ENGN2219/COMP6719
Computer Systems & Organization

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

Recap: Function Calls

C Code

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

void simple() {
  return;
}
```

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` — makes `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

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

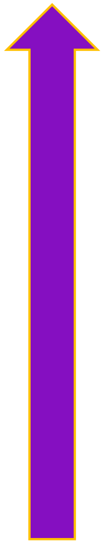
- How can we pass more than 4 function arguments?
- How can we ensure that registers in use by the caller are not corrupted?
  - `DIFFOFSUMS` overwrites `R4, R8, R9`
  - `MAIN` may need these registers after return
- The Stack
  - An area in memory used across function calls
  - Preserving/saving registers, passing extra arguments, local variables, temporary space

The Stack

- Abstract view
  - Last In First Out (LIFO) Queue
- push
  - Put a new plate on top
- pop
  - Remove a plate from top
- Stack expands and contracts as plates are added and removed

The Stack

- Stored at some arbitrary address in memory
- `push {R0}`
  - Store `R0` onto the stack
- `pop {R0}`
  - Restore `R0` with whatever is at the top of the stack
- Caller & callee can preserve registers on the stack, place arguments, and use it for temporary data

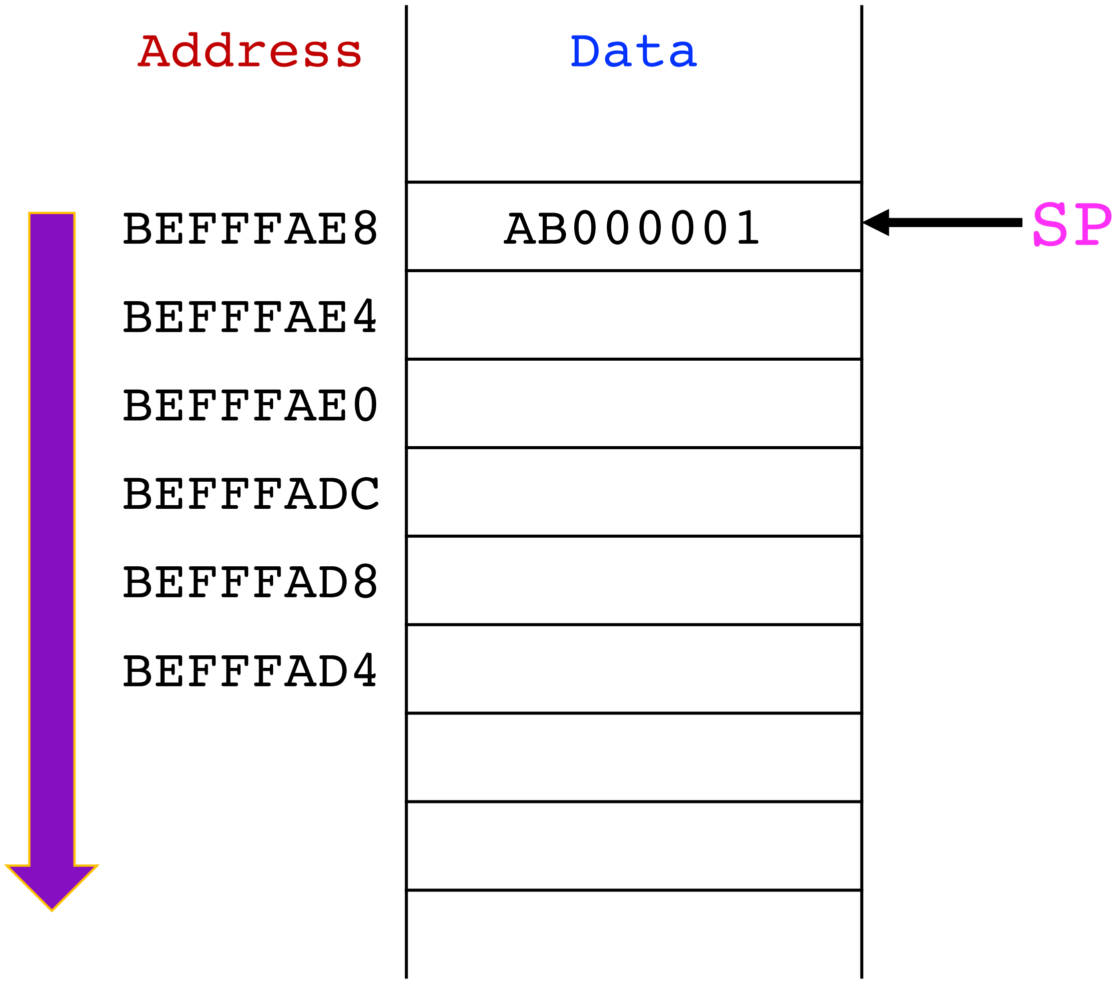


The Stack

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

contents of stack pointer

SP 0xBEFFFAE8

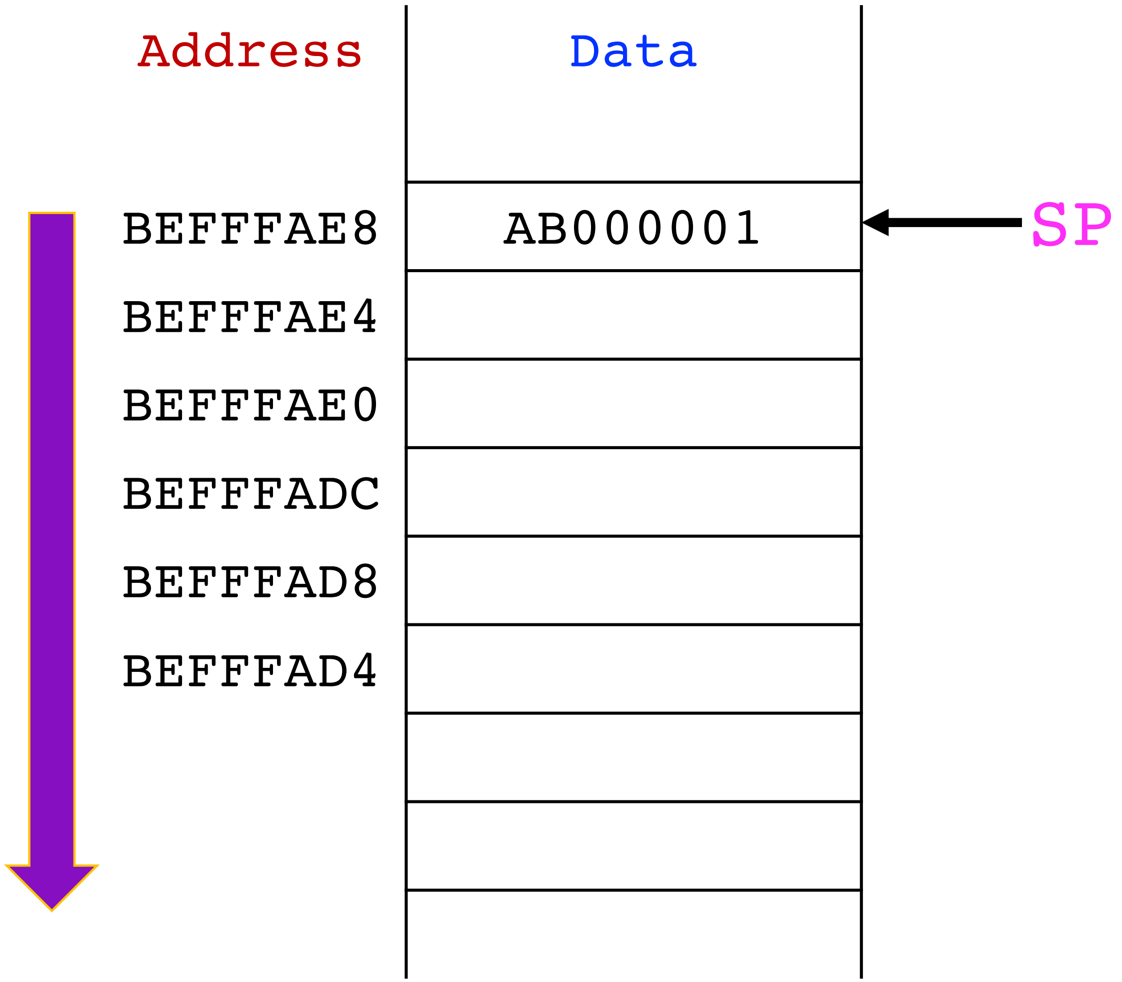


Growing the Stack

- Let's `push` two items on the stack
  - `0x12345678`
  - `0xFFFFDDCC`
- Where does the `SP` points now?
- How does the stack look?

contents of stack pointer

`SP` `0xBEFFFAE8`

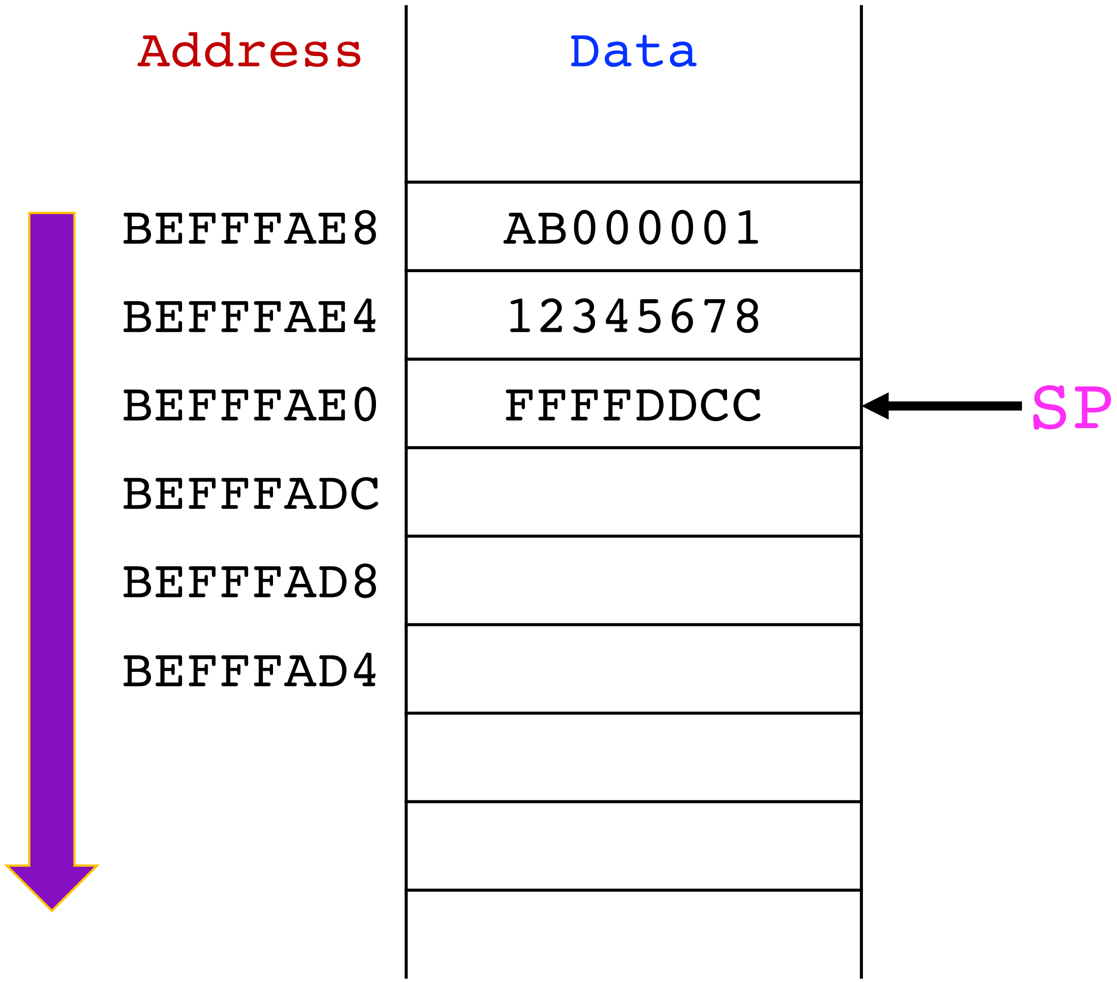


Growing the Stack

- 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/Restoring Registers

- `DIFFOFSUMS` (previous lecture) corrupts 3 registers
  - Spy must not reveal their actions
  - No unintended side-effects (except using `R0` for result)
  - Callee should not corrupt caller's execution

Saving/Restoring Registers

- `DIFFOFSUMS` (previous lecture) corrupts 3 registers
  - Spy must not reveal their actions
  - No unintended side-effects (except using `R0` for result)
  - Callee should not corrupt caller's execution
- 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

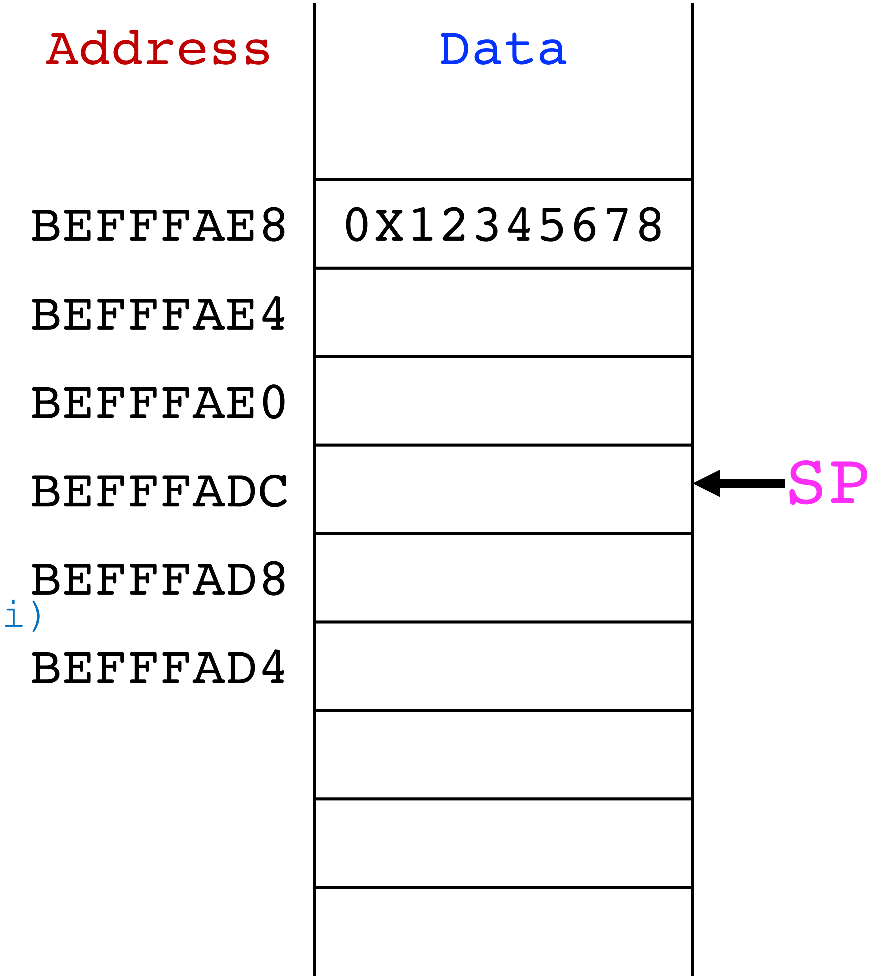
ARM Assembly Code

```
; 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
```

Address	Data	
BEFFFAE8	0X12345678	← SP
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
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
```



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`
- We need a convention/contract that callers and callees must follow
- Functions compiled by two different compilers can interoperate
- You can use a library function (written by third party) 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
- **`POP`:** Restores registers from the stack
  - `POP {R4}` stores `[SP]` in `R4`
- 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
```

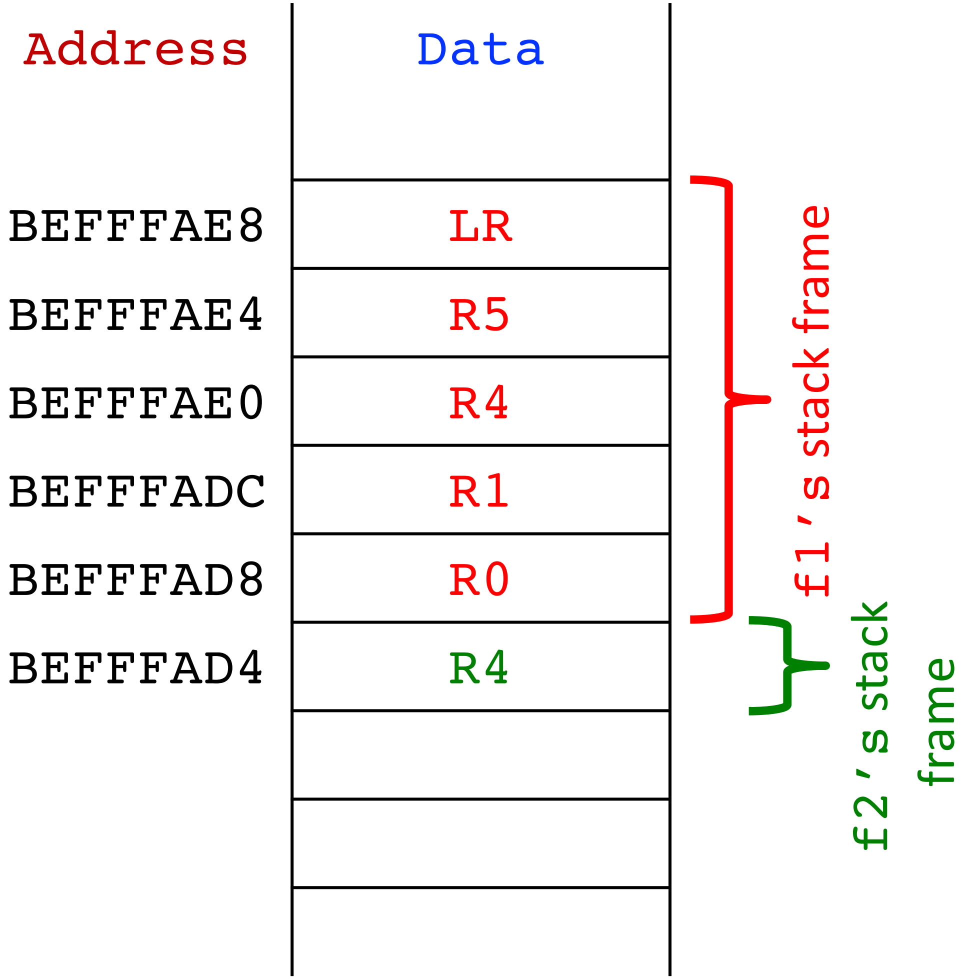
```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD    R4, R0, 5
   ADD    R0, R4, R0
   POP    {R4}
   MOV    PC, LR
```

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

Question: *Can you spot a register being pushed on the stack needlessly?*

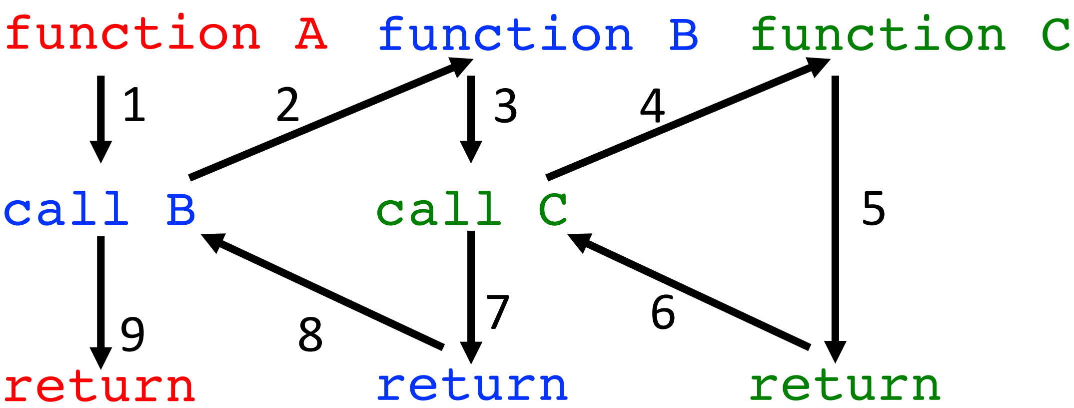
Stack Frame

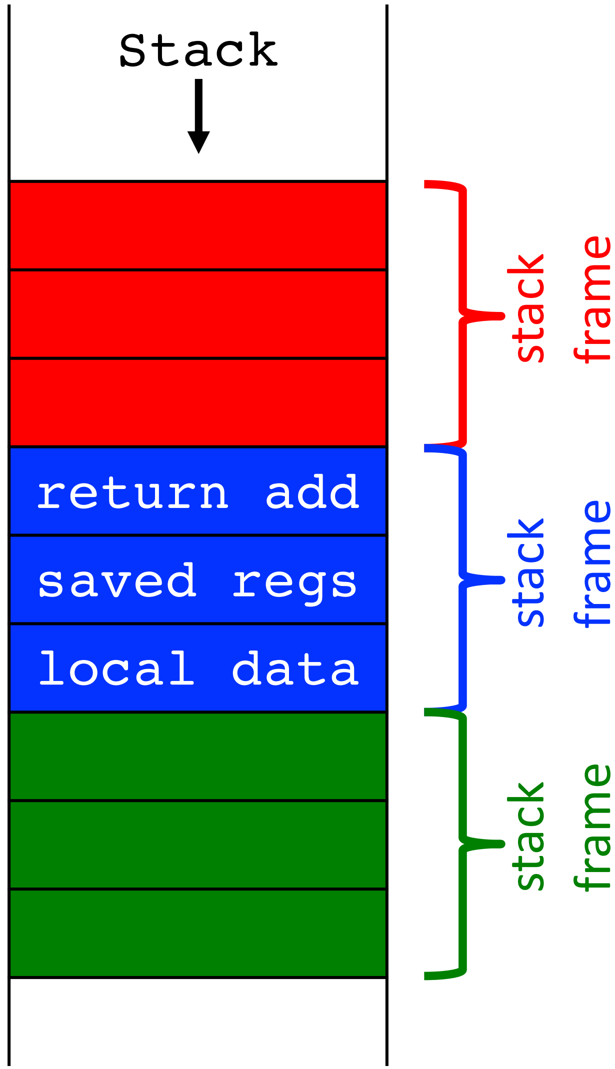
- The space that a function allocates on the stack is called its stack frame
- Also called *activation record* or *activation frame* or *execution environment (env)*
- In general,
  - Caller's execution env must be preserved b/w call & return
  - Callee's execution env must be installed on function activation (invocation)



Stack Frame

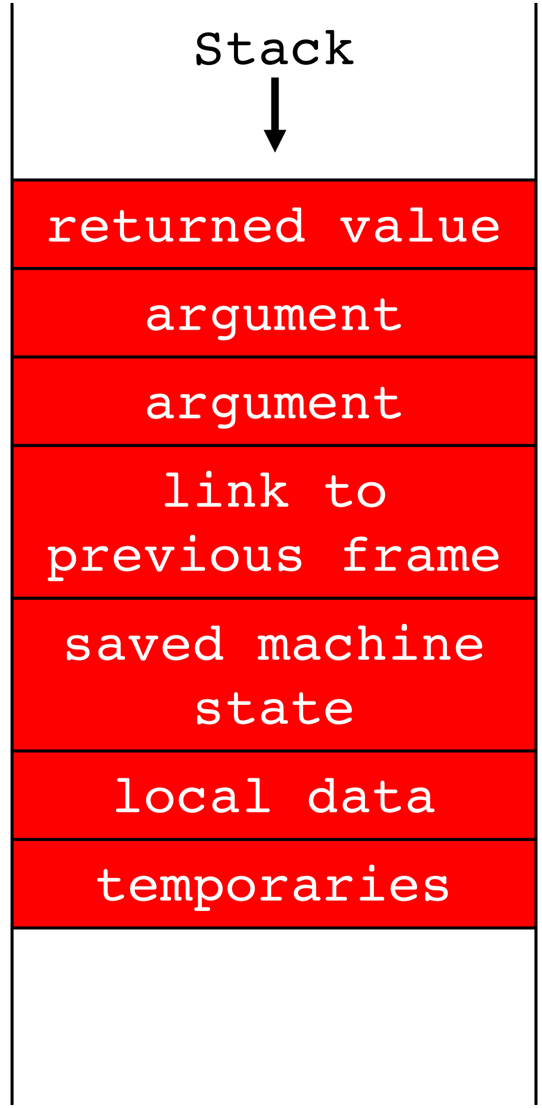
- Many frames can be active on the stack during program execution





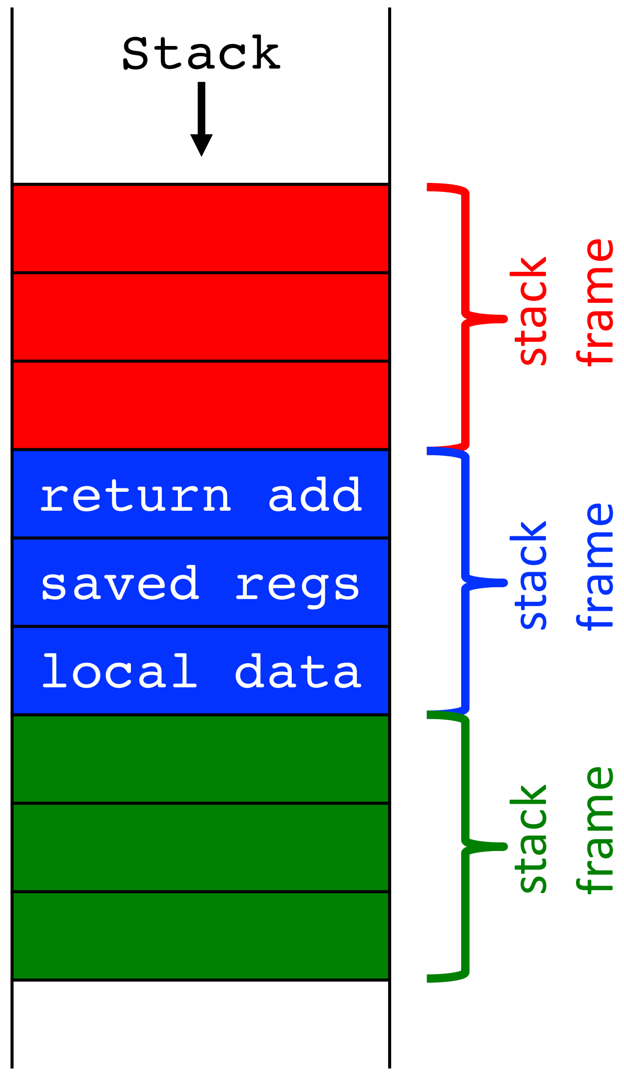
Stack Frame

- 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 by having a pointer to the top
- A more general stack frame is shown to the right



Fancy Names

- Execution Stack
- Program Stack
- Run-time Stack
- Control Stack
- Machine Stack
- Activation Stack
- .... Its The Stack



Summary

- **Caller**
  - Puts arguments in `R0-R3`
  - Saves any needed registers (`LR`, maybe `R0-R3, R8-R12`)
  - Calls function: `BL CALLEE`
  - Restores registers
  - Looks for result in `R0`
- **Callee**
  - Saves registers that might be disturbed (`R4-R7`)
  - Performs function
  - Puts result in `R0`
  - Restores registers
  - Returns: `MOV PC, LR`

Recursion

- Recursion is a powerful programming tool
  - Clarity, simplicity, 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

```
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
BEFFFAE8	LR (0x1000)
BEFFFAE4	R0 (3)
BEFFFAE0	LR (0x8520)
BEFFFADC	R0 (2)
BEFFFAD8	LR (0x8520)
BEFFFAD4	R0 (1)
BEFFFAD4	
BEFFFAD4	
BEFFFAD4	

SP

Is recursion worth the trouble?

- Alternative to recursion
  - Any recursive solution has an equivalent iterative solution (mathematically sound)
  - (Exercise) Write `factorial(2)` with an iterative (for/while) statement
- Overheads of recursion
  - (CPU) Extra instructions due to function calls
  - (Memory) Extra memory consumed by the stack frames
- In many areas, the convenience is worth the trouble
  - Neural networks, data structures, recursive descent parsers

Summary of `factorial`

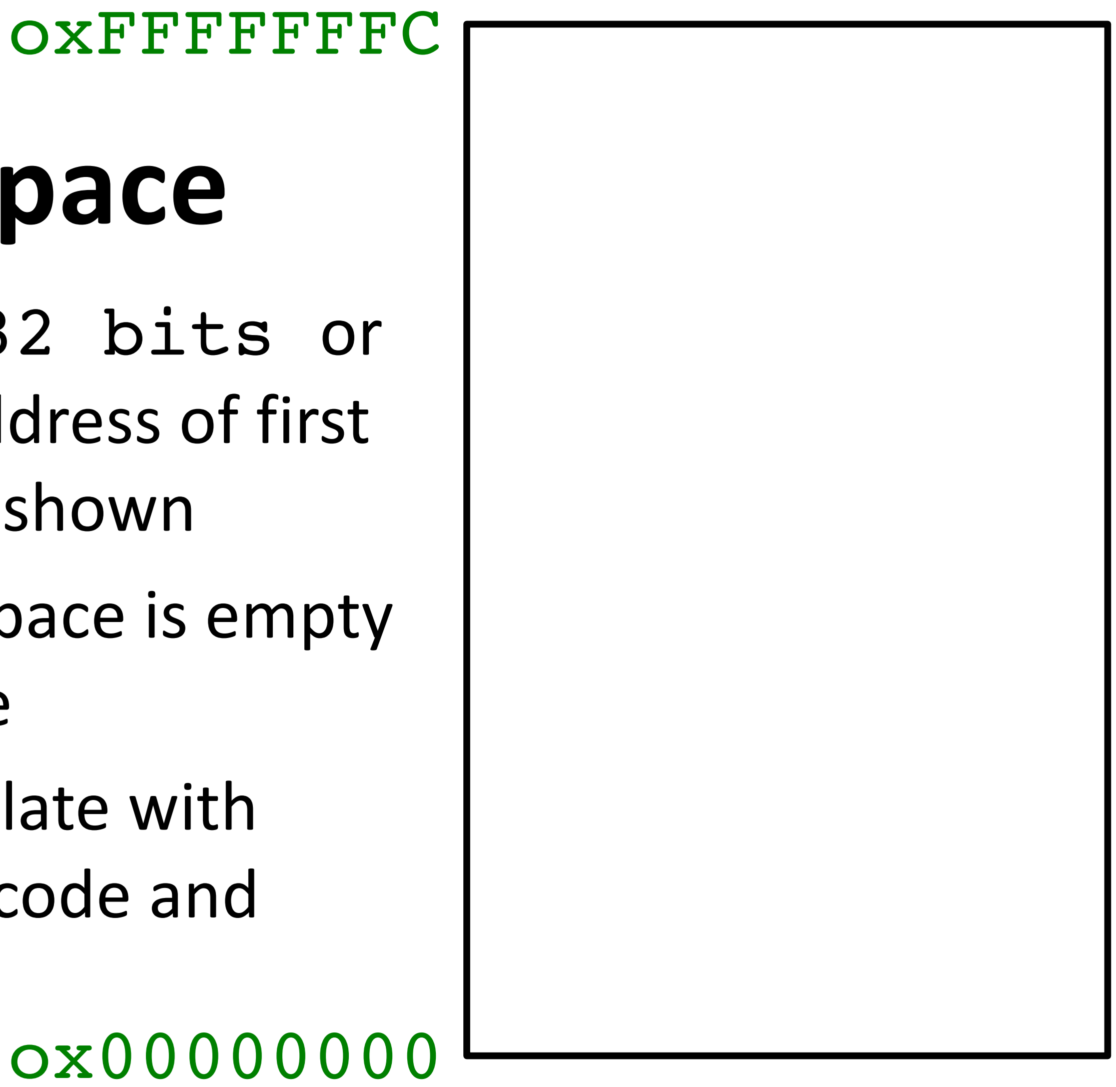
- `factorial` saves `LR` according to the callee save rule
- `factorial` saves `R0` according to the caller save rule, because it will need `n` after calling itself
- if `n is less than or equal to 1` put the result (1) in `R0` and return (no need to restore `LR` because it is unchanged)
- Use `R1` for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

Address Space

- **Address range**
  - A `32-bit` (ARM) CPU generates addresses in the range `0` to `0xFFFFFFFC` (`4294967292`)
  - With a 4×10^9 address range, the CPU can access `4 billion` individual bytes
- **Address space**
  - The address space of a `32-bit` CPU is 2^{32} bytes which equals `4 Gigabytes (GB)`

Address Space

- Each word is `32 bits` or `4 bytes`. Address of first & last word is shown
- The address space is empty as shown here
  - Let's populate with stack and code and data

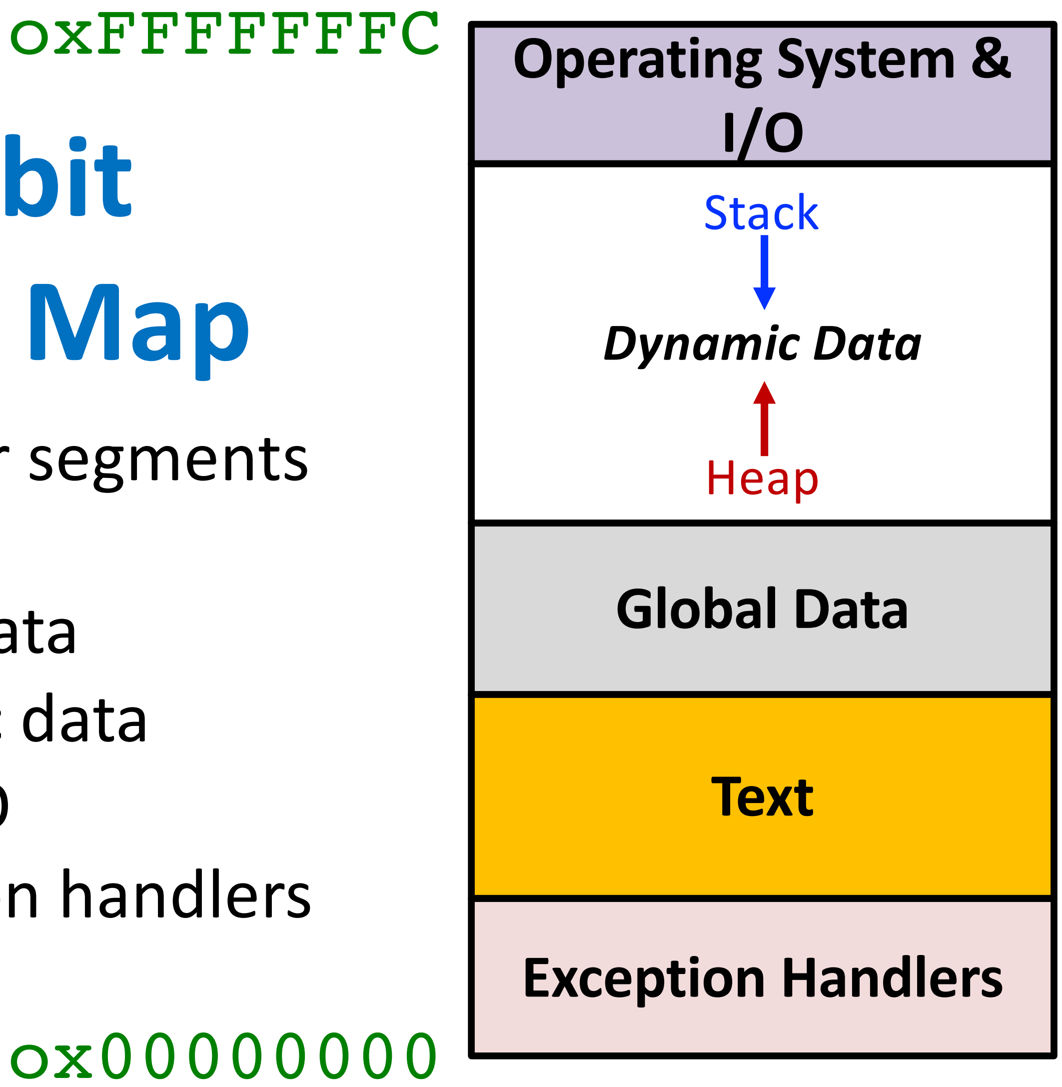


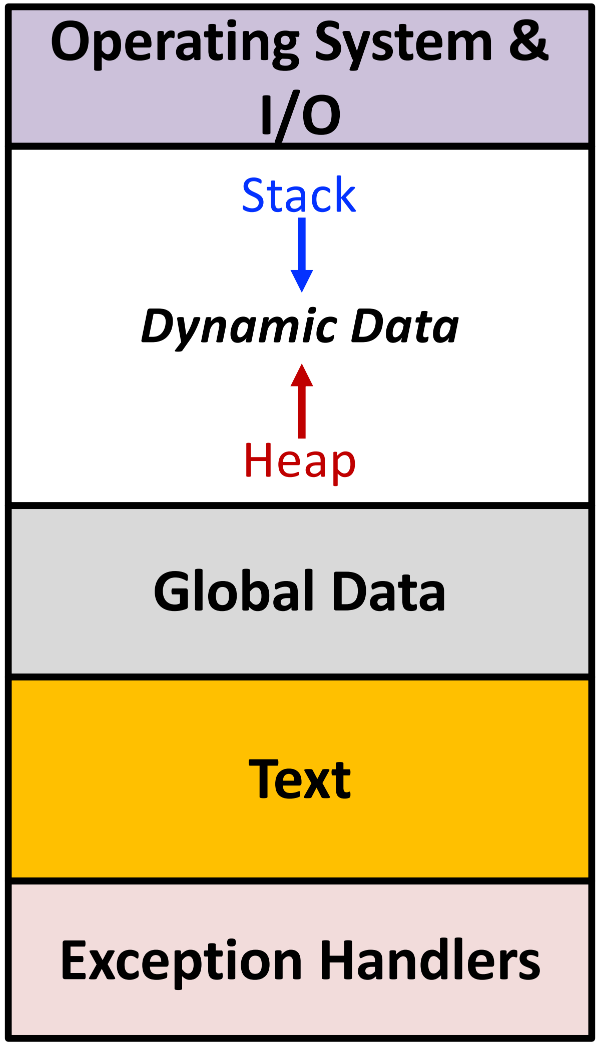
Questions

- Where is the code, data, and the stack in the address space?
- **Memory map**
  - Defines where code, data, and stack memory are in the program address space
  - Differs from architecture to architecture
  - The subsequent discussion pertains to ARM

ARM 32-bit Memory Map

- Five parts or segments
  - text
  - global data
  - dynamic data
  - OS & I/O
  - Exception handlers





- *Data in this segment is dynamically allocated and deallocated during program execution*
- *Heap data is allocated by the program at runtime*
  - `malloc()` and `new`
- *Heap grows upward, stack grows downward*

- *Global variables visible to all functions (contrasted with local variables that are only visible to a function)*

- *Machine language program*
- *Also called read-only (**RO**) segment*
- *Literals (constants) such as "Hello"*

Translating/Starting Programs