFAD4	
BEFFFAD4	
BEFFFAD4	

Is recursion worth the trouble?

- There is an alternative to solving a problem using recursion
  - Any recursive solution has an equivalent iterative solution (**mathematically sound statement**)
  - **Exercise:** Write `factorial(int n)` with an iterative statement

- Overheads of recursion
  - (CPU) Extra branch instructions due to function calls
  - (Memory) Extra memory is consumed by the stack frames

- In many areas, the convenience is worth the trouble
  - Neural networks, data structures, recursive descent parsers

Summary of `factorial`

- `factorial` saves `LR` according to the callee save rule
- `factorial` saves `R0` according to the caller save rule, because it will need `n` after calling itself
- if `n is less than or equal to 1` put the result (i.e., 1) in `R0` and return (no need to restore `LR` because it is unchanged)
- Use `R1` for restoring `n`, so as not to overwrite the returned value
- The multiply instruction `(MUL R0, R1, R0)` multiplies `n (in R1)` and the returned value `(in R0)` and puts the result in `R0`